Configuration Prefetching Techniques for Partial Reconfigurable Coprocessor with Relocation and Defragmentation

Zhiyuan Li
Department of Electrical and Computer Engineering
Northwestern University
Evanston, IL 60208-3118 USA
zl@ee.washington.edu

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

Abstract
One of the major overheads for reconfigurable computing is the time it takes to reconfigure the devices in the system. This overhead limits the speedup possible in this paradigm. In this paper we explore configuration prefetching techniques for reducing this overhead. By overlapping the configuration loadings with the computation on the host processor the reconfiguration overhead can be reduced. Our prefetching techniques target to the reconfigurable systems containing a Partial Reconfigurable FPGA with Relocation + Defragmentation (R+D model) since the R+D FPGA showed high hardware utilization. We have investigated various techniques including static configuration prefetching, dynamic configuration prefetching, and hybrid prefetching. We have developed prefetching algorithms that significantly reduce the reconfiguration overhead.

1. INTRODUCTION
In recent years, reconfigurable computing systems such as Garp [Hauser97], OneChip [Wittig96], PipeRench [Schmit97], and Chimaera [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. Reconfiguration overhead becomes a major concern because the systems must sit idle during reconfiguration. By reducing the overall reconfiguration overhead, the performance of the system can be improved. Therefore, many studies have involved investigating techniques to reduce the configuration overhead. Some of these techniques include configuration caching [Li00] and configuration compression [Li99, Li01, Dandilas01].

FPGA'02, February 24-26, 2002, Monterey, California, USA.

Another interesting technique that effectively reduces this overhead is configuration prefetching [Hauck98]. The basic idea of configuration prefetching, similar to that of prefetching in general purpose computer system, is to overlap the configuration loading with computation. Targeted at the Single Context FPGA model, [Hauck98] described an algorithm that can reduce the reconfiguration overhead by a factor of 2.

As technology moves forward more advanced devices and systems, such as Xilinx Virtex families, Xilinx XC6200 families, Garp [Hauser 97], and Chimaera [Hauck97], can be partially reconfigured at run time. For a system containing a Partial Run-Time Reconfigurable device, a configuration can be loaded into part of the device while the rest of the system continues computing. Compared with the Single Context FPGA, the Partial Run-Time Reconfigurable devices provide greater flexibility and higher hardware utilization. Based on the Partial Run-Time Reconfigurable model, a new model called Relocation + Defragmentation [Compton00] is built to further improve the hardware utilization. The relocation allows the final placement of a configuration within the FPGA to be determined at run-time, while defragmentation provides a method to consolidate unused area within an FPGA during run-time without unloading useful configurations. The configuration caching techniques [Li00] applied on the Relocation + Defragmentation model (Partial R+D) have demonstrated a significant reduction in reconfiguration overhead. Note that this overhead reduction was based on the demand fetch of configurations, meaning that effective prefetching approaches will further reduce this overhead.

2. PARTIAL R+D FPGA
Similarly to the partially reconfigurable FPGA, the memory array of the Partial R+D FPGA is composed of an array of SRAM bits. These bits are read/write enabled by the decoded row address for the programming data. However, instead of using a column decoder, a SRAM buffer called a "staging area" is built. This buffer is essentially a set of memory cells equal in number to one row of programming bits in the FPGA memory array at the row location indicated by the row address. To configure the chip every row of a configuration is loaded into the staging area and then transferred to the array. By providing the run-time-determined row address to the row decoder rows of a configuration can be relocated to locations specified by the system.

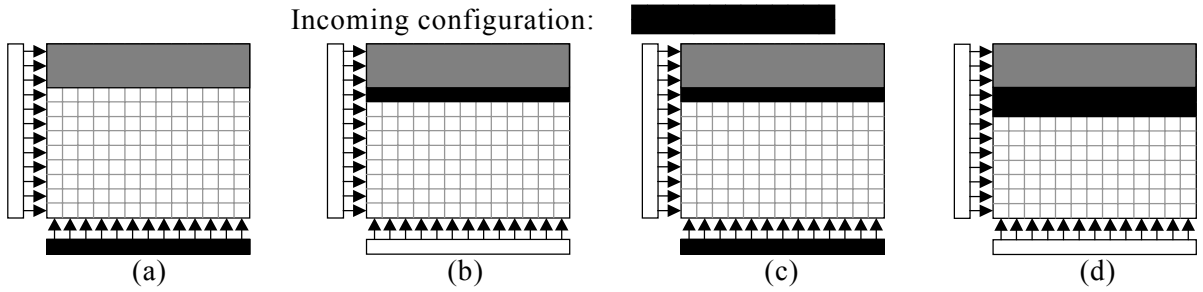

Figure 1. An example of configuration Relocation. The incoming configuration contains 2 rows. The first row is loaded into staging area (a) and then transferred to the desired location that is determined at run-time (b). Then the second row of the incoming configuration is loaded to the staging area (c) and transferred to the array (d).

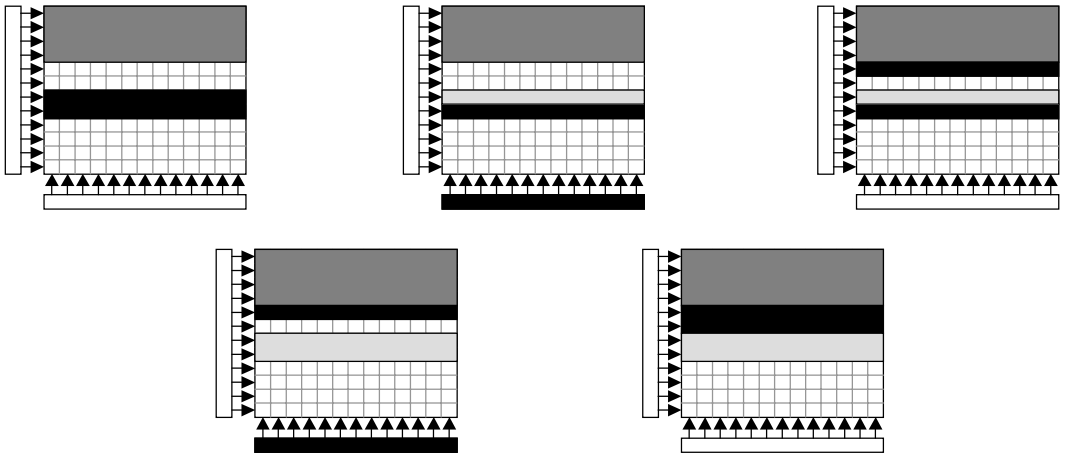

Figure 2. An example of Defragmentation. By moving the rows in a top-down fashion into the staging area and then moving upwards into the array, the smaller fragments were collected.

Figure 1 shows the steps of relocating a configuration into the array. Defragmentation operation is slightly more complicated than a simple relocation operation. In order to collect the fragments within that array each row of a particular configuration is read back into the staging area and then moved to a new location in the array. Figure 2 demonstrates the steps of a defragmentation operation. It has been demonstrated in [Compton00] that with a very minor area increase over standard architectures the Relocation + Defragmentation model has a considerably lower reconfiguration overhead than the Partial Run-Time Reconfigurable model.

3. PREFETCHING OVERVIEW

As demonstrated in [Li00], an FPGA can be viewed as a cache of configurations. Prefetching configurations on an FPGA, which is similar to prefetching in a general memory system, overlaps the reconfigurations with computation to hide the reconfiguration latency. Before we will discuss the details for the configuration prefetching, we first reexamine the factors very important to the effectiveness of the prefetching for a general purpose system:

1) *Accuracy*. This is the ratio of the executed prefetched instructions or data to the overall prefetched instructions or data.

2) *Coverage*. This is the fraction of cache misses eliminated by the effectiveness of a prefetching technique. An accurate prefetch technique will not significantly reduce the latency without a high coverage.

3) *Pollution*. One side-effect of prefetching techniques produce is the cache lines that would have been used in the future will be replaced by some prefetched instructions or data that may not be used. This is known as cache pollution.

These issues are even more critical to the performance of configuration prefetching. In general purpose systems the atomic data transfer unit is cache block. Cache studies consistently showed that the average access time will likely to drop when the block size increases until it reaches a certain value (usually fewer than 128 bytes), then the access time will increases as the block size continue to increase since a very large block will result an enormous penalty for every cache miss. The atomic

data transfer unit in the configuration caching or configuration prefetching domain, rather than a block, is the configuration itself, which normally is significantly larger than a block. Therefore the system suffers severely if a demanded configuration is not present on chip. In order for a system to minimize this huge latency accurate prediction of the next required configurations is highly desired.

As bad as it could be in general memory systems, cache pollution plays a much more malicious role in the configuration prefetchng domain. As demonstrated in [Li00], due to the large configuration size and relatively small on-chip memory, very few configurations can be stored on chip. As the result, a wrong prefech will be very likely to cause a required configuration to be replaced and a significant overhead will be generated when the required configuration is brought back later. Thus, rather than reducing the overall configuration overhead a poor prefetching approach can actually significantly increase this overhead.

In general, prefetching algorithms can be divided into 3 categories: static prefetching, dynamic prefetching and hybrid prefetching. A compiler-controlled approach, static prefetching inserts prefetch instructions after performing control flow or data flow analysis based on profile information and data access patterns. One major advantage of static prefetching is that it requires no additional hardware. However, since a significant amount of access information is unknown at compile time, the static approach is limited by the lack of run-time data access information. Dynamic prefetching determines and dispatches prefetches at run-time without compiler intervention. With the help of the extra hardware, dynamic prefetching uses more data access information to make accurate predictions. Hybrid prefetching tries to combine the strong points of both approaches—it utilizes both compile-time and run-time information to become a more accurate and efficient approach.

4. PREFETCHING FACTORS

In order to better discuss the factors that will affect the prefeching performance, we first make following definitions.

L_k: The latency of loading a configuration k.
S_k: The size of the configuration k.
D_{ik}: The distance between an operation i (can be either a normal instruction or a prefetch operation) and the execution of the configuration k.
P_{ik}: The probability of i to reach the configuration k.
C: The capacity of the chip.

The probability factor could have a significant impact on the prefetching accuracy. The combination of S_k and C determines whether there is enough space on the chip to load the configuration k. The combination of L_k and D_{ik} determines that the amount of the latency of the configuration k can be hidden if it is prefetched at i. It is obvious that there will not be a significant reduction if the D_{ik} is too short because most of the latency of the configuration k cannot be eliminated if it is prefetched at i.

In addition, the non-uniform configuration latency will affect the order of the prefetches that need to be performed. Specifically, we might want to perform out-of-order prefetch (prefetch configuration j before configuration k even if k is required earlier than j) for some situations. For example, suppose we have 3 configurations *1, 2*, and *3* to be executed in that order. Given that $S_3 >> S_1 >> S_2$, $S_1 + S_3 < C < S_1 + S_2 + S_3$, and $D_{12} >> L_3 >> L_2 > D_{23}$, prefetching the configration *3* before the configuration *2* when *1* is executed results in a penalty of L_2 at most. This is because the latency of the configuration *3* can be completely hidden and the configuration *2* can be either demanded fetched (penalty of L_2) or prefetched once the execution of the configuration *1* completes. However, if the in-order prefetches are performed the overall penalty is calculated as $L_3 - D_{23}$, which is much larger than L_2.

5. CONFIGURATION PREFETCHING

Most current reconfigurable computing architectures consist of a FPGA connected, either loosely or tightly, to a host microprocessor. In these systems the microprocessor will act as controller for the computation, and will perform computations which are more efficiently handled outside the FPGA logic. The FPGA will perform multiple, regular operations (contained in FPGA configurations) during the execution of the computation. In normal operation the processor executes until a call to the reconfigurable coprocessor is found. These calls (RFUOPs) contain the ID of the configuration required to compute the desired function. In this work, we target systems containing Partial R+D FPGAs. In this work we seek to find efficient prefetching techniques for such systems. We have developed algorithms applying the different configuration prefetching techniques. Based on the available access information and the additional hardware required, our configuration caching algorithms can be divided into 3 categories: Static Configuration Prefetching, Dynamic Configuration Prefetching, and Hybrid Configuration Prefetching. Before presenting the details of the algorithms, we first discuss the experimental setup.

5.1. Experimental Setup

In order to evaluate the different configuration prefetching algorithms we must perform the following steps. First, some method must be developed to choose which portions of the software algorithms should be mapped to the reconfigurable coprocessor. In this work, we apply the approach presented in [Hauck98] (these mappings of the portions of the source code will be referred to as RFUOPs). Second, a simulator of the reconfigurable system must be employed to measure the performance of the prefetching algorithms. Our simulator is developed from SHADE [Cmelik93]. It allows us to track the cycle-by-cycle operation of the system, and get exact cycle counts. We will compare the performance of the prefetching algorithms as well as the performance assuming no prefetch occurs at all.

5.2. Static Configuration Prefetching

Similar to the prefetching approach used in [Hauck98], a program running on this reconfigurable system can insert prefetch operations into the code executed on the host processor. However, the system described in [Hauck98] contains a Single Context FPGA rather than a Partial R+D FPGA. The prefetch instructions for systems containing a Single Context FPGA are executed just like any other instructions, occupying a single slot in the processor's pipeline. The prefetch instruction specifies the ID of a specific configuration that should be loaded into the coprocessor. If the desired configuration is already loaded, or is

in the process of being loaded by some other prefetch instruction, this prefetch instruction becomes a NO-OP. Since a Single Context FPGA can only hold one configuration, at any given point only one configuration needs to be prefetched. This greatly simplifies prefetches in these systems.

Unlike the Single Context FPGA a Partial R+D FPGA can hold more than one configuration, and at a given point multiple RFUOPs may need to be prefetched. Thus, the method to effectively specify the IDs of the RFUOPs to prefetch becomes an issue. One intuitive approach is to pack the IDs into one single instruction. However, since the number of IDs need to be specified could be different for each prefetch instruction, it is not possible to generate prefetch instructions with equal length. Another option is to use a sequence of prefetch instructions when multiple prefetching operations need to be performed. However, to make it an effective approach a method that can terminate previously issued prefetches is required. This is because during the execution certain previous unfinished prefetch instructions may become obsolete and these unwanted prefetching operations will significantly harm performance. In Figure 3 for example, 1, 2, 3, and 4 are RFUOPs and P1, P2, P3, and P4 are prefetching instructions. It is obvious that when P1 is executed, configurations 3 and 4 will not be reached, therefore the prefetches of 3 and 4 are wasted. This waste may be negligible for a general purpose system since the load latency of an instruction or a data block is very small. However, because of the large configuration latency in reconfigurable systems, it is likely the prefetch of configuration 3 has not completed or even not started when P1 is reached. As a consequence, if we use the same approach used in general purpose systems, letting the prefetches of P3 and P4 complete before prefetching P1 and P2, the effectiveness of the prefetches of P1 and P2 will be severely damaged since they cannot completely or mostly hide the load latencies of configurations 1 and 2. Therefore, we must find a way to terminate previously issued prefetches if they become unwanted.

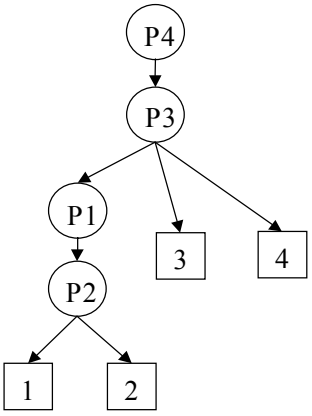

Figure 3. Example of prefetching operation control.

A simple approach we used in this work to solve this problem is to insert termination instructions when necessary. The format of a termination instruction is just like any other instructions, consuming a single slot in the processor's pipeline. Once a termination instruction is encountered the processor will terminate all previously issued prefeches so the new prefetches can start immediately. For the example in **Figure 3**, a termination instruction will be inserted immediately before P1 to eliminate the unwanted prefetches of P4 and P3.

Now we have demonstrated the way to handle the prefetches, the remaining problem is to determine where the prefetch instructions will be placed given the RFUOPs and the control flow graph of the application. Since the algorithm used in [Hauck98] demonstrates good-quality results for the Single Context reconfigurable systems, we will extend it for systems containing a Partial R+D.

The algorithm we used to determine the prefetches contains 3 stages:

1) *Penalty calculation*. In this stage the algorithm will compute the potential penalties for a set of prefetches at each instruction node of the control flow graph.
2) *Prefetch scheduling and generation*. In this stage the algorithm will determine the configurations that need to be prefetched at each instruction node based on the penalties calculated in stage 1. Prefetches will be generated under the restriction of the size of the chip.
3) *Prefetch reduction*. In this stage the algorithm will trim the redundant prefetches generated in the previous stage. In addition, the termination instructions are inserted.

In order to make our analysis clearer in the control flow graph, we will use circles to represent the instruction nodes and squarse to represent the RFUOPs. Since a prefetch should be executed at the top node of a single entry and single exit path, but not other nodes contained in the path, only the top node is considered as the candidate where prefetch instructions can be inserted. Therefore, we simplify the control graph by packing other nodes in the path. In the following sections we will discuss the details for every stage.

5.2.1. Penalty Calculation

In [Hauck98] a bottom-up approach is applied to calculate the penalties using the probability and distance factors. Since the probability is dominant in deciding the penalties, and the average prefetching distance is mostly greater than the latencies of RFUOPs, it is adequate to use probability to represent penalty. Note that control flow graph can be complex with nested loops, the bottom-up algorithm cannot be performed without loop detection and conversion. In this work we apply the loop detection and conversion algorithm presented in [Hauck98].

Given the simplified control flow graph and the branch probabilities, we will use a bottom-up approach to calculate the potential probabilities for an instruction node to reach a set of RFUOPs. The basic steps of the bottom-up algorithm are outlined below:

1. For each instruction, initialize the probability of each reachable configuration to 0, and set the num_children_searched to be 0.
2. Set the probability of the configuration nodes to 1. Place the configuration nodes into a queue.
3. While the queue is not empty, do
   3.1. Remove a node *k* from the queue, if it is not a configuration node, do

3.1.1. $P_{kj} = \Sigma(P_{ki} \times P_{ij})$, for all children i of node k.

3.2. For each parent node, if it is not a configuration node, do

3.2.1. Increase num_children_searched by 1, if num_children_searched equals to the total number of children of that node, insert the parent node into the queue.

Once this stage is complete, each instruction will contain the probabilities of the reachable RFUOPs. The following stages will use the results to schedule the necessary prefetches.

5.2.2. Prefetch Scheduling and Generation

The prefetch scheduling is quite trivial once the probabilities are calculated. Based on the decreasing order the probabilities a sequence of prefetches could be generated for each instruction node. Since the aggregate size of the reachable RFUOPs for a certain instruction may exceed the capacity of the chip, the algorithm will only generate prefetches under the size restriction of the chip. The rest of the reachable RFUOPs are ignored.

Assume in Figure 4 that the chip can at most hold 2 RFUOPs at a time, the generated prefetches at each instruction are showed on the right side of the figure.

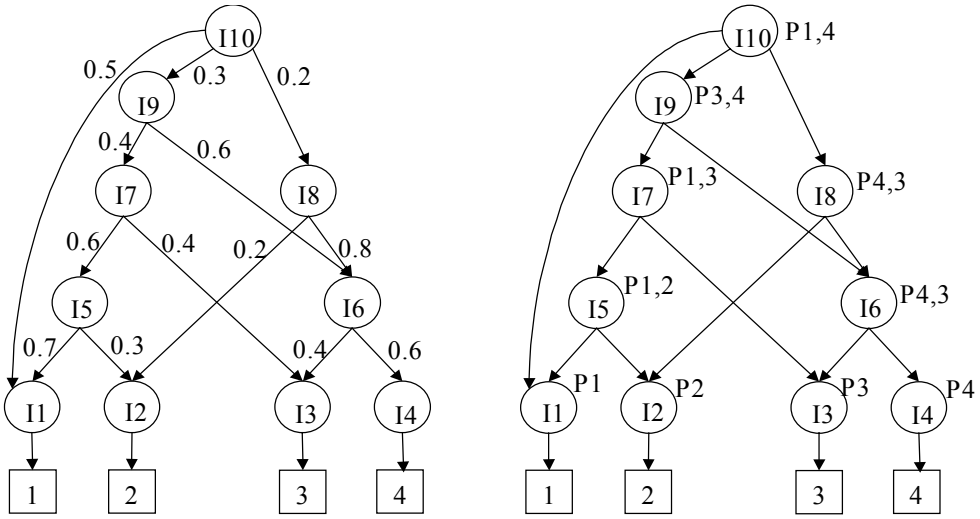

Figure 4. An example of prefetch scheduling and generation. The control flow is shown on the left. The corresponding prefetches are shown on the right.

5.2.3. Prefetch Reduction

The prefetches generated at a child node are considered as redundant if they match the beginning sub-sequence generated at its parents. Our algorithm will check and eliminate these redundant prefetches. One may argue that the prefetches at a child node should be considered as redundant if they are a sub-sequence, but not necessary the beginning sub-sequence, of its parents because the parents represent a superset of prefetches. However, by eliminating the prefetch instructions at the child node, the desired the prefetches cannot start immediately since the unwanted prefeches at the parents might have not completed. For example in Figure 4, P2 at the instruction I2 cannot be eliminated even though it is a sub-sequence of P1,2 because when I2 is reached P2 may not be able to start if P1 has not completed. In Figure 4, the prefetches at instructions I1, I4, and I6 can be eliminated.

Once the prefetch reduction is complete, for each instruction node where prefetches need to be performed a termination instruction is inserted followed by a sequence of prefetch instructions. Note that a termination instruction does not flush the entire FPGA, but merely clears the prefetch queue. RFUOPs that have already been loaded by the preceding prefetches are often retained.

5.3. Dynamic Configuration Prefetching

Among the various dynamic prefetching techniques available for general purpose computing systems, Markov prefetching [Joseph 1997] is a very unique approach. Markov Prefetching does not tie itself to particular data structure accesses and is capable of prefetching both instructions and data.

5.3.1. Markov Prefetching

As the name implies, Markov prefetching utilize a Markov model to determine what blocks should be brought in from the higher-level memory. A Markov process is a stochastic system for which the occurrence of a future state depends on the immediately preceding state, and only on it. A Markov process can be represented as a directed graph, with probabilities associated with each vertex. Each node represents a specific state, and a state transition is described by traversing an edge from the current node to a new node. The graph is built and updated dynamically using the available access information. As an example, the access string **A B C D C C C A B D E** will result in the Markov model described in Figure 5. Using the Markov graph multiple prefetches with different priorities can be issued.

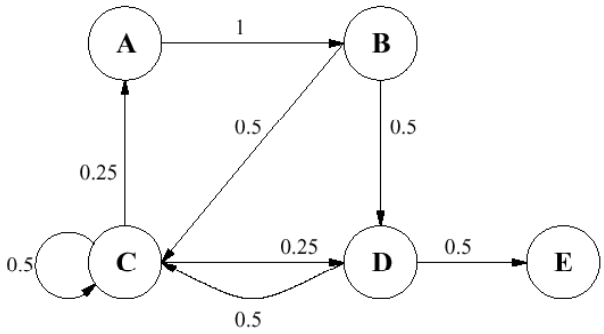

Figure 5. The Markov model generated from access string A B C D C C C A B D E. Each node represents a specific state and each edge represents a transition from one state to another. The number on an edge represents the probability of the transition occurring.

5.3.2. Dynamic Prefetching Algorithm

Markov prefetching can be extended to handle the configuration prefetching for reconfigurable systems. Specifically, the RFUOPs can be represented as the vertices in the Markov graph and the transitions can be built and updated using the RFUOP access sequence. However, the Markov prefetching needs to be modified because of the differences between general purpose systems and reconfigurable systems.

As mentioned in previous sections, due to their large sizes only a few RFUOPs can be stored on chip at a given time. Therefore, it is very unlikely that all the transitions from the current RFUOP node can be executed. This feature requires the system to make accurate predictions to guarantee that only the highly probable RFUOPs are prefetched. In order to find good candidates to prefetch, Markov prefetching keeps updating the probability of each transition using the currently available access information. This probability represents the overall likelihood this transition could happen during the course of the execution and may work well for a general purpose system which a large number of instructions or data blocks with small loading latency can be stored in a cache. However, without emphasizing the recent history, Markov prefetching could make poor prediction during a certain period of the execution. Due to the features of the reconfigurable systems mentioned above, we believe the probability calculated based on the recent history is more important than the overall probability.

In this work, a weighed probability in which recent accesses are given higher weight in probability calculation is used as a metric for candidate selection and prefetching order determination. The weighted probability of each transition is continually updated as the RFUOP access sequence progresses. Specifically, probabilities out of a node u will be updated once a transition *(u, v)* is executed:

For each transition starting from u,

$$P_{u,w} = P_{u,w} / (1 + C) \text{ if } w \neq v;$$

$$P_{u,v} = (P_{u,v} + C) / (1 + C);$$

Where C is a weight coefficient.

For general purpose systems, the prefetching unit generally operates separately from the cache management unit. Specifically, the prefetching unit will pick candidates and then send prefetch requests into the prefetching buffer (usually a FIFO). Working separately, the cache management unit will vacate the requests one by one from the buffer and load the instructions or data blocks into the cache. Since more accurate run-time access information is available for prefetching, integrating caching with prefetching will allow cache manager to choose better victims to replace. The dynamic prefetching algorithm can be described as following:

Upon the finish of each RFUOP *k* execution, do:

1. Sort the weighted probabilities in decreasing order of all transitions starting from *k* in the Markov graph.
2. Terminate all previously issued prefetches. Select *k* as the first candidate.
3. Select the rest of the candidates in sorted order under the size constraint of the chip. Issue prefetch requests for each candidate that is not currently on chip.
4. Update the weighted probability of each transition starting from *j*, where *j* is the RFUOP executed just before *k*.

Though the replacement is not presented in the algorithm, it is carried out indirectly. Specifically, any RFUOPs that are currently on chip will be marked for eviction if they are not selected as candidates. One last thing to mention is that the RFUOP just executed is treated as the top candidate automatically since generally each RFUOP is contained in a loop and likely to be repeated multiple times. Correspondingly, the self-loop transitions in the Markov graph are ignored.

5.3.3. Hardware Requirements of Dynamic Prefetching

A data structure is required to maintain and continually update the Markov graph as execution progresses. A table showed in Figure 6 can be used to represent the Markov graph. Each node (RFUOP) of the Markov graph occupies a single row of the table. The first column of each row is the ID of an RFUOP, and the rest of the columns are the RFUOPs it can reach. Since under chip size constraint only the high probable RFUOPs out of each node are used for the prefetching algorithm, keeping all transitions out of a node will simply waste precious hardware resources. As can be seen in Figure 6, the number of transitions retained is limited to K.

RFUOP 1	Next 1	Next 2	…	Next K
RFUOP 2	Next 1	Next 2	…	Next K
…	…	…	…	…
RFUOP M	Next 1	Next 2	…	Next K

Figure 6. A table is used to represent the Markov graph. The first column of each row is the ID of a RFUOP and the rest of the columns are k reachable RFUOPs from this row's RFUOP with highest probability.

In addition, a small FIFO buffer is required to store the prefetch requests. The configuration management unit will take the requests from the buffer and load the corresponding RFUOPs. Note that the buffer will be flushed to terminate previous

prefetches before the new prefetches are sent to the buffer. Furthermore, the configuration management unit can be interrupted to stop the current loading if an RFUOP not currently loaded is invoked. In order for the host process to save execution time in updating the probabilities, the weight coefficient *C* is set to 1. This means when a transition need to be updated, the host processor will simply right shift the register retaining the probability by one bit. Then the most significant bit of the register representing the currently occurring transition is set to 1. To balance the hardware cost and retain enough history, we use 8-bit registers in this work.

5.4. Hybrid Configuration Prefetching

The dynamic prefetching using recent history works well for the transitions occurred within a loop. However, this approach will not be able to make accurate predictions for the transitions jumping out a loop. For example on the left side of Figure 7 we assume only one RFUOP can be store on chip at any given point. By applying the dynamic prefetching approach, RFUOP 2 is always prefetched after RFUOP 1 assuming the inner loop will always be taken for several times. Thus, the reconfiguration penalty for RFUOP 3 can never be hidden due to the wrong prediction.

This misprediction can be avoided if the static prefetching approach can be integrated with the dynamic approach. More specifically, before reaching RFUOP3 a normal instruction node will likely be encountered and the static prefetches determined at that instruction node can be used to correct the wrong predictions determined by the dynamic prefetching. As illustrated on the right side of the Figure 7, a normal instruction I1 will be encountered before RFUOP 3 is reached and our static prefetching will correctly predict 3 will be the next required RFUOP. As the consequence, the wrong prefetch of RFUOP 2 determined by our dynamic prefetching can be corrected at I1.

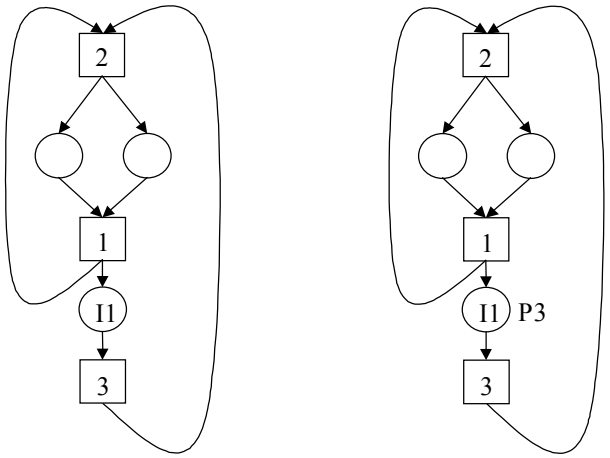

Figure 7. An example illustrates the ineffectiveness of the dynamic prefetching.

The goal of combining the dynamic configuration prefetching with the static configuration prefetching is to take advantage of the recent access history without exaggerating it. Specifically, dynamic prefetching using the recent history will make accurate predictions within the loops while static prefetching using the global history will make accurate predictions between the loops. The challenge of integrating dynamic prefetching with static prefetching is to coordinate the prefetches such that the wrong prefetches are minimized. When the prefetches determined by the dynamic prefetching do not agree those determined by the static prefetching a decision must be made. The basic idea we use to determine the beneficial prefetches for our hybrid prefetching is to penalize the wrong prefetches. We add an per-RFUOP flag bit to indicate the correctness of the prefetch made by previous static prefetching. When the prefetches determined by the static prefetching conflict with those determined by the dynamic prefetching, the statically predicted prefetch of a RFUOP is issued only if the flag bit for that RFUOP was set to 1. The flag bit of a RFUOP is set to 0 once the static prefetch of the RFUOP is issued, and will remain 0 until the RFUOP is executed. As the consequence, statically predicted prefetches, especially those made within the loops, are ignored if they are not correctly predicted. On the other hand, those correctly predicted static prefetches, especially those made between the loops, are chosen to replace the wrong prefetches made by the dynamic prefetching. The basic steps of the hybrid prefetching are outlined as following:

1. Perform the static configuration prefetching algorithm. Set the flag bit of each RFUOP to 1. An empty priority queue is created.
2. Upon the finish of a RFUOP execution, perform the dynamic prefetching algorithm. Set the flag bit of the RFUOP to 1. Clear the priority queue first, then place the Ids of the dynamically predicted RFUOPs into the queue.
3. When a static prefetch of a RFUOP is encountered and the flag bit of the RFUOP is 1, terminate current loading. Set the flag bit of the RFUOP to 0. Give the highest priority to this RFUOP and insert its ID into the priority queue. The RFUOPs with lower priorities are replaced or ignored to make room for the new RFUOP.
4. Load the RFUOPs from the priority queue.

6. RESULTS AND ANALYSIS

All algorithms are implemented in C++ on a Sun Sparc-20 workstation and are simulated with the SHADE simulator [Cmelik93]. We choose to use the SPEC95 benchmark suite to test the performance of our prefetching algorithms. Note that these applications have not been optimized for reconfigurable systems, and may not be as accurate in predicting exact performance as would real applications for reconfigurable systems. However, such real applications are not in general available for experimentation. In addition, the performance of the prefetching techniques will be compared against the previous caching techniques, which also use SPEC95 benchmark suite.

As can be seen in Figure 8, 5 algorithms are compared: Least Recently Used (LRU) Caching, Off-line Caching, Static Prefetching, Dynamic Prefetching, and Hybrid Prefetching. The LRU algorithm chooses victims to be replaced based on run-time information, while the Off-line algorithm takes consideration of future access patterns to make more accurate decisions. Note that our static prefetching uses the Off-line algorithm to pick victims. Since the cache replacement is integrated into the dynamic prefetching and the hybrid prefetching, no additional replacement algorithms are used for both prefetching algorithms. Clearly, all prefetching techniques substantially outperformed the solely caching techniques, especially when cache size is small. As cache

size grows, the chip is able to hold more RFUOPs and the cache misses will be reduced. However, the prefetching distance will not be changed. As the consequence, the performance due to prefetching will not significantly improve as the cache size grows. Among the prefetching techniques, dynamic prefetching performs consistently better than static prefetching because dynamic prefetching can use the RFUOP access information. Hybrid prefetching performs slightly better than dynamic prefetching because of its ability to correct some wrong prediction made by dynamic prefetching. However, the advantage of hybrid prefetching becomes negligible as the cache size grows.

Figure 9 demonstrates the effect of the different replacement algorithms that used for the static prefetching. As can be seen, the off-line replacement algorithm performs slightly better than the LRU since it has more complete information on the applications.

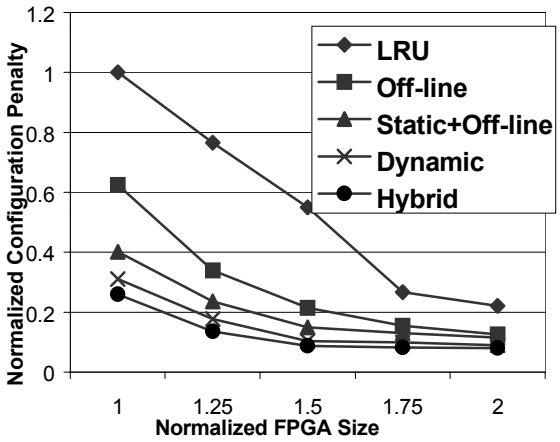

Figure 8. Reconfiguration overhead comparison.

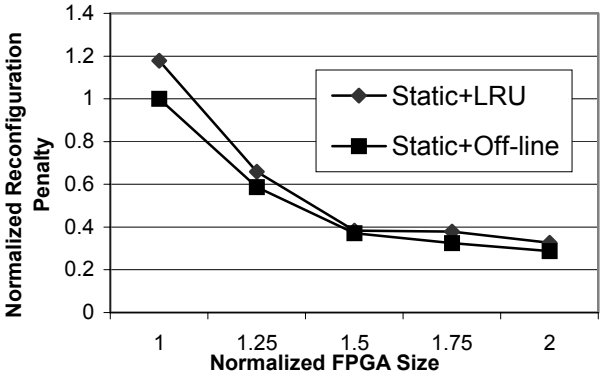

Figure 9. Effect of the replacement algorithms for the static prefetching.

7. CONCLUSIONS

Configuration prefetching, where configurations are preloaded on chip before they are required, is a technique to reduce the reconfiguration overhead. However, the limited on-chip memory and the large configuration latency add complexity in deciding which configurations to prefetch.

In this work we developed efficient prefetching techniques for reconfigurable systems containing a Partial Reconfigurable FPGA with Relocation and Defragmentation (R+D). We have developed algorithms applying the different configuration prefetching techniques. Based on the available access information and the additional hardware required, our configuration prefetching algorithms can be divided into 3 categories: Static Configuration Prefetching, Dynamic Configuration Prefetching, and Hybrid Configuration Prefetching. Compare with the caching techniques presented in our previous work [Li00], our prefetching algorithms can further reduce the reconfiguration overhead by more than a factor of 2.

References

[Babb97] J. Babb, M. Frank, V. Lee, E. Waingold, R. Barua, M. Taylor, J. Kim, S. Devabhaktuni, A. Agarwal, "The RAW Benchmark Suite: Computation Structures for General Purpose Computing", IEEE Symposium on FPGAs for Custom Computing Machines, pp. 134-143, 1997.

[Bolotski94] M. Bolotski, A. DeHon, T. F. Knight Jr., "Unifying FPGAs and SIMD Arrays", *2nd International ACM/SIGDA Workshop on Field-Programmable Gate Arrays*, 1994.

[Callahan91] D. Callahan, K. Kennedy, A. Porterfield, "Software Prefetching", *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 40-52, 1991.

[Cmelik93] R. F. Cmelik, *Introduction to Shade, Sun Microsystems Laboratories*, Inc., February, 1993.

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

[Dandalis01] Andreas Dandalis, Viktor Prasanna, "Configuration Compression for FPGA-based Embedded Systems", *ACM/SIGDA International Symposium on Field-Programmable Gate Array*s.

[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*, 1997.

[Hauck98] S. Hauck, "The Configuration prefetching for the Single Context FPGA", *ACM/SIGDA International Symposium on FPGA*s, pp65-74, 1998.

[Hauser97] John Hauser, John Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor", *IEEE Symposium on FPGAs for Custom Computing Machines,* pp. 24-33, 1997

[Joseph97] Doug Joseph, Dirk Grunwald, "Prefetching using Markov Predictors", *Proceedings of the 24th International Symposium on Computer Architecture*, pp 252-263, 1997.

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

[Li00] Zhiyuan Li, K. Compton, Scott Hauck, "Configuration Caching Management Techniques for Reconfigurable Computing", *IEEE Symposium on FPGAs for Custom Computing Machines,* pp. 87-96, 2000

[Li01] Zhiyuan Li, Scott Hauck, "Configuration Compression for Virtex FPGA", *IEEE Symposium on FPGAs for Custom Computing Machines,* pp 143-154, 2001

[Schmit97] Herman Schmit, "Incremental Reconfiguration for Pipelined Applications", *IEEE Symposium on FPGAs for Custom Computing Machines*, pp 47-55, 1997.

[Spec95] *SPEC CPU95 Benchmark Suite*, Standard Performance Evaluation Corp., Manassas, VA, 1995.

[Wittig96] R. Wittig, P. Chow, "OneChip: An FPGA Processor with Reconfigurable Logic", *IEEE Symposium on FPGAs for Custom Computing Machines*, pp126-135, 1996.