Cost-Driven Hybrid Configuration Prefetching for Partial Reconfigurable Coprocessor

Ying Chen[1], Simon Y. Chen[2]

[1]School of Engineering
San Francisco State University
1600 Holloway Ave
San Francisco, CA 94132
http://online.sfsu.edu/~yingchen
yingchen@sfsu.edu

[2]DSP Group Inc.
3120 Scott Blvd, Santa Clara, CA 95054
http://www.dspg.com
simon.yingchen@gmail.com

Abstract

Reconfigurable computing systems have developed the capability of changing the configuration of the reconfigurable coprocessor multiple times during the course of a program. However, in most systems the reconfigurable coprocessor wastes computation cycles while waiting for the reconfiguration to complete. Therefore, the high demand for frequent run-time reconfiguration directly translates into higher reconfiguration overhead. Some studies have introduced the concept of prefetching to reduce the reconfiguration overhead. However, these prefetching algorithms are probability-driven. We believe that including configuration size information in the prediction algorithm directly links the training of the predictor with the performance gain. Therefore we proposed a performance-oriented cost-driven algorithm for coarse-grained configuration prefetching. Our cycle accurate simulation results show that the proposed cost-driven algorithm outperforms the probability-driven predictor by 10.8% to 29.6% in reducing reconfiguration overhead.

1. Introduction

Reconfigurable computing systems that consist of a general purpose processor and a reconfigurable coprocessor have been a promising solution to a lot of practical problems [1] [2] [3]. The reconfigurable coprocessor provides high performance, particularly in repetitive logic and arithmetic, while the general purpose processor provides the flexibility in handling all other tasks [4]. As the reconfigurable computing systems evolve over the years, they developed capability of changing the configuration of the reconfigurable coprocessor multiple times during the course of a program. However, their ever-increasing flexibility comes with a higher demand for frequent run-time reconfigurations. This directly translates into higher reconfiguration overhead because in most systems the reconfigurable coprocessor has to wait before the reconfiguration completes, wasting cycles that can be used to perform computations. For example, applications on the DISC system have spent 25% to 71% of their execution time on reconfiguration [4]. It is obvious that the performance of these systems can be improved by reducing the reconfiguration overhead.

Many studies have been focusing on reducing the reconfiguration overhead. These studies include configuration compression [6] and configuration caching [7]. Another promising technique that has been studied is configuration prefetching [5] [4]. This technique targets at absorbing reconfiguration overhead by overlapping the reconfiguration with computation. A prefetching technique is developed in [5] for single context FPGA systems, which are much simpler than most of the modern reconfigurable computing systems where part of the device can be programmed while the rest of the system continues computing so that multiple configurations can be loaded into different sections of the reconfigurable coprocessor. Configuration prefetching techniques for partial reconfiguration is introduced in [4]. Their dynamic prefetching technique uses a Markov chain predictor to predict the next Reconfigurable Unit OPeration (RFUOP). However, an important factor of reconfiguration overhead, the configuration size, was not considered. Without

1-4244-0910-1/07//$20.00©2007IEEE

considering the configuration size, the predictor will make prefetching decision merely based on probability or correlation instead of the real reconfiguration overhead, which is an important metric.

In this study, we propose a cost-driven coarse-grained configuration prefetching algorithm by incorporating configuration size into the prediction decision function. We incorporate the cost-driven algorithm into both Markov prefetcher and Least Mean Square Algorithm (LMSA) prefetcher. We used algorithmic experiments to access the effect of RH capacity, prediction algorithm training step size, configuration size information, and history information on the prefetching error rate. We also modeled the reconfigurable coprocessor access process in a cycle accurate simulator and accessed cost-driven algorithm's efficiency in redoing reconfiguration overhead. Our results from algorithmic Matlab experiments and cycle accurate simulations show that the cost-driven algorithm outperforms the probability-driven predictor by 4 to 29.6% in the cases of different RH capacities in reducing reconfiguration overhead. The most efficient probability-driven predictor is LMSA with history information and the reconfiguration overhead reduction is 10.8 to 29.6%.

The remainder of this paper is organized as follows: Section 2 presents the background of configuration prefetching. In Section 3 we describe the cost-driven hybrid configuration prefetching and related algorithms. The experimental setup is explained in Section 4 with the results shown in Section 5. Finally, Section 6 summaries and concludes.

2. Background

The idea of prefetching is widely used in general purpose computer systems, for both data and instruction. Prefetching is first introduced into reconfigurable systems in [5] where a prefetching technique is developed for single context FPGA systems. Their work demonstrated the potential of configuration prefetching in reducing the reconfiguration overhead. However, it targets at single context FPGA systems, which are much simpler than most of the modern reconfigurable computing systems where multiple configurations can be loaded into different sections of the reconfigurable coprocessor.

Configuration relocation and defragmentation are introduced for partial reconfigurable coprocessor in [8]. They provide a flexible mechanism by which configurations of different functions can be rearranged in the coprocessor configuration space and make room for potentially more configurations. Based on such system, several configuration prefetching techniques were introduced in [4] to reduce the reconfiguration overhead, including static prefetching, dynamic prefetching, and hybrid prefetching. Their study shows that hybrid prefetching algorithm is the most efficient. The dynamic prefetching technique uses a Markov chain predictor to predict the next RFUOP. However, an important factor of reconfiguration overhead, the configuration size, was not considered in the dynamic prefetching algorithm. This potentially affects the effectiveness of the technique because even if an RFUOP has higher probability to be the next RFUOP, it may not present the highest reconfiguration overhead risk if its configuration latency is very small. On the other hand, an RFUOP that takes a long time to load may need more attention prior to its appearance at the top of the RFUOP queue. In this study, we use configuration size as an important metric for prefetching the next configuration.

3. Cost-Driven Hybrid Configuration Prefetching

The size of the RFUOP has significant impact on the prefetching performance. Without considering the configuration size, the predictor will make prefetching decision merely based on probability or correlation instead of the real reconfiguration overhead, which is the meaningful metric. In this study we used the configuration sizes in hybrid prefetching, which includes both static process and dynamic process.

In the static process, the coarse-grained configurations, which will be run on the reconfigurable coprocessor, are chosen, and the configuration sizes are used as part of the training weight for training the optimal step size for prefetching algorithms (e.g., Markov and LMSA). Instead of calculating the probability of missing a needed configuration, the predictor calculates the cost (reconfiguration overhead) associated with missing the needed configuration.

In the dynamic process, configuration sizes are used in the prefetching scheduling process. When the prefetcher prioritizes the configurations in the prefetch queue, it considers the distance (time gap) between the current RFUOP and the RFUOP being scheduled for prefetching. Bigger configuration with short distance is given lower priority since it is likely the prefetching of that configuration will not complete due to short of time, wasting time that can be used to prefetch smaller configuration.

We incorporated the cost-driven method in both Markov predictor and LMSA predictor. Furthermore, we use history information to improve the performance of predictor.

Markov Predictor was first used as a prefetching method between on-chip and off-chip cache [9]. A first-order Markov process was used to simplify the prediction process. Also, a table is used to represents the transition probability to reduce the memory usage. These simplifications are used in the configuration prefetching

studies [4]. However, the number of configurations in a reconfigurable computing system is much smaller than the size of the address space in a general purpose computer, making storing the transition probability matrix a much smaller concern.

The Markov predictor models the configuration execution as a first-order Markov process. A first-order Markov process is a discrete-time stochastic process where state c_k at time k is one of a finite number of states and is determined only by c_{k-1}.

The Markov predictor views each RFUOP as a state. The execution pattern, the sequence that the program executes these RFUOP, is viewed as a sequence of state transition in a Markov process. To predict the next RFUOP as prefetching candidate is to find the transition from the current state to the next state that has the largest probability, where the probability information is obtained dynamically by collecting statistics of the program.

Markov predictor gains the statistical information about the execution pattern dynamically. The process where it collects the statistics is the training process. There are many ways to train the predictor. Conceptually they all serve the same purpose: obtain statistics to evaluate the probability of state transitions. However, they differ subtly in terms of the prediction performance.

In previous work, [4] a simplified mechanism is used to update the state transition probability. For each occurrence of state transition (*u, v*), the probability of state transition (*u, w*) is updated as:

$$\begin{aligned} P_{u,w} &= P_{u,w} / (1+C), w \neq v \\ P_{u,w} &= (P_{u,w} + C) / (1+C), w = v \end{aligned} \quad (3.1)$$

Here *C* is the training step size, or learning rate, which determines how fast the predictor adapts to changes in the execution pattern. When *C* is set to 1 as in [4], this greatly reduces the hardware complexity. In this study we tune the step size and thus find the optimal one to improve the prediction performance.

Least Mean Square Algorithm (LMSA). An alternative to the transition probability used in Markov predictor is to view them as the correlation between the current RFUOP (state) and the next RFUOP. The Least Mean Square Algorithm (LMSA) is widely used in adaptive filters. It quickly learns the correlation between the input and the output when applied to the correlation context. It is also very simple and flexible. The basic rule of adaptively training the LMSA is (tailored for this application):

$$\begin{aligned} w^1 &= 0 \\ w^{k+1} &= w^k + \mu \cdot (d_i - y_i) \end{aligned} \quad (3.2)$$

Here *w* can be viewed as the correlation between two RFUOP; d_i is the desired output of the predictor, e.g., 1 if *i* is the next RFUOP, 0 if *i* is not the next RFUOP; y_i is the output of the predictor, in this case, the product of w_k and x_k; μ is the training step size. Obviously tuning the value of μ does not involve the use of divider in the hardware. μ can even be picked as 2^{-A} where *A* is an integer so the multiplication by μ becomes a simple shift operation.

History Information is useful in prediction. The assumption that the RFUOP execution is a first-order Markov process as mentioned previously serves as a good simplification and approximation. But this is not always true (almost always not true) in real program. Using more than 1 step of the execution history sometimes is useful. Consider the nested loop ((AB)CD) where AB is the inner loop followed by CD in the outer loop. A first-order predictor will always predict B follows A, A and C follows B, D follows C. It will not consider any correlation between B and D, but only that between C and D. If the time gap between C and D is small, there may not be enough time to prefetch D after the prefetcher sees C. A predictor that considers history will know that there is certain probability that D will appear two steps after B.

A straightforward scheme to use history information in Markov predictor is to use higher order of Markov process. Instead of computing $P(c_k \mid c_{k-1})$, it computes $P(c_k \mid c_{k-1}, c_{k-2}, ...)$. This can be implemented as a higher order of state transition probability matrix.

In LMSA, a similar scheme is explored where, instead of learning the correlation between the current RFUOP and the next one, the LMSA tries to predict the next RFUOP based on recent history of RFUOP. For instance, in the case when one extra RFUOP is used, each pair of the $RFUOP_{k-1}$ and $RFUOP_k$ will have a correlation with the $RFUOP_{k+1}$, which is updated every time the pair occurs.

The performance gain obtained by using history information comes with a high cost. In both Markov predictor and LMSA, storing history information increases the memory requirement exponentially. The original methods are used in algorithmic Matlab simulations (Section 5.1). A more practical approach is necessary in the benchmark simulation (Section 5.2) and in real system. It is termed as back annotation. The basic idea is that in training, the predictor not only updates correlation value $C(c_{k-1}, c_k)$, but also $C(c_{k-2}, c_k)$ or even $C(c_{k-3}, c_k)$. The back annotation uses a weight as 2^{-D} where *D* is the distance (in steps) between the 2 RFUOP.

4. Experimental Setup

In this study we conduct two sets of experiments. In the first set of algorithmic experiments, we used Matlab on the benchmark trace data to test the effect of

Reconfigurable Hardware (RH) capacity, training step size, configuration size information, and history information on the prefetching performance. In the second set of experiments we used cycle accurate simulator SimpleScalar3.0 [10] to test the effect of the various cost-driven algorithms over the probability-driven algorithm on reducing reconfiguration overhead by running the real work load benchmark. For both sets of experiments we used SPEC2000 benchmark suite [11].

4.1 Algorithmic experimental setup

Various algorithmic experiments are done in Matlab. They include both Markov predictor and LMSA predictor with different RH sizes, training step sizes, configuration size information, and history information. The choice of Matlab is based on its flexibility and short development cycle. This is ideal for initial exploration of the design space. Some important mathematical tools used in the algorithms are also readily available in Matlab. The tradeoff is that Matlab is not suitable to model details of the programs, particular those involve the timing aspect of the program. It also is too slow to run any real benchmark data.

To gain meaningful and realistic information from Matlab experiments, a set of trace data is obtained by running a real benchmark program, which is 164.gzip from the SPEC2000 benchmark suite in a modified SimpleScalar simulator. According to the benchmark profiling, 15 program blocks in 164.gzip are chosen as coarse-grained RFUOPs for reconfigurable coprocessors. The generated trace data contains real instruction execution pattern and is used as the primary input to the Matlab functions. The configuration size for the 15 RFUOPs are normalized and distribute evenly from 1 to 3.

The performance of the Matlab simulation is evaluated by the error rate of the prediction weighted with the RFUOP configuration size as shown in the following formula:

$$\text{weighted prediction error rate} = \sum_{i=1}^{N} SC_i \qquad (4.1)$$

Here N is the total number of mispredictions, SC_i is the configuration size of *i*th mispredicted RFUOP.

4.2 Cycle-accurate performance simulation setup

Cycle-accurate benchmark simulation is done using SimpleScalar3.0 and a SPEC2000 benchmark program 164.gzip. Each time an assigned RFUOP is executed in the benchmark program, the RFUOP event is captured and sent to the SimpleScalar simulator.

The configurations of the simulated general purpose processor and reconfigurable coprocessor are as the following: a 32KB date L1 cache of 32-byte block size, 2-way associativity and 1 cycle latency; a 32KB instruction L1 cache of 32-byte block size, 2-way associativity and 1 cycle latency; a unified 512KB L2 cache of 64-bytle block size, 4-way associativity and 4-way associativity; 50 cycle main memory access latency; 8 issue rate; a 64-entry load/store queue; 2-level branch predictor with 128 entries; 8 integer ALU's with 1 cycle latency; 2 integer multiplier/dividers with 1 cycle latency; 6 floating point ALU's with 2 cycles latency; 2 floating point multiplier/dividers with 12 cycle latency; 2 cycle latency for each step of defragmentation.

Simulating Access to Reconfigurable Coprocessor. We modified SimpleScalar to simulate the access to reconfigurable coprocessor. At the beginning of each RFUOP execution on reconfigurable coprocessor, the following operations are modeled in the SimpleScalar

- Defragmentation of existing configurations in the RH based on the algorithm in [8]
- Execute the prioritized prefetching queue based on available time gap
- Check hit/miss events, execute least recently used (LRU) replacement in the case of prefetching miss
- Update prefetcher statistics

At the end of each RFUOP, the following operations are done in the simulator:

- Update RFUOP timestamp
- Build prefetching queue based on statistical data

Reconfiguration Overhead. The metric used to evaluate the performance is the overall reconfiguration overhead. The configuration time used by a RFUOP includes defragmentation latency, prefetching latency, and configuration replacement latency in case of a configuration miss.

The total reconfiguration overhead is computed as:

$$t_{overhead} = t_{pred} + t_{miss} \quad (4.2)$$

where t_{pred} is the total overhead caused by the prediction algorithm, and t_{miss} is the configuration replacement latency in the case of a configuration miss.

$$t_{pred} = \begin{cases} t_{defrag} + t_{pref} - t_{dist}, & t_{defrag} + t_{pref} > t_{dist} \\ 0, & t_{defrag} + t_{pref} < t_{dist} \end{cases} \quad (4.3)$$

where t_{defrag} is the time spent on defragmentation, t_{pref} is the time spent on prefetching, and t_{dist} is the time interval between two RFUOPs.

$$t_{defrag} = step_{defrag} \cdot T_{defrag}$$
$$t_{pref} = s_{RFUOP} \cdot T_{offchip} \quad (4.4)$$
$$t_{dist} = t_{RFUOP_k} - t_{RFUOP_{k-1}}$$

The steps need to be performed for defragmentation is based on the algorithm in [8]. s_{RFUOP} is the size of the prefeteched RFUOP. t_{RFUOP_k} is the time when $RFUOP_k$ is executed. T_{defrag} and $T_{offchip}$ are simulation parameters that denote the unit latency of defragmentation steps and accessing off-chip memory. In this study we chose $T_{defrag} = 2$ and $T_{offchip} = 50$.

In the case of a configuration miss, t_{miss} is the time spent on fetching the needed RFUOP using the LRU algorithm:

$$t_{miss} = \begin{cases} s_{RFUOP} \cdot T_{offchip}, & miss \\ 0, & hit \end{cases} \quad (4.5)$$

5 Experimental Results

The results for algorithmic experiments in Matlab are shown in Section 5.1. The efficiency metric is prediction error rate as described in Section 4.1. These algorithms include both Markov predictor and LMSA predictor with different RH sizes, training step sizes, configuration size information, and history information. The cycle accurate simulation results for testing the efficiency of reducing reconfiguration overhead are shown in Section 5.2.

5.1 Algorithmic experiments results

We test the effect of RH size, training step size, configuration size information, and history information on the prefetching performance.

RH Capacity. Figure 1 shows the normalized weighted prediction error of the probability-driven LMSA predictor and the probability-driven Markov predictor for the cases of various RH capacities (4, 6, 8). We started with RH capacity of 4 because the maximum single configuration size is 3 as mentioned in Section 4.1. As shown the larger the RH capacity the smaller the error rate is. When RH capacity increases from 4 to 6 and 8, the corresponding normalized error rate reductions are 40% and 65% for probability-driven LMSA predictor, and 39% and 59% for probability-driven Markov predictor.

Furthermore, LMSA predictor outperforms Markov predictor (11%, 13%, and 13% reduction in normalized error rate for RH capacity of 4, 6, and 8) because LMSA converges faster and thus adapts to the application faster. For the rest of Section 5.1 we only show the algorithmic results for LMSA prediction.

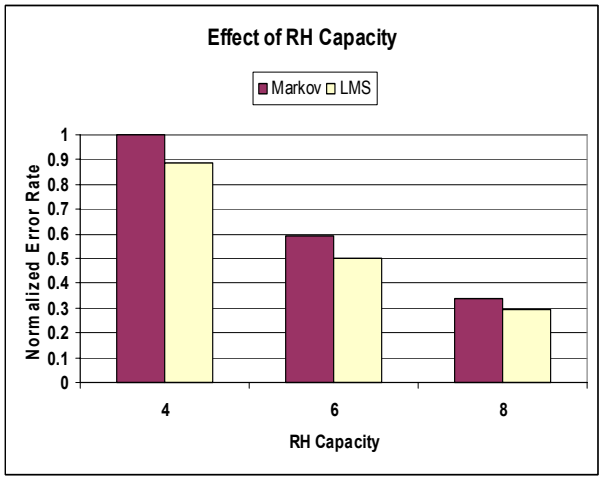

Figure 1. Normalized Prediction Error of probability-driven LMSA and probability-driven Markov for the cases of various RH capacities (4 6, 8). The baseline is Markov predictor with RH capacity of 4.

Training step Size. Training step size plays an important role in prefetching performance. Figure 2 shows the prediction error rate for different value of the training step size C in probability-driven LMSA. The RH capacity sizes are 4, 6 and 8 in this experiment. As shown in the figure, the larger the capacity the lower the error rate because more space is available for configurations. As the step size (C) increases from 0.025 to 1, the error rate decreases in the range [0.05, 0.1] and then increases. The optimal training step sizes with various RH capacity sizes are around 0.05 (e.g., RH4, RH6) to 0.075 (e.g., RH8) as shown in Table 1.

Training Step Sizes	0.025	0.05	0.075	0.1
RH4	8.43%	***7.96%***	8.06%	8.09%
RH6	3.40%	***3.33%***	3.37%	3.54%
RH8	1.56%	1.50%	***1.46%***	1.46%

Table 1. Optimal training step sizes with different RH capacities (4, 6, 8) for probability-driven LMSA predictor.

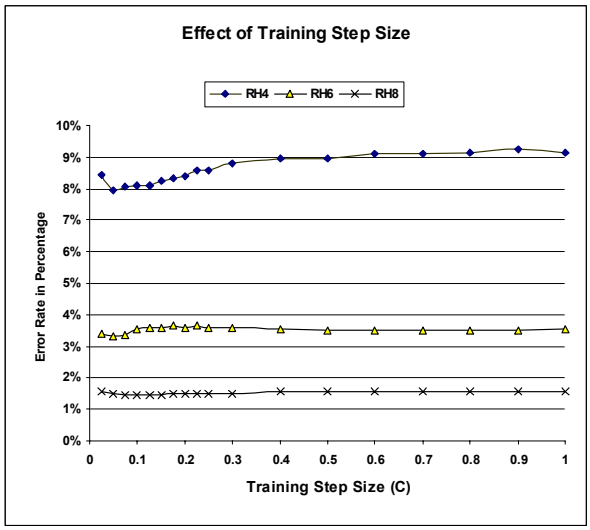

Figure 2. The effect of training step sizes (from 0.025 to 1) on error rate for different RH capacities (4, 6, 8) in probability-driven LMSA predictor.

Configuration Size Information. Configuration size is significant for prefetching decision. Without it, the prefetching decision is merely based on probability or correlation instead of the real reconfiguration overhead. Figure 3 shows the normalized weighted prediction error without (probability-driven) and with RFUOP configuration size information (cost-driven LMSA) in training weight. Step size is fixed to be 0.05 in this experiment.

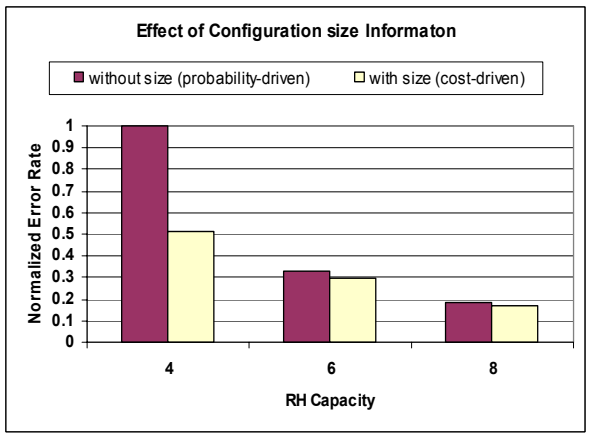

Figure 3. Normalized prediction error rate without (probability-driven LMSA) and with configuration size information (cost-driven LMSA) for different RH capacities (4, 6, 8). The baseline is the error rate of probability-driven LMSA with RH capacity of 4.

Using RFUOP configuration size information (cost-driven LMSA) the results show 49%, 10%, and 7% error rate reduction respectively for RH capacities of 4, 6, and 8. The reason that the error rate reduction decreases as the RH capacity increases is because there is less potential for performance improvement.

History Information. History information helps prediction. Figure 4 shows the normalized weighted prediction error with and without extra history information (1st order, 2nd order, and 3rd order LMSA). The training step size is fixed to be 0.05 and it is cost-driven LMSA predictor. As shown, the larger order of history is efficient for small RH capacity (e.g., RH4). When the RH capacity increases, the efficiency of using more history information decreases, for instance the most efficient history is the 3rd order when RH capacity is 4, the 2nd order when RH capacity is 6, and the 1st order when RH capacity is 8. This trend means aggressive prediction using history information is helpful for small RH capacity, while its complexity offsets its efficiency when RH capacity is large. Therefore, we use the 2nd order history for cycle accurate performance simulations in Section 5.2.

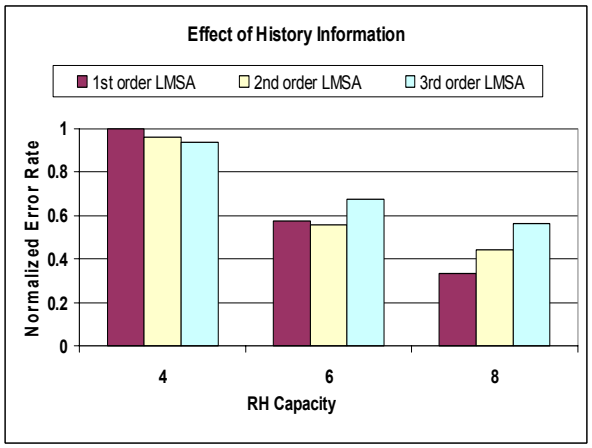

Figure 4. The effect of history information on performance of cost-driven LMSA with different RH capacities (4, 6, 8). The baseline is the error rate of the cost-driven 1st order LMSA with RH capacity of 4.

5.2 Benchmark simulation results

We used cycle accurate benchmark simulations to show the reconfiguration overhead reduction. Figure 5, 6, 7 show the results of a group of simulated algorithms (*LRU, Markov, Markov_ht2, LMSA, LMSA_hty2*). The *Least Recently Used (LRU)* uses no prefetching technique. In the case of missing a configuration, it simply replaces the least recently used configuration in the RH with the

needed one. All the other four algorithms (*Markov, Markov_ht2, LMSA, LMSA_hty2*) use configuration prefetching and LRU replacement in case of missing configuration. *Markov* is the probability-driven Markov predictor without history information. *Markov_hty2* is the cost-driven Markov predictor with 2nd order history information. *LMSA* is the cost-driven LMSA predictor without history information. *LMSA-hty2* is the cost-driven LMSA predictor with history information. The cycle accurate simulations give the amount of clock cycles each algorithm use for reconfiguration, which are the reconfiguration overheads. For each prefetching algorithm the optimal training step sizes are obtained through Matlab algorithmic experiments (Table 2) and used in the benchmark simulations.

RH Capacity	4	6	8
Markov	0.125	0.125	0.05
Markov_hty2	0.125	0.100	0.075
LMSA	0.085	0.015	0.020
LMSA_hty2	0.040	0.035	0.055

Table 2. The optimal training step sizes for the four prefetching algorithms (*Markov, Markov_ht2, LMSA, LMSA_hty2*) with different RH capacities (4, 6, 8).

We compared the reconfiguration overheads of the four predictors with that of LRU where no configuration prefetching is used (Figure 5). The amount of the simulated clock cycles spent in reconfiguration in LRU is used as the standard reconfiguration overhead.

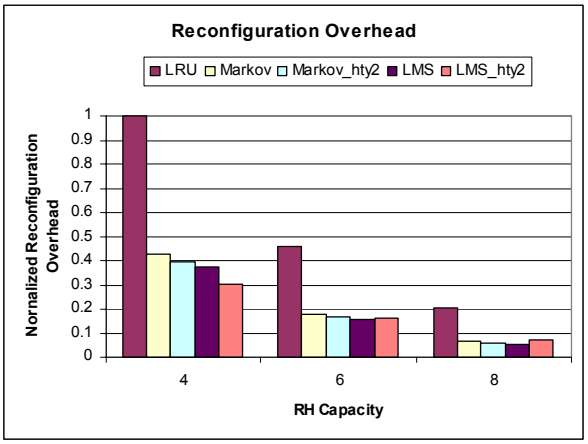

Figure 5. The normalized reconfiguration overhead of different algorithms (*LRU, Markov, Markov_ht2, LMSA, LMSA_hty2*). The baseline is the amount of the simulated clock cycles spent in reconfiguration in LRU for RH capacity of 4.

All the reconfiguration overheads in Figure 5 are normalized to that of LRU. Obviously all prefetching algorithms outperform the LRU algorithm considerably, particularly for the low capacity RH. As shown using configuration prefetching algorithm is more efficient than merely increasing RH capacity. For instance, the reconfiguration overheads of prefetching algorithms with RH capacity of 4 are smaller than LRU with RH capacity of 6; and it is the same trend between the groups of RH capacity 6 and 8.

Furthermore, the efficiency of prefetching algorithms increase as the RH capacity increases. This is because the amount of reconfiguration overheads reduction increases when the RH capacity increases as shown in Figure 6, where the reconfiguration overhead reduction is calculated using different LRU reconfiguration overhead as baseline in each RH capacity group.

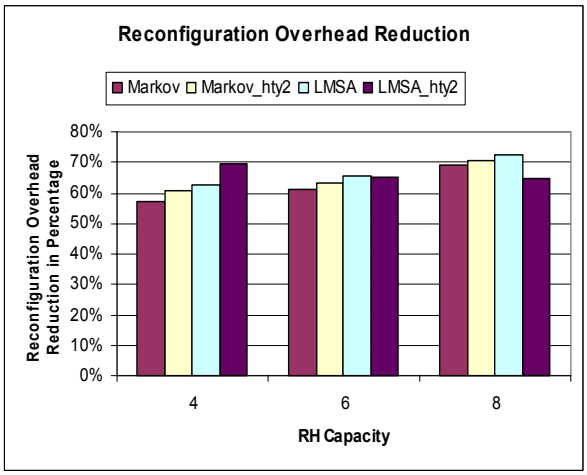

Figure 6. The reconfiguration overhead reduction for the prefetching algorithms (*Markov, Markov_ht2, LMSA, LMSA_hty2).* The baselines are *LRU* reconfiguration overheads for each RH capacity.

Figure 7 compares the cost-driven predictors (*Markov_ht2, LMSA, LMSA_hty2*) with the probability-driven predictor (*Markov*). It shows cost-driven predictors outperform the probability-driven predictor across the cases of different RH capacities except for RH capacity of 8. For instance, the reconfiguration overhead reductions over probability driven *Markov* are 4% to 7.5% for *Markov_hty2*, 11% to 12.5% for *LMSA*, and 10.8% to 29.6% for *LMSA_hty2*.

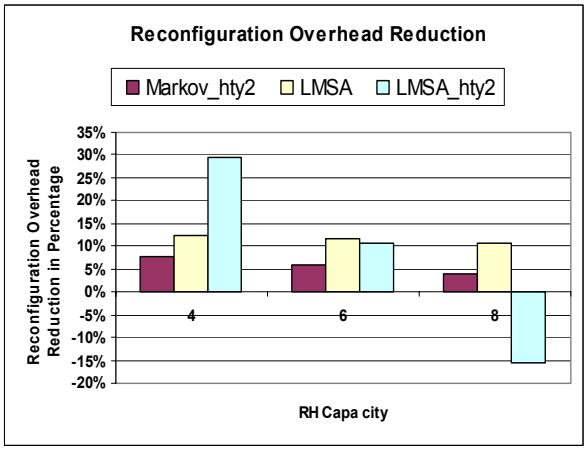

Figure 7. Reconfigurable overhead reduction for the cost-driven prefetching algorithms (*Markov_ht2, LMSA, LMSA_hty2).* The baselines are *Markov* reconfiguration overheads for each RH capacity.

6 Conclusions

This study introduced cost-driven hybrid configuration prefetching algorithm for coarse-grained reconfigurable coprocessor. The cost-driven prefetching algorithm is based on the fact that configuration size plays a significant role in reconfiguration overhead, which impacts the prefetching performance. Without considering the configuration size the predictor will make prefetching decision merely based on probability or correlation instead of the real reconfiguration overhead. We used algorithmic experiments to access the effect of RH capacity, prediction algorithm training step size, configuration size information, and history information on the prefetching error rate. And we also modeled the reconfigurable coprocessor access process in a cycle accurate simulator SimpleScalar and accessed cost-driven algorithm's efficiency in reducing reconfiguration overhead. Our results from algorithmic Matlab experiments and cycle accurate simulations show that prefetching algorithms outperform the LRU algorithm considerably, particularly for the low capacity RH; using configuration prefetching algorithms is more efficient than merely increasing RH capacity; the efficiency of prefetching algorithms increase as the RH capacity increases; cost-driven predictors outperform the probability-driven predictors by 4% to 29.6% in reducing the reconfiguration overhead. The most efficient probability-driven predictor is LMSA with history information and the reconfiguration overhead reduction is 10.8 to 29.6%.

Acknowledgements

Thanks for the constructive discussions from Prof. Kia Bazargan from Electrical and Computer Engineering Department of University of Minnesota.

References

[1] J. M. Arnold, et al. 'The Splash 2 Processor and Applications,' in Proc. IEEE International Conference on Computer Design (ICCD): VLSI in Computers and Processors, Oct. 1993.

[2] J. R. Hauser, Wawrzynek, J., 'Garp: A MIPS Processor with a Reconfigurable Coprocessor,' in Proc. IEEE Workshop FPGA's Custom Comput. Machiens, J. Anold and K. L. Pocek, Eds., Napa, CA, pp. 12-21, Apr. 1997.

[3] SC Goldstein, et al. 'PipeRench: A Coprocessor for Streaming Multimedia Acceleration,' in Proceed. Of the 26th Annual International Symposium on Computer Architecture, pp. 38-49, 1999.

[4] Z. Li, and S. Hauck, 'Configuration Prefetching Techniques for Partial Reconfigurable Coprocessor with Relocation and Defragmentation,' in ACM/SIGDA Symposium on Field-Programmable Gate Arrays, 2002.

[5] S. Hauck, 'Configuration Prefetch for Single Context reconfigurable Coprocessors,' ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 1998.

[6] Z. Li, and S. Hauck, 'Discovery for FPGA Configuration Compression,' ACM/SIGDA Symposium on Field-Programmable Gate Arrays, 1999.

[7] Z. Li, et. al, 'Configuration Caching management Techniques for Reconfigurable Computing,' IEEE Symposium on. FPGAs for Custom Computing Machines, pp. 87-96, 2000.

[8] K. Compton, et. al, 'Configuration Relocation and Defragmentation for Run-Time Reconfigurable Computing,' IEEE Transactions on VLSI Systems, 2002.

[9] D. Joseph, et. al, 'Prefetching Using Markov Predictors,' IEEE Transactions on Computers, VOL 48., No. 2, Feb. 1999

[10] www.simplescalar.com

[11] www.spec2000.com