

Improved Configuration Prefetch for Single Context Reconfigurable Coprocessors

Scott Hauck

Department of Electrical Engineering
University of Washington
Seattle, WA 60208-3118 USA
hauck@ee.washington.edu

Zhiyuan Li

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

Abstract

Current reconfigurable systems suffer from a significant overhead due to the time it takes to reconfigure their hardware. In order to deal with this overhead, and increase the compute power of reconfigurable systems, it is important to develop hardware and software systems to reduce or eliminate this delay. In this paper we propose one technique for significantly reducing the reconfiguration latency: the prefetching of configurations. By loading a configuration into the reconfigurable logic in advance of when it is needed, we can overlap the reconfiguration with useful computation. We demonstrate the power of this technique, and propose an algorithm for automatically adding prefetch operations into reconfigurable applications. This results in a significant decrease in the reconfiguration overhead for these applications.

1 Introduction

When FPGAs were first introduced in the mid 1980s they were viewed as a technology for replacing standard gate arrays for some applications. In these first generation systems a single configuration is created for the FPGA, and this configuration is the only one loaded into the FPGA. A second generation soon followed, with FPGAs that could use multiple configurations, but reconfiguration was done relatively infrequently [Hauck98a]. In such systems the time to reconfigure the FPGA was of little concern.

Many of the most exciting applications being developed with FPGAs today involve run-time reconfiguration [Hauck98a]. In such systems the configuration of the FPGAs may change multiple times in the course of a computation, reusing the silicon resources for several different parts of a computation. Such systems have the potential to make more effective use of the chip resources than even standard ASICs, where fixed hardware may only be used in portion of the computation. However, the advantages of run-time reconfiguration do not come without a cost. By requiring multiple reconfigurations to complete a computation, the time it takes to reconfigure the FPGA becomes a significant concern. In most systems the FPGA must sit idle while it is being reconfigured, wasting cycles that could otherwise be performing useful work. For example, applications on the DISC and DISC II system have spent 25% [Withlin96] to 91% [Wirthlin95] of their execution time performing reconfiguration.

It is obvious from these overhead numbers that reductions in the amount of cycles wasted to reconfiguration delays can have a significant impact on the performance of run-time reconfigured systems. For example, if an application spends 50% of its time in reconfiguration, and we were somehow able to reduce the overhead per reconfiguration

by a factor of 2, we would reduce the application's runtime by at least 25%. In fact, the performance improvement could be even higher than this. Specifically, consider the case of an FPGA used in conjunction with a host processor, with only the most time-critical portions of the code mapped into reconfigurable logic. An application developed for such a system with a given reconfiguration delay may be unable to take advantage of some optimizations because the speedups of the added functionality are outweighed by the additional reconfiguration delay required to load the functionality into the FPGA. However, if we can reduce the reconfiguration delay, more of the logic might profitably be mapped into the reconfigurable logic, providing an even greater performance improvement. For example, in the UCLA ATR work the system wastes more than 95% of its cycles in reconfiguration [Villasenor96, Villasenor97]. This overhead has limited the optimizations explored with the algorithm, since performance optimizations to the computation cycles will yield only limited improvement in the overall runtimes. This has kept the researchers from using higher performance FPGA families and other optimizations which can significantly reduce the computation cycles required.

Because of the potential for improving the performance of reconfigurable systems, reducing the reconfiguration delay is an important research area. In this paper we consider one method for reducing this overhead: the overlapping of computation with reconfiguration via the prefetching of FPGA configurations.

2 Configuration Prefetch

Run-time reconfigured systems use multiple configurations in the FPGA(s) during a single computation. In current systems the computation is allowed to run until a configuration not currently loaded is required to continue the computation. At that point the computation is stalled while the new configuration is loaded. These stall cycles represent an overhead to the computation, increasing runtimes without performing useful work.

One method to reduce this reconfiguration overhead is to begin loading the next configuration before it is actually required. Specifically, in systems with multiple contexts [Bolotski94], partial run-time reconfigurability [Hutchings95], or tightly coupled processors [DeHon94, Razdan94, Wittig96, Hauck97] it is possible to load a configuration into all or part of the FPGA while other parts of the system continue computing. In this way, the reconfiguration latency is overlapped with useful computations, hiding the reconfiguration overhead. We will call the process of preloading a configuration before it is actually required *configuration prefetching*.

The challenge in configuration prefetching is determining far enough in advance which configuration will be required next. Many computations (especially those found in general-purpose computations) can have very complex control flows, with multiple execution paths branching off from any point in the computation, each potentially leading to a different next configuration. At a given point in the computation it can be difficult to decide which configuration will be required next. Even worse, the decision of which configuration to prefetch may need to be done thousands of cycles in advance if we wish to hide the entire reconfiguration delay. In a system where it

takes a thousand cycles to load a configuration, if we do not begin fetching the configuration at least a thousand cycles in advance we will be unable to hide the entire reconfiguration latency.

Not only is it necessary to decide which configuration to load far in advance of a configuration's use, it is also important to correctly guess which configuration will be required. In order to load a configuration, configuration data that is already in the FPGA must be overwritten. An incorrect decision on what configuration to load can not only fail to reduce the reconfiguration delay, but in fact can greatly increase the reconfiguration overhead when compared to a non-prefetching system. Specifically, the configuration that is required next may already be loaded, and an incorrect prefetch may require the system to have to reload the configuration that should have simply been retained in the FPGA, adding reconfiguration cycles where none were required in the non-prefetch case.

Prefetching for standard processor caches has been extensively studied. These efforts are 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 and stride issue prefetches for locations one or more strides away [Mowry92, Santhanam97, Zucker98, Callahan99]. 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 prefetching instructions. 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 lookahead 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. Data prefetching techniques are inappropriate since there is usually no data structure to trace – the configurations are part of the operation of the system, and are invoked when they are reached in the execution stream. Lookahead PCs will not hide much of the huge reconfiguration penalties for FPGAs because the chances of misprediction means they can only be effective a moderate distance ahead of the true PC. For example, if branch prediction is 95% accurate, on average it will correctly predict approximately twenty branches in a row. With current FPGA reconfiguration times in at least the thousands of cycles, this will not be enough to hide much of the reconfiguration penalty.

Target prefetch [Hsu98] shows some promise, but would be difficult to implement. In target prefetch each block remembers the block accessed directly after itself. This successor is then prefetched the next time this block is accessed. If this approach was applied directly to configurations, each configuration would remember its last successor. However, current FPGAs can hold only one configuration, and each configuration is normally executed multiple times in series. Thus, configurations would either immediately unload themselves after their first execution after a different configuration has run, radically increasing reconfiguration penalties, or no prefetching would be done at all. Alternatively, each instruction block could remember what configuration was next executed. However, a large number of instructions may be executed between configurations, and all of these instructions would have to be updated with the configuration next called, requiring very complex hardware support.

In this paper we will demonstrate the potential of prefetching for reconfigurable systems, and present a new algorithm for automatically determining how this prefetching should be performed. We first present a model of the reconfigurable systems that we consider in this paper. We then develop an upper bound on the improvements possible from configuration prefetching under this model via an (unachievable) optimal prefetching algorithm. Finally, we present a new algorithm for configuration prefetching which can provide significant decreases in the per-reconfiguration latency in reconfigurable systems.

3 Reconfigurable System Model

Most current reconfigurable computing architectures consist of general-purpose FPGAs 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, arbitrary operations (contained in FPGA configurations) during the execution of the computation, but most current FPGAs can hold only one configuration at a time. In order to perform a different computation the FPGA must be reconfigured, which takes a fixed (but architecture-depend) time to complete. Note that the reconfiguration system for the FPGA and the processor's memory structures are often separate, with the FPGAs often having associated FLASH or other memory chips to hold configurations. Thus, conflicts between reconfiguration and processor memory accesses are typically not present.

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. At this point the coprocessor checks if the proper configuration is loaded. If it is not, the processor stalls while the configuration loads. Once the configuration is loaded (or immediately if the proper configuration was already present), the coprocessor executes the desired computation. Once a configuration is loaded it is retained for future executions, only being unloaded when another coprocessor call or prefetch operation specifies a different configuration.

In order to avoid this latency, a program running on this reconfigurable system can insert prefetch operations into the code executed on the host processor. These prefetch instructions 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. If the specified configuration is not present, the coprocessor trashes the current configuration and begins loading the configuration specified. At this point the host processor is free to perform other computations, overlapping the reconfiguration of the coprocessor with other useful work. Once the next call to the coprocessor occurs it can take advantage of the loading performed by the prefetch instruction. If this coprocessor call requires the configuration specified by the last prefetch operation, it will either have to perform no reconfiguration if the coprocessor has had enough time to load the entire configuration, or only require a shorter stall period as the remaining reconfiguration is done. Obviously, if the prefetch instruction specified a different configuration than was required by the coprocessor call, the processor will have to be stalled for the entire reconfiguration delay to load the correct configuration. Because of this, an incorrect prefetch operation can not only fail to save reconfiguration time, it can in fact increase the overhead due to the reconfigurable coprocessor. This occurs both in the wasted cycles of the useless prefetch operations, as well as the potential to overwrite the configuration that is in fact required next, causing a stall to reload a configuration that should have been retained in the reconfigurable coprocessor.

For simplicity we assume that the coprocessor can implement arbitrary code sequences, but these sequences must not have any sequential dependencies. This is enforced by requiring that the code mapped to the coprocessor appear sequentially in the executable, have a single entry and exit point, and have no backwards edges. Note that while assuming a reconfigurable coprocessor could implement any such function is optimistic, it provides a reasonable testbed with properties similar to reconfigurable systems that have been proposed [Razdan94, Hauck97].

In this study we assume a single-issue in-order processor with a separate reconfiguration bus. In systems with a more advanced processor, or in which reconfiguration operations may conflict with other memory accesses, the relative delay of reconfiguration to instruction issue may be significantly increased over current systems. Alternatively, reconfigurable logic with on-chip configuration caches may decrease this ratio. To handle this uncertainty we consider a wide range of reconfiguration times. Also, we measure the impact of the techniques on reconfiguration time, which can be reasonably measured in this system, as opposed to overall speedup, which cannot accurately be measured due to uncertainties in future system architectures and exact program features.

4 Experimental Setup

To investigate the impact of prefetching on the reconfiguration overhead in reconfigurable systems we have tested prefetching on standard software benchmarks from the SPEC benchmark suite [Spec95]. 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. Although initial work has been done in collecting benchmarks, they either represent single kernels without run-time-reconfiguration [Babb97], or are constructed primarily to stress-test only a

portion of the reconfigurable system [Kumar98]. Thus, we feel that the only feasible way to investigate optimizations to the reconfiguration system is to use current, general-purpose applications, and make reasonable assumptions in order to mimic the structure of future reconfigurable system.

In order to conduct these experiments we must perform three steps. First, some method must be developed to choose which portions of the software algorithms should be mapped to the reconfigurable coprocessor. Second, a prefetch algorithm must be developed to automatically insert prefetch operations into the source code. Third, a simulator of the reconfigurable system must be employed to measure the performance of these applications. Each of these three steps will be described in the paragraphs that follow.

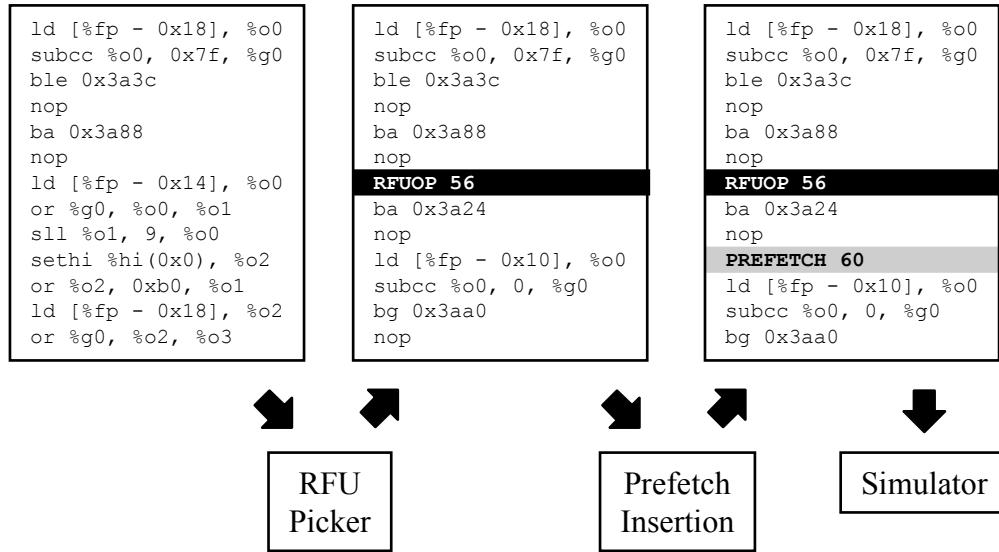


Figure 1. Experimental setup. Source code is augmented with calls to the reconfigurable coprocessor (RFUOPs) by the RFU picker. This code then has prefetch instructions inserted into it. The performance of a given set of RFUOPs and PREFETCHes is measured by the simulator.

The first step is to choose which portions of the source code should be mapped to the reconfigurable coprocessor (these mappings will be referred to as RFUOPs here). As mentioned before, we assume that arbitrary code sequences can be mapped to the reconfigurable logic as long as they have a single entry and a single exit point, and have no backward branches or jumps. In order to pick which portions of the code to map to the reconfigurable logic we find all potential mappings and then simulate the impact of including each candidate assuming optimal prefetching (described later). We then greedily choose the candidate that provides the best performance improvement, and retest the remaining candidates. In this way a reasonable set of RFUOPs can be developed which produces a significant performance improvement even when reconfiguration delay is taken into consideration.

Our simulator is developed from SHADE [Cmelik93a]. It allows us to track the cycle-by-cycle operation of the system, and get exact cycle counts. This simulator reports the reconfiguration time and overall performance for the

application under both normal and optimal prefetching, as well as performance assuming no prefetching occurs at all. These numbers are used to measure the impact of the various prefetching techniques.

Note that for simplicity we model reconfiguration costs as a single delay constant. Issues such as latency verses bandwidth in the reconfiguration system, conflicts between configuration load and other memory accesses in systems which use a single memory port, and other concerns are ignored. Such effects can be considered to simply increase the average delay for each configuration load, and thus should not significantly impact the accuracy of the results. We consider a very wide range of reconfiguration overheads, from 10 cycles to 10,000 per reconfiguration. This delay range should cover most systems that are likely to be constructed, including the very long delays found in current systems, as well as very short delays that might be achieved by future highly cached architectures [Li 02].

The remaining component is the prefetch insertion program. This algorithm decides where prefetch instructions should be placed given the RFUOPs determined by the RFU picker and the control flow graph of the program.

5 Optimal Prefetch

In order to measure the potential prefetching has to reduce reconfiguration overhead in reconfigurable systems, we have developed the *Optimal Prefetch* concept. Optimal Prefetch represents the best any prefetch algorithm could hope to do given the architectural assumptions and choice of RFUOPs presented earlier.

In Optimal Prefetching we assume that prefetches occur only when necessary, and occur as soon as possible. Specifically, whenever an RFUOP is executed we determine what RFUOP was last called. If it was the same RFUOP it is assumed that no PREFETCH instructions occurred in between the RFUOPs since the correct RFUOP will simply remain in the coprocessor, requiring no reconfiguration. If the last RFUOP was different than the current call, it is assumed that a PREFETCH operation for the current call occurred directly after the last RFUOP. This yields the greatest possible overlap of computation with reconfiguration.

In Table 1 we present the results of Optimal Prefetch in systems with reconfiguration delays of 10 to 10,000 cycles per reconfiguration. This delay range should cover most systems that are likely to be constructed, including the very long delays of current systems, as well as the very short delays that might be achieved by future highly cached architectures. It is important to realize that the reductions in reconfiguration delay shown represent only an upper bound on what is possible within the architecture described in this paper. It is unlikely that any actual prefetching algorithm will be able to achieve improvements as the Optimal Prefetch technique suggests.

6 Prefetching Factors

Since most computations have very complex control flows, with multiple execution paths branching from each instruction to multiple configurations, the correct prediction of the next required configuration becomes an

extremely difficult task. Furthermore, in a system where it takes hundreds or thousands of cycles to load a configuration, we need to make decision to prefetch the required configuration far in advance of its use.

The first factor that can affect the prefetch decision is the distances from an instruction to different configurations. This factor states that the configuration that can be reached in the least number of clock cycles is probably the one that should be the most important to prefetch. From our experiments and analysis, we found that prefetching the closest configuration could work well on applications where the reconfiguration latency is relatively small [Hauck98b]. In this case, the prefetches are not required to be determined as far in advance which RFUOP to load to hide the entire latency, and the prefetch of the closest configuration will result in correct decision most of the time. The prefetch of the closest configuration will lead to more direct benefit than the prefetch of other configurations, whose latency can be overlapped by later prefetches.

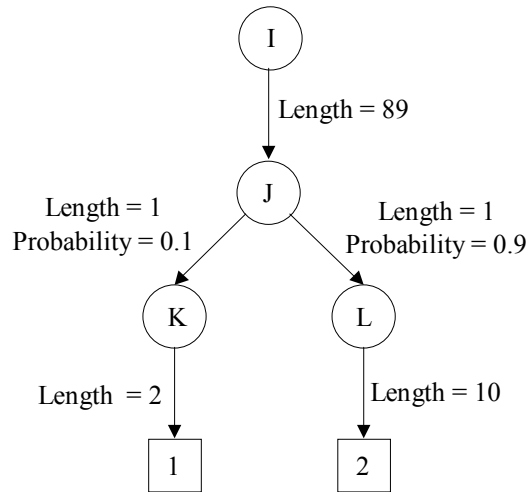


Figure 2. An example for illustrating the ineffectness of the directed shortest path algorithm.

However, for applications with very large reconfiguration latencies, prefetching the closest configuration will not lead to the desired solution because the other factor, the probabilities of reaching each configuration from the current instruction, can have a more significant effect. With large reconfiguration delays, the insertion of PREFETCHs to load one configuration will mean that the reconfiguration latency of other configurations cannot be adequately hidden. Consider the example in Figure 2, where the instruction nodes are represented by circles and the RFUOPs are represented by squares. The “Length” and the “Probability” next to an arrow represents the number of instruction cycles and the probability to execute that path, respectively. If the closest configuration is required to be prefetched, a PREFETCH of Configuration 1 will be inserted at Instruction I and a PREFETCH of the Configuration 2 will be inserted at Instruction L. All reconfiguration latency will be eliminated if we assume it takes 10 cycles to load each configuration because the path from Instruction L to Configuration 2 is long enough to hide the reconfiguration latency of Configuration 2. However, if we assume that the reconfiguration latency of each configuration is 100 cycles, then the majority of the latency of Configuration 2 still remains. Furthermore, although

90% of the time Configuration 2 will be executed once Instruction I is reached, the long path from Instruction I to Instruction J cannot be utilized to hide the reconfiguration cost 90% of the time. However, if Configuration 2 is prefetched at Instruction I, the rate of correct prediction of the next configuration improves from 10% to 90%, and thus the path from Instruction I to Instruction J can provide more benefit in reducing the overall reconfiguration cost.

As can be seen in Figure 2, the reconfiguration latency is also a factor that affects the prediction of the next required configuration. With the reconfiguration latency changed from 10 cycles to 100 cycles, the determination of the next required configuration at Instruction I has changed. For the prefetching model we consider, in order to have a correct prediction of the next required configuration all three factors—the probability, the distance, and the reconfiguration latency—need to be considered. Failing to do so will lower the efficiency of configuration prefetching.

7 Cost Function

Given the branch probabilities and control flow graph, we need to create a general algorithm to predict the next required configuration for a given reconfiguration latency. Considering that the processor must stall if the insertions of PREFETCH operations cannot completely hide the reconfiguration latency, we can use the number of cycles stalled as a cost function for a given prefetch. Minimizing the overall reconfiguration cost by inserting prefetch instructions is our goal.

For paths where only one configuration can be reached the determination of the next required configuration is trivial, since simply inserting a PREFETCH of the configuration at the top of the path is the obvious solution. The problem becomes more complex for paths where multiple configurations can be reached. We call these paths shared paths. Inserting a PREFETCH of any single configuration at the top of a shared path will prevent this path to be used to hide the reconfiguration latency for any other configurations. Before further discussing the reconfiguration cost calculation and the determination of the next required configuration, we first make the following definition.

Definition 1: For an instruction i and a configuration j that can be reached by i , we define:

P_{ij} : the probability of i to reach j

D_{ij} : the distance from Instruction i to the Configuration j

C_{ij} : the potential minimum cost if j is prefetched at the Instruction i

P_{ij} and D_{ij} are calculated by using profile information provided by our simulator, and C_{ij} is calculated based on P_{ij} and D_{ij} . Since the prefetches should be performed only at the top node of a single entry and single exit path, but not at other nodes contained in that path, only the top node is considered as a candidate position for PREFETCH insertion. Therefore, we simplify the control graph by replacing other nodes in the path by an arrow with a length that represents the number of nodes contained in that path. Since for most current systems the reconfiguration time is fixed, we define the latency to load a configuration as L .

We first start our cost calculation on the basic case that is shown in Figure 3. There are two ways to insert prefetches in this example. One is to prefetch *Configuration 1* at instruction i and prefetch *Configuration 2* at instruction m. The other is to prefetch *Configuration 2* at instruction i and prefetch *Configuration 1* at instruction k. The decision is made depending on the calculation of the overall reconfiguration cost resulting from each prefetching sequence, which is affected by the factors of the probability, the distance and the reconfiguration latency. The reconfiguration cost of each prefetching sequence is calculated as follows:

$$C_{i1} = P_{i1} \times (L - D_{i1}) + P_{i2} \times (L - D_{m2}) = P_{i1} \times (L - Z - X - 2) + P_{i2} \times (L - Y)$$

$$C_{i2} = P_{i2} \times (L - D_{i2}) + P_{i1} \times (L - D_{k1}) = P_{i2} \times (L - Z - Y - 2) + P_{i1} \times (L - X)$$

Since the cost of each prefetch cannot be negative, we modify the functions as:

$$C_{i1} = P_{i1} \times \max(0, (L - Z - X - 2)) + P_{i2} \times \max(0, (L - Y)) \quad (1)$$

$$C_{i2} = P_{i2} \times \max(0, (L - Z - Y - 2)) + P_{i1} \times \max(0, (L - X)) \quad (2)$$

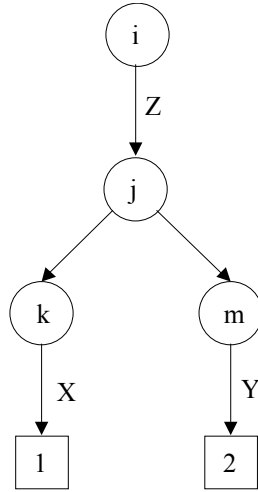


Figure 3. The control flow graph for illustrating the cost calculation.

C_{i1} and C_{i2} are the potential minimum costs at instruction i with different prefetches performed. Each sequence composed of the costs of C_{31} and C_{42} and the benefit resulted by prefetching either configuration on the shared path from instruction i to the instruction j. As can be seen from (1) and (2), the costs of upper nodes can be calculated by using the costs of lower nodes. Therefore, for more complex control graphs we can apply a bottom-up scheme to calculate the potential minimum costs at each node, with upper level nodes using the results calculated at the lower level nodes.

8 The Bottom-up Algorithm for Prefetching

The bottom-up algorithm works on the control flow graph with the edges from RFUOPs to their successors removed. The algorithm starts from the configuration nodes, calculating the potential minimum costs at each instruction node once the costs of children nodes are available. This scheme continues until a configuration node or

the beginning of the program is reached. Once finished, the top most nodes will contain a series of costs reflecting the different prefetch sequences. The sequence with the minimum cost represents the prefetch sequence that has the best potential to hide the reconfiguration overheads. Before presenting the algorithm, we will first discuss the information that must be calculated and retained at each instruction node.

From (1) and (2), we can see that the length of the paths are important to the generation of the best prefetch sequences. If the shared path in Figure 3 is not long enough to hide the remaining latency of both configurations, then by subtracting (2) from (1) we will have:

$$\begin{aligned} C_{i1} - C_{i2} &= P_{i1} \times (L - Z - X - 2) + P_{i2} \times (L - Y) - (P_{i2} \times (L - Z - Y - 2) + P_{i1} \times (L - X)) \\ &= P_{i2} \times (Z + 2) - P_{i1} \times (Z + 2) \quad (3) \end{aligned}$$

As can be seen in (3), for the case the configuration with largest probability should be prefetched. However, this may not be true for the case that the shared path is long enough to hide the remaining reconfiguration latency of at least one configuration. Suppose in Figure 3 that X is much longer than Y, and by prefetching Configuration 1 at Instruction i the entire reconfiguration latency of Configuration 1 can be eliminated. Then by subtracting (2) from (1), we will have:

$$C_{i1} - C_{i2} = P_{i2} \times (Z + 2) - P_{i1} \times (L - X) \quad (4)$$

As can be seen in (4), the probability is not the only factor in deciding the prefetches, the length of each path for an instruction to reach a configuration also affects the way to insert prefetches. In control flow graph, for any two nodes connected by an edge, we call the node that the edge points from parent node and the node that the edge points to the child node. Notice that although any pair of parent node and child node connected by a shared edge can reach same set of configurations, the parent node may not agree the prefetches determined at the child node. This because the configuration that generates the minimum cost at the parent node may not be the one that results in the minimum cost at the child node. For example in Figure 3, we assume that the $X, Y, Z, L, P_{i1}, P_{i2}$ equal 90, 10, 80, 100, 0.6, 0.4 respectively. When we perform a simple bottom-up scheme, the best configuration to prefetch at Instruction j is Configuration 1 and the configuration to prefetch at instruction i is 2. By combining all of these observations a new bottom-up prefetching algorithm can be developed.

In order to perform the bottom-up algorithm, at each instruction node the cost to prefetch each reachable configuration is calculated. The costs will be used to determine the prefetch at current node and the cost calculation at the parent nodes. The basic steps of the bottom-up scheme are outlined below:

For each instruction node i, set C_{ij}, P_{ij}, D_{ij} and num_children_searched to 0 for any reachable configuration j.

1. For each configuration node i, set C_{ii}, P_{ii} and D_{ii} to 1. Place configuration nodes into a queue.
2. While the queue is not empty, do
 - 2.1. Remove a node k from the queue, if it is not a configuration node, do
 - 2.1.1. CALCULATE_COST(k)

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

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

Before we present the details of subroutine CALCULATE_COST(k), we must define some terms.

B_{ij} : the branch probability for instruction i to reach instruction j .

$Min_cost(i)$: the minimum cost achievable by prefetching certain configuration at the instruction i .

Now we outline the basic steps of the subroutine CALCULATE_COST(k):

1. For each configuration j that can be reach by k , do

1.1. For each children node i that can reach the configuration j , do

1.1.1. $Temp_probability = B_{ki} \times P_{ij}$

1.1.2. $P_{kj} = P_{kj} + Temp_probability$

1.1.3. $Temp_length = Temp_length + \min(Latency, D_{ij} + D_{ki}) \times Temp_probability$

1.1.4. $Temp_cost = Temp_cost + C_{ij} - Temp_probability \times \max(0, Latency - D_{ij} - D_{ki})$

1.2. $D_{kj} = Temp_length / P_{kj}$

1.3. $C_{kj} = C_{kj} + Temp_cost$

1.4. For each children node i that cannot reach the configuration j , do

1.4.1. $C_{kj} = C_{kj} + Min_cost(i)$

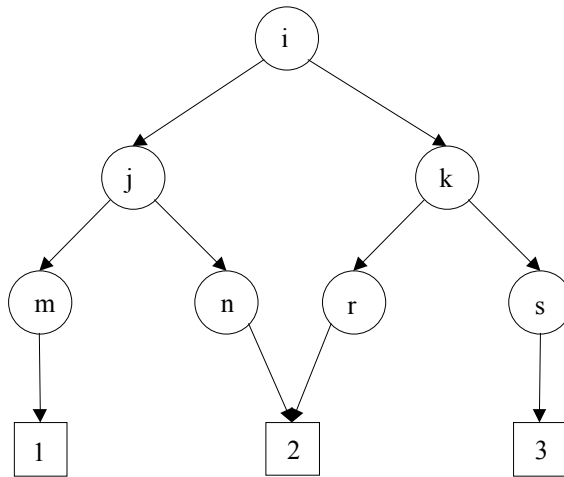


Figure 4. An example that multiple children nodes reach the same configuration.

The function call CALCULATE_COST(k) at instruction k is to calculate probability, distance, and cost of k to prefetch each reachable configuration j . The probability is calculated based on the probabilities of children nodes to reach j and the branch probability of k to reach each child node. Since the control flow graph can be very complex, there may exist several different paths for an instruction to reach a configuration. Therefore, we calculate the weighted average length of these paths. This could cause some inaccuracy in the cost calculation, but in most of the cases the inaccuracy is tolerable because the path with the high probability dominates the weighted average length. Cost calculation considers both probability and length of k to reach configuration j and the costs computed at the

children nodes. For example in Figure 4, node j and k are the two children nodes of i and C_{i3} is being calculated. C_{i3} is the summation of the costs of two children nodes subtracted by the costs can be reduced on edges (i, j) and (i, k) . However, since k is the only node to reach 3, we only calculate the cost reduction on edge (i, k) , and C_{i3} is the summation of the reduced cost from k and $\min_cost(j)$. For the case that multiple children can reach a same configuration, the cost of the parent is consisted of the reduced cost from those children. Consider the example in Figure 4, both node j and node k can reach Configuration 2. C_{i2} is the summation of the reduced costs of C_{j2} and C_{k2}

9 Loop Detection and Conversion

Our bottom-up algorithm requires loop detection and conversion. With loops existing in the control flow graph, deadlock situation will occur when a child node is also the ancestor of that node. In addition, inserting prefetches in the loop may not reduce the overhead but increase it. This because the same PREFETCH instruction is executed multiple times in the loop, wasting CPU cycle. For example in Figure 5 (a), inserting PREFETCH of either configuration in the loop (contains instructions j and m) results one waste instruction.

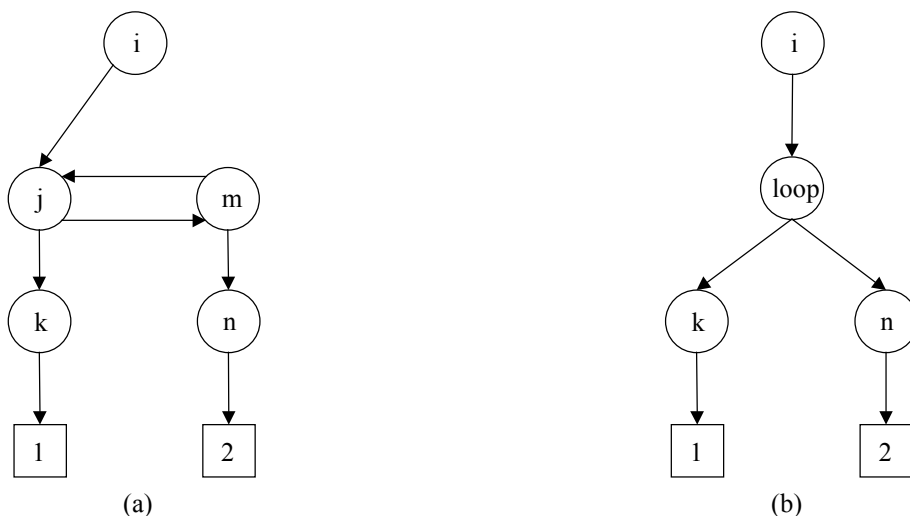


Figure 5. Loop conversion. The original control flow graph with a loop containing nodes j and m is shown on the left, while at right they are replaced by a dummy loop node.

In this work, we use a strongly connected components algorithm to identify the loops existing in the control flow and then convert the loops into dummy nodes in the graph. The strongly connected components algorithm is a standard method to identify a set of nodes in the graph such that every node in the set can reach to each other. In our model, a nested loop can be viewed as a set of strong connected components, in which every node in the loop can reach each other. Furthermore, in order to perform the cost calculation, information that necessary at the dummy nodes, including the length of the loop, the probability to reach each loop exit node needs to be computed. The basic steps of loop detection and conversion are as following:

1. Run the strongly connected component algorithm.
2. For each loop, do
 - 2.1. Compute the summation of the number of executions of all the nodes
 - 2.2. Calculate the total number of executions of paths exiting the loop.
 - 2.3. Divide the value calculated from 2.1 by the value of 2.2, producing the average length of the loop.
 - 2.4. For each path exiting the loop, do
 - 2.4.1. The branch probability of the path is calculated as the execution of the path divided by the total number of executions of paths exiting the loop.

The execution information used in the algorithm can be gathered from the profile information provided by our simulator. Step 2.2 calculates the total control flow that exits from the loop. Dividing this number by the total number of executions of all loop nodes, the average length of the loop nodes can be calculated. Figure 5 (b) shows the loop containing instructions j and m in Figure 5 (a) is converted to a dummy instruction node.

10 Prefetch Insertion

Once the cost calculation is complete, the prefetch operations need to be inserted in the control flow graph. Note that parent nodes and children nodes can require same configuration and thus the same prefetch. However, the redundant prefetches at children nodes will not reduce the reconfiguration penalty but represent a significant overhead. Therefore, only the necessary prefetches are required.

The prefetch insertion is performed in a top-down style starting from the nodes that have no parents. The prefetch of each of these nodes is determined by the minimum cost calculated. Once the PREFETCH instruction is inserted, each node will pass the prefetch information to its children. Upon receiving the prefetch information, each child checks whether it can reach the configuration that is prefetched by the parents. If so, no prefetch is inserted and the prefetch information is passed down. For each of the nodes that cannot reach the configuration prefetched by the parents, the prefetch to the configuration that was calculated with the minimum cost is inserted and this information is passed to its children. This top-down method continues until all the instruction nodes are traversed.

11 Results and Analysis

The simulation results are shown in Table1. Each benchmark is tested at four different per-reconfigurationdelay values. The different reconfiguration delays results different number of RFUOP picked by out simulator. The “No Prefetch”, “Optimal Prefetch” and “Prefetch” columns report the total number of cycles spent stalling the processor while the coprocessor is reconfigured, plus the number of cycles spent on PREFETCH opcodes. The “Opt/No”, “Pre/NO” and “Pre/Opt” columns list the ratios of Optimal Prefetching delays to No Prefetching delays, of delays of out Prefetching algorithm to No Prefetching and of delays of our Prefetching algorithm to Optimal Prefetching. The “Geo. Mean” rows calculate geometric means of the ratios across the entire benchmark suite.

As can be seen in Table 1, the prefetching algorithm provides an overall 43%-75% reduction in reconfiguration overhead when compared to the base case of no prefetching. While this is not nearly as good as the 72%-98% improvement suggested by the Optimal Prefetch technique, it is important to realize that the Optimal Prefetch numbers may not be achievable by any static configuration prefetch algorithm. More specifically, if there exists a very long shared path for an instruction to reach multiple configurations, then static prefetch approaches will not gain much since the PREFETCH instruction inserted on the top of the shared path will only reduce the overhead of one configuration. If the probability to reach one configuration is significantly greater than the combined probability to reach rest configurations, the static approaches may not suffer too much compared with the Optimal Prefetch. However, if the probabilities for an instruction to reach different configurations are almost identical, the performance of static prefetch approaches will be much worst than that of the Optimal Prefetch, which can dynamically issue PREFETCH on the top of the shared paths. Consider the benchmark “Perl” with Latency of 10,000, only 2 prefetches are needed for 2 configurations by using Optimal Prefetch, issuing one at the beginning of the execution and issuing another one after the first RFUOP is finished. However, since the two configurations engage a long shared path, no static approach can perform nearly as good as Optimal Prefetch. Furthermore, as can be seen in Table 1, the shared path is more critical when the Latency is high. When the Latency is small, the shared path may not be necessary to hide reconfiguration latency for most configurations. With the increase of the Latency, it is more likely that the shared path is necessary to hide overhead for any configurations.

12 Conclusions

In this paper we have introduced the concept of configuration prefetch for reconfigurable systems. By adding instructions into the code of an application, configurations for a reconfigurable coprocessor can be loaded in advance. This allows the overlapping of computation and reconfiguration, reducing the reconfiguration overhead of reconfigurable systems. We have also developed an algorithm which can automatically determine the placement of these prefetch operations, avoiding burdening the user with the potentially difficult task of placing these operations by hand. Finally, we have developed the Optimal Prefetch technique, which provides a bound on the potential improvement realizable via configuration prefetch. The results indicate that these techniques can reduce the reconfiguration overhead of reconfigurable systems by 43% to 75%, which will have a direct impact on the performance of reconfigurable systems.

We believe that such techniques will become even more critical for more advanced reconfigurable systems. When one considers techniques such as partial Run-Time Reconfiguration [Hutchings95] or multiple contexts, this greatly increases the amount of computation available to overlap with the reconfiguration, since prefetching can be overlapped with other computations in the reconfigurable logic. We plan to explore the application of prefetching to such advanced systems in our future work.

Benchmark	Latency	No Prefetching	Optimal Prefetch (Opt/No)	Prefetch (Pre/No)	(Pre/Opt)		
Go	10	6,239,090	2,072,560	33.2%	2,738,360	43.9%	132.1%
	100	6,860,700	1,031,739	15.0%	3,064,593	44.7%	289.8%
	1,000	2,520,000	225,588	9.0%	932,476	37.0%	413.6%
	10,000	1,030,000	314,329	30.5%	534,400	51.9%	170.8%
Compress	10	344,840	63,403	18.4%	80,183	23.3%	126.5%
	100	127,100	46,972	37.0%	76,157	59.9%	162.1%
	1,000	358,000	289,216	80.8%	297,630	83.1%	102.8%
	10,000	520,000	12,535	2.4%	35,738	6.9%	285.1%
Li	10	6,455,840	958,890	14.9%	1,949,380	30.3%	255.3%
	100	4,998,800	66,463	1.3%	1,640,860	32.8%	2468.8%
	1,000	55,000	21,325	38.8%	39,082	71.1%	183.1%
	10,000	330,000	43,092	13.1%	128,802	39.0%	297.8%
Perl	10	4,369,880	656,210	15.0%	1,673,620	38.3%	240.7%
	100	3,937,600	398,493	10.1%	1,826,453	46.4%	458.2%
	1,000	3,419,000	9,801	0.3%	1,842,425	53.8%	17933.3%
	10,000	20,000	2	0.0%	5,714	28.6%	285700.0%
Fpppp	10	2,626,180	1,415,924	53.9%	1,490,894	56.9%	106.6%
	100	11,707,000	6,927,877	59.2%	7,663,829	66.3%	112.4%
	1,000	19,875,000	5,674,064	28.5%	5,739,582	28.9%	101.4%
	10,000	370,000	4,485	1.2%	135,094	37.6%	3042.6%
Swim	10	600,700	265,648	44.2%	323,458	53.8%	121.8%
	100	10,200	4,852	47.6%	8,250	80.9%	170.0%
	1,000	91,000	79,905	87.8%	87,932	96.6%	109.7%
	10,000	330,000	43,019	13.0%	52,183	15.9%	121.6%
Geo. Mean	10			26.3%		39.2%	153.9%
	100			16.5%		52.8%	320.0%
	1,000			16.6%		56.6%	340.1%
	10,000			2.3%		24.9%	1073.7%

Table 1. Results of the prefetching algorithm.

Acknowledgments

This research has been sponsored in part by a grant from the Defense Advanced Research Projects Agency, and a grant from the National Science Foundation.

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.
- [Chen94] T.-F. Chen, J.-L. Baer, “A Performance Study of Software and Hardware Data Prefetching Schemes”, *International Symposium on Computer Architecture*, pp. 223-232, 1994.

- [Chen97] I.-C. Chen, C.-C. Lee, T. Mudge, "Instruction Prefetching Using Branch Prediction Information", *International Conference on Computer Design*, pp. 593-601, 1997.
- [Cmelik93a] R. F. Cmelik, *Introduction to Shade*, Sun Microsystems Laboratories, Inc., February, 1993.
- [Cmelik93b] R. F. Cmelik, "SpixTools Introduction and User's Manual", SMLI TR93-6, February, 1993.
- [DeHon94] A. DeHon, "DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century", *IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 31-39, 1994.
- [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.
- [Hauck98a] S. Hauck, "The Roles of FPGAs in Reprogrammable Systems", *Proceedings of the IEEE*, Vol. 86, No. 4, pp.615-638, 1998.
- [Hauck98b] S. Hauck, "Configuration Prefetching for Single Context Reconfigurable Coprocessor", *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 65-74, 1998.
- [Hsu98] W.-C. Hsu, J. E. Smith, "A Performance Study of Instruction Cache Prefetching Methods", *IEEE Transactions on Computers*, Vol. 47, No. 5, May 1998.
- [Hutchings95] B. L. Hutchings, M. J. Wirthlin, "Implementation Approaches for Reconfigurable Logic Applications", in W. Moore, W. Luk, Eds., *Lecture Notes in Computer Science 975 - Field-Programmable Logic and Applications*, London: Springer, pp. 419-428, 1995.
- [Kim93] S. B. Kim, M. S. Park, S. H. Park, S. L. Min, H. Shin, C. S. Kim, D. K. Jeong, "Threaded Prefetching: An Adaptive Instruction Prefetch Mechanism", *Microprocessing and Microprogramming*, Vol. 39, No. 1, pp. 1-15, November 1993.
- [Kumar98] S. Kumar, L. Pires, D. Pandalai, M. Vojta, J. Golusky, S. Wadi, B. Van Voorst, M. Agrawal, "Honeywell Adaptive Computing Systems Benchmarking", <http://www.htc.honeywell.com/projects/acsbench/index.html>, 1998.
- [Li02] Z. Li, S. Hauck, "Configuration Prefetching Techniques for Reconfigurable Coprocessor with Relocation and Defragmentation", *ACM International Symposium on Field-Programmable Gate Arrays*, pp 187-196, 2002.
- [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.
- [Razdan94] R. Razdan, *PRISC: Programmable Reduced Instruction Set Computers*, Ph.D. Thesis, Harvard University, Division of Applied Sciences, 1994.
- [Santhanam97] V. Santhanam, E. H. Gornish, W.-C. Hsu, "Data Prefetching on the HP PA-8000", *International Symposium on Computer Architecture*, pp. 264-273, 1997.
- [Smith78] A. J. Smith, "Sequential Program Prefetching in Memory Hierarchies", *Computer*, pp. 7-21, December 1978.
- [Spec95] *SPEC CPU95 Benchmark Suite*, Standard Performance Evaluation Corp., Manassas, VA, 1995.
- [Tse98] J. Tse, A. J. Smith, "CPU Cache Prefetching: Timing Evaluation of Hardware Implementations", *IEEE Transactions on Computers*, Vol. 47, No. 5, pp. 509-526, May 1998.
- [Villasenor96] J. Villasenor, B. Schoner, K.-N. Chia, C. Zapata, H. J. Kim, C. Jones, S. Lansing, B. Mangione-Smith, "Configurable Computing Solutions for Automatic Target Recognition", *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 70-79, 1996.
- [Villasenor97] J. Villasenor, Personal Communications, 1997.
- [Wirthlin95] M. J. Wirthlin, B. L. Hutchings, "A Dynamic Instruction Set Computer", *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 99-107, 1995.
- [Wirthlin96] M. J. Wirthlin, B. L. Hutchings, "Sequencing Run-Time Reconfigured Hardware with Software", *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 122-128, 1996.
- [Wittig96] R. Wittig, P. Chow, "OneChip: An FPGA Processor with Reconfigurable Logic", *IEEE Symposium on FPGAs for Custom Computing Machines*, 1996.

[Zucker98] D. F. Zucker, R. B. Lee, M. J. Flynn, "An Automated Method for Software Controlled Cache Prefetching", *Hawaii International Conference on System Sciences*, Vol. 7, pp. 106-114, 1998.