ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 1998.

Configuration Prefetch for Single Context Reconfigurable Coprocessors

Scott Hauck
Department of Electrical and Computer Engineering
Northwestern University
Evanston, IL 60208-3118 USA
hauck@ece.nwu.edu

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 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 [Hauck97a]. 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 [Hauck97a]. 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 a 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 idlewhile 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 71% [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 75% 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, developing techniques for 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) in the system during a single computation. In current systems the computation is allowed to run until a configuration that is 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 on the actual computation.

A simple method to reduce or eliminate 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, Hauck97b] 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 hundreds or 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 actual 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.

Note that prefetching has already been used successfully in other domains. Standard processors can use prefetching to load data into the processor's caches, or load data from disk into the processor's memory. However, the demands of configuration prefetching are quite different than those of other prefetching domains. In the case of prefetching data from disks into memory, or from memory into the processor's cache, the system can look for regular access patterns in order to predict the next required data. For configurations, the calling pattern will be extremely irregular. Because of this, new algorithms for determining how to best perform prefetching in reconfigurable systems must be developed in order to make this a viable approach to reducing the reconfiguration overhead.

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 simple model of a reconfigurable system that can allow us to experiment with configuration prefetching. 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. To the best of our knowledge, this is the first configuration prefetch algorithm developed for reconfigurable computing.

3 Reconfigurable System Model

In order to explore the potential of configuration prefetching, we will assume a reconfigurable computing architecture similar to that of the PRISC system [Razdan94]. This system will allow us to easily measure the benefits of configuration prefetch, while representing one of the most difficult systems for which to develop prefetching algorithms. In our experiments, we assume that the reconfigurable computing system consists of a standard microprocessor coupled with a reconfigurable coprocessor. This coprocessor is capable of implementing custom instructions for arbitrary computations. While the coprocessor can support multiple configurations for a given application, we assume that it is only capable of holding one computation at a time. In order to use the reconfigurable coprocessor to compute a different computation a new configuration must be loaded, which takes a fixed latency before it is ready for operation. The actual reconfiguration latency will be varied in our experiments to demonstrate the sensitivity of prefetching to reconfiguration latency, yet each individual experiment will have a fixed latency for all reconfigurations.

In normal operation, the processor executes instructions until a call to the reconfigurable coprocessor is found. These calls to the reconfigurable coprocessor (RFUOPs) contain the ID of the configuration required to compute the desired function. At this point, the coprocessor checks to see if the proper configuration is loaded. If it is not, the host processor is stalled while the configuration is loaded. Once the configuration is loaded (or immediately if the proper configuration was already present), the reconfigurable coprocessor executes the desired computation in a single clock cycle. Once a configuration is loaded it is retained for future executions, only being unloaded when some other 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,

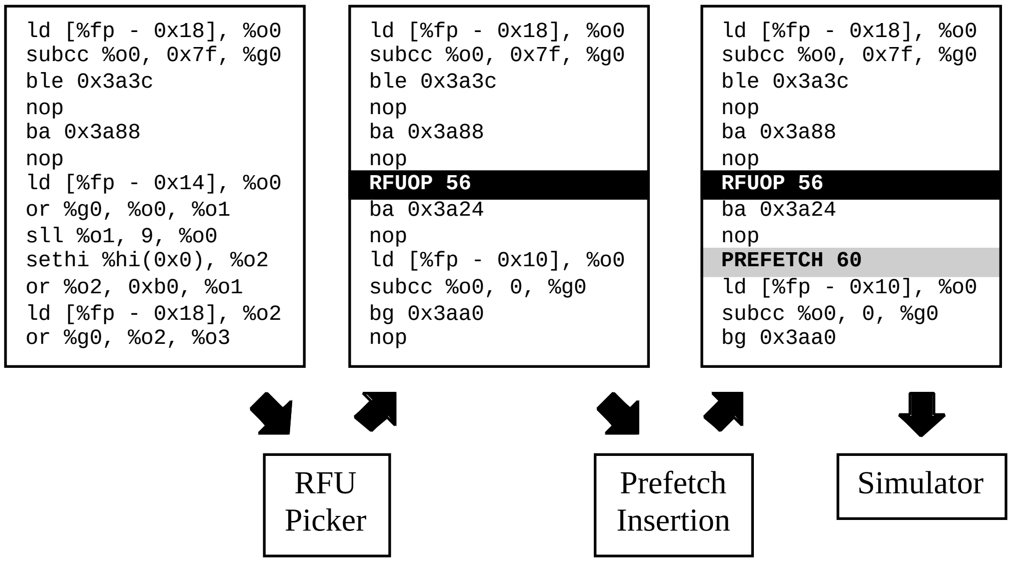

Figure 1. Experimental setup for the prefetch tests. Source code is augmented with calls to the reconfigurable coprocessor (RFUOPs) by the RFU picker. This code then has prefetch instructions inserted into it. The performance of a given set of RFUOPs and PREFETCHes is measured by the simulator.

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 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 code sequences must not have any sequential dependencies. This is enforced by requiring that the code sequences mapped to the reconfigurable coprocessor appear sequentially in the executable, have a single entry point and a single 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, Wittig96, Hauck97b].

4 Experimental Setup

In order to investigate the impact of prefetching on the reconfiguration overhead in reconfigurable systems, we have tested prefetching on some 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. Also, applications of reconfigurable systems are tailored to a specific system, and can be carefully optimized in reaction to a specific reconfiguration overhead. These applications may change significantly if they were mapped to a system with a much higher or lower reconfiguration delay, with different portions of the source code mapped to the reconfigurable logic. 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 paragraphs that follow.

The first step in the experiments 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, in this paper we will 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. This ensures that only combinational code sequences are considered. This is a somewhat conservative assumption, since in many reconfigurable systems it is possible to implement loops and other sequential control flow operations in the reconfigurable logic.

One complexity in deciding which portions of the source code should be mapped to the reconfigurable logic is to find that set of mappings that provide the best performance improvement in the face of a potentially substantial delay for each reconfiguration. In general this is a complex problem, and one we do not attempt to solve here. Our solution is to simply find all potential mappings to the reconfigurable logic, and then simulate the impact of including each candidate. This is done by repeatedly calling the reconfigurable system simulator, and assuming optimal prefetching (both of which are described later in this paper). Our algorithm then greedily chooses the candidate which provides the best performance improvement, and retests the remaining candidates. These retests examine the impact of including any one candidate in with the already chosen candidates. This repeats until the simulator determines that there is at most a potential 1% improvement available in 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. The result of this operation is to create a file that specifies which portion of the source executable should be mapped into RFUOPs, and which can be given to the simulator to compute the delays seen in the target reconfigurable system.

The simulator we have developed takes in an executable for a Sun SPARCstation, a specification of the location of RFUOPs and PREFETCH instructions in the executable, and a parameter that specifies the number of cycles it takes to reconfigure the coprocessor. This simulator is developed from the SHADE simulator [Cmelik93a]. This allows us to track the cycle-by-cycle operation of the system, and get exact cycle counts. Note that only one program can be executed at a time, and operating system calls are not instrumented, so context switch effects and the potential to overlap reconfiguration with cycles in the operating system are not considered. 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.

The remaining component of the experimental setup is the prefetch insertion program. This algorithm decides where prefetch instructions should be inserted into the executable given the set of RFUOPs determined by the RFU picker. The specific algorithm used will be described in a later section. The prefetch insertion program takes in the specification of RFUOPs from the RFU picker, as well as a control flow graph for the executable, and produces a file for the simulator that specifies the PREFETCH locations. Note that in a production system both the RFU picker and the prefetch insertion program would directly modify the executable. However, in order to allow us to use a standard processor simulator to simulate the reconfigurable system this information is maintained in a separate file.

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, instead of choosing specific locations for prefetch operations we assume that prefetch operations occur only when necessary, and occur as soon as possible. Specifically, whenever an RFUOP is encountered in the code, 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.

```
...
03a28 RFUOP 56
      ?? PREFETCH ??
03a2c ble 0x3a28 ! Branch to RFUOP 56
03a30 nop
03a34 ld [%fp - 0x10], %o0
03a38 subcc %o0, 0, %g0
03a3c ble 0x3a60 ! Branch beyond RFUOP 60
03a40 nop
03a44 RFUOP 60
...
```

Figure 2. Example of the optimism of the Optimal Prefetch technique. Once RFUOP 56 in line 3a28 is executed, there are multiple possible next RFUOPs which might be encountered. If the branch at 3a2c is taken, RFUOP 56 is executed again, and no intermediate prefetch cycle occurs. If neither the branch at 3a2c nor at 3a3c is taken, RFUOP 60 is the next to occur, and it is assumed that a PREFETCH 60 occurs right after the call of RFUOP 56, overlapping 6 cycles of computation with the reconfiguration. No fixed prefetching scheme could achieve both results for the code sequence shown.

It is important to realize that the Optimal Prefetch technique, while providing a bound on the potential of prefetching, potentially produces results better than what could possibly be

Benchmark	Latency	No Prefetching	Optimal Prefetching	Ratio
Go	10	6,239,090	2,072,560	33.2%
	100	6,860,700	1,031,739	15.0%
	1,000	2,520,000	225,588	9.0%
	10,000	1,030,000	314,329	30.5%
Compress	10	344,840	63,403	18.4%
	100	127,100	46,972	37.0%
	1,000	358,000	289,216	80.8%
	10,000	520,000	12,535	2.4%
Li	10	6,455,840	958,890	14.9%
	100	4,998,800	66,463	1.3%
	1,000	55,000	21,325	38.8%
	10,000	330,000	43,092	13.1%
Perl	10	4,369,880	656,210	15.0%
	100	3,937,600	398,493	10.1%
	1,000	3,419,000	9,801	0.3%
	10,000	20,000	2	0.0%
Fpppp	10	2,626,180	1,415,924	53.9%
	100	11,707,000	6,927,877	59.2%
	1,000	19,875,000	5,674,064	28.5%
	10,000	370,000	4,485	1.2%
Swim	10	600,700	265,648	44.2%
	100	10,200	4,852	47.6%
	1,000	91,000	79,905	87.8%
	10,000	330,000	43,019	13.0%
Cumulative	10			26.2%
	100			16.6%
	1,000			16.5%
	10,000			2.3%
	All			11.4%

Table 1. The results of Optimal Prefetch on the benchmark programs. Each benchmark is tested at four different per-reconfiguration delay values. The "No Prefetch" and "Optimal 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 ratio column lists the ratio of Optimal Prefetching delays to No Prefetching delays. "All" is the average of all benchmarks at all reconfiguration delays considered.

done by an actual prefetching algorithm. As shown in Figure 2, Optimal Prefetching may assume that a prefetch instruction occurs at a given point in the code during some portions of the execution, while the same location does not contain a prefetch at other times. However, the bound provided by Optimal Prefetching is useful to demonstrate the limits of configuration prefetching.

As can be seen in Table 1, optimal prefetching has the potential to significantly reduce reconfiguration times. This ranges from an average factor of almost 4 for reconfigurable systems with a 10 cycle reconfiguration delay, to a factor of almost 44 for systems with a reconfiguration delay of 10,000 cycles. Averaged across the reconfiguration delays considered, this produces a reduction in reconfiguration delay of 88.6%, or a factor of almost 9.

It is important to realize that the reductions in reconfiguration delay shown in Table 1 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 quite as good as the Optimal Prefetch technique suggests. In the next section, we will present an algorithm for configuration prefetch. This algorithm determines

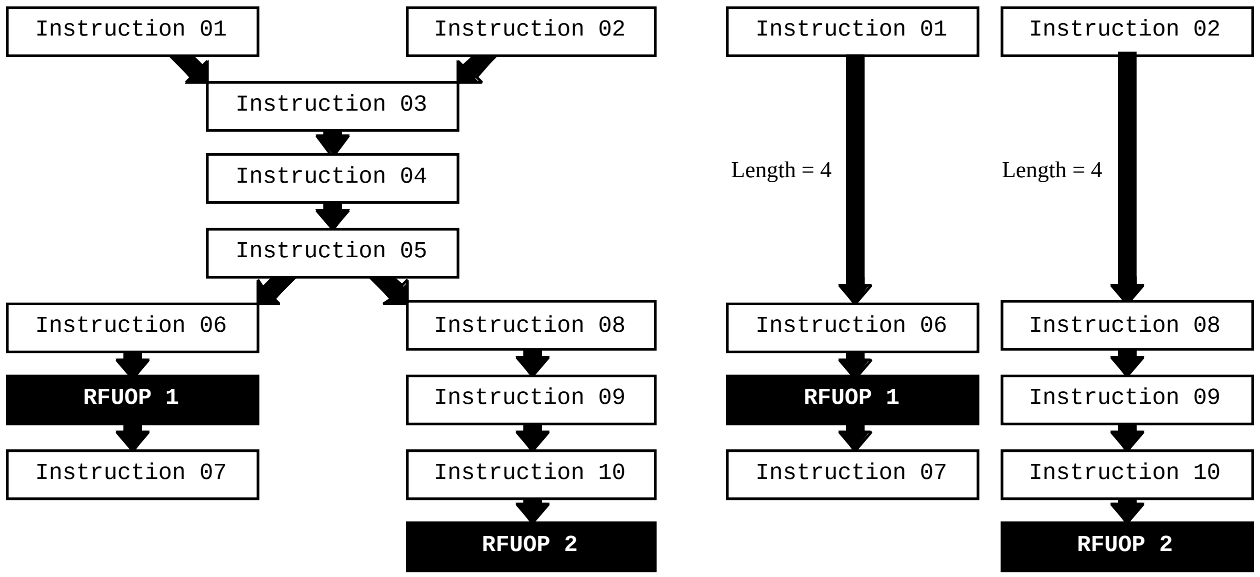

Figure 3. An example for the prefetch insertion algorithm (left), and the same example with the subroutine at instructions 3-5 removed (right).

specific locations where prefetch instructions should be inserted in order to overlap computation with reconfiguration.

6 Prefetch Insertion

In the previous sections we have proposed the concept of configuration prefetch, and have demonstrated that this technique has the potential to significantly reduce the reconfiguration overhead in reconfigurable systems, thus improving the performance of these systems. In this section we detail a specific algorithm which has the capability to realize some of these potential gains.

The challenge in developing a prefetch algorithm is to statically determine which RFUOP is the next to be needed at some point in the code. This decision must be done as far in advance of the RFUOP's execution as possible, so that most or all of the reconfiguration can be overlapped with useful computations. However, the earlier the PREFETCH operation occurs the more complicated the control flow between the PREFETCH and the RFUOP, increasing the likelihood that the wrong configuration will be loaded. In fact, from a given point in the code there may be many different RFUOPs that might occur next, since subsequent branches may lead to many different RFUOPs. Thus, at best we can hope to make an educated guess as to what configuration should be loaded, hoping that on average this prefetch will reduce the reconfiguration overhead as much as possible.

Our prefetch insertion algorithm starts with a control flow graph for the benchmark being considered. This graph contains information on the potential execution paths within the program, and thus forms the basis for determining which RFUOP will occur next. In a production system this control flow graph would be extracted from the source code. In our experimental system we construct the control flow graph from the executable via information provided by the SpixTools [Cmelik93b] code profiler, as well as some additional information from the simulator. The insertion algorithm also takes the locations of RFUOPs produced by the RFU picker.

The basis of our algorithm is a directed shortest-path algorithm on the control flow graph, starting from each RFUOP location. This is based upon the belief that the RFUOP that can be reached in the least number of clock cycles is the RFUOP configuration that should be loaded. We determine for each instructions in the executable which RFUOP can be reached in the shortest number of steps. Note that we only consider forward arcs in the control flow graph from an instruction to an RFUOP, or alternatively backward edges from RFUOP to preceding instructions, since this corresponds to the direction of control flow. This closest RFUOP is assumed to *own* the instruction, in that we will insert PREFETCH operations such that that RFUOP will either be present in the coprocessor when that instruction is executed, or will begin prefetching it at this time.

Once we determine which RFUOP owns each instruction, we have broken the code into *ownership regions*, where each region represents the portion of the executable's instructions owned by a given RFUOP. For example, in the code segment in Figure 3 left, instructions 01-06 are in RFUOP 1's ownership region, while instructions 08-10 are in RFUOP 2's ownership region. The next step in our prefetch insertion algorithm is to add PREFETCH operations before any instruction in one ownership region which has a direct predecessor in another ownership region. This PREFETCH operation will prefetch whichever RFUOP owns that instruction. Thus, in Figure 3 left we would insert a single PREFETCH operations, and that would be a prefetch for RFUOP 2 just before Instruction 08. Prefetches for RFUOP 1 would appear somewhere before Instruction 01 and Instruction 02, although their exact placement would depend on the exact control flow. In this way, we have multiple cycles in which to prefetch RFUOP 1, while we will change to prefetching RFUOP 2 once it becomes clear that that is the next RFUOP to occur, which

Benchmark	Latency	No Prefetching	Basic Prefetch	(Bas/No)	Pruned Prefetch	(Pru/No)	(Pru/Opt)
Go	10	6,239,090	3,134,360	50.2%	2,862,128	45.9%	138.1%
	100	6,860,700	4,126,293	60.1%	2,989,912	43.6%	289.8%
	1,000	2,520,000	2,599,305	103.1%	996,300	39.5%	441.6%
	10,000	1,030,000	5,562,593	540.1%	706,611	68.6%	224.8%
Compress	10	344,840	86,284	25.0%	78,284	22.7%	123.5%
	100	127,100	78,821	62.0%	78,821	62.0%	167.8%
	1,000	358,000	311,677	87.1%	311,651	87.1%	107.8%
	10,000	520,000	1,156,939	222.5%	263,213	50.6%	2099.8%
Li	10	6,455,840	2,043,246	31.6%	1,929,195	29.9%	201.2%
	100	4,998,800	3,209,090	64.2%	2,395,041	47.9%	3603.6%
	1,000	55,000	6,898,414	12542.6%	42,082	76.5%	197.3%
	10,000	330,000	150,720	45.7%	150,720	45.7%	349.8%
Perl	10	4,369,880	1,873,472	42.9%	1,579,463	36.1%	240.7%
	100	3,937,600	2,241,365	56.9%	1,965,287	49.9%	493.2%
	1,000	3,419,000	5,616,728	164.3%	2,015,812	59.0%	20567.4%
	10,000	20,000	5,715	28.6%	5,714	28.6%	285700.0%
Fpppp	10	2,626,180	1,505,906	57.3%	1,490,467	56.8%	105.3%
	100	11,707,000	7,660,039	65.4%	7,656,892	65.4%	110.5%
	1,000	19,875,000	11,782,888	59.3%	5,805,461	29.2%	102.3%
	10,000	370,000	79,616,610	21518.0%	350,002	94.6%	7803.8%
Swim	10	600,700	325,339	54.2%	324,589	54.0%	122.2%
	100	10,200	139,174	1364.5%	5,573	54.6%	114.9%
	1,000	91,000	4,004,510	4400.6%	81,265	89.3%	101.7%
	10,000	330,000	41,995,371	12725.9%	56,126	17.0%	130.5%
Cumulative	10			41.8%		38.9%	148.3%
	100			103.3%		53.4%	321.2%
	1,000			411.1%		58.6%	355.2%
	10,000			591.8%		44.0%	1906.5%
	All			180.0%		48.1%	423.8%

Table 2. The results of the prefetching algorithm on the benchmark programs. Each benchmark is tested at four different per-reconfiguration delay values. The "Basic Prefetch" and "Pruned 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 ratio of prefetch to non-prefetch latency is also reported. The final column lists the ratio of Pruned Prefetch to Optimal Prefetch. "All" is the average of all benchmarks at all reconfiguration delays considered.

happens when we branch to Instruction 08. Note that an RFUOP is considered to be owned by itself, and thus if Instruction 07 is in REFUOP 1's ownership region we will not waste a PREFETCH by inserting it before Instruction 07, while if Instruction 07 is owned by some other RFUOP we would insert a PREFETCH for that RFUOP at this location.

There is one refinement to this initial prefetch insertion algorithm that can be important to creating the best prefetching. The issue is that subroutine calls may combine multiple different regions of the control flow graph, creating "false paths". Specifically, imagine that Instructions 03-05 represent a subroutine in the software, called by Instructions 01 and 02. If we use the algorithm just discussed, RFUOP 1 would be considered to own Instruction 02, even though there may be no execution path that would lead from Instruction 02 to RFUOP 1 without passing through some other RFUOP. The solution to this is simple: we replace most subroutine calls in the control flow graph with control flow edges from the instruction just before the subroutine call to the corresponding instruction just after the call, and this edge has a "length" (used in the shortest path algorithm) equal to the shortest execution path through that subroutine. Thus, if

Instructions 03-05 in Figure 3 left were in fact a subroutine, we would remove these instructions, and replace them with a control flow arc from Instruction 01 to Instruction 06, and another arc from Instruction 02 to Instruction 08, with both of them having a length of 4 (Figure 3 right). Normal arcs have a length of 1. In this way, Instruction 01 would be owned by RFUOP 1, and Instruction 02 would be owned by RFUOP 2, giving each of them a much longer time to prefetch their configurations without sacrificing any accuracy in the prefetching decisions.

In order to do this simplification of the control flow graph we classify procedures as *pure* or *impure*. Any subroutine that does not contain an RFUOP, and does not call any impure subroutines, is considered *pure*. All others are considered *impure*. This distinction is important, because we do not want to remove any impure subroutines from the control flow graph. The reason for this is that an impure subroutine will contain RFUOPs which should block the ownership regions of RFUOPs following this subroutine call. For example, assume that Instructions 03-05 in Figure 3 left are a subroutine, and Instruction 04 is in fact an RFUOP 3 instruction. In this case, it should be clear that Instructions 01 and 02 should be owned by RFUOP 3, since that will always be the next RFUOP encountered after these instructions. However, if we remove this subroutine from the control flow graph we would not discover this fact. To deal with this, we only remove *pure* subroutines from the control flow graph, leaving all *impure* subroutines as is. Our algorithm would then properly label Instructions 01 and 02 as being owned by RFUOP3 and prefetch accordingly. The classification of subroutines as *pure* or *impure* can be made by a very simple search of the control flow graph.

As shown in Table 2, the prefetching algorithm as described so far (referred to here as the "Basic Prefetch" algorithm) does a reasonable job of prefetching in most cases, but can do a poor job in others. For example, the Basic Prefetch algorithm reduces the reconfiguration overhead by 58% on average for systems with a reconfiguration delay of 10 cycles, but can in fact increase the reconfiguration delay by a factor of almost 6 for systems with a reconfiguration delay of 10,000 cycles. The problem is that the algorithm sometimes makes poor decisions for some prefetch placements, causing the coprocessor to unload the configuration that is in fact the next one needed in the system. Obviously, something must be done to improve the consistency of the algorithm's results.

Our solution is to use a profiler-based pruning of the prefetch operations. We maintain statistics, on a per PREFETCH operation basis, of whether the outcome of that PREFETCH operation was beneficial or not. In those cases where it begins loading the configuration that is in fact the next RFUOP to be called, we credit it with the number of cycles saved. If it is the first RFUOP to overwrite the configuration that is required next, we reduce it's benefit by the number of cycles the system has to stall while reloading that configuration (note that a subsequent PREFETCH of the proper configuration can reduce this penalty). Finally, we also reduce the PREFETCH's benefit by the total number of times that prefetch operation is executed, since every time a PREFETCH operation is executed the processor must waste a cycle performing this operation. All of these statistics are easy to maintain, and could be reported by techniques similar to those found in prof, gprof, and other program profilers.

The information gathered on a per PREFETCH basis measures the effect this instruction has on the operation of the system. We go through these statistics and remove ("prune") any PREFETCH instruction that has a net loss on the operation of the system. This operation is similar to the performance optimization performed on standard software algorithms, with the added benefit that it can be easily automated, requiring no user intervention. Note that the pruning of one PREFETCH operation can cause another PREFETCH to have a negative impact on the system operation. For example, in between two calls to the same RFUOP there may be two different PREFETCH operations for other RFUOPs. During the first pruning step the first PREFETCH operation would be penalized for unloading the RFUOP, and would be removed. At this point, the second PREFETCH is responsible for overwriting the RFUOP that should have been retained. Our solution is to run the pruning process iteratively, continuing to remove PREFETCH operations that have a negative impact on the system operation. Note that this takes at most a handful of pruning cycles.

As can be seen in Table 2, when we combine our original prefetch insertion algorithm with a pruning step ("Pruned Prefetch"), we get a much more consistent result. This greatly improves the performance of the prefetching algorithm, providing an overall 52% reduction in reconfiguration overhead when compared to the base case of no prefetching. While this is not nearly as good as the 89% 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. With the algorithm described here, we are capable of providing a significant reduction in the reconfiguration overhead of reconfigurable systems. As shown in Table 3, this speedup has a direct impact on the runtime of the reconfigurable system, providing a 10% reduction in overall runtime over the case of no prefetching.

7 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 more than a factor of two, 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

Benchmark	Latency	Basic Prefetch	Pruned Prefetch	Optimal Prefetch
Go	10	82.4%	80.8%	76.4%
	100	89.5%	85.1%	77.6%
	1,000	100.3%	93.8%	90.7%
	10,000	118.5%	98.7%	97.1%
Compress	10	87.4%	87.0%	86.3%
	100	97.9%	97.9%	96.5%
	1,000	98.2%	98.2%	97.3%
	10,000	119.5%	92.1%	84.5%
Li	10	79.2%	78.7%	74.1%
	100	92.6%	89.2%	79.6%
	1,000	134.3%	99.9%	99.8%
	10,000	99.1%	99.1%	98.6%
Perl	10	83.8%	81.9%	75.9%
	100	90.5%	88.9%	80.1%
	1,000	111.6%	92.6%	82.0%
	10,000	99.9%	99.9%	99.9%
Fpppp	10	79.2%	78.9%	77.6%
	100	80.0%	80.0%	76.3%
	1,000	84.0%	72.2%	72.0%
	10,000	205.9%	100.0%	99.5%
Swim	10	84.4%	84.4%	81.1%
	100	106.7%	99.8%	99.7%
	1,000	293.3%	99.5%	99.5%
	10,000	1732.8%	89.3%	88.8%
Cumulative	10	82.7%	81.9%	78.4%
	100	92.5%	89.9%	84.5%
	1,000	124.0%	92.2%	89.6%
	10,000	192.0%	96.4%	94.5%
	All	116.2%	89.9%	86.6%

Table 3. Relative performance numbers for different prefetch techniques. The numbers represent the ratio of the total runtime (execution plus reconfiguration time) under the specified prefetch technique to the delay with no prefetching. "All" is the average of all benchmarks at all reconfiguration delays considered.

techniques such as partial Run-Time Reconfiguration [Hutchings95] or multiple contexts [Bolotski94], 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.

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

[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.

[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.

[Hauck97a] S. Hauck, “The Roles of FPGAs in Reprogrammable Systems”, submitted to *Proceedings of the IEEE*, 1997.

[Hauck97b] 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.

[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.

[Razdan94] R. Razdan, *PRISC: Programmable Reduced Instruction Set Computers*, Ph.D. Thesis, Harvard University, Division of Applied Sciences, 1994.

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

[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.