An Optimal Algorithm for Minimizing Run-Time Reconfiguration Delay

SOHEIL GHIASI, ANI NAHAPETIAN, and MAJID SARRAFZADEH
University of California, Los Angeles

Reconfiguration delay is one of the major barriers in the way of dynamically adapting a system to its application requirements. The run-time reconfiguration delay is quite comparable to the application latency for many classes of applications and might even dominate the application run-time. In this paper, we present an efficient optimal algorithm for minimizing the run-time reconfiguration (context switching) delay of executing an application on a dynamically adaptable system. The system is composed of a number of cameras with embedded reconfigurable resources collaborating in order to track an object. The operations required to execute in order to track the object are revealed to the system at run-time and can change according to a number of parameters, such as the target shape and proximity. Similarly, we can assume that the applications comprising tasks are already scheduled and each of them has to be realized on the reconfigurable fabric in order to be executed.

The modeling and the algorithm are both applicable to partially reconfigurable platforms as well as multi-FPGA systems. The algorithm can be directly applied to minimize the application run-time for the typical classes of applications, where the actual execution delay of the basic operations is negligible compared to the reconfiguration delay. We prove the optimality and the efficiency of our algorithm. We report the experimental results, which demonstrate a 2.5–40% improvement on the total run-time reconfiguration delay as compared to other heuristics.

Categories and Subject Descriptors: B.8.2 [**Hardware Performance**]: Performance Analysis and Design Aids

General Terms: Performance, Algorithms

Additional Key Words and Phrases: Reconfigurable computing, reconfiguration delay, instantiation ordering

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 and thus can speedup the application run-time significantly. While dedicated hardware implementations address the application latency problem, they are not flexible. New version of the application has to go through all of the implementation steps

Authors' address: S. Ghiasi, A. Nahapetian, and M. Sarrafzadeh, Computer Science Department, UCLA, Los Angeles, CA, 90095; email: soheil@cs.ucla.edu.

in order to be realized in dedicated hardware [DeHon 1994; Burns et al. 1997; Adario et al. 1999; Hauser and Wawrzynek 1997].

Reconfigurable systems, however, provide flexibility and the ability to reuse hardware for multiple applications. Also, reconfigurable hardware resources can be used to execute applications that are too large to fit on them completely. In such cases, each part of the large application is executed on the hardware one at a time. Namely, by reusing the reconfigurable hardware, all of the parts of the application can be loaded and executed on the hardware at run-time. This technique, known as run-time reconfiguration, has been used by many researchers for realizing large designs [Compton and Hauck 2002; Maestre et al. 2001; Trimberger 1998; Chang and Marek-Sadowska 1997; Liu and Wong 1998].

Run-time reconfiguration, in particular, is suitable for realizing intensive applications that take different paths at run-time. Consider the target-tracking application. Based on available information from a scene, such as the number of targets, their distance from the camera and their resolution, different algorithms should be executed in order to track the targets efficiently in real-time. Hence, run-time reconfigurable hardware resources are utilized for implementing this application.

A major drawback of using run-time reconfiguration is the significant delay of reprogramming the hardware. The total run-time of an application includes the actual computation delay of each task on the hardware along with the total time spent on hardware reconfiguration between computations. The latter may dominate the total run-time, especially for classes of applications with a small amount of computation between two consecutive reconfigurations. Much of the previous work has tried to tackle the reconfiguration delay problem using different approaches [Li and Hauck 2001, 2002; Li et al. 2000].

In many applications, only a small portion of the design changes at a time, and so there is no need to reconfigure the entire hardware for instantiating a new design. This has led the industry to add the capability for partial reconfiguration to most of its recent products. FPGAs are an example of such reconfigurable hardware, and most of the recent FPGA devices have the capability of partial run-time reconfiguration [Xilinx; Altera].

Some earlier work has used partial reconfiguration to realize and execute an application. For example, Taylor et al. present a partially reconfigurable system in which there are a limited number of identical locations on the FPGA to plug in and run a module [Taylor et al. 2002; Horta et al.]. Figure 1 shows the basic idea of such a reconfigurable platform. Each module (operation) is instantiated on one of the identical places on the chip through partial reconfiguration, and hence the instantiation of each operation will not affect the other existing modules.

For example, in Figure 1, operation g can be instantiated by reconfiguring the physical resources currently executing operation d. This will not affect the configuration of resources executing operations a, c, and e. Note that multi-FPGA systems, such as the target-tracking system described above, are another example of a partially reconfigurable system in which there is more than one FPGA on the system. Each FPGA realizes a part of the design, which can be reconfigured independent of the state of the other FPGAs.

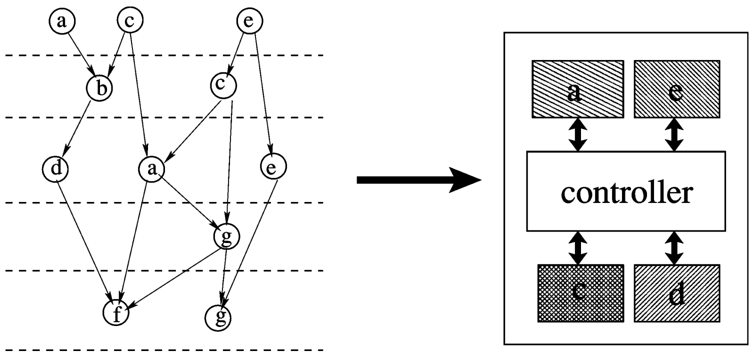

Fig. 1. Executing an application on a partially reconfigurable hardware.

Partial reconfiguration allows the user to change only the part of the design that needs to be updated and hence decrease the reconfiguration delay [Sezer et al. 1998]. The partial reconfiguration overhead, however, is still significant, and it dominates the computation delay for many applications. Reconfiguration delay is usually on the order of tens to hundreds of milliseconds for today's FPGAs [Xilinx; Altera]. While the partial reconfiguration approach is very effective and many different applications can be executed using the existing basic operations, the partial reconfiguration delay is still a barrier. Therefore, minimizing it can lead to faster execution of applications [Li et al. 2000].

In this paper, we formally state the problem of minimizing the run-time reconfiguration delay. We present a provably optimal algorithm to minimize the total delay incurred by partial reconfiguration. The input to the algorithm is an application, which is modeled 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 for the operations on the hardware resources such that the total run-time reconfiguration is minimized.

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

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

1.1 Object Tracking System

A system consisting of a number of cameras and a controller is depicted in Figure 2. The figure demonstrates a collaborative intruder detection and object

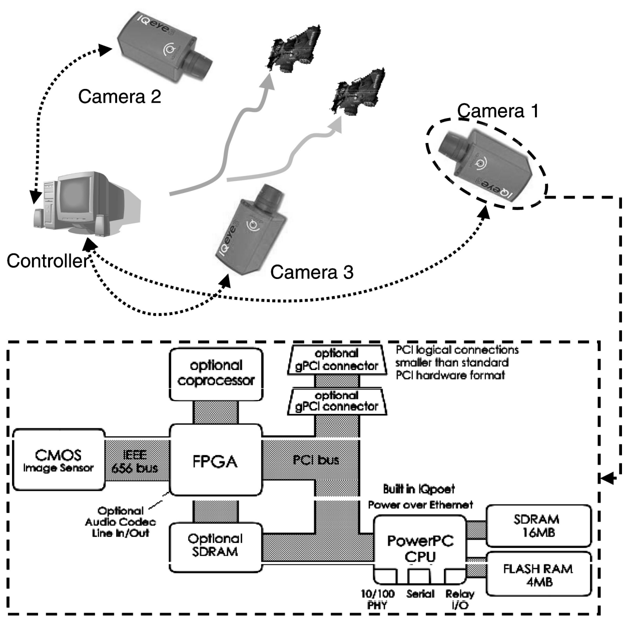

Fig. 2. The implemented target tracking system.

tracking system that has been implemented as part of this work [Kumar et al. 2003; Nguyen et al. 2002; Ghiasi et al. 2003a, 2003b, 2003c]. The system consists of multiple IQeye3 cameras [IQinVision]. The cameras communicate with the control unit in order to collaborate, distribute their information and execute the controller's commands.

As shown in Figure 2, each of the cameras has a number of embedded computational resources. These resources can be utilized to implement any application that takes the streaming scene data as input. The processing power embedded in the cameras eliminates the need to transfer the scene data to a remote processing station, and hence reduces the load of system communications. For cases where data communication is slow or expensive, the aforementioned embedded processing scheme improves system performance. Furthermore, colocating the processing and the data acquisition improves system scalability. The issue of partitioning a given application among different available resources has been studied by many researchers with different objectives. A general technique which is applicable to many objective functions is called *budgeting-based resource management* [Bozorgzadeh et al. 2003; Ghiasi et al. 2003a, 2003b; Chen et al. 2002; Sarrafzadeh et al.]. Throughout this paper, we assume that the application partitioning has already been performed. Thus, the portion of the application that has to run on the reconfigurable device is known. The task of partitioning the given application onto system resources shall not be discussed, since it is out of the scope of this paper.

Figure 3 demonstrates the abstract model of the computational resources existing in the system. A general-purpose processor (IBM PowerPC) and a Xilinx Virtex1000E FPGA [Xilinx] are embedded in each of the cameras. The processor

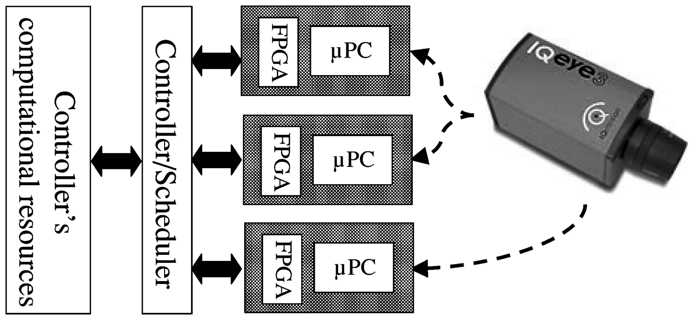

Fig. 3. The abstract model of the system resources. The controller has to schedule the tasks on reconfigurable hardware resources. Each camera has an FPGA embedded in it. There are several of cameras in the system (three in this picture).

communicates with a main controller which sends commands to the cameras in order to instantiate the proper design on their FPGAs. Then, each camera reconfigures its FPGA to realize a particular design. Finally, the instantiated design is executed on the FPGA using the real-time streaming scene data as input. The computation results are either sent back to the controller or stored locally in the memory blocks embedded in the camera. Note that the control unit schedules the application processes on the system resources through run-time reconfiguration [Nahapetian et al. 2003].

In order to highlight the effect of hardware implementation on system performance, the underlying tracking algorithms were implemented in C. These algorithms, namely KLT feature selection and feature tracking [Tomasi and Kanade 1991], perform intensive computations and cannot process the real-time streaming data when executed on the camera's processor (PowerPC).

To speedup the algorithms various simplifications have been made, which have resulted in five different versions of each algorithm [Ghiasi et al. 2003c]. Each version contains a new simplification in addition to all the changes made to previous version numbers. For example, version 4 of each algorithm contains all simplifications made to version 3 plus some further adaptation of the algorithm to the constrained camera platform. Figure 4 demonstrates the latency of each implementation when executed on camera PowerPC.

Furthermore, a nonembedded computing scheme is considered. In nonembedded processing, each image is first transmitted to a powerful processing unit where the computation is performed. Since the communication overhead dominates the computation latency, all the different versions of the algorithms perform similarly in this scheme. Figure 4 supports the fact that the image-processing algorithms are computationally intensive and do not show real-time performance if they are implemented in software.

Fortunately most of the image-processing algorithms, including feature selection and tracking, perform similar computations on all pixels of the image. Therefore, hardware implementations can take advantage of the intrinsic parallelism of these algorithms and boost their performance. For instance, Benedetti et al. report real-time performance for feature selection algorithm when implemented on a reconfigurable system [Benedetti and Perona 1998].

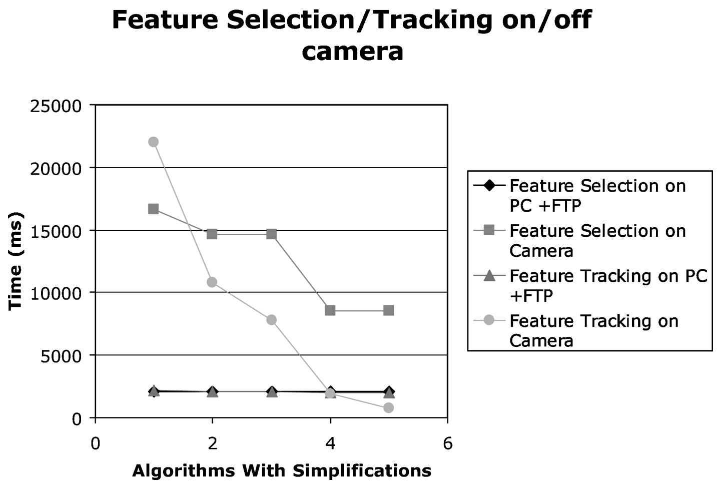

Fig. 4. None of the simplified versions of the algorithms or non-embedded computing scheme exhibit real-time performance.

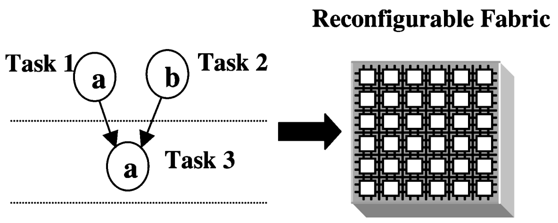

Fig. 5. Instantiation order of the blocks on the FPGA affects the number of required reconfigurations.

On the other hand, implementing the tracking application on a reconfigurable hardware requires the instantiation of different algorithms at different points of the application lifetime. Therefore, it becomes necessary to reduce the reconfiguration delay to further improve performance. This problem, motivated by our reconfigurable object tracking system, is the focus of this paper.

1.2 Example

Figure 5 is an example where different execution orders of nodes lead to different number of hardware reconfigurations. Tasks (nodes) 1 and 3 have the same type a and Task 2 has another type b. The reconfigurable hardware is capable of fitting one operation at a time in this example. Executing such an application in $\langle 1, 2, 3\rangle$ order, requires loading of a, b and a, into the hardware respectively. Thus the hardware has to be reconfigured three times, which incurs a cost of 3 units, whereas execution of the same application in $\langle 2, 1, 3\rangle$ order requires loading of b and a, respectively, which costs 2 units. Therefore, the execution order of basic tasks can impact the number of required reconfigurations and hence the total reconfiguration delay.

2. PROBLEM STATEMENT

In this section, we first present some preliminaries and definitions that will be used throughout the paper. Then, we formalize the problem of minimizing the reconfiguration delay when executing a given application on a system with multiple fully or partially reconfigurable resources.

2.1 Preliminaries and Assumptions

Let $G(V, E)$ be a DAG representing the given application, where V is the set of vertices that represent operations, and E is the set of directed edges that correspond to the dependencies among the operations (Figure 1). We assume that at each time step, one task or multiple tasks are revealed to the system for execution. The arriving tasks have to be executed before the next set of upcoming tasks to maintain the precedence constraints among tasks. Equivalently, we can assume that vertices of G have already been *scheduled* to execute at the time step at which they are revealed to the system. Furthermore, we assume that the entire DAG is known *a priori*. This assumption holds both for applications whose structure is fixed and for dynamic applications that have been extensively profiled. As a result, our technique's applicability is restricted to these two classes of applications, namely, scheduled fixed structure applications or applications with known profiling information. Note that profiling information can serve as a guideline, probably with provable error rates and approximation bounds, for determining the control-data flow graph (CDFG) structure of the application.

We assume that a *partially reconfigurable hardware* (PRH) is selected as the target platform to execute the application. The functional units corresponding to each operation should exist on the hardware before its execution. Due to area constraints, all of the comprising operations of an application might not fit into the PRH at the same time. In this case, a subset of the operations can be instantiated in the PRH, and it can be partially reconfigured to realize the remaining operations when needed. In such cases, partial reconfiguration for instantiating operations in the PRH, imposes a delay on the total application run-time. Reconfiguration delay is one of the major barriers when using PRH for real-time systems, and it is the main focus of this paper.

Reconfiguration delay is roughly proportional to the number of bits that need to be transmitted to the PRH in order for it to change its state. Partial reconfiguration bits contain both data and control information for altering the logic and the interconnect of a particular block on the chip [Xilinx; Altera]. Hence, the length of the sequence of reconfiguration bits is proportional to the reprogrammed area on the chip. Therefore, the reconfiguration delay for instantiation of different operations is the same for a number of different platforms. These platforms include multi-FPGA systems with identical FGPAs and architectures in which there are a number of identical places on the chip to plug in an operation (Figure 1) [Taylor et al. 2002; Horta et al.].

Our system presented in Section 1 uses similar FPGAs in all of its cameras. Hence, the reconfiguration delays for all types of tasks are identical. Therefore, the number of required partial reconfigurations (RPR) accurately represents the total reconfiguration delay. It follows that our technique's effectiveness is limited to multi-FPGA platforms with identical FPGAs and/or architectures based on dynamic hardware plug-in idea presented in Taylor et al. [2002], and Horta et al. There are many issues such as connection to chip pins and heterogeneous routing and logic resources that do not allow easy relocation of tasks for other reconfigurable architectures.

Finally, we assume the target reconfigurable hardware can accommodate at most K different operations at a time. Namely, there are K identical plug-in locations in the target PRH, or the target reconfigurable hardware is composed of K identical fully reconfigurable devices. This implies that an upcoming new operation that does not currently exist in the PRH 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 total number of RPR by 1.

2.2 Problem Formulation

The partial reconfiguration delay minimization problem can be formally stated as:

—Given are the scheduled task graph $G(V, E)$ representing an application, and K the number of identical plug-in locations in the PRH (or similarly the number of fully programmable FPGAs in the system. This is the case for the system shown in Figure 2).

—The objective is to find the order in which the tasks in V, have to be instantiated in the PRH to execute the entire application, such that the number of required partial reconfigurations is minimized. Moreover, for instantiating each task $v \in V$, the existing task in PRH that has to be overwritten has to be determined.

—The constraints are that the entire application has to be executed using the K existing plug-in locations in the PRH. This implies that each node has to be instantiated in the PRH at some point after all of its inputs have been executed. Furthermore, at most K different type of operations can exist in the PRH at each time.

The problem, as formulated above, is somewhat similar to the standard paging problem that has been formulated and extensively studied in the domain of online algorithms. Specifically, reconfigurable hardware 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 the standard paging formulation interchangeably.

3. MINIMIZING THE NUMBER OF REQUIRED PARTIAL RECONFIGURATIONS

In this section, we present an optimal algorithm for solving the problem defined in Section 2, and we prove the optimality of our solution. First, we define the notations. Then, we model the problem using sequences and permutations and prove some theorems using this model. Finally, we present the algorithm. Its optimality will follow from the theorems.

3.1 Definitions

Any solution to the problem stated in Section 2 will form a sequence of operations. Moreover, the solution must specify which operation to evict from the PRH for loading a new operation. As we will show in the next subsection, there is a simple optimal algorithm for evicting the existing operations in PRH in order to minimize RPR for a given sequence of operations. Therefore, we focus our efforts on finding the optimal sequence.

Let S_i be the set of operations in cycle i. Also, we define P_i to be a permutation of the operations in S_i. Note that the operations in S_i are allowed to be repeated, because there can exist multiple operations of the same type in a cycle.

Let $Cost(P, K)$ represent the minimum number of RPR for the execution (processing) of sequence P with a PRH with capacity K. It follows that the optimal solution is equal to the minimum possible value of $Cost(P, K)$ for all P's. We define PRH(i) as the set of operations existing in PRH when the least immediately used (LIU) starts to process cycle i . Since PRH contains no operations when the application execution starts PRH(0) $= \emptyset$.

3.2 Modeling and Theoretical Results

In this section, we present some theoretical results that provide a basis for deriving the optimal algorithm for solving the problem defined in Section 2. First, we consider a special case in which a given DAG has only one operation per cycle. Such a DAG is a path, and there is already an optimal method developed for this special case. We extend this method for all DAGs.

Consider the case when G is a sequence of operations in which an operation has to wait for its predecessor to run. Hence, the scheduled version of G has only one operation in each cycle. Therefore, the algorithm is forced to select the nodes according to their original order for execution. In other words, there is only a unique P, and the optimal cost would be equal to $Cost(P, K)$.

The optimal algorithm 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 [1966]. It has been proven that the LIU operation existing in the cache is the best candidate for overwriting. This algorithm (LIU) leads to the minimum number of page faults.

THEOREM 3.1. *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 to minimize the number of RPR.*

PROOF. There is a one-to-one correspondence between the current problem and the offline paging problem. The LIU is known to be optimal for the latter problem. Hence, it is also optimal for the current problem [Belady 1966]. □

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 $P = \langle P_1, P_2, \ldots, P_n\rangle$ to meet the data dependency constraint of the problem formulation. According to Theorem 3.1, 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 3.2. *Adding an operation to any location 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 has a cost equal to *Cost*(Q, K), that is, 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 3.3. *For any sequence of operations Q and any subsequence P of Q: Cost$(Q, K) \geq$ Cost(P, K)*

LEMMA 3.4. *Let $P = \langle P_1, P_2, \ldots, P_i, \ldots, P_n\rangle$ be an optimal solution for a given instance of the problem. Let Q_i be a subsequence of P_i that contains all of the operations in P_i that are in PRH(i). Similarly, let R_i be a subsequence of P_i that is composed of all of the operations in P_i but not in PRH(i). Then, $S = \langle P_1, P_2, \ldots, P_{i-1}, Q_i, R_i, P_{i+1}, \ldots, P_n\rangle$ is also an optimal solution.*

PROOF. The cost of $T = \langle P_1, P_2, \ldots, P_{i-1}, R_i, P_{i+1}, \ldots, P_n\rangle$ 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 3.2. Therefore, *Cost*$(S, K) \leq$ *Cost*(P, K). On the other hand, P is an optimal solution. Therefore *Cost*$(S, K) =$ *Cost*(P, K) and S also has the optimal cost. □

COROLLARY 3.5. *There exists an optimal algorithm, which executes operations already existing in PRH before other nodes at each cycle.*

COROLLARY 3.6. *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, that is, they have different types.*

LEMMA 3.7. *Let $P = \langle A_1, A_2, \ldots, A_i, A_{i+1}, \ldots, A_j, \ldots, A_k, \ldots, A_m\rangle$ be a solution for a given instance of the problem in which A_i is the ith operation of P. Let A_j and A_k be the next instances of A_i and A_{i+1} respectively (Figure 6). If A_i and A_{i+1} both belong to the same cycle c and neither of them is in PRH(c), then $Q = \langle A_1, A_2, \ldots, A_{i+1}, A_i, \ldots, A_j, \ldots, A_k, \ldots, A_m\rangle$ is also a solution and Cost$(P, K) \geq$ Cost(Q, K).*

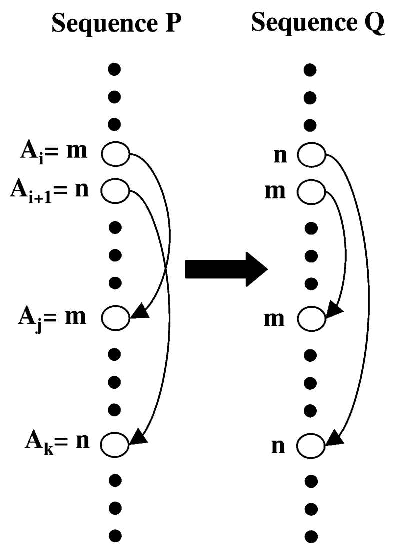

Fig. 6. Converting sequence P to Q will not increase the cost, provided that m and n are not in PRH(i).

PROOF. We prove that Q is a valid solution and can be processed with cost equal to $Cost(P, K)$, that is, 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 ith node will overwrite the same operation for both sequences since they both have the same PRH configuration after processing the ith node. Loading the $(i+1)$th operation, however, might replace different existing modules, since ith operations in P and Q are different.

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 processing cost will be the same.

However, if $x = A_i$ we replace the ith operation with the $(i+1)$th operation when processing Q. This implies that PRH configuration is identical for P and Q up to position $i+2$, except for one operation. In particular, Q has an operation of type m instead of n (Figure 6). 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. Since the rest of P and Q are the same, they will have the same cost. However, if such a case does not happen until 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

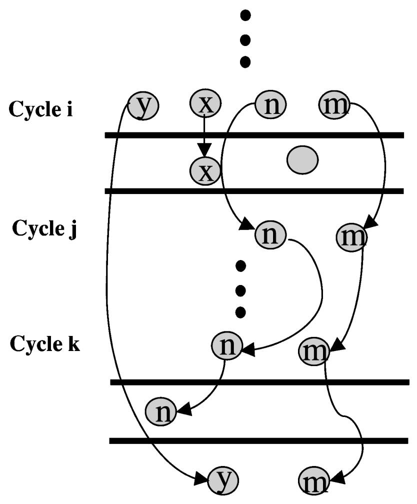

Fig. 7. Tie breaking at cycle i.

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 processing cost up to this point while the rest of two sequences are the same. This completes the proof. □

3.3 Optimal Algorithm

The theorems proved in the previous section imply that an optimal algorithm can order all nodes appearing in a cycle, according to their next occurrence. According to Corollary 3.5, at each cycle, the optimal algorithm can execute nodes existing in PRH before others. Furthermore, according to Lemma 3.7, the remaining nodes can be executed according to such ordering without increasing the cost, as compared with any other possible ordering.

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 the location of the next instance is trivial when two operations do not have their next occurrence in the same cycle. If an operation does not have any future occurrence, it will not be needed in the future cycles. Therefore its next repetition can be thought to happen at infinity, and the same approach can be applied. For example, in Figure 7, either $\langle y, m, n, x\rangle$ or $\langle y, n, m, x\rangle$ would be an optimal ordering for cycle i.

If two operations in cycle i have their next instances in cycle j (Figure 7), their relative ordering in cycle j determines their ordering in cycle i. However, the same argument applies to cycle j and operations' relative ordering in cycle j depends on their next occurrence. To tackle this problem, the ordering can be done in the 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.

```
Input: scheduled $G(V,E)$, $K$.
Output: OptimalSequence

let $PRH(0) = \emptyset$;
let $OptimalSequence = \emptyset$;
for all $v \in V$ do
  Find its next occurrence.
end for
for all cycles traversed in reverse order do
  Sort nodes in this cycle according to their next occurrence.
end for
for all cycles do
  If any of the operations is already in PRH: append it to the OptimalSequence;
  Append the remaining operations to OptimalSequence based on their previously known sorting;
  Update PRH configuration by processing the ordered list for this cycle using LIU;
end for
Return OptimalSequence;
```

Fig. 8. Algorithm *min*–RPR generates the optimal sequence of reconfigurations.

This procedure can be summarized by the *min*–RPR algorithm shown in Figure 8. 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 in this step. 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 the current cycle. Lemmas 3.4 and 3.7 guarantee that the *min*–RPR algorithm will find a valid 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. 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 are fixed. Hence, the algorithm's run-time is expected to scale linearly with respect to the application size.

3.4 Interesting Extensions

The problem presented in this paper can be extended to model other realistic application problems. One extension of the problem assumes that tasks occupy different areas on the chip. Hence, when overwriting a task, one has to consider not only its next occurrence, but also the amount of area that the task will free-up upon removal from the chip. This problem occurs in web caching, where pages have different sizes and request frequencies. A complete discussion along with effective algorithms can be found in Irani [2002].

An assumption of our paper is that complete information about the application DFG is known. Many real life applications do not conform to this assumption, instead they are an online version of the problems. For instance, a web caching policy has almost no information about the next page that a user might request. However, probabilistic and/or statistical information, can be taken advantage of to predict the tasks coming after a particular node. For example, a web caching algorithm might realize that in practice a request for a cartoon website is not likely to follow a request for a news website. In the online algorithm domain, this property is referred to as *locality of reference*. Borodin et al. [1995] and Irani et al. [1996] have discussed this problem and presented some strongly competitive algorithms to tackle it.

In the tracking system (Figure 2) implemented as part of this work, the tasks are revealed to the system as events happen in a scene. Furthermore, each set of revealed tasks has to be executed before the next upcoming set. Therefore, we assume that the application DFG has the scheduling information embedded in it. However, scheduling information is not available for all applications. Examples include signal-processing applications whose DFGs are often known *a priori*. For such application domains, a scheduling technique has to be utilized prior to applying our methodology.

An interesting extension of the current work would address the aforementioned issue, where there is no scheduling information available, and the precedence constraint is the only constraint that has to be met to guarantee a valid evaluation of the computation. This problem also arises in other application domains such as compiler optimization. The problem can be formally states as: *Given a DFG with color information for each node, what is the best topological order that minimizes the number of color changes among consecutive nodes?*

This problem assumes a PRH of capacity 1 (a system consisting of only one fully reconfigurable FPGA), if transferred to reconfigurable computing domain. Kennedy and McKinley [1993] and Darte [2000] have studied this problem when applying loop fusion to code generation in the area of compiler optimization. By reduction from vertex cover problem, it has been shown that the general formulation of the problem is NP-hard.

4. EXPERIMENTAL RESULTS

This section describes the experiments carried out to verify our algorithm, which tackles the reconfiguration sequence problem. Section 4.1 will describe the experimental setup and the testbenches. Section 4.2 will follow with a detailed discussion of the experimental results and their implications.

4.1 Experimental Setup

Six different signal-processing applications running on a partially reconfigurable hardware have been used as testbenches. The applications' DFG have been manually extracted from their MATLAB implementation, available through the signal-processing tool box of the software [Mathworks]. The testbenches are standard functions commonly used in many signal-processing applications, such as digital filter design. Each DFG has been scheduled using a path-based scheduler [Memik et al. 2001] with two different sets of resource constraints. Table I summarizes the application testbenches. It also depicts the complexity of each DFG using the number of nodes and the number of cycles in the scheduled version. Note that in Table I, two testbenches with the same

Table I. List of DFGs Extracted from MATLAB and Scheduled for Experiments

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

name and different indices refer to the same DFG, which is scheduled using different resource constraints. The examples are Firls1 and Firls2.

Each node in these DFGs is a complex matrix manipulation operation such as matrix inversion, matrix multiplication or a sine of matrix elements. Since the matrix dimensions can be very large, these operations can be complex enough to be implemented as separate blocks on the PRH for real-time applications. Therefore, they agree with the assumptions that we have made throughout this paper.

We have implemented our proposed technique along with three other algorithms using the C language. These other three algorithms are Left First (LF), Least Recently Used (LRU), and Most Recently Used (MRU) policies for ordering nodes at each cycle. The first algorithm, LF, executes the leftmost first at each cycle. The LRU algorithm gives a higher priority to least-recently-used nodes at each cycle. The MRU, on the other hand, selects the most recently used node to execute. The last algorithm is *min*–RPR, whose optimality we have proven in Section 3.

It is assumed that all the aforementioned applications are to be executed on a PRH. Extensive simulations using the four mentioned algorithms have been performed with three different PRH capacities. Moreover a number of randomly generated DFGs have been used to perform another set of simulations. The next subsection will describe our results and explain our observations.

4.2 Simulation Results

Each DFGs was executed using all 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 was passed to the LIU algorithm [Belady 1966] to measure the number of RPRs. The number of RPRs for each algorithm is reported as its cost.

The results for testbench DFGs are shown in Table II. The table contains the number of RPR for PRHs with 1, 2, and 3 module capacity (K). The experimental results show that the optimal algorithm outperforms the other algorithms

Table II. Number of Required Partial Reconfigurations for Different Algorithms on Real DFGs

	$K = 1$				$K = 2$				$K = 3$			
	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

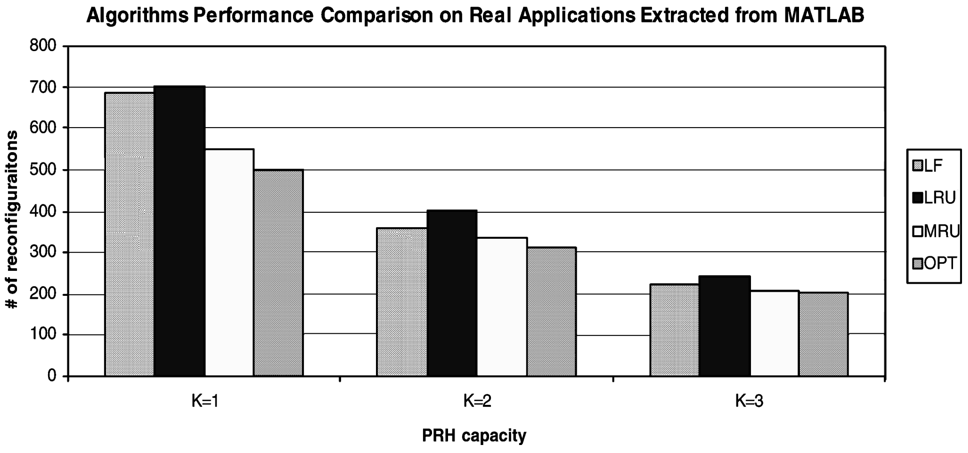

Fig. 9. Performance comparison of different context switching policies. Testbenches are signal processing applications.

significantly. For these DFGs, the overhead penalty that the other algorithms pay ranges from 2.5% to 40%.

Figure 9 summarizes the results from Table II. It compares the average performance of the four context switching policies. As shown in the figure, the optimal algorithm outperforms the other three policies significantly for the single FPGA systems ($K = 1$). However, the performance gap among the different algorithms decreases as the capacity of the reconfigurable hardware (K) is increased. This is due to the fact that the applications used for this set of experiments do not contain many varieties of tasks. They mainly use matrix addition, matrix subtraction, and matrix multiplication as the basic comprising operations. Other matrix manipulation operations such as matrix inversion and sine of matrix elements happen infrequently in the application DFGs. Therefore, different reconfiguration sequence management techniques perform similarly for reconfigurable systems with capacity three (Figure 9).

Table III. Number of Required Partial Reconfigurations for Different Algorithms on Randomly Generated 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

Intuitively speaking, increasing the PRH capacity reduces the performance gap between the different algorithms, because the frequent operations are less likely to be evicted from the PRH. In the extreme case, if K is equal to the number of different operation types occurring in DFG, referred to as p in Section 3, all algorithms would behave in exactly the same manner. In this case, all the algorithms would have to pay a unit cost for loading the first occurrence of each operation type. From that point on, future occurrences of the operations of the DFG will not incur any cost, because all the operations already exist in the PRH. As mentioned before, the DFGs listed in Table II do not contain many different types of operations. Therefore, a small performance penalty is incurred with small values of K. For instance in the case where $K = 3$, the performance penalty of MRU is 2.5%.

To further investigate this observation, we randomly generated 12 DFGs with 26 different operation types. Each of the DFGs had $500 \pm 10\%$ nodes. These DFGs were solely used to show that a small performance gap will occur at greater values of K, when there are many types of operations in DFG. The output of the four algorithms on this set of testbenches is summarized in Table III. The table reports the number of required reconfigurations for all of the generated DFGs.

The average number of RPR of the four algorithms on the testbenches is summarized in Figure 10. The figure demonstrates that the performance gap for all of the algorithms is significant when $K = 4$. This supports the previous expectation concerning the relation of PRH capacity and the algorithms' performance gap. A significant difference can be observed for LF and LRU algorithms even when $K = 16$. MRU, however, performs close to the optimal in this case.

An interesting observation is that MRU uses a policy similar to *min*–RPR to order nodes at each cycle. At each cycle, MRU gives higher priority to executing nodes that have been most recently executed. This utilizes the same idea in Lemma 3.7 and makes MRU perform more efficiently than the other two

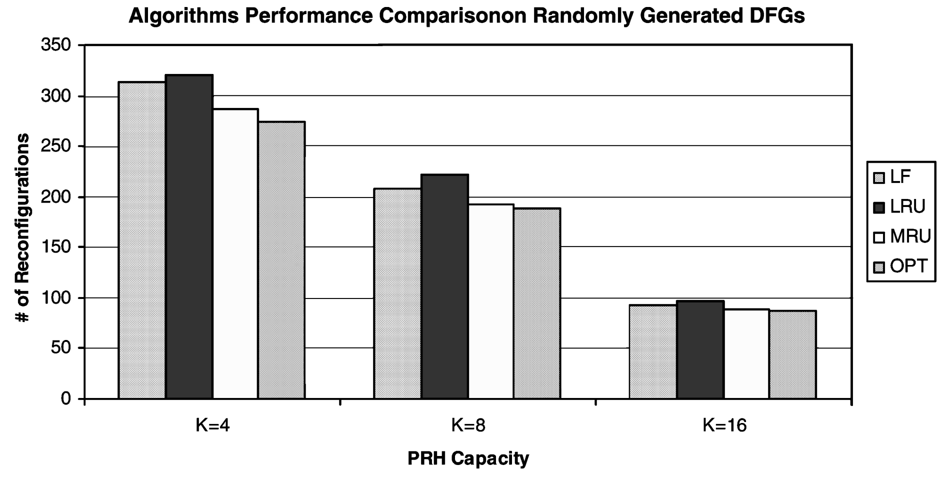

Fig. 10. Performance comparison of different context switching policies. Testbenches are generated randomly.

suboptimal algorithms. Therefore, one would expect MRU to exhibit a performance similar to the optimal algorithm. The experimental results reported in this section support this observation.

In summary, all of the experiments on real applications and randomly generated DFGs, for different values of K, show that our algorithm outperforms all the other candidates. The improvement ranges from a few percents to tens of percents depending on the DFG, the algorithm structure and the capacity of the PRH.

5. CONCLUSIONS AND FUTURE DIRECTIONS

We have presented an efficient optimal algorithm for minimizing the number of required partial reconfigurations (context switches) 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 nonpartially reconfigurable FPGA platforms. Since the total application run-time is dominated by the partial reconfiguration delay for many classes of applications, this algorithm can directly minimize the total application run-time.

Future research will focus on extensions with operation area and delay considerations. Currently, all of the operations are assumed to occupy the same area on the chip and are assumed to have delays negligible compared to the reconfiguration delay. These assumptions, however, might not apply to all applications. We will work towards extending our results to more complicated models by incorporating module delay and area.

The current version of the algorithm finds the best instantiation sequence for a scheduled DAG. However, it does not provide any information about a good scheduling for the given application. Obviously, different schedules for the same DAG, incur different reconfiguration costs. Therefore, in the future we would like to investigate the effect of scheduling on the number of reconfiguration.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers whose helpful comments improved the quality of this paper.

REFERENCES

ADARIO, A., ROEHE, E., AND BAMPI, S. 1999. Dynamically reconfigurable architecture for image processor applications. In *Design Automation Conference*.

ALTERA. Altera Products' Online Documentation. *http://www.altera.com*.

BELADY, L. 1966. A study of replacement algorithms for virtual-storage computer. *IBM Systems Journal 5,* 2, 78–101.

BENEDETTI, A. AND PERONA, P. 1998. Real-time 2-d feature detection on a reconfigurable computer. In *IEEE Conference on Computer Vision and Pattern Recognition*.

BORODIN, A., IRANI, S., RAGHAVAN, P., AND SCHIEBER, B. 1995. Strongly competitive algorithms for paging with locality of reference. *Journal of Computer and System Science 50,* 2, 244–258.

BOZORGZADEH, E., GHIASI, S., TAKAHASHI, A., AND SARRAFZADEH, M. 2003. Optimal integer delay budgeting on directed acyclic graphs. In *Design Automation Conference*.

BURNS, J., DONLIN, A., HOGG, J., SINGH, S., AND WIT, M. 1997. A dynamic reconfiguration run-time system. In *IEEE Symposium on Field-Programmable Custom Computing Machines*.

CHANG, D. AND MAREK-SADOWSKA, M. 1997. Buffer minimization and time-multiplexed i/o on dynamically reconfigurable FPGAs. In *ACM 5th International Symposium on Field-Programmable Gate Arrays*. 142–148.

CHEN, C., BOZORGZADEH, E., SRIVASTAVA, A., AND SARRAFZADEH, M. 2002. Budget management with applications. *Algorithmica 34*, 3, 261–275.

COMPTON, K. AND HAUCK, S. 2002. Reconfigurable computing: A survey of systems and software. *ACM Computing Surveys 34*, 2, 171–210.

DARTE, A. 2000. On the complexity of loop fusion. *Parallel Computing 26*, 9, 1175–1193.

DEHON, A. 1994. Dpga-coupled microprocessors: Commodity ics for the early 21st century. In *IEEE Workshop on FPGAs for Custom Computing Machines*.

GHIASI, S., MOON, H., AND SARRAFZADEH, M. 2003a. Collaborative and reconfigurable object tracking. In *International Conference on Engineering of Reconfigurable Systems and Algorithms*.

GHIASI, S., MOON, H., AND SARRAFZADEH, M. 2003b. Improving performance and quality thru hardware reconfiguration: Potentials and adaptive object tracking case study. In *Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)*.

GHIASI, S., MOON, H., AND SARRAFZADEH, M. 2003c. A networked reconfigurable system for collaborative unsupervised detection of events. In *Tehnical Report, Computer Science Dept, UCLA*.

GHIASI, S., NGUYEN, K., BOZORGZADEH, E., AND SARRAFZADEH, M. 2003a. On computation and resource management in an FPGA-based computing environment. In *International Symposium on Field-Programmable Gate Arrays (poster)*.

GHIASI, S., NGUYEN, K., BOZORGZADEH, E., AND SARRAFZADEH, M. 2003b. On computation and resource management in networked embedded systems. In *International Conference on Parallel and Distributed Computing and Systems*.

GHIASI, S., NGUYEN, K., AND SARRAFZADEH, M. 2003c. Profiling accuracy-latency characteristics of collaborative object tracking applications. In *International Conference on Parallel and Distributed Computing and Systems*.

HAUSER, J. AND WAWRZYNEK, J. 1997. Garp: A mips processor with a reconfigurable coprocessor. In *IEEE Symposium on Field-Programmable Custom Computing Machines*.

HORTA, E., LOCKWOOD, J., TAYLOR, D., AND PARLOUR, D. Dynamic hardware plugins in an FPGA with partial run-time reconfiguration.

IQINVISION. Product manuals and online documentation. *http://www.iqinvision.com*.

IRANI, S. 2002. Page replacement with multi-size pages and applications to web caching. *Algorithmica 33*, 3, 384–409.

IRANI, S., KARLIN, A., AND PHILLIPS, S. 1996. Strongly competitive algorithms for paging with locality of reference. *SIAM Journal on Computing 25*, 3, 477–497.

Kennedy, K. and McKinley, K. 1993. Typed fusion with applications to parallel and sequential code generation. In *Rice University Dept. of Computer Science Technical Report TR93-208*.

Kumar, R., Ghiasi, S., and Srivastava, M. 2003. Dynamic adaptation of networked reconfigurable systems. In *Workshop on Software Support for Reconfigurable Systems*.

Li, Z., Compton, K., and Hauck, S. 2000. Configuration caching management techniques for reconfigurable computing. In *IEEE Symposium on FPGAs for Custom Computing Machines*. 22–36.

Li, Z. and Hauck, S. 2001. Configuration compression for virtex FPGAs. In *IEEE Symposium on FPGAs for Custom Computing Machines*.

Li, Z. and Hauck, S. 2002. Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation. In *ACM/SIGDA Symposium on Field-Programmable Gate Arrays*.

Liu, H. and Wong, D. 1998. Network flow based circuit partitioning for time-multiplexed FPGAs. In *IEEE/ACM International Conference on Computer-Aided Design*. 497–504.

Maestre, R., Kurdahi, F., Fernandez, M., Hermida, R., Bagherzadeh, N., and Singh, H. 2001. A framework for reconfigurable computing: Task scheduling and context management. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 9*, 6, 858–873.

Mathworks. Matlab product manual and help files. *http://www.mathworks.com*.

Memik, S. O., Bozorgzadeh, E., Kastner, R., and Sarrafzadeh, M. 2001. A super-scheduler for embedded reconfigurable systems. In *International Conference on Computer-Aided Design*.

Sarrafzadeh, M., Knol, D., and G.E. Tellez, t.

Nahapetian, A., Ghiasi, S., and Sarrafzadeh, M. 2003. Scheduling on heterogeneous resources with heterogeneous reconfiguration costs. In *International Conference on Parallel and Distributed Computing and Systems*.

Nguyen, K., Yueng, G., Ghiasi, S., and Sarrafzadeh, M. 2002. A general framework for tracking objects in a multi-camera environment. In *International Workshop on Digital and Computational Video*.

Sezer, S., Heron, J., Woods, R., Turner, R., and Marshall, A. 1998. Fast partial reconfiguration for fccms. In *IEEE Symposium on Field-Programmable Custom Computing Machines*.

Taylor, D., Turner, J., Lockwood, J., and Horta, E. 2002. Dynamic hardware plugins (dhp): Exploiting reconfigurable hardware for high-performance programmable routers. *Computer Networks 38*, 3, 295–310.

Tomasi, C. and Kanade, T. 1991. Detection and tracking of point features. In *Carnegie Mellon University Technical Report CMU-CS-91-132*.

Trimberger, S. 1998. Scheduling designs into a time-multiplexed FPGA. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 153–160.

Xilinx. Xilinx Products' Online Documentation. *http://www.xilinx.com*.

Received January 2003; revised July 2003; accepted August 2003