IEEE Symposium on FPGAs for Custom Computing Machines, 2000.

Configuration Caching Techniques for FPGA

Zhiyuan Li, Katherine Compton
Department of Electrical and Computer Engineering
Northwestern University
Evanston, IL 60208-3118 USA
{zl, kati}@ece.nwu.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 hardware configuration. In this paper, we explore the idea of configuration caching. 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.

Introduction

In recent years, reconfigurable computing systems such as Garp [Hauser97], OneChip [Wittig96] and Chimera [Hauck97] 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 computation.

For many applications, the systems need to be reconfigured frequently during run-time to exploit the full potential of using the reconfigurable hardware. Reconfiguration overhead is therefore a major concern because the CPU must sit idle during this process. By reducing the overall reconfiguration overhead, the performance of the system is improved. Therefore, many studies have involved examining techniques to reduce the configuration overhead. Some of these techniques include configuration prefetching [Hauck98a] and configuration compression [Hauck98b, Li99]. In this paper, we present another approach called configuration caching that reduces the reconfiguration overhead by buffering configurations on the FPGA.

Caching configurations on an FPGA, which is similar to caching instructions or data in a general memory, is to retain the configurations on the chip so the amount of the data that needs to be transferred to the chip can be reduced. In a general-purpose computational system, caching is an important approach to hide memory latency by taking advantage of two types of locality—spatial locality and temporal locality. Spatial locality states that items whose addresses are near one another tend to be referenced close together in time. Temporal locality addresses the tendency of recently accessed items to be accessed again in the near future. These two localities also apply to the caching of configurations on the FPGA in coupled processor-FPGA systems. 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 block represents the atomic data transfer unit, while in the coupled processor-FPGA systems, the latency of configurations may vary. This variable latency factor could have a great impact on the effectiveness of caching approaches. In traditional memory caching, the data items frequently accessed are kept in the cache in order to minimize the memory latency. However, this might not be true in coupled processor-FPGA systems because of the non-uniform configuration latency. For example, suppose that we have two configurations with configuration latencies 10ms and 1000ms respectively. Even though the first configuration is executed 10 times as often as the second one, the second configuration is likely to be cached in order to minimize the total configuration overhead.

2) Since the ratio of the average size of configurations to chip area is much smaller 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 suffer the “thrashing problem”, in which the configurations are swapped between the configuration memory and the FPGA very frequently.

The challenge in configuration caching is to 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 lead to a much higher reconfiguration overhead than a correct decision. The non-uniform configuration latency and the small number of configurations that reside simultaneously on the chip increase the complexity of this decision. Both frequency and latency factors of configurations need to be considered to assure the best reconfiguration overhead reduction. Specifically, in certain situations retaining configurations with high latency is better than keeping frequently required configurations that have lower latency. In other situations, keeping configurations with high latency and ignoring the frequency factor will result switching between other frequently required configurations because they cannot fit in the remaining area. The switching causes reconfiguration overhead in this case that will not occur if the configurations with high latency but low frequency are unloaded.

In addition, the different features of different 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 varies for a fixed area. In this paper, we will present the capacity analysis for three prevalent FPGA models and two new FPGA models. For each model, we implement realistic caching algorithms that use either run-time information or profile information of the applications. Furthermore, we develop lower bound algorithms or near lower bound algorithms that apply 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. 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 reconfigured, the whole chip is rewritten during the reconfiguration. Configuration caching for the Single Context model allocates multiple configurations that are likely to be accessed near in time into a single context to minimize switching of contexts. By caching configurations in this way, the reconfiguration latency is amortized over the configurations in a context. Since the reconfiguration latency for a Single Context FPGA is fixed (based on the total amount of configuration memory), minimizing the number of times the chip is reconfigured will minimize the reconfiguration overhead.

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

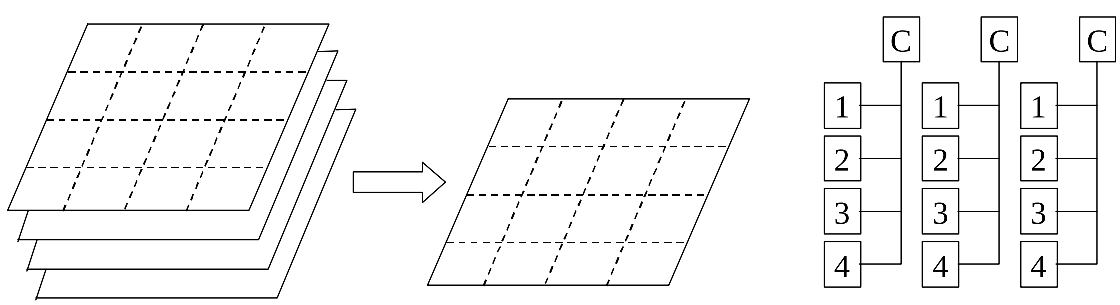

Figure 1: The structure of a 4-context FPGA [Trimberger97].

The configuration mechanism of the Multi-Context model [Trimberger97] is similar to that of the Single Context FPGA. However, instead of having one configuration stored in the FPGA, multiple complete configurations are stored. Each complete configuration can be viewed as multiple configuration memory planes contained within the FPGA. The structure of a 4-context FPGA is illustrated in Figure 1. For the Multi-Context FPGA, the configuration can be loaded into any of the contexts. When needed, the context containing the required configuration will be switched to control the logic and interconnect plane. Even though the latency of a single switch of is negligible, the frequency of the switching will lead to a significant overhead. Therefore, the reconfiguration latency for the Multi-Context FPGA includes not only the configuration loading latency, but also the configuration switching latency. Because every SRAM context can be viewed as a Single Context FPGA and the method for allocating configurations onto contexts for the Single Context FPGA can be applied.

We will present two new models based on the Partial Run-Time Reconfigurable (PRTR) FPGA. The current PRTR systems are likely to suffer a “thrashing problem” if two or more frequently used configurations occupy overlapping locations in the array. Simply increasing the size of the chip will not alleviate this problem. Instead, we present a new model, the Relocation model, 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 [Compton00], further improves the hardware utilization. In current PRTR systems, during reconfiguration portions of chip area could be wasted because they are too small to hold another configuration. These small portions are called fragments similar to the fragments in traditional memory systems, and they could represent significant percentage of chip area. In the Relocation + Defragmentation model, a special hardware unit called the Defragmentor can move configurations within the chip such that the small unused portions are collected as a single large fragment. This allows more configurations to be retained on the chip, increasing the hardware utilization and thus reducing the reconfiguration overhead. For example, Figure 2 shows three configurations currently on the chip with two small fragments. Without defragmentation, one of the three configurations has to be replaced when configuration 4 is needed. However, as shown in right side of the Figure 2, by pushing the configuration 2 and 3 upward the defragmentor produces one single fragment that is large enough to hold configuration 4. Notice that the previous three configurations are still present, and therefore the reconfiguration overhead caused by reloading a replaced configuration can be avoided.

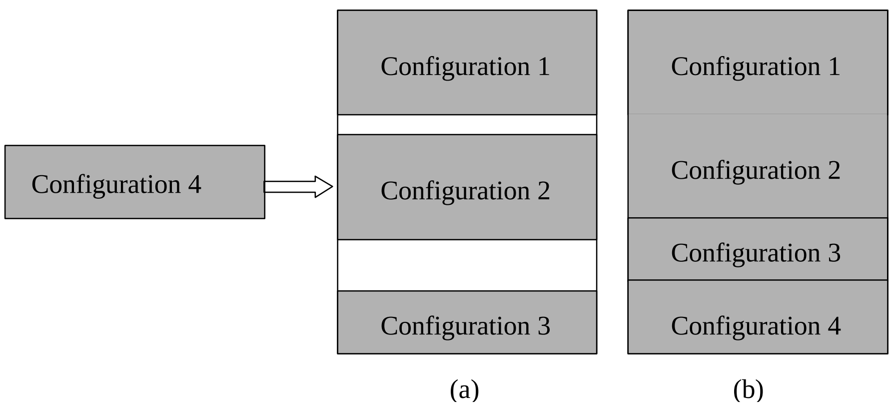

Figure 2: An example illustrating the effect of defragmentation. The two small fragments are located between configurations, and neither of them is large enough to hold configuration 4 (a). After defragmentation, configuration 4 can be loaded without replacing any of the three other configurations (b).

Experimental Setup

In order to investigate the performance of configuration caching for the five different models presented in the last section, we develop a set of caching algorithms for each model. A fixed amount of hardware resources (in the form of overall area) is allocated to each model. To conduct the evaluation, we must perform three steps. First, for each model, the capacity equation must be derived such that the number of programming bits can be calculated for a given architecture model and a given area. Second, we test the performance of the algorithms for each model by generating a sequence of configuration accesses from a profile of execution information from each benchmark. Third, 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 layouts for the programming structures of the different FPGA types: the Single Context, the Partial Run-Time Reconfigurable, the Multi-Context, the PRTR with Relocation, and the PRTR with Relocation and Defragmentation. These area models are based on the layout of tileable structures that composed the larger portions of the necessary architecture. This layout was performed using the Magic tool, and sizes (in lambda^2) were obtained for the tiles.

The Single-Context FPGA model is composed of a two-phase shift chain of programming bits, which forms the path for the input of configuration data. 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 the information found in [Trimberger97]. We use a four-context design in our representation of a Multi-Context device, where each context is similar to a single plane of a PRTR 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 Relocation + Defragmentation FPGA, require a short discussion on their basic operation. Both of these designs are one-dimensional row-based models, similar to Chimaera [Hauck97] and PipeRench [Goldstein99]. In this type of FPGA, a full row of computational structures is the atomic unit used when creating a configuration: configurations may use one or more rows, but any row used by one configuration becomes unavailable to other configurations. 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. One-dimensionality simplifies the creation of relocation and defragmentation hardware, however, because these problems become one-dimensional issues similar to memory allocation and defragmentation.

The PRTR design forms the basis of the PRTR with 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 memory array were added for the new design. This allows all configurations to be generated such that the "uppermost" address is 0. Relocating the configuration is therefore as simple as loading an offset into the offset register, and adding this offset to the addresses supplied when loading a configuration.

Finally, the PRTR with Relocation and Defragmentation [Compton00] 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 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 row decoder, since it is composed of several data words. A column decoder between the incoming configuration data and the buffer is not needed, as we design the structure to have multiple rows, but a single column. However, the column decoders connected to the main array (as appear in the basic PRTR design) are no longer 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].

In order to account for the size of the logic and interconnect in these FPGAs, we use the assumption that the programming layer of an FPGA uses approximately 25% of the area of the chip. Because most current FPGAs are of the Single Context variety, we base the other 75% of the area on this style of model, using the equation:

$$0.25 \times \text{TOTAL\_SIZE} = \text{MEM\_SINGLE\_CONTEXT} \Rightarrow 0.75 \times \text{TOTAL\_SIZE} = 3 \times \text{MEM\_SINGLE\_CONTEXT},$$

where $0.75 \times$ TOTAL_SIZE is the amount to add to the sizes computed using our area models in order to obtain full chip sizes for those models. See Appendix I for calculation details.

As mentioned before, all models are given the same total area. However, due to the differences in the hardware structures, the number of programming bits 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.

Configuration Sequence Generation and Size Definition

We use two sets of benchmarks to evaluate caching algorithms for FPGA models. One set of the benchmarks was compiled and applied on 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 [Hauck98c]. For both sets of the benchmarks, the mappings to the coprocessors are referred to RFUOPs. In order to evaluate the algorithms for different FPGA models, we need to create an RFUOP sequence for each benchmark that 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.

Configuration Caching Algorithms

In this work, we seek to find caching methods that target the different the 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 the complete application information. For the models where tight lower bound algorithms are unavailable, we will develop algorithms that we believe could be the near optimal solution.

We divide our algorithms into 3 categories: run time algorithms, complete prediction algorithms, and general off-line algorithms. The classification of the algorithms depends on the time complexity and input information needed for each algorithm. The run time algorithms require the information available only at run time. 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 upper bound on reconfiguration overhead. The complete prediction algorithms use complete execution information of the application. These algorithms attempt to search the whole application domain in order to lower the configuration overhead. We believe these algorithms provide the optimal (lower bound) or near optimal solution. Among these algorithms, some are realistic but with very high computation complexity, while others just provide a realistically unachievable bound. The general off-line algorithms use profile information of each application and provide realistic solutions for the caching models.

Simulated Annealing Algorithm for the 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 possibly one or more later required RFUOPs is loaded. This amortizes the configuration time over all of the configurations grouped into a context. Minimizing the number of the group (context) loading will minimize the overall reconfiguration overhead.

It is obvious that the method used for grouping has a great impact on the latency reduction; the overall reconfiguration overhead resulting 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 configurations 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 present a simulated annealing approach to acquire near optimal solution. For the simulated annealing algorithm, we use the reconfiguration overhead 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 RFUOP sequence. The steps below outline the complete algorithm:

Initially assign each RFUOP to a group such that for each group the total size of all RFUOPs is smaller than or equal to the size of the context. Set up parameters of initial temperature, the number of iterations under each temperature.

1. While the current temperature is greater than the terminating temperature:

   1.1. While the number of iterations is greater than 0:

      1.1.1. A candidate RFUOP is randomly chosen along with a randomly selected destination group to which the candidate will be moved.

      1.1.2. After the move, if the total size of the RFUOPs in the destination group exceeds the size of the context, a new candidate RFUOP in the destination group is randomly selected. This RFUOP is then moved to any group that can hold it. This step is repeated until all groups satisfy the size constraint.

      1.1.3. Execute the newly generated grouping on the RFUOP execution sequence and calculate the number of times reconfiguration is performed. The reconfiguration overhead, which is used as the cost function of this version of simulated annealing, can be calculated by multiplying the number of context switches by the loading latency of a context.

      1.1.4. Compare the new cost to the old cost to determine if the move is allowed, then decrease the number of iterations by one.

   1.2. Decrease the current temperature.

Since configuration caching for the Single Context FPGA is similar to the placement problem in CAD applications, the simulated annealing algorithm will provide near optimal solution.

General Off-line Algorithm for the Single Context FPGA

Although the simulated annealing approach can generate a near optimal solution, the high computation complexity and the requirement of knowledge of the exact execution sequences make this solution unreasonable for most real applications. We therefore suggest 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 there exists any consecutive appearance of A and B.

For the Single Context FPGA, the 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. The details of our grouping algorithm are as follows:

1. Create an *N ´ N* matrix, where *N* is the number of RFUOPs. All values of *A[i, j]* are initialized to 0, where *0 £ i, j £ N-1*.

2. Every RFUOP is assigned to a separate group, and thus initially *N* groups are formed.

3. Traverse the RFUOP sequence. If RFUOP i correlates to RFUOP j, increase *A[i, j]* by 1

4. While the largest *A[i, j]* > 0, do the following:

   4.1. Search the matrix to find two groups *i, j* such that value *A[i, j]* + *A[j, i]* is the largest in the matrix. If total size of the RFUOPs in the two groups is less than the size of the context, merge two the groups together, then go to step 4.2, else go to step 4.3.

4.2. For any other group k, set A[i, k] and A[k i] to A[k i] + A[k j], and A[j, k] and A[k, j] = 0;

4.3. For any RFUOP *i* in group 1 and any RFUOP *j* in group 2, *A[i, j]* and *A[j, i]* are assigned the new value 0.

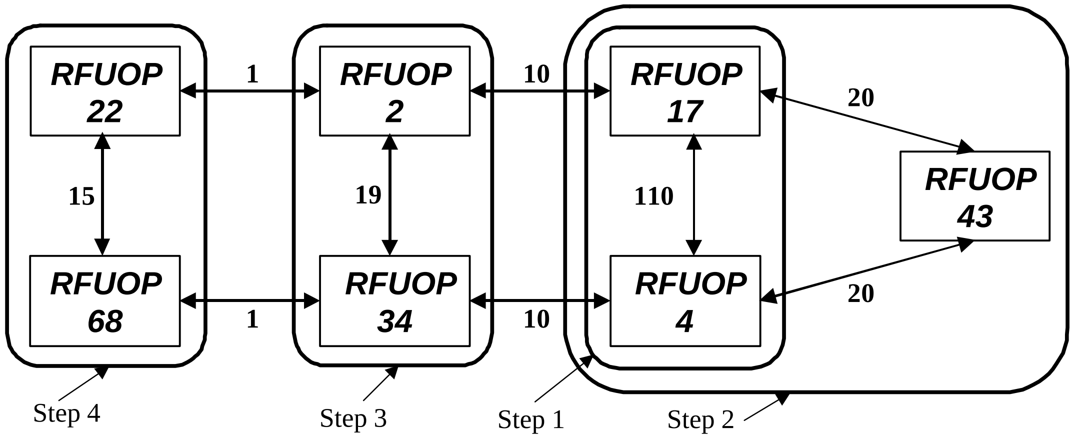

Figure 3: An example to illustrate the general off-line algorithm for Single Context FPGA.

Figure 3 illustrates an example of the general off-line algorithm. Each arrow 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. 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. 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.

Compared to the simulated annealing algorithm, this realistic 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 of the matrix instead of traversing the RFUOP sequence as the simulated annealing algorithm does. Therefore, the computation complexity is significantly lowered.

A Complete Prediction Algorithm for the 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 could be replaced once a demanded configuration is not in any of the contexts currently present on the chip. 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 tactics that worked for the general cache for the Multi-Context FPGA. More specifically, we have an existing optimal replacement algorithm called the Belady [Belady66] algorithm for the Multi-Context FPGA. The Belady [Belady66] algorithm is well known in the operating systems and computer architecture fields. It claims that the fewest number of replacements will occur 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. Since the RFUOP sequence is known, it is trivial to create the context access string by transforming the RFUOP sequence.

Combining the two key factors of grouping and replacement mentioned above, we present our algorithm as follows:

1. Apply the Single Context simulated annealing 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 result a context access sequence.
3. Apply the Belady algorithm to the context access string. Increase the total number of context loads by one if a replacement occurs.

The reconfiguration overhead for a Multi-Context FPGA is therefore the number of context loads multiplied by the configuration latency for a single context. We must also consider the context activation overhead that is incurred by each context switch. A context switch occurs when a demanded context is not currently in execution, but is present on one of the inactive contexts. Generally, the cost of a single switching operation is negligible compared to a single context load. However, the overall switching cost still must be calculated, given that the total number of switching operations could be much larger than that of the context loads. This switching cost can be calculated as the cost of a single switching operation multiplied by the number of switching operations. Considering the overhead caused by the context switches, we modify the third step of the algorithm as following:

3. Apply the Belady algorithm to the context access string. Increase the total number of context loads by one if a replacement occurs. Increase the total number of switching operations by one if the accessed context is not the active 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 first step of the algorithm can result in a near optimal grouping and the third step represents the optimal replacement policy, we consider this algorithm to be a lower bound on what is possible with a realistic algorithm. In addition, since the RFUOP sequence is contained in the profile information, we consider this algorithm to be a general off-line algorithm.

Least Recently Used (LRU) Algorithm for the Multi-Context FPGA

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 easier to implement than the Belady algorithm, but because of the limited amount of information used to make a decision, the reconfiguration overhead incurred through the use of the LRU is higher than that of the Belady algorithm. The LRU is a run-time algorithm.

Algorithms for the Partial Run Time Reconfigurable FPGA

Compared to the Single Context FPGA, an advantage of the Partial Run-Time Reconfigurable FPGA is its 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, which could be caused by the overlap of frequently used RFUOPs. 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 Simulated Annealing Algorithm for the PRTR FPGA

Similarly 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 optimal 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. Before presenting the full 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. Assign a random position for each RFUOP. Set up the parameters of initial temperature, number of iterations under each temperature, and terminating temperature. The parameter of old cost is initially set to be infinity.
2. While the current temperature is greater than the terminating temperature:
   - 2.1. While the number of iterations is greater than 0:
     - 2.1.1. A randomly selected RFUOP is moved to a random location within the chip.
     - 2.1.2. Traverse the RFUOP sequence. If the demanded RFUOP is not currently on the chip, load the RFUOP to the specified location, and increase the overall reconfiguration latency by the loading latency of the RFUOP. If the newly loaded RFUOP conflicts with any other RFUOPs on the chip, those conflicted RFUOPs are removed from the chip.
     - 2.1.3. Let the new cost be equal to the overall RFUOP overhead and determine whether the move is allowed. Decrease the number of iteration by one.
   - 2.2. Decrease the current temperature.

Since finding the location for each RFUOP is similar to the placement problem in physical design, where the simulated annealing algorithm usually provides good performance, so we believe our simulated annealing algorithm will create a near optimal solution.

An Alternate Simulated Annealing Algorithm for the PRTR FPGA

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. Obviously, a better algorithm is needed to reduce the computation complexity. Again, as for the Single Context FPGA, an adjacency matrix of size $N \times N$ is built, where N is the number of the RFUOPs. The main purpose of the matrix is to record the possible conflicts between RFUOPs. In order to reduce the reconfiguration overhead, the conflicts that will create larger configuration loading latency are distributed to unoverlapped locations. This is done by modifying the cost computation step of the previous algorithm. To clarify, we present the full algorithm:

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. Set up parameters of initial temperature, the number of iterations under each temperature, and terminating temperature. The parameter of old cost is set to be infinity. An $N \times N$ adjacency matrix B is created. All values of $B[i, j]$ are set to be 0, where $0 \le i, j \le N-1$.
4. While the current temperature is greater than the terminating temperature:
   - 4.1. While the number of iterations is greater than 0:
     - 4.1.1. 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.
     - 4.1.2. 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.
     - 4.1.3. Determine whether the new move is allowed and decrease the number of iterations by one.
   - 4.2. Decrease the current temperature.

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.

Still, one final concern is the quality of the algorithm because, instead of using configuration sequence, the matrix of potential conflicts derived from the sequence is used. Even the matrix may not represent the conflicts exactly, however, it gives enough information about the potential overall conflicts between any two configurations.

Algorithms for the PRTR FPGA with Relocation and Relocation + Defragmentation

For the PRTR FPGA with Relocation + Defragmentation, the replacement policies have a great impact on reducing reconfiguration overhead. This is because the high flexibility available for 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, even though the empty space exists in the way of many small portions. Instead of giving the algorithms for PRTR FPGA with only Relocation, we first analyze the case of PRTR with both Relocation and Defragmentation.

A Lower Bound Algorithm for the PRTR FPGA with Relocation + Defragmentation

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 a configuration is proportional to the size of the configuration, given fixed a configuration bandwidth.

The Belady [Belady66] algorithm gives the optimal replacement for the case that the memory access string is known and the data transfer unit is uniform. Given the RFUOP sequence for the PRTR with Relocation + defragmentation 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 is replaced while the other portion is still kept on the chip. Once the removed RFUOP is needed again, only the missing portion (could be the whole RFUOP) is loaded instead of loading the entire RFUOP. We present the Lower Bound Algorithm as follows:

1. Traverse the RFUOP sequence, if an RFUOP required is not on the chip, do following:

   1.1. Find the missing portion of the RFUOP. While the missing portion is greater than the free space on the chip, do 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 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.

The correctness of the algorithm is proven by the following theorem.

Theorem: Given RFUOPs $R_1, R_2, R_3, \ldots, R_m$, with sizes of $S_1, S_2, S_3, \ldots, S_m$ atomic configuration units, and RFUOP sequence $C_1, C_2, C_3, \ldots, C_n$, the Lower Bound Algorithm provides a lower bound for the PRTR with Relocation + Defragmentation model.

Proof: We can transform the RFUOP sequence $C_1, C_2, C_3, \ldots, C_n$ to $(R_{i1}, S_{i1}), (R_{i2}, S_{i2}), (R_{i3}, S_{i3}), \ldots, (R_{in}, S_{in})$, where R_{ij} belongs to $(R_1, R_2, R_3, \ldots, R_m)$ and S_{ij} belongs to $(S_1, S_2, S_3, \ldots, S_m)$. We will then further extend this sequence to:

$$\underbrace{R_{i1}, R_{i1}, \cdots, R_{i1}}_{S_{i1}}, \underbrace{R_{i2}, R_{i2}, \cdots, R_{i2}}_{S_{i2}}, \underbrace{R_{i3}, R_{i3}, \cdots, R_{i3}}_{S_{i3}}, \cdots, \underbrace{R_{in}, R_{in}, \cdots, R_{in}}_{Sin}$$

Now the size of every R_{ij} in the sequence is equal to the atomic configuration unit.

In our algorithm, we assumed that 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 [Belady66] algorithm provides the optimal replacement for the general caching problem, it can also provide the lowest configuration overhead for the PRTR with Relocation + Defragmentation.

A General Off-line Algorithm for the PRTR FPGA with Relocation + Defragmentation

Since the Belady [Belady66] algorithm can provide a lower bound for the fixed size problem, some ideas can be transferred into our algorithm. As in the Belady [Belady66] algorithm, for all RFUOPs that are currently on chip, we identify the one that will not appear in the RFUOP sequence until others have appeared. But instead of replacing that RFUOP, as in the Belady [Belady66] algorithm, the victim configuration is selected by considering the factors of size and loading latency. The details of the algorithm are as follows:

1. Traverse the RFUOP sequence. 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. For all RFUOPs on chip, find their next appearances. Among these appearances, find the one furthest in the future.

      1.1.2. For each RFUOP, calculate the total number of appearances between the current appearance and the furthest appearance identified.

      1.1.3. For each RFUOP, multiply the loading latency and the number of appearances, the RFU with the smallest value is replaced.

   1.2. Load the demanded RFUOP. Increase the overall latency by the loading latency of the RFUOP.

The steps 1.1.1 – 1.1.4 specify the rules to select the victim RFUOP. Counting the number of appearances of each RFUOP is to get the frequency of the RFUOP to be used in near future. As we mentioned, this is not adequate to determine a victim RFUOP, because an RFUOP with lower frequency may have much higher configuration latency. Therefore, by multiplying the latency and the frequency, we can find the possible overall latency in near future if the RFUOP is replaced. Moreover, by considering the size factor, we choose the victim configuration that has the smallest latency to size ratio.

Run Time Algorithms for the PRTR FPGA with Relocation + Defragmentation

In order to further evaluate the performance of the PRTR with Relocation + Defragmentation FPGA model two real-time algorithms, a LRU algorithm and a penalty oriented [Young94] algorithm, are implemented. These two algorithms take the RFUOP sequence as the input at run time and do not need future information.

LRU Algorithm for the PRTR FPGA with Relocation + Defragmentation

Since the PRTR with Relocation plus Defragmentation 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 LRU algorithm.

Penalty Oriented Algorithm for the PRTR FPGA with Relocation + Defragmentation

Since the non-uniform size of RFUOPs is not considered as a factor in LRU algorithm, a high RFUOP overhead could potentially 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 same order of the RFUOP sequence. It is obvious that 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 factors should be considered in the algorithm. Therefore, we use a variable "credit" to determine the victim. Every time an RFUOP is loaded onto the chip, its credit is set to its size. When a replacement occurs, the RFUOP with the smallest credit is evicted from the chip and the credit of all other RFUOPs on the chip is decreased by the credit of the victim. To make this more clear, we present the algorithm as follows:

1. Traverse the RFUOP sequence. 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.

A General Algorithm for the PRTR FPGA with Relocation

One major advantage that the PRTR with Relocation + Defragmentation has over the PRTR with Relocation 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 could be replaced and thus may result in extra overhead if the replaced RFUOP is demanded again very soon. In order to reduce the reconfiguration overhead for this model, the utilization of the fragments must be improved. We present the algorithm as following:

1. Traverse the configuration sequence. If a demanded configuration 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. For all RFUOPs on chip, find their next appearances. Among these appearances, find the furthest one from the current demanded RFUOP.

        1.1.2. For each RFUOP, calculate the total number of appearances between the demanded RFUOP and the furthest appearance identified on last step.

        1.1.3. For each RFUOP, multiply the loading latency and the number of appearances. Then divide this value by the size of the RFUOP, producing a cost.

        1.1.4. For each RFUOP on chip, presume that it to be the candidate victim, identify the next 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 the sum of each RFUOP, and victims 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 heuristic that applied to the PRTR with Relocation + Defragmentation is also implemented in this algorithm. The major difference for this algorithm is to use as many empty fragments as possible, reducing the reconfiguration overhead by replacing either fewer configurations or smaller configurations.

Simulation results

All algorithms are implemented in C++ on a Sun Sparc-20 workstation. Table 1 shows reconfiguration overhead of the algorithms on the benchmarks. The definition of “size” indicates capacity of the chip. “1” represents the base capacity, which is the least multiple of ten larger than the size of the largest RFUOP. The other values for size represent the ratio of the FPGA size to the base capacity. Figure 4 demonstrates the reconfiguration overhead vs. FPGA size. For each benchmark, we first normalize the reconfiguration penalty for each algorithm, then we calculate the average for each algorithm. As can be seen in Figure 4, the reconfiguration penalties of the PRTR and the Multi-Context models are about 75% and 85% less than that of the Single Context model. For the two new models we discussed, PRTR with Relocation and the PRTR with Relocation + Defragmentation, the reconfiguration penalties are about 85% and 90% less than that of the Single Context model.

Bench	Size	Single Context		Partial		Partial Reloc	Partial Relocation + Defragmentation				Multi-Context	
		Simulate A	General	Simulate I	Simulate II	General	Optimal	General	LRU	Penalty	General	LRU
Go	1	442560	461760	109652	121301	82854	64595	78542	123906	106554	90400	162400
	1.25	364800	379200	90654	94396	65599	46330	56517	117232	91432	80500	161500
	1.5	406080	400320	69179	79472	46242	30139	36710	107654	69845	72000	76800
	1.75	423360	423360	66066	30876	29264	16576	21652	84636	43399	29400	56300
	2	291840	288000	45666	29272	15580	7776	9720	31232	20866	4000	4800
Perl	1	10327680	11340960	2333743	2659040	2114457	1750762	2279647	2565527	2642261	2481500	3673700

	1.25	11946000	16258200	2145665	2345660	1927136	1509741	1917227	2447775	2455968	2464250	3352875
	1.5	11143500	15756480	2245434	2285316	1641843	1294499	1688077	2266006	2127938	2430000	3329700
	1.75	13500900	18189780	1967722	2095221	1438792	1113794	1465085	2141724	1901806	1945125	2989875
	2	9821760	13660320	1973449	2030283	1222860	944679	1207741	1938961	1756577	2350600	3960400
Fpppp	1	9992500	10294500	9534595	9672211	6695022	4305660	5619275	10411182	8443375	4161500	7781500
	1.25	11721250	14503120	5427786	6392619	4017768	2271989	3083254	8967020	5968650	1552500	2786875
	1.5	7385250	7956000	3098876	3265848	2098129	1073299	1369862	4084820	2919774	1168500	2251500
	1.75	5512500	5642000	1700895	1478998	967818	553594	785300	1746796	1503056	321125	571375
	2	4134000	4297000	1126934	945945	603276	290630	496440	1695305	931576	113000	126000
compress	1	219120	237360	83409	91152	56500	26127	32537	40358	37637	25600	38300
	1.25	239700	264900	30879	34095	56816	21377	24788	27644	34978	32000	47875
	1.5	235440	300600	28964	31586	50540	18321	21835	27480	37224	23100	42300
	1.75	67620	64260	27054	27497	48102	15321	15656	27264	18938	18375	40075
	2	42240	49440	20381	24436	43739	12324	13479	27171	21475	21000	41300
li	1	3861360	4382640	220403	273902	213907	126131	144793	358178	366564	421700	559400
	1.25	2949399	2288700	90977	125673	92936	47001	57871	143309	150583	54375	54625
	1.5	675720	783720	54862	63955	47754	7881	11063	54671	64650	63150	66300
	1.75	388800	459900	27636	33544	13525	2274	2369	6616	5373	70175	73675
	2	328800	360480	1203	2755	1521	852	874	1317	1319	1400	1400
Perl.garp	1	42840	51240	2028	2332	1594	1570	1570	1707	2119	3150	4550
	1.25	28323	34617	1694	1783	1570	1570	1570	1683	1912	2622	2622
	1.5	22680	26460	1638	1646	1570	1570	1570	1659	1672	2100	2100
	1.75	4407	4407	1594	1594	1570	1570	1570	1639	1646	1836	1836
	2	1680	1680	1590	1590	1570	1570	1570	1570	1570	2100	2100
Dss.garp	1	26400	26400	18766	19083	10674	4247	10155	14928	15298	160	160
	1.25	7320	7200	15436	16097	9799	860	1759	3939	3646	200	200
	1.5	135	135	135	135	135	135	135	135	135	180	180
	1.75	135	135	135	135	135	135	135	135	135	140	140
	2	135	135	135	135	135	135	135	135	135	160	160
Go.garp	1	9516000	10299840	604598	721357	204573	162659	199311	205791	227166	421800	549600
	1.25	8338800	10061400	601344	681920	185089	151097	182584	193556	212051	476500	613250
	1.5	8780400	10764720	554567	661089	172023	140516	167725	187284	199239	514200	663600
	1.75	9227400	12056520	543013	631345	161257	130740	154438	181633	187239	528850	685300
	2	7541760	13257600	498762	571709	149992	121485	144812	178732	179336	471600	602400

Table 1: Reconfiguration Penalties of different algorithms.

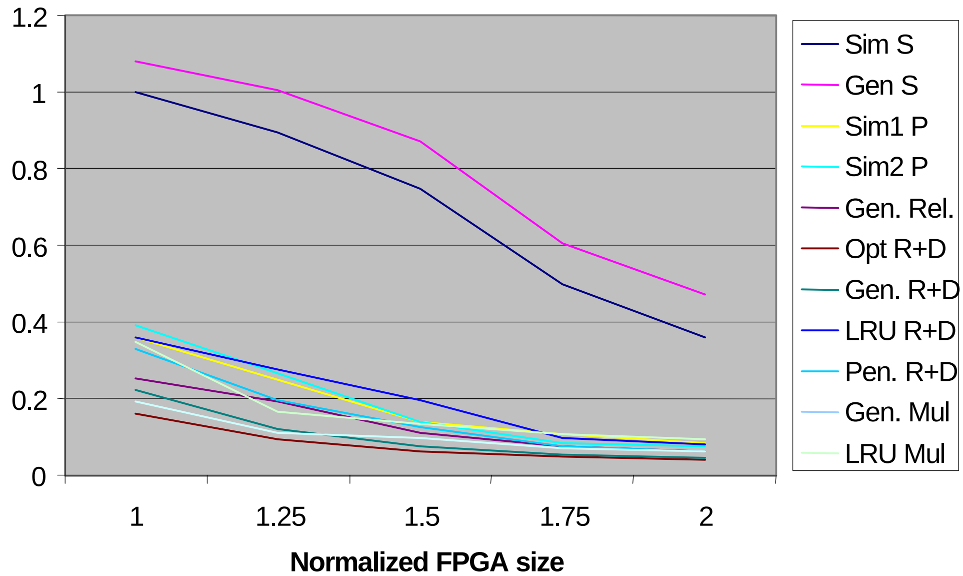

Figure 4: Reconfiguration penalties of different caching algorithms.

Conclusions

Configuration caching, which configurations are retained on chip until they are required again, is a technique to reduce the reconfiguration overhead. However, the limit the on-chip 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. To deal these problems, we have developed new caching algorithms targeted at a number of different FPGA models. 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 PRTR with Relocation model and the PRTR with Relocation + Defragmentation model. For each model, 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:

Row = number of rows of configuration bits

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

Single Context: $291264 \times \text{Row} \times \text{Col}$

PRTR: $260336 \times \text{Row} \times \text{Col} + 476 \times \text{Row} + 392 \times \text{Row} \times \lg \text{Row} + 367217.5 \times \text{Col} + 487.5 \times \text{Col} \times \lg \text{Col}$

Multi-Context (4 contexts): $636848 \times \text{Row} \times \text{Col} + 476 \times \text{Row} + 392 \times \text{Row} \times \lg \text{Row} + 385937.5 \times \text{Col} + 487.5 \times \text{Col} \times \lg \text{Col}$

PRTR Relocation: $260336 \times \text{Row} \times \text{Col} + 476 \times \text{Row} + 392 \times \text{Row} \times \lg \text{Row} + 367217.5 \times \text{Col} + 487.5 \times \text{Col} \times \lg \text{Col} + 20300 \times \lg \text{Row}$

PRTR Relocation & Defragmentation: $260336 \times \text{Row} \times \text{Col} + 476 \times \text{Row} + 392 \times \text{Row} \times \lg \text{Row} + 407404 \times \text{Col}$

$+ 392 \times \text{Col} \times \lg \text{Col} + 365040 + 30186 \times \lg \text{Row}$

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 Context: 9.544×10^9 lambda^2

PRTR: 8.547×10^9 lambda^2

Multi-Context (4 contexts): 20.885×10^9 lambda^2

PRTR w/ Relocation: 8.547×10^9 lambda^2

PRTR w/ Relocation & Defragmentation: 8.549×10^9 lambda^2

References

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

[Compton00] K. Compton, J. Cooley, S. Knol, S. Hauck, “Configuration Relocation and Defragmentation for FPGAs”, in preparation for *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000.

[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.

[Hauck98] S. Hauck, Z. Li, E. J. Schwabe, “Configuration Compression for the Xilinx XC6200 FPGA”, *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 138-146, 1998.

[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