

Configuration Prefetching Techniques for Partial Reconfigurable Coprocessors with Relocation and Defragmentation

Zhiyuan Li

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

Scott Hauck

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

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 reconfigurable systems containing a Partial Reconfigurable FPGA with Relocation + Defragmentation (R+D model) since the R+D FPGA showed high hardware utilization [Compton02]. 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 hardware, especially field programmable gate arrays (FPGAs) have been considered as a new and effective means of performing computation. The wide range of high-performance applications that have been developed with FPGAs indicate the great successes of this technology. Now many systems such as Garp [Hauser97], OneChip [Wittig96] and Chimera [Hauck97] combine reconfigurable hardware with a standard general-purpose processors to make the FPGAs suitable not only for specialized applications, but also for general-purpose applications. In such systems, portions of an application suited to hardware implementation are mapped to the FPGA, while other portions of computation are handled by the general-purpose processor.

In such systems, the multiple configurations generated for a computation demand run time reconfigurable devices. For run time reconfigurable systems, the configuration overhead is a major concern because the CPU must sit idle during reconfiguration. If the configuration overhead is reduced, the performance of the system could be improved. Therefore, many groups have studied techniques to reduce the configuration times. Some techniques, such as configuration caching [Li00] and configuration compression [Hauck98b] [Li99], have greatly reduced the configuration overhead.

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 systems, 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 [Hauser97], 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 [Compton02] 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. Reconfigurable Models

Frequently, the areas of a program that can be accelerated through the use of reconfigurable hardware are too numerous or complex to be loaded simultaneously onto the available hardware. For these cases, it is helpful to swap different configurations in and out of the reconfigurable hardware as they are needed during program execution, performing a run-time reconfiguration of the hardware. Because run-time reconfiguration allows more sections of an application to be mapped into hardware than can be fit in a non-run-time reconfigurable system, a greater portion of the program can be accelerated in the run-time reconfigurable systems. This can lead to an overall improvement in performance.

There are a few traditional configuration memory styles that can be used with reconfigurable systems, including the *Single Context* model [Xilinx94, Altera98, Lucent98], the *Partial Run-time Reconfigurable* model (PRTR) [Ebeling96, Schmit97, Hauck97] and the *Multi-Context* model [DeHon94, Trimberger97]. For Single Context FPGA shown in Figure 1, the whole array can be viewed as a shift register, and the whole chip area must be reconfigured during each reconfiguration. This means that even if only a small portion of the chip needs to be reconfigured, the whole chip is rewritten. Since many of the applications being developed with FPGAs today involve run-time reconfiguration, the reconfiguration of the Single Context architecture incurs a significant overhead. Therefore, much research has focused on the new generation of architectures or tools that can reduce this reconfiguration overhead.

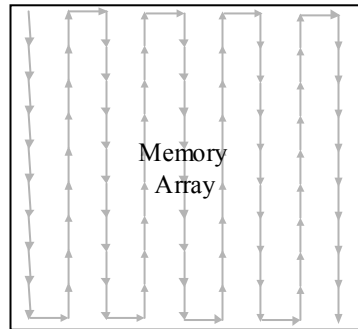


Figure 1. The structure of a Single Context FPGA.

Partial Run-time Reconfiguration (PRTR) is another typical reconfiguration model. By changing only a portion of the reconfigurable logic while other sections continue operating, the reconfiguration latency can be hidden by other computations. Many commercial devices (Xilinx Virtex series, Xilinx 6200 series) and systems (Garp [Hauser 97], Chimaera [Hauck 97], DISC [Wirthlin 96]) have applied the PRTR model.

The structure of a PRTR FPGA is shown in Figure 2. By sending configuration values with row and column addresses to the FPGA, a certain location can be configured.

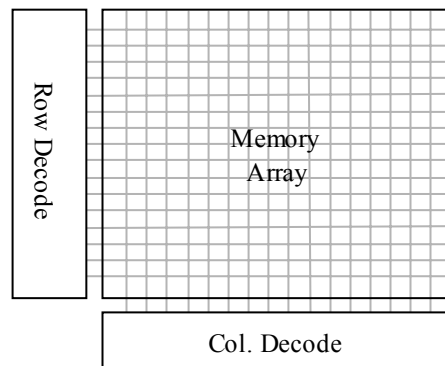


Figure 2. The structure of a Partial Run-time Reconfigurable FPGA.

Based on the Partial Run-time Reconfigurable model, a new model, called Relocation + Defragmentation (Partial R + D) [Compton00, Hauser97], was built to further improve hardware utilization. Relocation allows the final placement of a configuration within the FPGA to be determined at run-time, while defragmentation provides a way to consolidate unused area within an FPGA during run-time without unloading useful configurations.

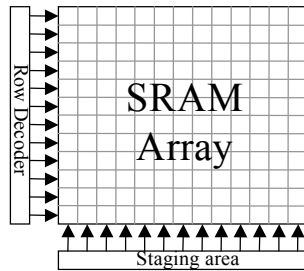


Figure 3. The architecture of the Relocation + Defragmentation model. Each row of the configuration bit-stream is loaded into the staging area and then moved into the array.

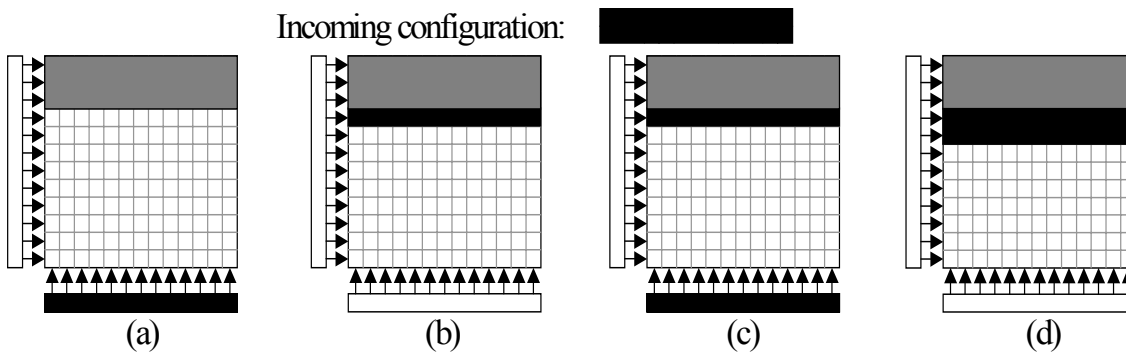


Figure 4. An example of configuration relocation. The incoming configuration contains two rows. The first row is loaded into the staging area (a) and then transferred to the desired location that was determined at run-time (b). Then the second row of the incoming configuration is loaded to the staging area (c) and transferred into the array (d).

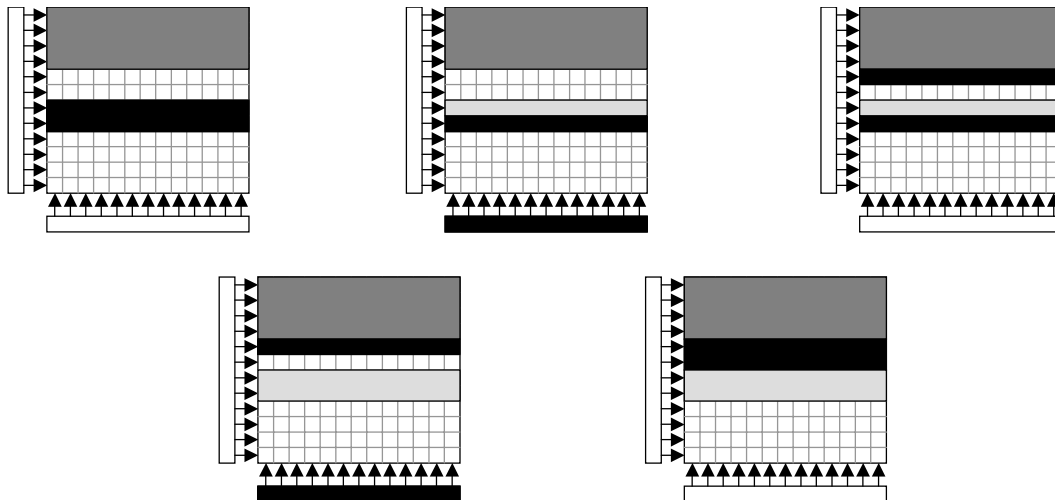


Figure 5. An example of defragmentation. By moving the rows in a top-down fashion into staging area and then moving upwards in the array, the smaller fragments are collected.

Like the PRTR 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. The structure view of a Partial R + D model is shown in Figure 3.

To configure a 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 4 shows the steps of relocating a configuration into the array. The defragmentation operation is slightly more complicated than a simple relocation operation. 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 5 presents the steps of a defragmentation operation. [Compton02] has shown that with a very minor area increase, the Relocation + Defragmentation model has a considerably lower reconfiguration overhead than the Partial Run-Time Reconfigurable model.

3. Prefetching Overview

Prefetching for standard processor caches has been extensively studied. Research is normally split into data and instruction prefetching. In data prefetching, the organization of a data structure and the measured or predicted access pattern are exploited to determine which portions of a data structure are likely to be accessed next. The simplest case is array accesses with a fixed stride, where the access to memory location N is followed by accesses to $(N+\text{stride})$, $(N+2*\text{stride})$, etc. Techniques can be used to determine the stride and issue prefetches for locations one or more strides away [Mowry92, Santhanam97, Zucker98, Callahan91]. For more irregular, pointer-based structures, techniques have been developed to prefetch locations likely to be accessed in the near future, either by using the previous reference pattern, by prefetching all children of a node, or by regularizing data structures [Luk96].

Techniques have also been developed for instruction prefetching. The simplest is to prefetch the cache line directly after the line currently being executed [Smith78, Hsu98], since this is the next line needed unless a jump or branch intervenes. To handle such branches information can be kept on all previous successors to this block [Kim93], or the most recent successor (“target prefetch”) [Hsu98], and prefetch these lines. Alternatively, a look-ahead PC can use branch prediction to race ahead of the standard program counter, prefetching along the likely execution path [Chen94, Chen97]. However, information must be maintained to determine when the actual execution has diverged from the lookahead PC’s path, and then restart the lookahead along the correct path.

Unfortunately, many standard prefetching techniques are not appropriate for FPGA configurations because of differences between configurations and single instruction or data block. Before we will discuss the details for the configuration prefetching, we first reexamine the factors very important to the effectiveness of prefetching for a general-purpose system:

Accuracy. This is the ratio of the executed prefetched instructions or data to the overall prefetched instructions or data. Prefetching accuracy estimates the quality of a prefetching technique as a fraction of all prefetches that are useful. Based on profile or run-time information, the system must be able to make accurate predictions on the instructions or data that will be used and fetch them in advance.

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.

Pollution. One side-effect of prefetching techniques produce is that 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 a cache block. Cache studies consistently show that the average access time will likely drop when the block size increases until it reaches a certain value (usually fewer than 128 bytes), and access time will then increase 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 prefetching domain. Due to the large configuration size and relatively small on-chip memory, very few configurations can be stored on reconfigurable hardware. As the result, an incorrect prefetch will be very likely to cause a required configuration to be replaced, and 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 overhead. In addition, the correct prefetch of a configuration needs to be performed much earlier than it will be required; otherwise, the large configuration loading latency cannot be entirely or mostly hidden. A correct prediction must be made as early as possible to make prefetching an effective approach.

In general, prefetching algorithms can be divided into three 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 very little 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. Factors Affecting Configuration Prefetching

In order to better discuss the factors that will affect prefetching performance, we first make the 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 .
- $Cost_{ik}$: The potential minimum cost if k is prefetched at Instruction i .
- C : The capacity of the chip.

The probability factor could have a significant impact on 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 three configurations 1, 2, and 3 to be executed in that order. Given that $S_3 \gg S_1 \gg S_2$, $S_1 + S_3 < C < S_1 + S_2 + S_3$, and $D_{12} \gg L_3 \gg L_2 > D_{23}$, prefetching configuration 3 before configuration 2 when 1 is executed results in a penalty of L_2 at most. This is because the latency of configuration 3 can be completely hidden and 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 paper 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). This approach is illustrated in Figure 6. 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.

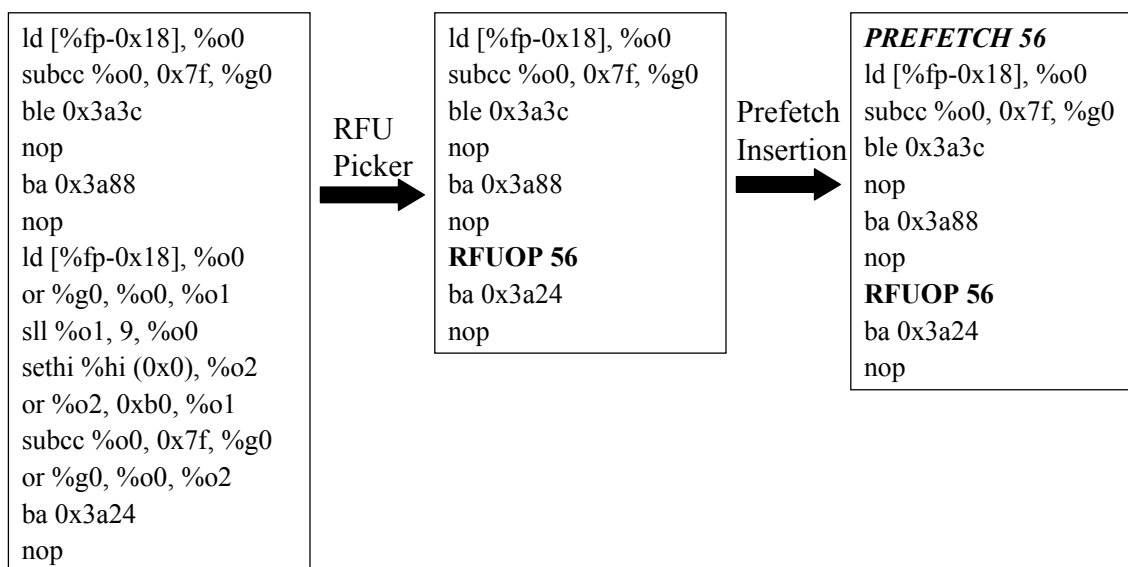


Figure 6. Experimental setup for prefetching. Source code is analyzed by the RFUP picker, mapping the portions that can be sped up to the reconfigurable coprocessor. The mappings are executed by inserting calls to the reconfigurable coprocessor (RFUOPs). Then prefetch instructions are inserted to load the RFUOPs in advance.

To make our analysis clearer in the control flow graph, we use circles to represent the instruction nodes and squares to represent the RFUOPs for the rest of the paper. Since a prefetch should be executed at the top node of a single entry and single exit path, but not at 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, with a length that represents the number of nodes contained in the path.

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 that 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 damage performance.

In Figure 7, 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.

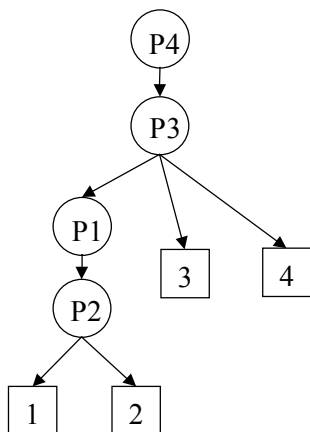


Figure 7. Example of prefetching operation control.

A simple approach used in this work to solve this problem is to insert termination instructions when necessary. The format of a termination instruction resembles 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 prefetches so the new prefetches can start immediately. For the example in Figure 7, a termination instruction will be inserted immediately before P1 to eliminate the unwanted prefetches of P4 and P3.

Now that we have demonstrated how 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 presented in [Hauck98] demonstrates high-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 three stages:

Penalty calculation. In this stage the algorithm computes the potential penalties for a set of prefetches at each instruction node of the control flow graph.

Prefetch scheduling and generation. In this stage the algorithm determines the configurations that need to be prefetched at each instruction node based on the penalties calculated in Stage 1. Prefetches are generated under the restriction of the size of the chip.

Prefetch reduction. In this stage the algorithm trims the redundant prefetches generated in the previous stage. In addition, it inserts termination instructions.

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.

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:

For each instruction, initialize the probability of each reachable configuration to 0, and set the num_children_searched to be 0.

Set the probability of the configuration nodes to 1. Place the configuration nodes into a queue.

While the queue is not empty, do

Remove a node k from the queue; if it is not a configuration node, do:

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

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

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.

The left side of Figure 8 shows a control flow graph with branch probability on each edge. Table 1 shows the probabilities calculated at the instruction node with more than one reachable RFUOP.

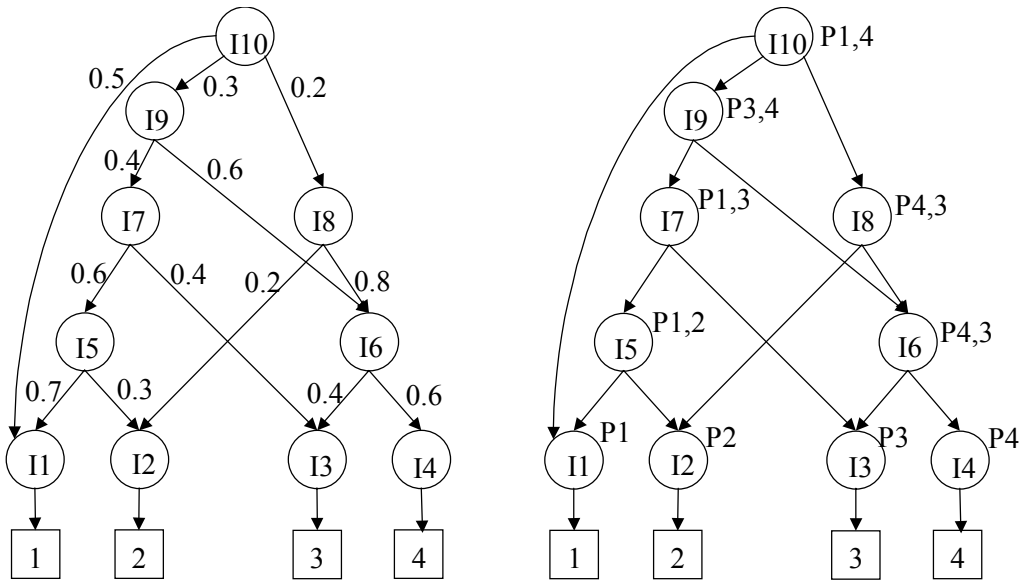


Figure 8: An example of prefetch scheduling and generation after the probability calculation.

Once this stage is complete, each instruction contains the probabilities of the reachable RFUOPs. We use the results to schedule necessary prefetches.

The prefetch scheduling is quite trivial once the probabilities have been calculated. Based on the decreasing order of probabilities, a sequence of prefetches could be generated for each instruction node. Since the aggregate size of the reachable RFUOPs for a certain instruction could exceed the capacity of the chip, the algorithm only generates prefetches under the size restriction of the chip. The rest of the reachable RFUOPs are ignored. For example, in Figure 8 we assume that the chip can at most hold 2 RFUOPs at a time. The generated prefetches at each instruction are shown on the right side of Figure 8.

	1	2	3	4
I5	0.7	0.3	0	0
I6	0	0	0.4	0.6
I7	0.42	0.18	0.4	0
I8	0	0.2	0.32	0.48
I9	0.168	0.072	0.4	0.36
I10	0.5504	0.0616	0.184	0.204

Table 1. Probability calculation for example in Figure 8.

The prefetches generated at a child node are considered to be 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 prefetches cannot start immediately, since the unwanted prefetches at the parents might have not completed. For example, in Figure 8, $P2$ 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 8, the prefetches at instructions $I1$, $I4$, and $I6$ can be eliminated.

Once the prefetch reduction is complete, for each instruction node where prefetches must 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.

5.2. Dynamic Configuration Prefetching

Among the various dynamic prefetching techniques available for general-purpose computing systems, Markov prefetching [Joseph97] 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.

As the name implies, Markov prefetching uses a Markov model to determine which 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 $A B C D C C C A B D E$ results in the Markov model shown in Figure 9. Using the Markov graph multiple prefetches with different priorities can be issued.

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.

Note that only a few RFUOPs can be retained on-chip at any given time because of the large size of each RFUOP. 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 where 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.

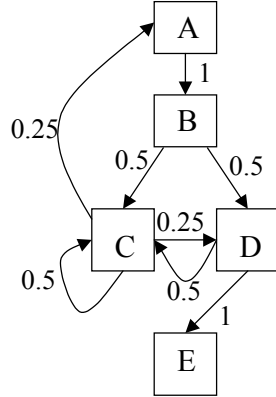


Figure 9. The Markov model generated from access string A B C D C C C A B D E. Each node represents a specific state and each edge represents a transition from one state to another. The number on an edge represents the probability of the transition occurring.

In this work, a weighed probability in which recent accesses are given higher weight in probability calculation is used 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 picks candidates and then sends 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. If there is not enough space in the cache, the cache management unit will apply a certain replacement policy to select victims and evict them to make room for the instructions or data to be loaded.

Though this works well for general-purpose systems, this separated approach may not be efficient for configuration prefetching because of the large size and latency of each RFUOP. For example, at a certain point 3 equal-size RFUOPs A, B, C are stored on chip, and equal-size RFUOPs D, A, B are required in sequence with a very short distance between each other. Suppose there is no room on chip and a FIFO replacement policy is used. The system will evict A first to make room for D, then replace B with A and C with B in order. It is obvious that the overall latency will not be significantly reduced because of the short distance between D, A, and B. This situation can be improved by combining the prefetching with caching techniques. For the example above, we can simply replace C with D and retain A and B on chip to eliminate the latencies of loading them. The intuition behind this approach is to use the prefetching unit as predictor as well as a cache manager. The dynamic prefetching algorithm can be described as following:

Upon the completion of each RFUOP k 's execution, do:

Sort the weighted probabilities in decreasing order of all transitions starting from k in the Markov graph.

Terminate all previously issued prefetches. Select k as the first candidate.

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.

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 point 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. Hardware Requirements of Dynamic Prefetching

A data structure is required to maintain and continually update the Markov graph as execution progresses. A table, as shown in Figure 6.8, 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 Table 2, 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

Table 2. 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 that when a transition needs 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. An N -bit register representing any transition (u, v) will become 0 if (u, v) does not occur for N executions of u . In that sense, only the most recent N accesses of u are used to compute the probability for each transition starting from u . As a consequence, the number of transitions that needs to be stored in the table representing the Markov graph can be set to N (i.e. $K = N$). To balance the hardware cost and retain enough history, we use 8-bit registers in this work.

5.4. Hybrid Configuration Prefetching

Dynamic prefetching using recent history works well for the transitions occurring within a loop. However, this approach will not be able to make accurate predictions for transitions jumping out a loop. For example, on the left side of Figure 10, we assume only one RFUOP can be stored on chip at any given point. By applying 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 10, 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.

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 with those determined by the static prefetching a decision must be made.

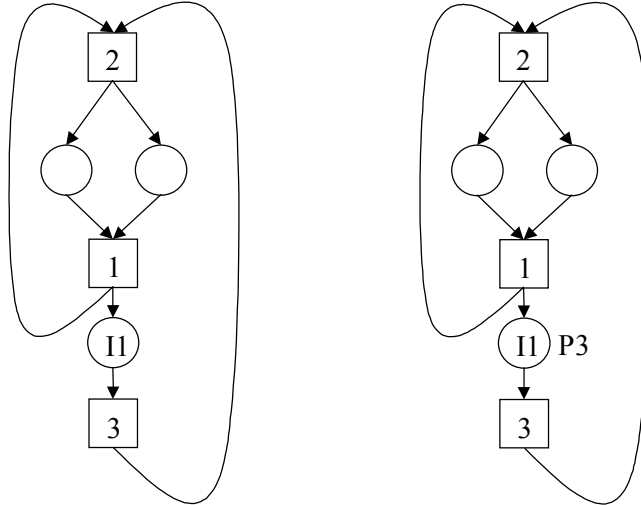


Figure 10: An example illustrates the ineffectiveness of the dynamic prefetching.

The basic idea we use to determine the best prefetches for our hybrid prefetching is to penalize wrong prefetches. We add a per-RFUOP flag bit to indicate the correctness of the prefetch made by previous static prefetching. When the prefetches determined by static prefetching conflict with those determined by 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 a 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.

5.5. 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 used the SPEC95 benchmark suite.

As can be seen in Figure 11, five 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 into consideration future access patterns to make more accurate decisions. Note that our static prefetching uses the Off-line algorithm to pick victims. Since cache replacement is integrated into dynamic prefetching and hybrid prefetching, no additional replacement algorithms are used for both prefetching algorithms. The graph's X-axis is the normalized FPGA size for all algorithms, and the Y-axis represents the configuration penalty that is normalized to the number of the instruction cycles of LRU algorithm at size 1.

Clearly, all prefetching techniques substantially outperformed caching-only techniques, especially when cache size is small. As cache size grows, the chip is able to hold more RFUOPs and cache misses are reduced. However, the prefetching distance is not changed. As a 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 it 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.

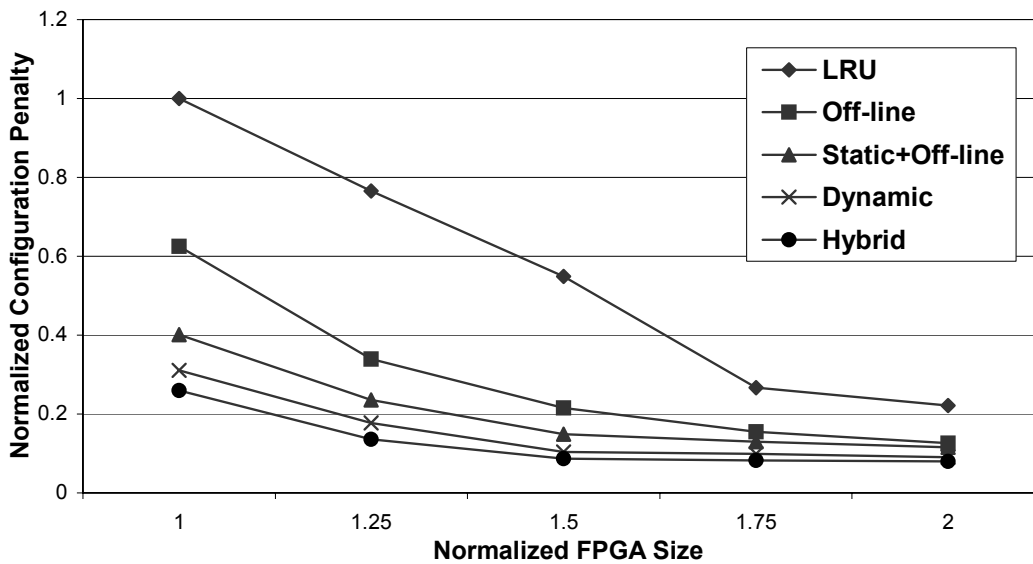


Figure 11. Reconfiguration overhead comparison.

Figure 12 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.

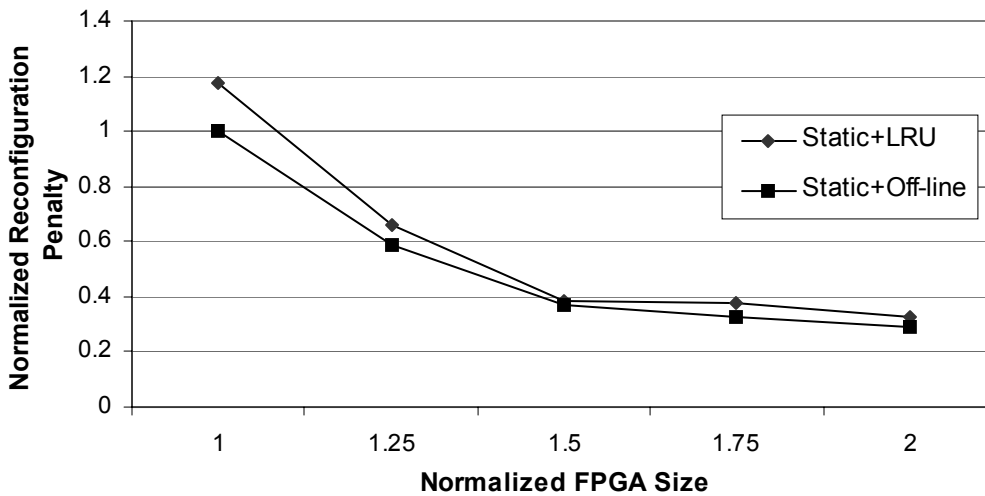


Figure 12. Effect of the replacement algorithms for the static prefetching.

6. 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

- [Altera98] Altera Inc.. Altera Mega Core Functions, <http://www.altera.com/html/tools/megacore.html>, San Jose, CA, 1998.
- [Cmelik93] R. F. Cmelik. Introduction to Shade. *Sun Microsystems Laboratories, Inc.*, February, 1993.
- [Compton02] K. Compton, Z. Li, S. Knol, S. Hauck, "A Configuration Relocation and Defragmentation for FPGAs", *IEEE Transactions on VLSI*, Vol. 10, No. 3., pp. 209-220. June 2002.
- [DeHon94] Andre DeHon. DPGA-coupled microprocessors: Commodity ICs for the early 21st century. In Duncan A. Buell and Kenneth L. Pocek, editors, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pp 31--39, April 1994.
- [Ebeling96] C. Ebeling, D.C. Cronquist, P. Franklin. RaPiD – Reconfigurable Pipelined Datapath. *Lecture Notes in Computer Science 1142—Field-programmable Logic: Smart Applications, New Paradigms and Compilers*. R. W. Hartenstein, M. Glesner, Eds. Berlin, Germany: Springer-Verlag, pp. 126-135, 1996.
- [Hauck97] S. Hauck, T. Fry, M. Hosler, J. Kao. The Chimaera Reconfigurable Functional Unit. *IEEE Symposium on FPGAs for Custom Computing Machines*, 1997.
- [Hauck98] 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.

- [Joseph97] Doug Joseph, Dirk Grunwald. Prefetching Using Markov Predictors. *Proceedings of the 24th International Symposium on Computer Architecture*, 1997.
- [Li00] Z.Li, K. Compton, Scott Hauck. Configuration Caching Management Techniques for Reconfigurable Computing”. *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 87-96, 2000.
- [Lucent98] Lucent Technology Inc.. FPGA Data Book, 1998.
- [Luk96] C.-K. Luk, T. C. Mowry. Compiler-Based Prefetching for Recursive Data Structures. *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 222-233, 1996.
- [Mowry92] T. C. Mowry, M. S. Lam, A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 62-73, 1992.
- [Schmit97] H. 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.
- [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.