Configuration Caching Management Techniques for Reconfigurable Computing

Zhiyuan Li,
Motorola Labs, Motorola Inc.
1303 E. Algonquin Rd., Annex 2,
Schaumburg, Il 60196, USA
azl086@motorola.com

Katherine Compton
Department of Electrical and Computer Engineering
Northwestern University
Evanston, IL 60208-3118 USA
kati@ece.northwestern.edu

Scott Hauck
Department of Electrical Engineering
University of Washington
Seattle, WA 98195 USA
hauck@ee.washington.edu

Abstract

Although run-time reconfigurable systems have been shown to achieve very high performance, the speedups over traditional microprocessor systems are limited by the cost of configuration of the hardware. In this paper, we explore the idea of configuration caching, and create some of the first cache management algorithms for reconfigurable systems. We present techniques to carefully manage the configurations present on the reconfigurable hardware throughout program execution. Through the use of the presented strategies, we show that the number of required reconfigurations is reduced, lowering the configuration overhead. We extend these techniques to a number of different FPGA programming models, and develop both lower bound and realistic caching algorithms for these structures. Our simulation results show about a factor of 5 overhead reduction can be achieved over the commercial FPGA structures.

Introduction

In recent years, coupled processor-FPGA systems have attracted a lot of attention because of their promise to deliver the high performance provided by reconfigurable hardware along with the flexibility of general purpose processors. In such systems, portions of an application with repetitive logic and arithmetic computation are mapped to the reconfigurable hardware, while the general-purpose processor handles other portions of the computation.

For many applications, the systems need to be reconfigured frequently during run-time to exploit the full potential of using the reconfigurable hardware. With the increase of size of FPGAs, it could take milliseconds to reconfigure devices such as Xilinx Virtex II. Therefore, by reducing the reconfiguration overhead, the performance of the system is improved. In recent years, much research has focused on techniques to reduce the configuration overhead. Some of these techniques include configuration prefetching [Hauck98a] and configuration compression [Hauck98b, Li99]. In this paper, we exploit another approach called configuration caching that reduces the reconfiguration overhead by buffering configurations on the FPGA.

Caching configurations on an FPGA is similar to caching instructions or data for a processor from main memory. Careful preservatrion of configurations on the array can avoid the expensive reprogramming process when a previously used configuration is again needed. In configuration caching we view the area of the FPGA as a cache. If this cache is large enough to hold more than one computation, configuration cache management techniques will be used to determine when configurations should be loaded and unloaded to best minimize the overall reconfiguration times. However, the traditional caching approaches for general-purpose computational systems are unsuitable for the configuration caching for the following reasons:

1) In general-purpose systems, the data loading latency is fixed because the cache block represents the atomic data transfer unit, while in coupled processor-FPGA systems, the loading latency of configurations may vary because of non-uniform configuration sizes. This variable latency factor could have a great impact on the effectiveness of caching approaches and therefore traditional memory caching approaches such as LRU are not suitable.

2) Since the ratio of the average size of configurations to chip size is much larger than the ratio of the block size to the cache size, only a small number of configurations can be retained on the chip. This makes the system more likely to suffer the thrashing problem, in which the configurations are excessively swapped between the configuration memory and the FPGA.

Given the above limitations, the challenge in configuration caching is to accurately determine which configurations should remain on the chip and which should be replaced when a reconfiguration occurs. An incorrect decision will fail to reduce the reconfiguration overhead and instead lead to a much higher overhead. The non-uniform configuration latency and the small number of configurations that can reside simultaneously on the chip increase the complexity of this decision. Both the frequency and latency factors of configurations need to be considered to ensure the best reconfiguration overhead reduction.

In addition, the different features of various FPGA programming models, such as the Single Context, the Multi-Context, and the Partial Run-Time Reconfigurable models (discussed in depth later) add complexity to configuration caching. Specific properties of each FPGA model require unique caching algorithms. Furthermore,

because of the different architectures and control structures, the computational capacities of the different models vary for a fixed area. Some of the existing architectures treat their configuration memory as a cache. However, none of the research has done quantitative analysis or developed specific algorithms for different caching models. In this paper, we will present a capacity analysis for three prevalent FPGA models and two new FPGA models. Since each model has its unique architecture, there is not a single caching scheme that performs well for all models. Therefore, we have developed effective caching algorithms for each model. These algorithms use either run-time information or profile information of the applications. In order to verify the effectiveness of those realistic algorithms, We have also developed lower bound algorithms or near lower bound algorithms that utilize omniscient execution information of the applications.

FPGA Models

The three FPGA models mentioned previously, the Single Context FPGA, the Partial Run-Time Reconfigurable FPGA, and the Multi-Context FPGA, are the three dominant models for current run-time reconfigurable systems. Before we further discuss these models, we first give definitions of following terms.

RFUOP: The portions of an application that are executed on FPGA are referred as reconfigurable functional unit operation (RFUOPs).

Context: A configuration memory store is referred as a context. Basically each context contains the set of programming bits for configuring logic and interconnect of entire device.

Configuration: The sets of RFUOPs that fit in the context are referred as configurations.

For a Single Context FPGA, the whole chip area must be reconfigured during each reconfiguration. Even if only a small portion of the chip needs to reconfigure, the programming information for the whole chip is rewritten during the reconfiguration. Intuitively, configuration caching for the Single Context model needs to allocate multiple RFUOPs that are likely to be accessed temporally near each other into a single context to minimize switching between configurations. By configuring RFUOPs in groups, the reconfiguration latency can be amortized over the RFUOPs in a context. Since the reconfiguration latency for a Single Context FPGA is fixed (based on the total amount of configuration memory in the device), minimizing the number of times the chip is reconfigured will minimize the reconfiguration overhead.

Multi-Context FPGAs contain one logic and interconnect plane plus multiple memory planes where each memory plane contains the programming bits for configuring the logic and interconnect plane. The structure of a 4-context FPGA is illustrated in Figure 1. Multiple configurations can be stored on a device, however, only one configuration can be actively running at any given time. During reconfiguration, the requested configuration can be loaded into any of the contexts. The loading will not stop execution of the device unless the requested configuration needs to be active immediately. The context containing the required configuration will be switched to control the logic and interconnect in one cycle. Compared with the configuration loading latency, the single cycle configuration switching latency is negligible. In this paper, we consider a Multi-Context model that cannot be partially reconfigured, thus every SRAM context can be viewed as a Single Context FPGA and the methods for allocating RFUOPs onto contexts for the Single Context FPGA could be applied.

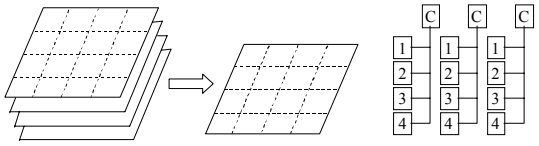

Figure 1. The structure of a 4-context FPGA [DeHon94]

For the Partial Run-Time Reconfigurable (PRTR) FPGA, the area that is reconfigured is just the actual portion required by the new RFUOP, while the rest of the chip remains intact. Unlike the configuration caching for the Single Context FPGA, where multiple RFUOPs are loaded to amortize the fixed reconfiguration latency, the configuration caching method for the PRTR is to load and retain RFUOPs that are required rather than to reconfigure the whole chip. The overall reconfiguration overhead is the summation of the reconfiguration latency of the individual RFUOPs. Compared to the Single Context FPGA, the PRTR FPGA provides more flexibility for performing reconfiguration.

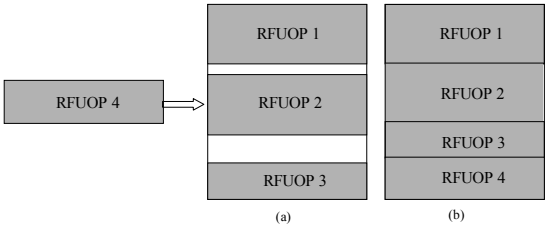

Figure 2**:** An example illustrating the effect of defragmentation. (a) The two small fragments are located between configurations, and neither of them is large enough to hold configuration 4. (b) After defragmentation, RFUOP 4 can be loaded without replacing any of the three other configurations.

Based on the PRTR devices, two new models will be discussed below. In standard PRTR devices, RFUOPs are mapped to fixed locations in the array, and whenever they are loaded they must be mapped to that specific location. Therefore, current PRTR systems are likely to suffer a thrashing problem, if two or more frequently used

RFUOPs occupy overlapping locations in the array. This could cause significant reconfiguration overhead as multiple RFUOPs with same location can be contained within a same loop. Simply increasing the size of the chip will not alleviate this problem. However, this problem can be solved by the Relocation model [Hauser97], which dynamically allocates the position of a configuration on the FPGA at run time instead of at compilation time. Another model, called the Relocation + Defragmentation model (R/D model) [Compton02], further improves the hardware utilization. Similar to the fragments in the memory system, portions of chip area in the current PRTR devices could be wasted because they are too small to hold another RFUOP. These small portions or fragments could represent a significant percentage of chip area. In the R/D model, a special hardware unit called the Defragmentor can move RFUOPs within the chip such that the small unused portions are collected as a single large fragment. This can allow more RFUOPs to be retained on the chip, increasing the hardware utilization and thus reducing the reconfiguration overhead. For example, Figure 2 shows three RFUOPs currently on the chip with two small fragments. Without defragmentation, one of the three RFUOPs would have to be replaced when RFUOP 4 is loaded. However, as shown in the right side of Figure 2, by pushing RFUOP 2 and 3 upward the defragmentor produces one single fragment that is large enough to hold RFUOP 4. The previous three RFUOPs are still present, and therefore the reconfiguration overhead from reloading one of these configurations can be avoided.

Experimental Setup

In order to investigate the performance of configuration caching for the five different programming models presented in the last section, we develop a set of caching algorithms for each model. To conduct the evaluation, An equal amount of hardware resources (in the form of overall area) is allocated to each model. Because the architectures and programming structures of the models vary, the actual areas that devote to computation also vary. Therefore, we compute the capacity of each model as the number of programming bits that can be implemented within the fixed chip area. This in turn affects the number and size of RFUOPs that can fit simultaneously on the different FPGA models. Once the capacity of each model is determined, we will perform 2 more steps. First, we test the performance of the algorithms for each model by generating a sequence of configuration accesses from an execution profile of each benchmark. Second, for each model, caching algorithms are executed on the configuration access sequence, and the configuration overhead for each algorithm is measured.

Capacity analysis

We created VLSI layouts for the programming structures of each of the different FPGA types: the Single Context, the Partial Run-Time Reconfigurable, the Multi-Context, the Relocation FPGA, and the Relocation FPGA and R/D FPGA. These area models are based on the size of tileable structures that comprise each programming architecture. This layout was performed using the Magic tool, and sizes (in λ^2) were obtained for the tiles.

The Single-Context FPGA model is built from shift chains or RAM structures. The PRTR FPGA, however, requires more complex hardware. The programming bits are held in 5-transistor SRAM cells, which form a memory array similar to traditional RAM structures. Row decoders and column decoders are necessary to selectively write to the SRAM cells. Large output tristate drivers are also required near the column decoder to magnify the weak signals provided by the SRAM cells when reading the configuration data off of the array. The Multi-Context FPGA is based on a previously published design [Trimberger97]. We use a four-context design in our representation of a Multi-Context device, where each context is similar to a programming structure of a Single-Context FPGA. A few extra transistors and a latch per active programming bit are required to select between the four contexts for programming and execution. Additionally, a context decoder must be added to determine which of those transistors should be enabled.

The two variants on the PRTR FPGA, the Relocation FPGA and the R/D FPGA, require a short discussion on their basic structure. Both of these designs are one-dimensional row-based models, similar to Chimaera [Hauck97], PipeRench [Goldstein99], DISC [Writhlin95], and Garp [Hauser97]. In this type of FPGA, a full row of computational structures is the atomic unit used when creating an RFUOP: RFUOPs may use one or more rows, but any row used by one RFUOP becomes unavailable to other RFUOPs. While a two-dimensional model could improve the configuration density, the extra hardware required and the complexities of two-dimensional placement limits the benefits gained through the use of the model.

The PRTR design forms the basis of the Relocation FPGA. A small adder and a small register, both equal in width to the number of address bits for the row address of the configuration memory array, were added for the new design. This allows all configurations to be generated such that the "uppermost" row address is 0. Relocating the configuration is therefore as simple as loading an offset into the offset register, and adding this offset to the row addresses supplied when loading a configuration.

Finally, the R/D FPGA [Compton02] is similar to the PRTR with Relocation, with the addition of a row-sized set of SRAM cells that form a buffer between the input of the programming information and the configuration memory array itself. A full row of programming information can be read back into this buffer from the array, and then written back to the array in a different position as dictated by the offset register. In order to make this operation efficient, an additional offset register and a 2:1 multiplexer to choose between the offset registers are

added. This provides one offset for the reading of configuration data from the array, and a separate one for writing the information back to a new location. This buffer requires its own decoder, since it is composed of several data words and is addressable. The column decoder connected to the main array in the basic PRTR design necessary, as the information written from the buffer to the array is the full width of the array. This structure is similar to an architecture proposed by Xilinx [Trimberger95], and used in Virtex devices [Virtex99].

In order to account for the size of the logic and interconnect in these FPGAs, we use the assumption that the programming layer of a Single Context FPGA uses approximately 25% of the area of the chip. All other models are assumed to require this same logic and interconnect area per bit of configuration. See Appendix I for calculation details.

As mentioned before, all models are given the same total silicon area. However, due to the differences in the configuration structures, the number of programming bits, and thus the capacity of the device, varies among our models. For example, according to Appendix I, a Multi-Context model with 1 Megabit of active configuration information and 3 Megabits of inactive information has same area as a PRTR with 2.4 Megabits of configuration information. Thus the PRTR devices has 2.4 times as many logic blocks as the Multi-Context device.

Configuration Sequence Generation

We use two sets of benchmarks to evaluate our caching algorithms for the various FPGA configuration models. One set of benchmarks was compiled and mapped to the Garp architecture [Hauser97], where the compute-intensive loops of C programs are extracted automatically for acceleration on a tightly-coupled dynamically reconfigurable coprocessor [Callahan99]. The other set of benchmarks was created for the Chimera architecture [Hauck97]. In this system, portions of the code that can accelerate computation are mapped to the reconfigurable coprocessor [Hauck98a]. In order to evaluate the algorithms for different FPGA models, we need to create an RFUOP access trace for each benchmark, which is similar to a memory access string used for memory evaluation.

The RFUOP sequence for each benchmark was generated by simulating the execution of the benchmark. During the simulated execution, the RFUOP ID is output when an RFUOP is encountered. After the completion of the execution, an ordered sequence of the execution of RFUOPs is created. In the Garp architecture, each RFUOP in the benchmark programs has size information in term of number of rows occupied. For Chimaera, we assume that the size of an RFUOP is proportional to the number of instructions mapped to that RFUOP.

Configuration Caching Algorithms

In this work, we seek to find caching methods that target the different FPGA models. For each FPGA model, we will develop realistic algorithms that can significantly reduce the reconfiguration latencies. In order to evaluate the performance of these realistic algorithms, we also attempt to develop tight lower bound algorithms by using complete application execution information. Notice that the complete execution information is not available at run time for the realistic algorithms. For the models where true lower bound algorithms are unavailable we will develop algorithms that we believe are near optimal.

We divide our algorithms into 3 categories: run time algorithms, general off-line algorithms, and complete prediction algorithms. The classification of the algorithms depends on the time complexity and input information needed for each algorithm.

The run time algorithms use only basic information on the execution of the program up to that point, and thus must make guesses as to the future behavior of the program. This is similar to run time cache management algorithms such as LRU. Because of the limited information at run time, a prediction of keeping a configuration or replacing a configuration may not be correct and can even cause higher reconfiguration overhead. Therefore, we believe that these realistic algorithms will provide a tight upper bound on reconfiguration overhead.

The complete prediction algorithms use entire execution information of the application, and can use computationally expensive approaches. These algorithms attempt to search the whole execution stream in order to lower the configuration overhead. These provide the optimal (lower bound) or near optimal solution. In some cases these algorithms relax restrictions on system behavior in order to make the algorithm a true (but unachieveable) lower bound.

The general off-line algorithms use profile information of each application, with computationally tractable algorithms. They represent realistic algorithms for the case where static execution information is available, or approximate algorithms where highly accurate execution predictions can be developed. These algorithms will typically perform between the run time and complete prediction algorithms in terms of quality, and are realistic for some situations.

With these three classes of algorithms, we can get upper bounds (the run time algorithms) and lower bounds (the complete prediction algorithms), as well as an estimate of behavior for executions without data-dependent execution (the general off-line algorithms).

Single Context Algorithms

In the next two sections we present a near lower bound algorithm based on simulated annealing, and a more realistic general off-line algorithm, which uses more

restricted information. Note that since there are no run-time decisions in a single context device (if a needed configuration is not loaded the only possible behavior is to overwrite all currently loaded configurations with the required configuration), we do not present a run-time algorithm.

Simulated Annealing Alg. for Single Context FPGA

When a reconfiguration occurs in a Single Context FPGA, even if only a portion of the chip needs to be reconfigured, the entire configuration memory store will be rewritten. Because of this property, multiple RFUOPs should be configured together onto the chip. In this manner, during a reconfiguration a group (context) that contains the currently required RFUOP, as well as possibly one or more later required RFUOPs, is loaded. This amortizes the configuration time over all of the RFUOPs grouped into a context. Minimizing the number of group (context) loadings will minimize the overall reconfiguration overhead.

The method used for grouping has a great impact on the latency reduction as the overall reconfiguration overhead resulted from a good grouping could be much smaller than that resulting from a bad grouping. For example, suppose there are 4 RFUOPs with equal size and equal configuration latency for a computation, and the RFUOP sequence is 1 2 3 4 3 4 2 1, where 1, 2, 3, and 4 are the RFUOP IDs. Given a Single Context FPGA that has the capacity to hold two RFUOPs, the number of context loads is 3 if RFUOPs 1 and 2 are placed in the same group (context), and RFUOPs 3 and 4 are placed in another. However, if RFUOPs 1 and 3 are placed in the same group (context) and RFUOPs 2 and 4 are placed in the other, the number of context loads will be 7.

In order to create the optimal solution for grouping, one simple method is to create all combinations of RFUOPs and then compute reconfiguration latency for all possible groupings, from which an optimal solution can be found. However, this method has exponential time complexity, and is therefore not applicable for real applications. In this paper, we instead use a simulated annealing approach to acquire a near optimal solution. For the simulated annealing algorithm, we use the exact reconfiguration overhead for a given grouping as our cost function, and the moves consist of shuffling the different RFUOPs between contexts. Specifically, at each step an RFUOP is randomly picked to move to a randomly selected group, and if there is not enough room in that group to hold the RFUOP, RFUOPs in that group are randomly chosen to move to other groups. Once finished, the reconfiguration overhead of the grouping is computed by applying the complete RFUOP sequence.

General Off-line Alg. for Single Context FPGA

Although the simulated annealing approach can generate a near optimal solution, the high computation complexity and the exact execution sequence make this solution unreasonable for most real applications. We therefore propose an algorithm more suited for general purpose use. The Single Context FPGA requires that the whole configuration memory will be rewritten if a demanded RFUOP is not currently on the chip. Therefore, if two consecutive RFUOPs are not allocated to the same group, a reconfiguration will result. Our algorithm attempts to compute the likelihood of RFUOPs following one another in sequence, and use this knowledge to minimize the number of reconfigurations required. Before we further discuss this algorithm, we first give the definition of a "correlate" as used in the algorithm.

Definition 1: Given two RFUOPs and an RFUOP sequence, RFUOP A is said to correlate to RFUOP B if in the RFUOP sequence there exists any consecutive appearance of A and B.

For the Single Context FPGA, highly correlated RFUOPs are allocated into the same group. Therefore the number of times a context is loaded is greatly decreased, and thus the reconfiguration overhead is minimized. In our algorithm, we first build an adjacency matrix of RFUOPs. Instead of using 0 or 1 as a general adjacency matrix does, the degree of correlation of every RFUOP pairs (the number of times two RFUOPs are next to each other) is recorded. These correlations could be estimated from expected behavior or determined via profiling. The details of our grouping algorithm are as follows:

1. Create COR, where COR[I, J]= number of times RFUOP I correlates to J
2. While any A[I, J] > 0, do
   - 2.1 Find I, J such that COR[I, J] + COR[J, I] is maximized
   - 2.2 If SIZE[I] + SIZE[J] <= Maximum Context Size
     - 2.2.1. Merge Group I and group J, and add together sizes
     - 2.2.2. Foreach group K other than I and J
       - 2.2.2.1. A[I, K] += A[J, K]; A[K, I] += A[K, J];
       - 2.2.2.2. A[J, K] = 0; A[K, J] = 0;
   - 2.3 A[I, J] = 0; A[J, I] = 0;

Figure 3 illustrates an example of the general off-line algorithm. Each line connects a pair of correlated RFUOPs and the number next to each line indicates the degree of the correlation. As presented in the algorithm, we will merge the highly correlated groups together under the size constraints of the target architecture. In this example, assume that the chip can only retain at most 3 RFUOPs at a time, although in reality this depends on the sizes of the RFUOPs. At the first grouping step we place RFUOP17 and RFUOP4 together. In the 2nd step we add RFUOP43 into the group formed at step 1, since it has a correlation of 30 (15+15) to that group. We then group RFUOP2 and RFUOP34 together in step 3, and they cannot be merged with the previous group because of the size restriction. Finally, in the 4th step RFUOP22 and RFUOP68 are grouped together.

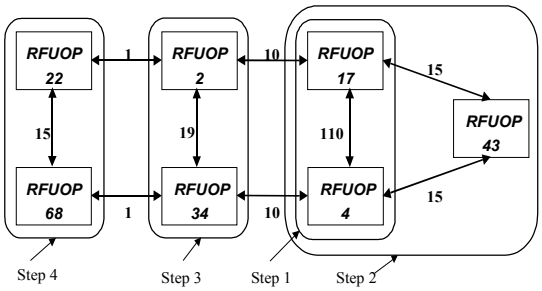

Figure 3**:** An example to illustrate the general off-line algorithm for Single Context FPGA.

Compared to the simulated annealing algorithm, this algorithm only requires profile information on the degrees of correlation between RFUOPs. In addition, since the number of RFUOPs tends to be much smaller than the length of the RFUOP sequence, it should be much quicker to find a grouping by searching the correlation matrix instead of traversing the RFUOP sequence as the simulated annealing algorithm does. Therefore, the computation time is significantly reduced.

Multi-Context Algorithms

In this section we present algorithms for multi-context devices. This includes a complete prediction algorithm that represents a near lower bound, and a general offline algorithm that couples the single-context general offline algorithm with a run-time replacement policy.

Complete Prediction Alg. for Multi-Context FPGA

A Multi-Context FPGA can be regarded as multiple Single Context FPGAs, since the atomic unit that must be transferred from the host processor to the FPGA is a full context. During a reconfiguration, one of the inactive contexts is replaced. In order to reduce the reconfiguration overhead, the number of reconfigurations must be reduced. The factors that could affect the number of reconfigurations are the configuration grouping method and the context replacement policies.

We have discussed the importance of the grouping method for the Single Context FPGA, where an incorrect grouping may have significantly larger overhead than a good grouping. This is also true for the Multi-Context FPGA, where a context (a group of configurations) remains the atomic reconfiguration data transfer unit. The reconfiguration overhead caused by the incorrect grouping remains very high even though the flexibility provided by the Multi-Context FPGA can somewhat reduce part of the overhead.

As mentioned previously, even the perfect grouping will not minimize the reconfiguration overhead if the policies used for context replacement are not considered. A context replacement policy specifies which context should be replaced once a demanded configuration is present. Just as in the general caching problem where frequently used blocks should remain in the cache, the contexts that are frequently used should be kept configured on the chip. Furthermore, if the atomic configuration unit (context) is considered as a data block, we can view the Multi-Context FPGA as a general cache and apply standard cache algorithms. More specifically, we can apply an existing optimal replacement algorithm called the Belady algorithm [Belady66] to the Multi-Context FPGA context replacement problem.

The Belady algorithm is well known in the operating systems and computer architecture fields. It states that the fewest number of replacements can be achieved provided the memory access sequence is known. This algorithm is based on the idea that a data item is most likely to be replaced if it is least likely to be accessed in the near future. For a Multi-Context FPGA, the optimal context replacement can be achieved as long as the context access string is available. When the RFUOP sequence is known, it is trivial to create the context access string by transforming the RFUOP sequence. We integrate the Belady algorithm into the simulated annealing grouping method used in the Single Context model to achieve the near optimal solution. Specifically, for each grouping generated, the number of the context replacements determined by the Belady algorithm is calculated as the cost function of the simulated annealing approach.

The reconfiguration overhead for a Multi-Context FPGA is therefore the number of context loads multiplied by the configuration latency for a single context. As mentioned above, the factors that can affect the performance of configuration caching for the Multi-Context FPGA are the configuration grouping and the replacement policies. Since the optimal replacement algorithm is integrated into the simulated annealing approach, this algorithm will provide the near optimal solution. We consider this algorithm to be a complete prediction algorithm.

Least Recently Used (LRU) Alg. for Multi-Context

The LRU algorithm is a widely used memory replacement algorithm in operating system and architecture. Unlike the Belady algorithm, the LRU algorithm does not require future information to make a replacement decision. Because of the similarity between the configuration caching problem and the data caching problem, we can apply the LRU algorithm for the Multi-Context FPGA model. The LRU is more realistic than the Belady algorithm, but the reconfiguration overhead incurred is higher. The basic steps are outlined below:

1. Apply the Single Context general off-line algorithm to acquire a final grouping of RFUOPs into contexts, and give each group formed its own ID.
2. Traverse the RFUOP sequence, and for each RFUOP appearing, change the RFUOP ID to the corresponding group ID. This will generate a context access sequence.

3. Apply the LRU algorithm to the context access string. Increase the total number of context loads by one when a replacement occurs.

Algorithms. for PRTR FPGA

An advantage that the PRTR FPGA has over the Single Context FPGA is greater flexibility of loading and retaining configurations. Any time a reconfiguration occurs, instead of loading the whole group, only a portion of the chip is reconfigured while the other RFUOPs located elsewhere on the chip remain intact. The basic idea of configuration caching for PRTR is to find the optimal location for each RFUOP. This is to avoid the thrashing problem that could be caused if RFUOPs used frequently in succession occupy overlapping positions on the FPGA. In order to reduce the reconfiguration overhead for the Partial Run-Time Reconfigurable FPGA, we need to consider two major factors: the reconfiguration frequency and the average latency of each RFUOP. Any algorithm that attempts to lower only one factor will fail to produce an optimal solution because the reconfiguration overhead is the product of the two. A complete prediction algorithm that can achieve near optimal solution and a general off-line algorithm that can significantly reduce the running time are presented below.

Simulated Annealing Algorithm for PRTR FPGA

Similar to the simulated annealing algorithm used for the Single Context FPGA, the purpose of annealing for the Partial Run-Time Reconfigurable FPGA is to find the mapping for each configuration such that the reconfiguration overhead is minimized. For each step, a randomly selected RFUOP is assigned to a random position within the chip and the exact reconfiguration overhead is then computed.

Alternate Simulated Annealing Algorithm for PRTR

In the simulated annealing algorithm presented in the last section, the computation complexity is very high since the RFUOP sequence must be traversed to compute the overall reconfiguration overhead after every move. To reduce the run time, we develop an alternative annealing algorithm that does not require to traverse the lengthy RFUOP sequence. An adjacency matrix of size $N \times N$, where N is the number of the RFUOPs, is built, to record the possible conflicts between RFUOPs. In order to reduce the reconfiguration overhead, the conflicts that will create larger RFUOP loading latency are distributed to unoverlapped locations. This is done by modifying the cost computation step of the previous algorithm. Before presenting the alternate simulated annealing algorithm, we first give the definition of a "conflict" as used in our discussion.

Definition 2: Given two configurations and their positions on the FPGA, RFUOP A is said to be in conflict with RFUOP B if any part of A overlaps with any part of B.

We now present our simulated annealing algorithm for the PRTR FPGA.

1. Create an $N \times N$ matrix, where N is the number of RFUOPs. All values of $A[i, j]$ are set to be 0, where $0 \le i, j \le N-1$.
2. Traverse the RFUOP sequence, for any RFUOP j that appears between two consecutive appearances of an RFUOP i, $A[i, j]$ is increased by 1. Notice that multiple appearances of an RFUOP j only count once between two consecutive appearances of an RFUOP.
3. Assign a random position for each RFUOP. An $N \times N$ adjacency matrix B is created.
4. At each step of in the annealing, recalculate matrix B:
   4.1. A random selected RFUOP is reallocated to a random location within the chip. After the move, if two RFUOPs i and j conflict, set $B[i, j]$ and $B[j, i]$ to be 1.
   4.2. For any $B[i, j]=1$, multiply the value of $A[i, j]$ by the RFUOP loading latency of j. The new cost is computed as the summation of the results of all the products.
   4.3. Accept the move based on the cost.

Generally, the number of total RFUOPs is much less than the length of the RFUOP sequence. Therefore, by looking up the conflict matrices instead of the whole configuration sequence, the time complexity can be greatly decreased. However, the quality of the algorithm may decrease because the matrix may not represent the conflicts exactly.

Algorithms for R/D FPGA.

For R/D FPGA, the replacement policies have a great impact on reducing the reconfiguration overhead. This is because a high degree of flexibility is available in choosing victim RFUOPs when a reconfiguration is required. With Relocation, an RFUOP can be dynamically remapped and loaded to an arbitrary position. With defragmentation, a demanded RFUOP can be loaded as long as there is enough room on the chip, since the small fragments existing on the chip can be merged. Instead of giving the algorithms for Relocation FPGA, we first analyze the case of R/D FPGA. This includes a lower bound algorithm that relaxes the restriction in the system, a general off-line algorithm integrating the Belady algorithm, and two run time algorithms using different approaches.

Lower Bound for R/D FPGA

As discussed previously, the major problems that prevent us from acquiring an optimal solution of configuration caching are the different sizes and different loading latencies of different RFUOPs. Generally, the loading latency of an RFUOP is proportional to the size of the configuration.

The Belady algorithm [Belady66] gives the optimal replacement for the case that the RFUOP access string is known and the data transfer unit is uniform. Given the RFUOP sequence for the R/D FPGA model, we can achieve a lower bound of our problem if we assume that a portion of any RFUOP can be transferred. Under this assumption, when a reconfiguration occurs, only a portion of an RFUOP might be replaced while the rest is still kept on the chip. Once the removed RFUOP is needed again, only the missing portion (which might be the whole RFUOP if it was previously completely removed) is loaded instead of always loading the entire RFUOP even if it is still partially programmed. We present the Lower Bound Algorithm as follows:

1. If a required RFUOP is not on the chip, do the following:
    1.1. Find the missing portion of the RFUOP. While the missing portion is greater than the free space on the chip, do the following:
        1.1.1. For all RFUOPs that are currently on the chip, a victim RFUOP is identified such that in the RFUOP sequence its next appearance is later than the appearance of all others.
        1.1.2. Let R = the size of the victim + the size of the free space – the missing portion.
        1.1.3. If R is greater than 0, a portion of the victim that equals R is retained on chip while the other portion is replaced and added to the free space. Otherwise add the space occupied by the victim to the free space.
    1.2. Load the missing portion of the demanded RFUOP into the free space. Increase the RFUOP overhead by the loading latency of the missing portion.

In our algorithm, we assumed that a portion of the any RFUOP can be retained on the chip, and during reconfiguration only the missing portion of the demanded RFUOP will be loaded. This can be viewed as loading multiple atomic configuration units. Therefore, this problem can be viewed as the general caching problem, with the atomic configuration unit as the data transfer unit. Since the Belady algorithm provides the optimal replacement for the general caching problem, it can also provide the lowest configuration overhead for the R/D FPGA.

General Off-line Algorithm for R/D FPGA.

Since the Belady algorithm can provide a lower bound for the fixed size problem, it can be modified into a more realistic off-line algorithm that can deal with non-uniform sizes of RFUOPs. As in the Belady algorithm, for all RFUOPs that are currently on chip, we identify the one that will not appear in the RFUOP sequence until all others have appeared. But instead of replacing that RFUOP, as in the Belady algorithm, the victim configuration is selected by considering the factors of size and loading latency. Before we further discuss the algorithms, we first give the definition of a reappearance window used in our algorithms.

Definition 3: A reappearance window W is the shortest subsequence of the RFUOP sequence, starting at the current RFUOP, which contains an occurrence of all currently loaded RFUOPs. If a loaded RFUOP does not occur again, the reappearance window is the entire remaining reconfiguration stream.

We now present our general off-line algorithm for the R/D FPGA:

1. If a demanded RFUOP is not currently on the chip, do the following.
    1.1. While there is not enough room to load the RFUOP, do the following:
        1.1.1. Find the reappearance window W.
        1.1.2. For each RFUOP, calculate the total number of appearances in W
        1.1.3. For each RFUOP, multiply the loading latency by the number of appearances. The RFUOP with the smallest such value is replaced.
    1.2. Load the demanded RFUOP. Increase the overall latency by the loading latency of the RFUOP.

LRU Algorithm for R/D FPGA.

Since the Relocation R/D FPGA model can be viewed as a general memory model, we can use a LRU algorithm for our reconfiguration problem. Here, we traverse the RFUOP sequence, and when a demanded RFUOP is not on the chip and there is not enough room to load the RFUOP, an RFUOP on the chip is selected to be removed by the LRU algorithm. Although simple to implement, this algorithm may display poor quality because it ignores the sizes and latencies of the RFUOPs.

Penalty Oriented Algorithm for R/D FPGA.

Since the non-uniform size of RFUOPs is not considered as a factor in LRU algorithm, a high RFUOP overhead could result. For example, consider an RFUOP sequence 1 2 3 1 2 3 1 2 3 …, RFUOPs 1, 2 and 3 have sizes of 1000, 10 and 10 programming bits respectively. Suppose also that the size of the chip is 1010 programming bits. According LRU algorithm, the RFUOPs are replaced in the same order of the RFUOP sequence. However, the configuration overhead will be much smaller if RFUOP 1 is always kept on the chip. This does not mean that we always want to keep larger RFUOPs on the chip as keeping larger configurations with low reload frequency may not reduce the reconfiguration overhead. Instead,

both size and frequency should be considered in the algorithm. Therefore, we use a variable "credit" to determine the victim [Young94]. The algorithm is as following:

1. If a demanded RFUOP is currently on the chip, set its credit equal to its size. Else do following:
    1.1. While there is not enough room to load the required RFUOP:
        1.1.1. For all RFUOPs on chip, replace the one with the smallest credit and decrease the credit of all other RFUOPs by that value.
    1.2. Load the demanded RFUOP and set its credit equal to its size.

General Off-line Algorithm for Relocation FPGA

One major advantage that the R/D FPGA has over the Relocation FPGA is the ability to have higher utilization of the space on the chip. Any small fragments can contribute to one larger area such that an RFUOP could possibly be loaded without forcing a replacement. However, for PRTR with only Relocation those fragments could be wasted. This could cause an RFUOP that is currently on chip to be replaced and thus may result in extra overhead if the replaced RFUOP is demanded again very soon. Therefore, the main focus is to minimize fragments resulted by reconfigurations. We present the algorithm as following:

1. If a demanded RFUOP is not currently on the chip, do the following.
    1.1. While there is not enough room to load the RFUOP, do the following:
        1.1.1. Find the reappearance window W.
        1.1.2. For each RFUOP, calculate the total number of appearances in W
        1.1.3. For each RFUOP, multiply the loading latency and the number of appearances, producing a cost.
        1.1.4. For each RFUOP on chip, presume that it is to be the candidate victim, identify the adjacent configurations that must also be removed to make room for the demanded RFUOP. Sum up the costs of all the potential victims.
        1.1.5. Identify the smallest sum and the victim(s) that produce the smallest cost are replaced.
    1.2. Load the demanded RFUOP. Increase the overall latency by the loading latency of the configuration

The general off-line heuristic that applied to the R/D FPGA is also implemented in this algorithm. The major difference for this algorithm is to consider the geometric positions of the RFUOPs. Since the R/D FPGA model has the ability to collect the fragments, the RFUOPs are replaced in the increasing order of their costs (load latency times appearance in the reappearance window). However, this scheme does not work for the Relocation FPGA if the chosen victim RFUOPs are separated by non-victim RFUOPs because the system cannot merge the non-adjacent spaces. Therefore, when multiple RFUOPs are to be replaced in the Relocation FPGA, these RFUOPs must be adjacent or separated only by empty fragments. Considering this geometric factor, the victims to be replaced are adjacent RFUOPs (or separated by fragments) that produce the overall smallest cost.

Simulation Results and Discussion

All algorithms are implemented in C++ on a Sun Sparc-20 workstation. As can be seen in Figure 4, the reconfiguration penalties of the PRTR is much smaller (64% to 85% smaller) than the Single Context model. This is because with almost the same capacity the PRTR model can significantly reduce the average reconfiguration latency of the Single Context model without incurring a much larger number of reconfigurations. The Multi-Context model has smaller reconfiguration overhead (20% to 40% smaller) than the PRTR when the chip silicon is small. With small silicon area, the Multi-Context model is more efficient because of its much larger configuration area. With the silicon area becomes larger, the number of conflicts incurred in the PRTR model is greatly reduced and thus the PRTR has almost the same reconfiguration penalty as the Multi-Context model. In fact, the PRTR performs even better than the Multi-Context model in some cases. The Multi-Context device must reload a complete context at each time, *resulting in a per-configuration penalty that increases with the size, whereas the per-reconfiguration penalty is unchanged with the PRTR FPGA.* This, combined with the reduction in PRTR conflicts as the FPGA size increases, the overall reconfiguration overhead of the PRTR FPGA is smaller than that of the Multi-Context FPGA.

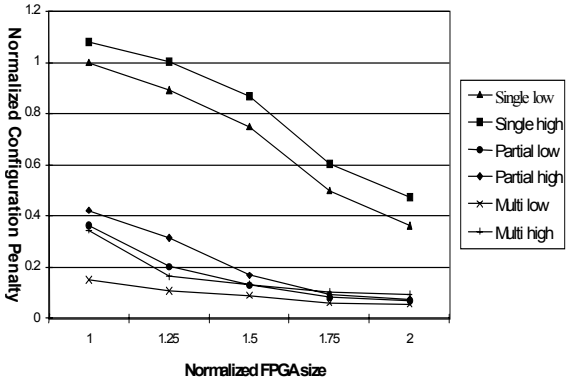

Figure 4. Reconfiguration overheads of the Single Context FPGA, the PRTR, and the Multi-Context models. The "low" represents the lower

bound or near optimal solution for each model, and the "high" represents the upper bound.

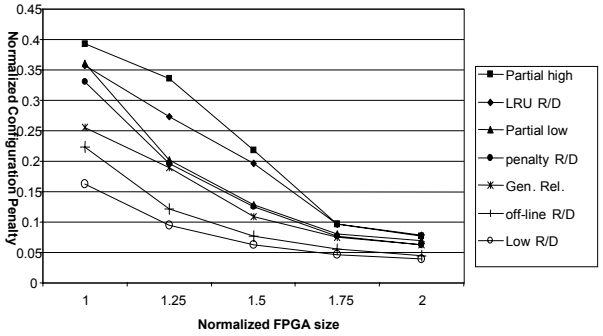

Figure 5. Reconfiguration overheads of the PRTR, the Relocation FPGA, and the R/D FPGA. The "Low R/D" represents the lower bound algorithm for the R/D FPGA.

Figure 5 demonstrates the reconfiguration overheads of the two new models we proposed. As can be seen, R/D FPGA significantly improves the performance of PRTR. For the R/D FPGA, the general off-line algorithm performs almost as well as the lower bound algorithm in the reconfiguration overhead reduction, especially when the chip silicon becomes larger. Note that the lower bound algorithm relaxes the PRTR model restrictions by allowing partial replacement of the RFUOPs. As can be seen in Figure 5, future information is very important, as the general off-line algorithm for the Relocation FPGA performs better than both the LRU and the penalty oriented algorithms for the R/D FPGA. The LRU algorithm has shown that it is not suitable for configuration caching since the sizes and latencies of RFUOPs are not considered.

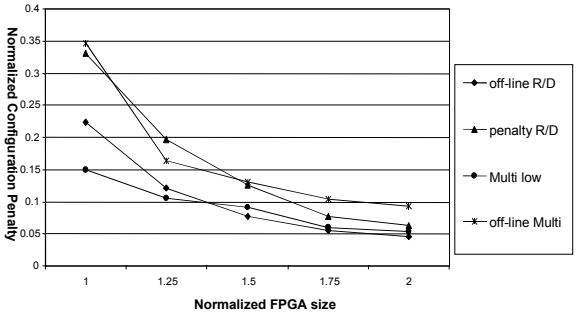

Figure 6. Comparison between the R/D FPGA model and the Multi-Context model.

Figure 6 compares the R/D FPGA model and the Multi-Context model. As we can see, when the chip silicon is small, the complete prediction algorithm for the Multi-Context FPGA performs better than the general off-line algorithm for the R/D FPGA. However, as the chip silicon increases, the general off-line algorithm for the R/D FPGA has almost the same ability to reduce the reconfiguration overhead as the complete prediction algorithm for the Multi-Context FPGA. In addition, the penalty oriented algorithm (run time algorithm) for the R/D FPGA performs slightly better than the general off-line algorithm for the Multi-Context FPGA.

Conclusions

Configuration caching, where configurations are retained on chip until they are required again, is a technique to reduce the reconfiguration overhead. However, the limited on-chip configuration memory and the non-uniform configuration latency add complexity in deciding which configurations to retain to maximize the odds that the required data is present in the cache.

In this work we present some of the first cache management algorithms for reconfigurable computing systems. We have developed new caching algorithms targeted at a number of different FPGA configuration models, and created lower bounds to quantify the maximum achievable reconfiguration reductions possible. In addition to the three currently dominant models (Single Context FPGA, Partial Run-Time Reconfigurable FPGA, and Multi-Context FPGA), we proposed two new models, the Relocation FPGAmodel and the R/D FPGA model, which significantly improve the performance of PRTR FPGA. For each of these five models, we have implemented a set of algorithms to reduce the reconfiguration overhead. The simulation results demonstrate that the Partial Run-Time Reconfigurable FPGA and the Multi-Context FPGA are significantly better caching models than the traditional Single Context FPGA.

Appendix I

Based on the structures given and presented in the paper, the size equations for the different FPGA models are as follows:

R = number of rows of configuration bits

C = number of word-size columns of configuration bits (we use 32 bits /word)

Single Context: 291264RC

PRTR: 260336RC + 476R + 392R × lg(R) + 367217.5C + 487.5C × lg(C)

Multi-Context (4 contexts): 636848RC + 476R+ 392R × lg(R) + 385937.5Col+ 487.5Col × lg(C)

PRTR Relocation: 260336RC + 476R + 392R × lg(R) + 367217.5C + 487.5C × lg (C) + 20300lg(R)

PRTR Relocation + Defragmentation: 260336RC + 476R + 392R × lg(R) + 407404C + 392C × lg(C) + 365040 + 30186 × lg(R)

Given these equations, the different styles will have the following area for 1 Megabit of configuration information (for the Multi-Context, 1 Megabit of active configuration information, 3 Megabits of inactive information).

	Single	PRTR	Multi (4)	PRTR Reloc	Reloc+ Defrag
Area(λ^2)	8.5×10^9	8.5×10^9	20.9×10^9	8.6×10^9	8.6×10^9

References

[Belady66] L. A. Belady "A Study of Replacement Algorithms for Virtual Storage Computers," *IBM Systems Journal* 5, 2, 78-101, 1966.

[Compton02] K. Compton, J. Cooley, S. Knol, S. Hauck, "Abstract: Configuration Relocation and Defragmentation for FPGAs", *IEEE Transactions on VLSI*, Vol. 10, No. 3., pp. 209-220. June 2002

[DeHon94] Andre DeHon, "DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century", *IEEE Workshop on FPGAs for Custom Computing Machines,* April 1994.

[Goldstein99] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, R. Laufer, "PipeRench: *A Coprocessor for Streaming Multimedia Acceleration", Proceedings of the 26th Annual International Symposium on Computer Architecture,* June 1999.

[Hauck97] S. Hauck, T. W. Fry, M. M. Hosler, J. P. Kao, "The Chimaera Reconfigurable Functional Unit", *IEEE Symposium on FPGAs for Custom Computing Machines,* pp. 87-96, 1997.

[Hauck98a] S. Hauck, "Configuration Prefetch for Single Context Reconfigurable Coprocessors", *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 65-74, 1998.

[Hauck98b] S. Hauck, Z. Li, E. Schwabe, "Configuration Compression for the Xilinx XC6200 FPGA", *IEEE Symposium on FPGAs for Custom Computing Machines,* 1998.

[Hauser97] J. R. Hauser, J. Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor", *IEEE Symposium on FPGAs for Custom Computing Machines,* pp. 12-21, 1997.

[Li99] Z. Li, S.Hauck, "Don't Care Discovery for FPGA Configuration Compression", *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 91-100, 1999.

[Trimberger95] S. Trimberger, "Field Programmable Gate Array with Built-In Bitstream Data Expansion", *U.S. Patent 5,426,379*, issued June 20, 1995.

[Trimberger97] S. Trimberger, D. Carberry, A. Johnson, J. Wong, "A Time-Multiplexed FPGA", *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 22-28, 1997.

[Wittig96] R. D. Wittig, P. Chow, "OneChip: An FPGA Processor with Reconfigurable Logic," *IEEE Symposium on FPGAs for Custom Computing Machines,* 1996.

[Young94] N. E. Young. "The k-server dual and loose competitiveness for paging", *Algorithmica, 11(6),* 535-541, June 1994