Optimal Reconfiguration Sequence Management

Soheil Ghiasi, Majid Sarrafzadeh

Computer Science Department
University of California, Los Angeles
{soheil,majid}@cs.ucla.edu

ABSTRACT

In this paper, we present an efficient optimal algorithm for minimizing runtime reconfiguration (context switching) delay of executing an application on a reconfigurable system. We assume that the basic operations of the application are already scheduled and each of them has to be realized on the reconfigurable fabric in order to be executed. The modeling and algorithm are both applicable to partially reconfigurable platforms as well as Multi-FPGA systems. The algorithm can be directly applied to minimize the application runtime for many typical classes of applications, where the actual execution delay of basic operations is negligible compared to reconfiguration delay. We prove the optimality and efficiency of our algorithm and report experimental results, which demonstrate 40% to 2.5% improvement in total runtime reconfiguration delay.

1. INTRODUCTION

Many applications contain computationally intensive blocks and hence they demand hardware implementation to exhibit real-time performance. Dedicated hardware solutions are capable of running many operations in parallel. Many researchers have used reconfigurable hardware units to speed up the application runtime [3, 6, 7, 8].

Reconfigurable systems provide the flexibility and reuse of hardware for multiple applications. Reconfigurable hardware can be used to execute designs, which are larger than the available hardware resources. In such cases, a part of a large application is executed on the hardware. By reusing the reconfigurable hardware, the remaining tasks of the application can be loaded and executed on the hardware at runtime. This is known as *runtime reconfiguration*. Another issue that necessitates the integration of reconfiguration in a hardware platform is that some applications require reconfiguration in different abstraction levels of system [11]. For example, some applications require different variations of an algorithm to execute their task. A non-flexible hardware realization for such applications has to fit all required algorithm variations on the die. This, if possible, makes the design and fabrication processes more complicated and expensive [17].

A major drawback of using runtime reconfiguration is the significant delay of reprogramming the hardware. The total runtime of an application includes the actual execution delay of each task on the hardware along with the total time spent for hardware reconfiguration between computations. The latter might dominate the total runtime, especially for classes of applications with a small amount of computation between two consecutive reconfigurations. Many previous works have tried to tackle the reconfiguration delay problem using different approaches [14, 15, 16].

In many applications, only a small portion of the design changes at a time and there is no need to reconfigure the entire hardware

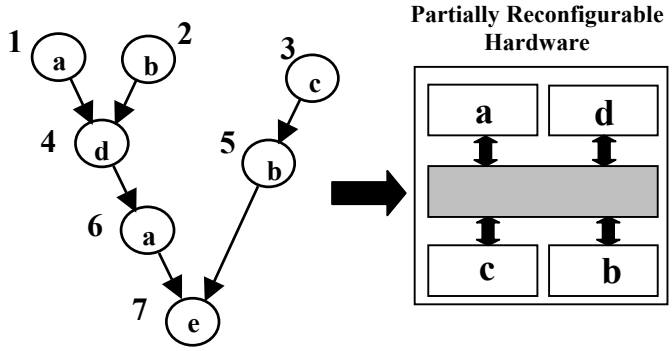

Figure 1. Executing an application on a partially reconfigurable hardware.

for instantiating a new design. This has led the industry to add the capability of partial reconfiguration to most of their recent products. FPGAs are examples of such reconfigurable hardware and most of the recent FPGA devices have the capability of runtime reconfiguration [9, 12].

Some earlier works have used partial reconfiguration to realize and execute an application. For example, researchers in [4, 13] present a partially reconfigurable system in which there are a limited number of similar places on the FPGA to plug in and run a module. Figure 1 shows such a reconfigurable platform. Each module (operation) is instantiated through partial reconfiguration and its instantiation will not affect other existing modules. In Figure 1, operation e can be instantiated by reconfiguring the physical resources currently executing operation d. This will not affect the configuration of resources executing operations a, b, and c. Multi-FPGA systems are another example of partially reconfigurable hardware in which there are more than one FPGA on a board. Each FPGA realizing a part of the design can be reconfigured independently.

Partial Reconfiguration allows the users to change only part of the design that needs to be updated and hence, decreases the reconfiguration time [5]. However, partial reconfiguration delay is still significant and it is dominating the whole computation delay for many applications. Reconfiguration delay is usually on the order of tens to hundreds of milliseconds for current FPGAs [9, 12]. While the partial reconfigurable approach is very effective and many different applications can be executed using the existing basic operations, the partial reconfiguration delay is still a barrier; hence minimizing it can lead to faster execution of the application [18].

In this paper, we formally state the problem of minimizing the runtime reconfiguration delay. We present a provably optimal algorithm to minimize the total delay incurred by partial reconfiguration. The input application is given as a set of

scheduled high-level operations (or simply operations). The data dependencies among the operations constitute a directed acyclic graph (DAG). Our algorithm outputs an execution order of the operations on hardware resources such that the total runtime reconfiguration is minimized.

The model and algorithm developed in this paper are directly applicable to current FPGA devices, multi-FPGA systems as well as other aforementioned partially reconfigurable systems. A special case of our algorithm applies to traditional non-partially reconfigurable FPGA platforms as well. We have conducted simulation-based experiments on some real applications. In terms of total runtime reconfiguration delay, our method outperforms other existing heuristics within the range of 40% to 2.5%.

The rest of this paper is organized as follows: In section 2 the problem of partial reconfiguration delay minimization is formally described. Section 3 describes our algorithm and proves its correctness and optimality. Some experimental results based on simulation are presented in section 4. Section 5 will conclude the paper along with some future directions and possible extensions.

2. Problem Statement

In this section, we formally present the problem of minimizing the reconfiguration delay for executing a given application on a system with multiple reconfigurable resources or with one partially reconfigurable resource. The application is composed of a set of basic tasks. At each time step, one or multiple tasks are revealed to the system. Arriving tasks have to be executed before next set of upcoming tasks. Executing these tasks (operations) in the specified order will lead to a correct execution of the application.

Suppose a *partially reconfigurable hardware* (PRH) is selected as the target platform to execute an arbitrary application. The application can be modeled as a set of operations that have to be executed in some specific order. Therefore, the functional unit corresponding to each operation should exist on the hardware before execution. Due to area constraint, only a subset of operations can be implemented in the PRH at each time. PRH can be partially reconfigured to realize the remaining operations. In such cases, partial reconfiguration delay for instantiating operations in the PRH imposes a delay on the total application runtime. Reconfiguration delay is one of the major barriers in using PRH for some real-time systems.

Partial reconfiguration delay is roughly proportional to the number of bits needed to transmit to PRH. Partial reconfiguration bits contain both data and control information for altering logic and interconnect of a particular block on the chip. The length of the sequence of reconfiguration bits is proportional to the reprogrammed area on the chip. In this paper, we assume the reconfiguration delay is equal for all types of operations. This assumption is exact for Multi-FPGA systems with identical FGPAs and for architectures in which we have some fixed places on the chip to plug in an operation [4, 13]. With this assumption, the number of *required partial reconfigurations* (RPR) represents the total reconfiguration delay. This delay mainly dominates the total application runtime for some classes of applications where the operation delay is negligible compared reconfiguration delay. Therefore, a reasonable metric for estimating the total application runtime is the number of RPRs for executing all operations.

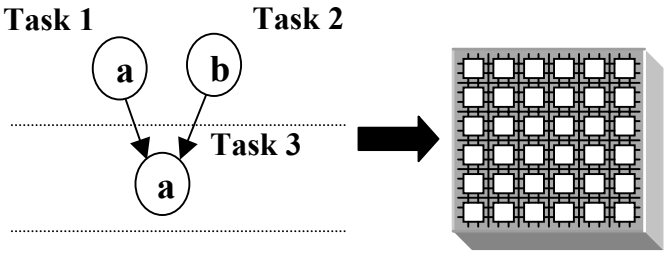

Figure 2. Different execution order of basic tasks leads to different number of required reconfigurations.

Figure 2 demonstrates an example in which different execution order of nodes, leads to different number of RPR. Tasks (nodes) 1 and 3 have the same type ‘a’ and Task 2 has another type ‘b’. PRH is able to fit one operation at a time, in this example. Executing such an application in <1 2 3> order, requires loading of ‘a’, ‘b’, and ‘a’ into the PRH respectively. This will cost 3 units, whereas execution of the same application in <2 1 3> order requires loading of ‘b’, and ‘a’ respectively, which costs 2 units. Therefore, execution order of basic tasks can impact the number of RPR and hence, total reconfiguration delay.

Let G (V, E) be a directed acyclic graph (DAG) representing a given application, where V is a set of vertices that represent operations and E is a set of directed edges that corresponds to the dependencies between operations (Figure 1). Assume that vertices of G have been already scheduled according to the time step at which they are revealed. Moreover, suppose the target reconfigurable hardware can accommodate at most K different operations in it. This implies that an upcoming new operation has to overwrite one of the K existing operations in PRH. Loading a new operation requires the PRH to be partially reconfigured. Therefore, it incurs a unit cost and increases the number of RPR by one. The partial reconfiguration delay minimization problem can be formally stated as:

Given such a scheduled G (V, E) and K as inputs, the objective is to load and execute all nodes of G on PRH so that the number of RPR is minimized. The constraints are that there are at most K different operations existing on hardware at all times and all nodes at cycle i have to be executed before nodes at cycle j if i < j. This ensures that the resulting execution order of operations leads to a valid evaluation of the computation and maintains the data dependencies among the nodes of G. We denote the minimum number of RPR to execute the scheduled G on a PRH with capacity K by Cost(G, K). Note that this problem can capture the case when a K-FPGA system is serving as the target architecture and each of the FPGAs can realize a single operation. In particular, the special case of K=1 represents the conventional single FPGA platforms (Figure 2).

The problem, as formulated above, is somewhat similar to standard paging problem that has been formulated and extensively studied in the domain of Online Algorithms. Particularly, reconfigurable hardware in present terminology corresponds to a cache unit with capacity K and each partial reconfiguration request is similar to a page fault (miss) that has a unit cost. However, to the best of our knowledge, the problem presented in this paper has not been studied and the current formulation is novel for modeling partial reconfiguration cost. Throughout this

paper, we may use terms from our formulation and standard paging formulation interchangeably.

3. Optimal Algorithm

In this section, we present an optimal algorithm for solving the problem defined in Section 2. First, we consider a special case in which the given DAG has only one operation in each cycle. Such a DAG is a path and there is already an optimal method developed for this special case. We extend this method for DAG.

Consider the case when G is a simple sequence of operations in which an operation depends on the previous one. Hence, the scheduled version of this sequence has only one operation in each cycle. Therefore, the algorithm is forced to select the nodes according to their original order for execution. However, it has to select an operation to overwrite if there are K operations existing in PRH at some cycle. This problem, which is known as the offline paging problem has been optimally solved by Belady [1]. It has been proved that the Least Imminently Used (LIU) operation existing in the cache is the best candidate to overwrite. This algorithm (LIU) leads to the minimum number of page faults.

Theorem 1: *Given a sequence of operations and a PRH to run the operations on, LIU is an optimal method to execute the operations in the given order and minimize the number of RPR.*

Proof: There is a one to one correspondence between the present problem and the offline-paging problem. The LIU is known to be optimal for the latter; hence, it is also optimal for the current problem [1]. □

We define S_i and P_i as the set of operations in cycle i and a permutation of operations in S_i respectively. Note that the operations in S_i are allowed to be repeated, since there can exist multiple operations of the same type in a cycle. Moreover, we define PRH(i) as the set of operations existing in PRH when the LIU starts to process cycle i . Therefore, PRH(0) = ∅.

Any solution to the general problem proposed in section 2, will be a permutation of operations reflecting their execution order. This permutation has to be in the form of $P = <P_1P_2 \ldots P_n>$ to meet the data dependency constraint of the problem formulation. According to Theorem 1, executing P using LIU algorithm will lead to the minimum number of RPR. Therefore, the generalized optimal algorithm only needs to find the optimal sequence of operations among all possible choices for P.

The following lemmas will aid in generating the optimal sequence:

Lemma 1*: Adding an operation to any place in a sequence of operations P cannot decrease Cost(P, K).*

Proof: Let Q be the new sequence created by adding an operation to P. We can process P exactly the way LIU processes Q, namely we can load/evict the same operations the optimal algorithm loads/evicts for processing Q. This processes P with a cost equal to Cost(Q, K), i.e., there is at least one way to process P with cost equal to Cost(Q, K). Hence Cost(P, K) cannot be greater than Cost(Q, K). □

Corollary 1: *For any sequence of operations Q and any subsequence P of Q: Cost(Q, K) ≥ Cost(P, K).*

Lemma 2: *Let $P=<P_1P_2 \ldots P_i \ldots P_n>$ be an optimal solution for a given instance of the problem. Let Q_i be a subsequence of P_i which contains operations in P_i that are in PRH(i). Similarly, let R_i be a subsequence of P_i that includes operations not in PRH(i). Then, $S=<P_1P_2 \ldots P_{i-1}Q_i\ R_i\ P_{i+1} \ldots P_n>$ is also an optimal solution.*

Proof: The cost of $T=<P_1P_2 \ldots P_{i-1}R_i\ P_{i+1} \ldots P_n>$ is equal to S since operations in Q_i are in PRH(i) when LIU starts to process cycle i. Therefore, they will neither incur any RPR nor alter the PRH configuration. On the other hand, T is a subsequence of P. Therefore, its cost cannot be greater than Cost(P, K) according to Lemma 1. Since P is an optimal solution, S also has the optimal cost. □

Corollary 2: *There exists an optimal algorithm, which executes operations previously existing in PRH before other nodes at each cycle.*

Corollary 3: *There exists an optimal ordering in which nodes of the same type appear adjacent to each other in each cycle. Therefore, an optimal algorithm can merge nodes with the same type in each cycle and assume that nodes occurring in each cycle are distinct.*

Lemma 3: *Let $P=< A_1\ A_2 \ldots A_i\ A_{i+1} \ldots A_j \ldots A_k \ldots A_m >$ be a solution for a given instance of the problem in which A_i is the i'th operation of P. Let A_j and A_k be the next instances of A_i and A_{i+1} respectively (Figure 3). If A_i and A_{i+1} both belong to the same cycle c and neither of them is in PRH(c), Then $Q=< A_1\ A_2 \ldots A_{i+1}\ A_i \ldots A_j \ldots A_k \ldots A_m>$ is also a solution and Cost(P, K) ≥ Cost(Q, K).*

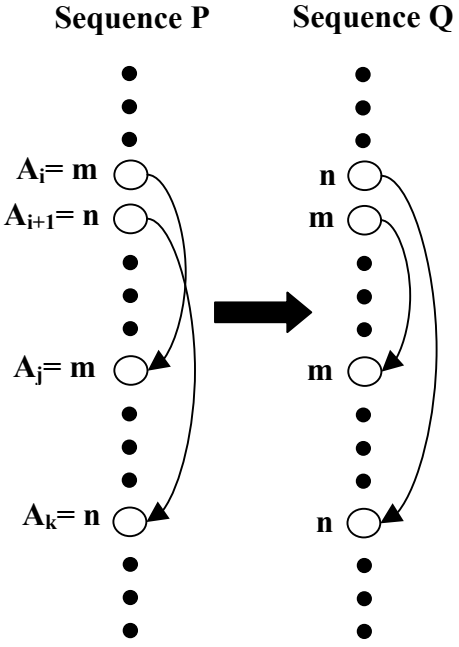

Figure 3. Converting sequence P to Q will not increase the cost, provided that m and n are not in PRH(i).

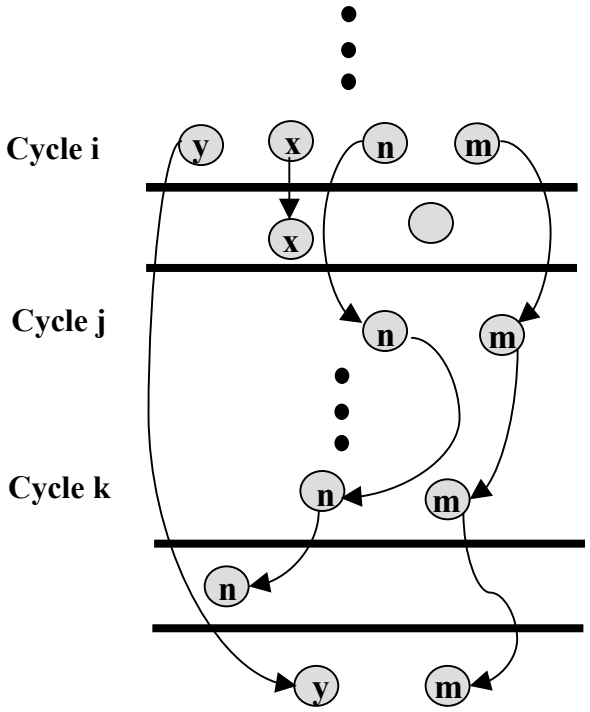

Figure 4. Tie breaking at cycle i.

Proof: We prove that Q is a valid solution and can be processed with cost equal to Cost(P, K), i.e., the optimal cost of processing Q is not greater than Cost(P, K).

Since A_i and A_{i+1} both belong to the same cycle, swapping them will produce a valid permutation. Note that relative positions of A_i and A_{i+1}, compared to other operations in P and Q, do not change. Therefore, optimal processing of P and Q up to position i, will lead to the same cost and PRH configuration. Executing A_i and A_{i+1} for both P and Q will incur two RPRs, since neither of them is in PRH(c). Loading the i'th node will overwrite the same operation for both sequences since they both have the same PRH configuration after processing the i'th node. Loading the i+1'th operation, however, might replace different existing modules, since i'th operations are different in P and Q.

Suppose loading A_{i+1} overwrites operation x when we are processing P optimally. If $x \neq A_i$ then we can overwrite x with the (i+1)'th operation for Q and have the exact cost and PRH configuration up to position i+2. Since the rest of Q is exactly same as P, its total cost will be the same. However, If $x=A_i$ we replace the i'th operation with the (i+1)'th operation when processing Q. This implies that except for one operation, PRH configuration is identical for P and Q up to position i+2. In particular, Q has an operation of type m instead of n (Figure 3). We continue processing Q exactly as LIU would process P up to position j. Note that RPRs for this span are the same, since type of operations between i+1 and j cannot be either m or n.

If there is an operation overwriting n for P, we overwrite m with the same operation for Q. This will make both cost and PRH configuration up to that point equal and since the rest of P and Q are the same, they will have the same cost. However, if such a case does not happen up to position j, P has to increase the cost by one to load m and execute A_j while Q has m on its PRH and does not issue a RPR. If m overwrites n in PRH of P, both sequences will have the same PRH configuration while Q has a lower cost up to this point. However, if m does not overwrite n, we can overwrite the same module with n after executing A_j for Q. Again, we will have the same PRH configuration and cost up to this point while the rest of two sequences are the same. This completes the proof. □

Therefore, an optimal algorithm can order all nodes appearing in a cycle, depending on their next occurrence. According to Corollary 2, at each cycle, the optimal algorithm can execute nodes existing in PRH before others. The remaining nodes can be executed according to such ordering without increasing the cost compared to any other possible ordering, according to Lemma 3.

Note that next instance of each operation happens in some S_i and all operations in S_i will come before all operations in S_j provided that $i < j$. Hence, comparing location of the next instance is trivial when two operations don't have their next occurrence in the same cycle. If an operation does not have another repetition in the next cycles, we can think of its next repetition at infinity and apply the same approach. For example in Figure 4, either <y m n x> or <y n m x> would be an optimal ordering for cycle i.

If two operations in cycle i have their next instances in cycle j (Figure 4), their relative order in cycle j determines their ordering in cycle i. However, the same argument applies to cycle j and operations relative order in cycle j depends on their next occurrence. To tackle this problem, the ordering can be done in reverse order. Starting the ordering process from the last cycle, all nodes occurring in cycle j, have their future occurrences already ordered. Therefore, they can be ordered deterministically using their next occurrences.

This procedure can be summarized as the min-RPR algorithm depicted in Figure 5. After the initialization step in which the next occurrence of a node is determined, nodes are ordered according to their next instance. Cycles are examined in reverse order for this step for efficient implementation. For determining the optimal execution order of nodes, operations already in the PRH are executed before other operations in each cycle. The remaining operations are executed according to their calculated ordering. PRH configuration is then updated for next cycle by processing the partial sequence generated in current cycle. Lemma 2 and 3 guarantee the min-RPR algorithm (Figure 5) finds a valid

Algorithm min-RPR(G, K):

```
PRH(0) = ∅;
For each operation
  Find its next occurrence;
For each cycle (traversing in reverse order)
  Sort nodes in this cycle according to their next occurrence;
For each cycle
  If any of the operations is already in PRH:
    append it to the optimal sequence;
  Append the remaining operations to the optimal;
    sequence based on their previously known sorting;
  Update PRH configuration by processing the ordered list for
    this cycle using LIU;
```

Figure 5. Algorithm min-RPR pseudo code

sequence of operations with the minimum cost.

The time complexity for algorithm min-RPR is O(n.p.log(p)), where n is the number of operations and p is the number of distinct operation types appearing in the scheduled DAG respectively. Note that at each cycle, it takes O(p.log(p)) to sort the nodes and there are O(n) cycles in the scheduled DAG. For practical applications, p does not grow with n. In realistic scenarios, the number of distinct operation types occurring in the application DAG is fixed; hence, the algorithm runtime is expected to scale linearly with respect to the application size.

4. Experimental Results

We have implemented our proposed algorithm along with three other algorithms using the C language. These four algorithms have been executed on 12 different scheduled DFGs extracted from real applications. These DFGs are all extracted from the signal processing toolbox of Matlab software. They are standard functions used in many signal-processing applications such as digital filter design. Each node of these DFGs is a complex matrix manipulation operation such as matrix inversion, multiplication and sine of matrix elements. Since matrix dimensions can be large, these operations could be complex enough to be implemented on the PRH.

Each DFG has been scheduled using a path-based scheduler [10] with two different sets of resource constraints. In Table 1, the two DFGs with the same name and different indices refer to the same DFG but different resource constraints (and hence schedules). Examples are Firls1 and Firls2. Table 1 demonstrates the basic characteristics of these test benches including number of nodes and number of cycles.

All scheduled DFGs are executed using four different algorithms. These algorithms differ in the manner in which they order nodes in a cycle. Once the order of the nodes at each cycle is determined, the generated sequence of nodes will be passed to the LIU algorithm [1] to measure the number of RPRs. For each algorithm, the number of RPRs is recorded as the cost.

The first algorithm, LF (Left First), executes the left node before other nodes occurring at its right in each cycle. LRU (Least Recently Used), at each cycle, executes a node that has been least recently used, while MRU selects the most recently used node to execute. Finally, the last algorithm is min-RPR, which we proved its optimality in section 3.

Results of four aforementioned algorithms on these DFGs are shown in Table 2. This table contains the number of RPR for PRHs with 1, 2 or 3 module capacity (K). The experimental results show that the optimal algorithm outperforms the other algorithms significantly. The overhead penalty that other algorithms have to pay ranges from 2.5% to more than 40% for these DFGs.

Intuitively, increasing the number of partitions on PRH reduces the algorithms' performance gap. In the extreme case, if K is equal to the number of module types occurring in DFG, all algorithms would behave the same. In this case, all algorithms have to pay a unit cost for loading the first occurrence of each operation type. From that point on, future occurrences of operations of the same type will not incur any cost. DFGs listed in Table 1 do not have many different types of operations. Therefore, the small performance penalty happens at small values of K. For instance for case K=3, the performance penalty of MRU is 2.5%.

We have randomly generated a set of DFGs with 26 different operation types and 500±10% nodes. These DFGs were used to show that this small performance gap would occur at greater values of K in case there are many types of operations in DFG. The output of the aforementioned four algorithms on this set of test benches is summarized in Table 3. The performance gap for all algorithms is significant for K=4. This significant gap can be observed for LF and LRU algorithms even for K=16.

Scheduled DFG	Number of nodes	Number of cycles
Fircls1	63	24
Fircls2	63	22
Firls1	64	32
Firls2	64	20
Firrcos1	79	42
Firrcos2	79	42
Invfreq1	41	25
Invfreq2	41	23
Maxflat1	115	51
Maxflat2	115	42
Spectrum1	55	28
Spectrum2	55	21

Table 1. Scheduled DFGs used for experiments.

In summary, all experiments on real applications and randomly generated DFGs, for different values of K, show that our algorithm outperforms other candidates. This improvement ranges from a few percents to tens of percents depending on DFG, algorithm structure, and capacity of the PRH. An interesting point is MRU uses a policy similar to min-RPR to order nodes at each cycle. MRU exhibits close to optimal results and behaves more efficiently than LF and LRU.

5. Conclusion

We presented an efficient optimal algorithm for minimizing the number of required partial reconfigurations when a partially reconfigurable or multi-FPGA system is used to run an application. A special case of the algorithm also solves the problem for single non-partially reconfigurable FPGA platforms. Since total application runtime is mainly dominated by the partial reconfiguration delay for many classes of applications, this algorithm can be directly applied for minimizing total application runtime.

Future research will focus on extensions with module area and delay considerations. Currently, all modules are assumed to occupy the same area on the chip and to have delays negligible compared to reconfiguration delay. These assumptions, however,

might not be the case for all applications. We will work toward extending our results to more complicated models incorporating module delay and area.

6. REFERENCES

[1] L. Belady, "A Study of Replacement Algorithms for Virtual-Storage Computer", *IBM Systems Journal, vol* 5, no 2, pp. 78-101, 1966

[2] A. DeHon, J. Wawrzynek, "Reconfigurable Computing: What, Why, Design Automation Requirements" *Design Automation Conference*, 1999.

[3] A. DeHon. "DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century". In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, April 1994.

[4] D. Taylor, J. Turner, J. Lockwood, E. Horta, "Dynamic Hardware Plugins (DHP): Exploiting Reconfigurable Hardware for High-Performance Programmable Routers", *Computer Networks*, vol. 38, no. 3, pp. 295-310, Feb 2002.

[5] S. Sezer, J. Heron, R. Woods, R. Turner, A. Marshall, "Fast Partial Reconfiguration for FCCMs", *IEEE Symposium on for Custom Computing Machines*, 1998.

[6] J. Burns, A. Donlin, J. Hogg, S. Singh, M. Wit. "A Dynamic Reconfiguration Run-Time System", *IEEE Symposium on FPGAs for Custom Computing Machines*, 1997.

[7] A. Adario, E. Roehe, S. Bampi, "Dynamically Reconfigurable Architecture for Image Processor Applications", *Design Automation Conference*, 1999.

[8] J. Hauser, J. Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor", *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 1997.

[9] Xilinx Inc. online documentaion, http://www.xilinx.com/ .

[10] S. Ogrenci Memik, E. Bozorgzadeh, R. Kastner, M. Sarrafzadeh. "A Super-Scheduler for Embedded Reconfigurable Systems", *International Conference on Computer-Aided Design (ICCAD)*, November 2001.

[11] P. Schaumont, I. Verbauwhede, K. Keutzer, M. Sarrafzadeh, "A Quick Safari through the Reconfigurable Jungle", *Design Automation Conference*, 2001.

[12] Altera Inc. online documentaion, http://www.altera.com/ .

[13] E. Horta, J. Lockwood, D. Taylor, D. Parlour, "Dynamic Hardware Plugins in an FPGA with Partial Run-time Reconfiguration", *Design Automation Conference*, 2002

[14] Z. Li, S. Hauck, "Configuration Prefetching Techniques for Partial Reconfigurable Coprocessor with Relocation and Defragmentation", *ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, pp. 187-195, 2002

[15] Z. Li, S. Hauck, "Configuration Compression for Virtex FPGAs", *IEEE Symposium on FPGAs for Custom Computing Machines*, 2001.

[16] Z. Li, K. Compton, S. Hauck, "Configuration Cache Management Techniques for FPGAs", *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 22-36, 2000

[17] K. Compton, S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software", *ACM Computing Surveys*, Vol. 34, No. 2. pp. 171-210. June 2002.

[18] Z. Li, K. Compton, S. Hauck, "Configuration Caching Management Techniques for Reconfigurable Computing", *Symposium on Field-Programmable Custom Computing Machines*, 2000.

	K=1				K=2				K=3			
	LF	LRU	MRU	OPT	LF	LRU	MRU	OPT	LF	LRU	MRU	OPT
Fircls1	59	60	50	46	36	43	35	32	25	30	24	23
Fircls2	60	57	49	44	38	40	34	33	27	29	25	24
Firls1	53	58	45	39	23	28	26	23	13	14	14	13
Firls2	46	46	34	32	23	27	20	19	13	18	13	13
Firrcos1	56	61	50	45	29	32	28	27	15	15	14	14
Firrcos2	47	47	42	36	27	26	23	22	14	14	12	12
Invfreq1	35	39	30	27	22	23	21	20	14	15	14	14
Invfreq2	32	38	30	27	20	24	21	19	14	15	14	14
Maxflat1	102	109	88	80	53	63	52	46	34	36	32	30
Maxflat2	106	94	69	62	46	49	40	37	27	29	24	24
Spectrum1	42	48	35	34	22	26	19	19	14	16	12	12
Spectrum2	47	44	28	28	21	21	15	15	11	11	9	9
Total	685	701	550	500	360	402	334	312	221	242	207	202
Penalty(%)	**37**	**40.2**	**10**	**NA**	**15.4**	**28.8**	**7.1**	**NA**	**9.4**	**19.8**	**2.5**	**NA**

Table 2. Number of required partial reconfigurations for different algorithms on real DFGs.

	K=4				K=8				K=16			
	LF	LRU	MRU	OPT	LF	LRU	MRU	OPT	LF	LRU	MRU	OPT
DFG1	315	320	296	278	209	224	200	192	98	101	91	90
DFG2	305	313	282	273	203	216	192	188	93	100	91	89
DFG3	311	315	285	270	207	219	195	186	89	96	87	86
DFG4	314	319	284	272	207	219	195	189	96	97	89	88
DFG5	330	336	304	290	220	233	205	197	97	103	96	94
DFG6	324	329	295	284	218	232	200	195	95	99	87	86
DFG7	306	311	277	266	202	216	185	181	90	97	85	85
DFG8	306	310	279	267	200	211	184	180	93	96	88	86
DFG9	320	326	291	278	213	222	196	191	92	94	87	85
DGF10	308	316	278	266	208	222	189	184	94	98	90	89
DFG11	312	317	283	271	204	217	189	183	87	94	83	83
DFG12	313	327	285	275	205	227	187	186	87	93	83	83
Average	313.7	319.9	286.6	274.2	208	221.5	193.1	187.7	92.6	97.3	88.1	87
Penalty(%)	**14.4**	**16.7**	**4.5**	**NA**	**10.8**	**18.0**	**2.9**	**NA**	**6.4**	**11.9**	**1.2**	**NA**

Table 3. Number of required partial reconfigurations for different algorithms on randomly generated DFGs.