Configuration Caching and Swapping

Suraj Sudhir — Department of ECE
Suman Nath — Computer Science Department
Carnegie Mellon University
Pittsburgh PA 15213
{ssudhir+@ece, sknath+@cs}.cmu.edu

Abstract

Speedups of run-time reconfigurable systems over traditional microprocessor systems are limited by the cost of hardware reconfiguration. In the case of multitasking system, this is further restricted by the cost of swapping the configurations among processes. In this paper we present techniques to carefully manage the configurations present on the reconfigurable hardware throughout program execution and context switching. We show that using the techniques we present, total latency of loading the configurations is reduced, lowering the configurable overhead. Our result shows feasibility and techniques to extend reconfigurable systems to multitasking environment.

1 Introduction

Configurable computing systems exhibit significant performance benefits over conventional microprocessors by mapping portions of executable code to the reconfigurable function unit (RFU). In order to achieve speedups two requirements must be satisfied. Firstly the code mapped must execute significantly faster on the RFU as compared to native binary execution on the traditional microprocessors. Secondly the overhead associated with configuring the RFU must not be so large as to negate the speedup obtained by migrating code execution to the RFU. Further, current microprocessors are increasingly turning towards advanced computing paradigms like ILP, multitasking and multithreading. Effectively coupling a RFU with such a processor requires an effective mechanism for swapping configurations in and out of the RFU upon context switches of processes or threads.

The problem of selecting code that would execute faster on an RFU than on the traditional microprocessor can be handled statically using compiler techniques. The bit-value algorithm [Goldstein00] is one such technique. The second requirement must be satisfied at runtime by effective configuration cache replacement policies. In this report we describe the results of our study on configuration cache management and configuration swapping.

As reconfigurable logic becomes mature enough to enable advanced reconfiguration techniques like partial and run-time reconfiguration the focus shifts towards reduction of reconfiguration overhead. Some of the independent techniques researchers have proposed for reducing configuration overhead include configuration prefetching [Hauck00] (loading configuration before it is actually needed, thereby reducing waiting time to load configuration), configuration compression [Hauck98] (loading compressed configuration and thereby reducing bus transfer time and load latency) and configuration caching [Compton00] (retaining configurations on chip efficiently so as to minimize the number of times they are loaded).

In this paper we extend the idea of configuration caching further. We propose some effective cache replacement policies and propose a 3 level exclusive configuration-caching model. More specifically, our contribution here is:

1. We formally characterize the cache replacement problem in the context of reconfigurable computing systems and point out the theoretical complexity of achieving the optimal

performance.

2. We propose some lightweight online algorithms for dominant FPGA models. We show simulation results that indicate the algorithms outperform the performance of similar algorithms existing in literature.
3. We extend the caching to a 3 level cache model and show how the performance varies when the inclusion property is avoided.
4. We study how multitasking can affect configuration-caching overhead and reduce performance and examine tradeoffs needed to handle this.

The rest of the paper is organized as follows. In section 2, we describe the configuration caching problem, its complexity and issues involved with regard to optimizing its performance. In section 3, we describe different FPGA models, with emphasis on the specific model we deal with throughout the paper. We describe our own cache replacement algorithms in section 4. In section 5, we present and analyze performance results obtained from our configuration caching algorithms in different models. In section 6, we deal with the problem of context switching and the associated problem of configuration swapping and its effect on performance. We show the results obtained for configuration swapping arising from context switches in section 7. Finally, in section 8, we conclude our report with a summary of the work done and future directions.

2 Configuration Caching Problem

The idea behind caching configurations on an FPGA, which is very similar to caching instructions or data in a general memory, is to retain the configurations (or RFU operations - *rfuOps*) on the chip or a nearby cache so that the data that needs to be transferred to the chip can be reduced. Like general-purpose computational system, configuration caching can also take advantage of the spatial and temporal locality and can be a good source of performance gain. However, the configuration-caching problem is different from the traditional caching problem in following ways:

1. Configuration caching is variable-size, variable-cost caching. In general-purpose systems, the data loading latency is fixed because a block represent the basic data transfer unit, while in the coupled processor-FPGA systems, the configuration latency is not fixed since the size of configurations may vary. This variable latency factor may have a great impact on the caching approaches. In traditional caching, the data items frequently accessed are kept in cache in order to minimize the memory latency. This might not be true, however, in coupled processor-FPGA systems because of the latency factor. In this case we need to consider a replacement strategy that is globally effective, since the total latency depends not only on the number of times a block is loaded but also its size. It is therefore possible that few loads of a very large configuration might be costlier than many loads of a smaller configuration, so that it makes more sense to keep the larger configuration in cache longer. Therefore an algorithm that efficiently implements rfuOp caching must consider both these factors to obtain a reduction in latency, not just frequency.
2. The number of configurations that could reside on FPGA is much smaller than the number of blocks that could be kept in a cache. So, without decent replacement algorithms, the system is more likely suffer the "thrashing problem", in which the configuration loading operations are performed very frequently, and therefore the reconfiguration overhead might not be greatly reduced.
3. Configuration caching is an *optional* caching problem. In conventional memory system, the block must be first loaded to memory before using it, but in reconfigurable system, if mapping some code to rfuOp seems to be infeasible (in terms of its own load latency or interaction with other on fabric rfuOps), the code can be run in conventional way (without

taking advantage of reconfigurable system).

The *size* and *cost* concerns make configuration caching a much more complicated problem than traditional caching problem. The challenge in configuration caching is to determine that what configurations should be kept on chip and what should not when a reconfiguration occurs. An incorrect decision on what configurations to be kept on chip can not only fail to reduce the reconfiguration latency, but in fact can increase the configuration overhead because of the decrease of the chip area. Specifically, by keeping a large configuration that is no longer used for the computation, other frequently used configurations cannot fit in the decreased area, and thus switch between those configurations may greatly increase the configuration overhead that is not the case if the large configuration is unloaded.

A similar class of problem has been explored in some other domains of computer systems. One example is Web Caching, where web pages of different sizes and different load latencies are to be cached on the proxy servers to reduce network traffic, load latency and load on busy web servers. Aside from having variable size and cost, the caching policy here is optional: some *non-profitable* pages should not be cached for making room for the other *profitable* pages. This Multi-size Optional Paging Problem [Irani97, Young98] is the same as the configuration-caching problem: configurations are of variable sizes, and loading into the fabric is optional, if infeasible, the code can be executed in conventional way.

The general offline algorithm for Multi-size Optional Paging Problem is NP-Hard [Cohen99] (while in fixed-size paging problem, Belady's [Belady66] polynomial time offline algorithm gives optimal performance). There is an approximation algorithm [Irani97] that can approximate the optimal to within a factor that is logarithmic in the maximum number of pages that can fit in the cache. Other algorithms tried in this context include LRU, Hybrid[Wooster97], Lowest-Relative-Value[Loren97], Greedy-Dual[Cao97], Size (evict the largest page), LFU, LRU-Threshold[Abrams95], Log(Size)+LRU[Abrams95], Hyper-G, Pitkow/Recker[Williams96], Lowest-Latency-First[Wooster97] etc. of which the first 4 have been found to perform better.

In spite of the similarity of the two problems, the web-cache replacement algorithms cannot directly be applied to the configuration-caching problem for flowing two reasons.

- First, web cache is much larger and can generally hold more pages than the number of configurations a fabric can hold. This suggests that the algorithm must be efficient to utilize the small area in the fabric to avoid thrashing.
- Second, configuration cache replacement decision should be made more quickly than web cache replacement decision. This suggests that configuration cache replacement algorithm should be cheap to implement.

So, we need to explore new algorithms for this context.

3 FPGA Models

Different Field Programmable Gate Array (FPGA) programming models have been proposed in literature. Three basic types are as follows.

a. *Single Context* FPGA: the whole chip area must be reconfigured during each reconfiguration. Even if a small portion of the chip needs to be reconfigured, the whole chip is rewritten during the configuration.

b. *Multi-Context* [Trimberger97]: the model is similar to that of the Single Context FPGA. However, instead of having one configuration stored in the FPGA, multiple complete configurations are stored. Each complete configuration can be viewed as multiple configuration memory planes contained within the FPGA.

c. *Partial Run-Time Reconfigurable* (PRTR): the area of the chip that is reconfigured is just the portion required by the new configuration, while the rest of the chip remains intact. Any *rfuOp* can be loaded anywhere in the chip as long as there is enough room for it.

PRTR seems to be the most promising model and throughout the paper we shall confine ourselves to this model. More specifically, we shall develop configuration caching for two new models based on PRTR FPGA. In conventional PRTR model, systems are likely to suffer a "thrashing problem" if more than one frequently used configurations occupy overlapping locations in the FPGA. Simply increasing the size of the chip will not alleviate the problem. So, two different models have been proposed in literature.

In the *Relocation* model, configurations can be dynamically allocated to any location of the chip in run time, instead of at compile time. In another model, called *Relocation + Defragmentation* model [Compton00] further improves chip area utilization. In this model, a special hardware unit called Defragmentor can move configurations within the chip to collect the small unused spaces (that are too small to be used by any configuration) in the chip to make a large fragment (like memory compaction). Including a defragmentor increases utilization of the fabric space, by using each small fragment of free spaces; but it also increase the total latency by adding the defragmentation cost.

Both of these models promise to deliver high performance through making maximum utilization of the chip area. In this paper, we shall focus on these two models.

4 Cache Replacement Algorithms

In this section we shall describe cache replacement algorithms for different FPGA models. We shall consider both the online and offline algorithms. The offline algorithms are not for implementation in hardware, they are just for comparing the performance of the online algorithms. We shall compare the performance of the algorithms in next section.

4.1 PRTR with Relocation

4.1.1 Online Algorithms

In this model, configurations cannot be moved once it has been loaded into the fabric. So, in case more than one victim needs to be evicted, we must select contiguous victims to make enough room for the new configuration. We describe the algorithms for this model in two phases: in the first phase, we assign scores to the configurations on the fabric. In the next phase we select victims (which are contiguous, if more than one configurations are selected) based on their score computed in the first phase. The overall algorithm is as follows: whenever a configuration miss occurs and there is not much empty space on fabric to load the configuration, phase-1 algorithm is invoked to score the configurations currently in the fabric. Phase-2 algorithm then selects the victims depending on their score computed in phase-1. The victims are then evicted and the new configuration is loaded.

Different algorithms we present here differ in the way they compute the score of the on-chip configurations. Phase 2 is same for all the algorithms. We give the phase 2, *SelectVictim* here. It basically finds the minimum number of contiguous configurations that makes enough room for the new configuration to be loaded, and have minimum total scores over all such possible configuration combinations.

procedure SelectVictim (*required_size*)
// Return the minimal set of contiguous configurations whose total size ≥ required_size
// rfu_count = *number of configurations on fabric*, s_i *is the i'th set*, rfu_i *is the i'th*
// configuration on fabric, each of the configurations has a cost associated with it
// total_cost *function computes the total cost of all the configurations in the set*
min_cost ← ∞
for *i* = 1 **to** *rfu_count*
    $s_i \leftarrow \{rfu_i\}$
    $j \leftarrow i + 1$
    while ($j \leq$ rfu_count **and** total_size(s_i) < required_size)
        $s_i \leftarrow s_i \cup rfu_j$
        $j \leftarrow j+1$
    endwhile
    if ($j \leq$ *rfu_count* **and** total_size(s_i) ≤ *min_cost*)
        min_cost ← total_cost(s_i)
        min_set ← s_i
    endif
endfor
return *min_set*
end procedure

We now present phase 1 of different online algorithms. The algorithms vary in the way they calculate the score of the configurations.

Algorithm 1: History based algorithm

This algorithm tries to predict the future sequence of rfuOps based on the recent history. It maintains a table *Next* where the *i*'th entry is the rfuOp that occurred after the last occurrence of rfuOp *i*. When it needs to select a victim, it predicts future occurrence of the configurations based on the *Next* entry and assign score based on their appearance in future. The further in future the appearance of the configuration, the lower is its score. The algorithm does not care about size of the configuration when calculating their score, but the SelectVictim used by it takes size into account in the sense that evicting a large rfuOp makes more room for loading the new rfuOp, while evicting smaller rfuOp makes smaller room and probably requires evicting more than one small rfuOp, thus making total cost comparable to the cost of one large rfuOp. As shown later, performance of this algorithm is very good, that can be explained by a theoretical result [Irani97] saying, if the page size is variable but load latency is proportional to the page size (*i.e.* latency per unit size is fixed), ignoring size to decide on replacement does not hurt much.

procedure History_Based_Decision(*rfuOp_to_load*)
// *rfuOp is the current configuration needed*
// Prev_rfuOp is the rfuOp needed just before this one
  R← *rfuOp_to_load*
  Next[*prev_rfuOp*] ← *rfuOp_to_load*
  if *rfuOp_to_load* is on fabric **then return**
  if there is enough space on fabric to load *rfuOp_to_load* **then**
    load *rfuOp_to_load* and **return**

  Make the chain *R*, Next[*R*], Next[Next[*R*]]?
  for each configuration *C* on fabric **do**
      C.cost ← -(distance of *C* on the sequence)

$S_i \leftarrow$ SelectVictim(size(*rfuOp_to_load*))
Load *rfuOp_to_load* overwriting the configurations in S_i
end procedure

Implementation of the algorithm in hardware needs to maintain a *Next* table of size $k \times log(k)$ bits, where k is the number of total rfuOp. But since the total number of the rfuOp varies widely from program to program, and this is a very large number, having such a table is not feasible. Our proposal is to have a smaller table and organize it as TLB. This table will contain the recently used next entries, and the rest entries are kept in some lower level of the hierarchy. Since the chain constructed is generally small, and chain constructed at some time step depends on the chain constructed at the previous step, TLB like table will perform well.

Algorithm 2: Penalty Based Algorithm
This is a kind of algorithm referred in most literature on variable size page replacement problem. The algorithm we give here is a slight modification of the Greedy-Dual algorithm, which is used in web cache replacement and shown to outperform other widely used web cache replacement algorithms. The modifications we make here are based on the following two motivations. First, we need something that can be implemented in hardware, so we should replace costly operations with cheaper one. This is not difficult though, since we assume that load latency of per unit configuration is constant. Second, we may need to replace more than one configuration and they must be physically contiguous. The algorithm works as follows: When a configuration is brought to fabric (or accessed when there is a hit), its cost is set to some large constant. Each time a fabric hit/miss occurs, the cost of each on-fabric configuration is reduced by (fabric_size - configuration_size). Intuitively this penalizes smaller configurations more than the larger configurations. During replacement, we look for some configuration that has the smallest cost (that has been penalized most).

procedure penalty_based_decision(*rfuOp_to_load*)
// *rfuOp_to_load is the current configuration needed by the program*
// FABRIC_SIZE is the size of the fabric and LARGE_CONSTANT is a large constant

if *rfuOp_to_load* is on fabric **then goto** L1

if there is enough space on fabric to load *rfuOp_to_load* **then**
load *rfuOp_to_load*
goto L1

$S_i \leftarrow$ SelectVictim(size(*rfuOp_to_load*))
Load *rfuOp_to_load* overwriting the configurations in S_i

L1: **for** each configuration C on fabric **do**
C.cost $\leftarrow$ *C*.cost - (FABRIC_SIZE - *C*.size)
rfuOp.cost $\leftarrow$ LARGE_CONSTANT
end procedure

Note that the algorithm is essentially similar to the one given in [Cao97] and [Hauck00]. Implementation of this algorithm in hardware is straightforward; we need to have one *cost* entry per rfuOp on fabric. Each time the algorithm is invoked, the cost entries will be accessed and updated. Replacement decision will be taken on the value of these cost entries.

4.1.2 Offline Algorithms

The motivation for finding an offline algorithm is to compare the performance of the online algorithms we propose. As we mentioned earlier, finding the optimum offline replacement policy for variable sized pages is NP Hard. So, rather than trying to find out the optimal policy, we propose two offline algorithms that work similar to the Belady's algorithm. They uses the whole rfuOp reference string, scores the on-fabric rfuOps based on the distance of their next occurrence from the current point (Distance of a rfuOp is the sum of the sizes of the rfuOps that occur before it), in different way. This score is then used to find contiguous victims. The algorithms are presented below.

- **Offline Algorithm 1**

procedure *offline1*(*rfuOp_to_load*)
for each rfuOp s_i in the reference string **do**
  if s_i is not in memory **then**
    for each on-fabric rfuOp r_i **do**
      distance ← sum of the sizes of rfuOps that occur between s_i and next occurrence of r_i in reference string
      score[r_i] ← -*distance*
    endfor
  endif
endfor
S_i ← SelectVictim(size(*rfuOp_to_load*))
Load *rfuOp_to_load* overwriting the configurations in S_i
end procedure

- **Offline Algorithm 2**

procedure *offline2* (*rfuOp_to_load*)
if enough space to load *rfuOp_to_load* in cache
  load rfuOp_to_load in cache;
  return;
endif
for each on-fabric rfuOp r_i **do**
  score[r_i] ← size(r_i) * LARGE_CONSTANT / distance between r_i and the next occurrence of rfuOp r_i in the reference string
while (not enough space to load new rfuOp)
  victim ← r_i such that score(r_i) is the lowest of all rfuOps in cache;
  drop victim from cache;
endwhile
load *rfuOp_to_load* in cache;
end procedure

For comparison with the online algorithms, we take the best of the performance of these two algorithms.

4.2 PRTR with Relocation + Defragmentation

This model offers greater flexibility for loading the required configuration, since rfuOp can be loaded in arbitrary location on fabric and on-fabric configurations can be moved around to merge small holes to a large one to accommodate it. We here consider two models here. In the first model (*full evection*), we assume that the whole rfuOp must be evicted even if the whole of its space is not needed by the rfuOp to be loaded. In the second model (*partial eviction*), we can evict only a part of a rfuOp from the fabric and later when it is needed again, only the part not in fabric is loaded.

It can be shown that cache replacement problem in partial eviction model is the same as the general caching problem, with the atomic configuration unit as the data transfer unit in general cache. Loading a rfuOp (full or in part) can be viewed as loading multiple atomic configuration units. The more formal proof will be found in [Hauck00]. So, the Belady's algorithm can be used as the optimal offline algorithm to compare. And since this model can be viewed as the general memory model, we used LRU as online algorithm.

But in the *full eviction* model, where the rfuOp must be evicted (and loaded) in full, the same problem arises: optimality cannot be assured by polynomial time algorithm. Our simulation results shows, LRU is not a very good candidate here (because it does not count the size of the rfuOp, so does not care for the waste space due to eviction). So, we tried a different algorithm that scores each on fabric rfuOp using the technique used in GreedyDual algorithm, that counts both size and occurrence history to select victims.

procedure full_evict (*rfuOp_to_load*)
$L \leftarrow 0$
for each rfuOp s_i in the reference string **do**
    if s_i is in memory **then**
        $cost[s_i] \leftarrow L + 1$
    else
        while there is not enough space **do**
            $L \leftarrow$ min (cost[R], for all rfuOp R in fabric)
            Evict the rfuOp with cost = L
        endwhile
    endif
endfor
Load *rfuOp_to_load*
$cost[rfuOp\_to\_load] \leftarrow L + 1$
end procedure

To compare the performance of the algorithm, we used an offline algorithm that selects victims using Belady's approach as well as reduce the amount of excess eviction (excess eviction = evicted rfuOp size - loaded rfuOp size). But the effect of reducing excess eviction is not much important here, because the excess space on fabric will be used up by next loaded rfuOp. So we also tried to count only distance on reference string (Pure Belady's algorithm), and the performance is almost same.

4.3 3 Level Caching Model

We now present algorithms for a three-level organization of configuration loading hierarchy, comprising the main memory, configuration cache and the actual RFU fabric that is tightly integrated with the chip. In case of any fabric miss, first the cache will be accessed, and only in the case of cache miss, main memory will be accessed. We consider two separate organizations of the overall cache hierarchy.

The first approach is to maintain inclusion between the fabric and the configuration cache, so that any rfuOp in the fabric is also in the configuration cache. The success of this method depends on the sizes of the fabric and the cache being significantly different. Unless the cache were much larger it is unlikely that this technique will be very effective, since few rfuOps not in the fabric will be in the configuration cache in case of a fabric miss.

An alternative approach we propose, called the *exclusion* technique, is to evict an rfuOp from the cache when it is loaded into the fabric from the configuration cache. Further, when rfuOps are dropped from the fabric they will be loaded in to the configuration cache. This technique permits a much larger working set of rfuOps to be present in the memory of the fabric and configuration caches and as we will show from experimental results, provides a much-improved performance as compared to inclusion. In this approach, the replacement policy to be applied to both the fabric and the configuration cache.

Below we describe algorithms for both the three-level inclusion and exclusion models. Both models call the replacement algorithm for both the fabric and the cache. They differ subtly in how they handle loads from cache to fabric. In both algorithms first a load is made from cache to fabric if the rfuOp is present in the cache. The exclusion algorithm then deletes the rfuOp from cache while inclusion model does not. Then any rfuOps dropped from fabric are loaded into cache. If the rfuOp were not in cache to begin with then it is loaded into fabric from memory. However in the inclusion model it is also loaded into cache, to satisfy inclusion condition.

3-Level Algorithm Implementation with Inclusion
procedure *threelevel* (rfuOp to load)
if (rfuOp is present in cache)
    perform *Replacement_algorithm(*rfuOp) on fabric;
else
    perform *Replacement_algorithm(*rfuOp) on configuration cache;
    load into fabric by calling *offline(*rfuOp) on fabric;
endif
end procedure

3-Level Algorithm Implementation without Inclusion
procedure *threelvl_noincl*(rfuOp to load)
if (rfuOp is in cache)
    perform *Replacement_algorithm*(rfuOp) on fabric from configuration cache;
else
    perform *Replacement_algorithm*(rfuOp) on fabric from memory;
endif
victims ← rfuOps evicted from fabric in previous step;
foreach (r_i in *victims*)
        perform *Replacement_algorithm*(r_i) on configuration cache;
endforeach
end procedure

5 Caching Performance Results

5.1 Experimental Setup

We used the *findcustom* script from the SUIF/Impact suite to extract rfuOps from our benchmarks. Each benchmark is compiled using the SUIF compiler and analyzed by the *findcustom* tool that determines which portions of the code, if any, will execute faster on RFU.

To determine this *findcustom* determines how many bits of variables used by functions are redundant and simulates the performance of a custom implementation in RFU of this function that only utilizes the non-redundant bits of the variables. The idea is that for lower bitwidths the same computation (say *add* for simplicity) will compute faster. More details of this algorithm are found in [Budiu00].

Once *findcustom* extracts functions that are to be implemented on RFU, they are extracted from the benchmark and implemented as separate functions, called *rfuOps*. We then use the Impact compiler from UIUC [August98] to generate an instrumented binary that generates trace information of the benchmark as it executes, enabling us to determine when and where these rfuOp functions. It also enables us to determine such things as the sequence in which rfuOps are called, their total execution time, etc. We modified the *findcustom* script as well as the Impact code generator and simulator to provide the rfuOp call sequence information as well as size and frequency statistics.

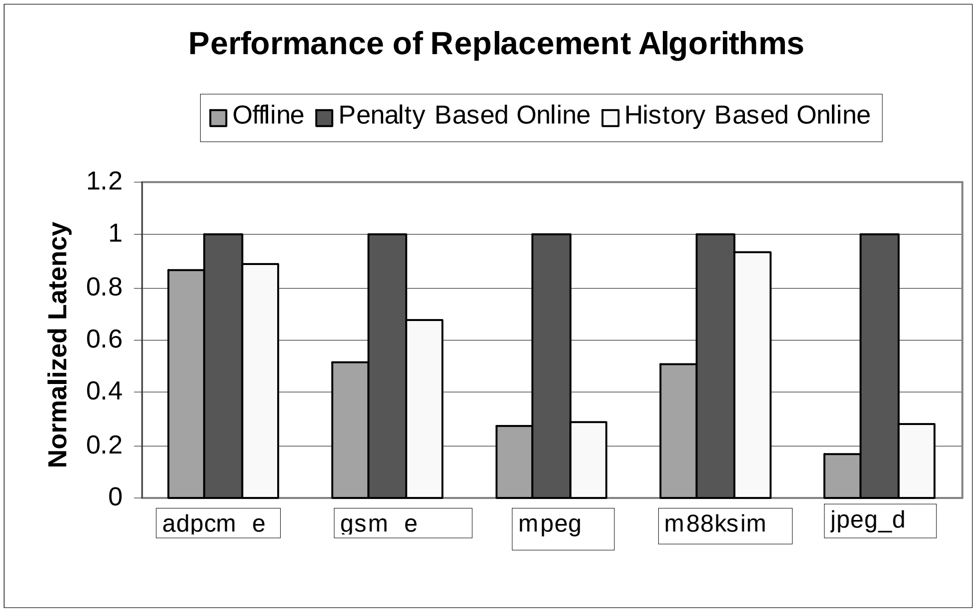
We used benchmark applications from the SPECInt95(http://www.spec.org) and MediaBench (http://www.cs.ucla.edu/~leec/mediabench/applications.html) [Potkonjak97] suites. Due to problems with the compiler infrastructure we were unable to compile all benchmarks. 6 benchmarks from the Mediabench suite and 4 from the SPECInt 95 suite were compiled in all, for generating sequence information.

5.2 Experimental Result

In this section we present and analyze the experimental result we got. We have tried out different algorithms for different FPGA models and tested them with different benchmark programs. Here we present some of the representative results, and discuss their implications. In the discussions and graphs below, we use some of the representative benchmarks and present only the data for fabric size of 50% of the total rfuOp size. Rest of the data can be found in appendix 1. Since miss penalties for different benchmarks vary a lot, we normalize them before putting them on graph.

5.2.1 Performance of Replacement Algorithms

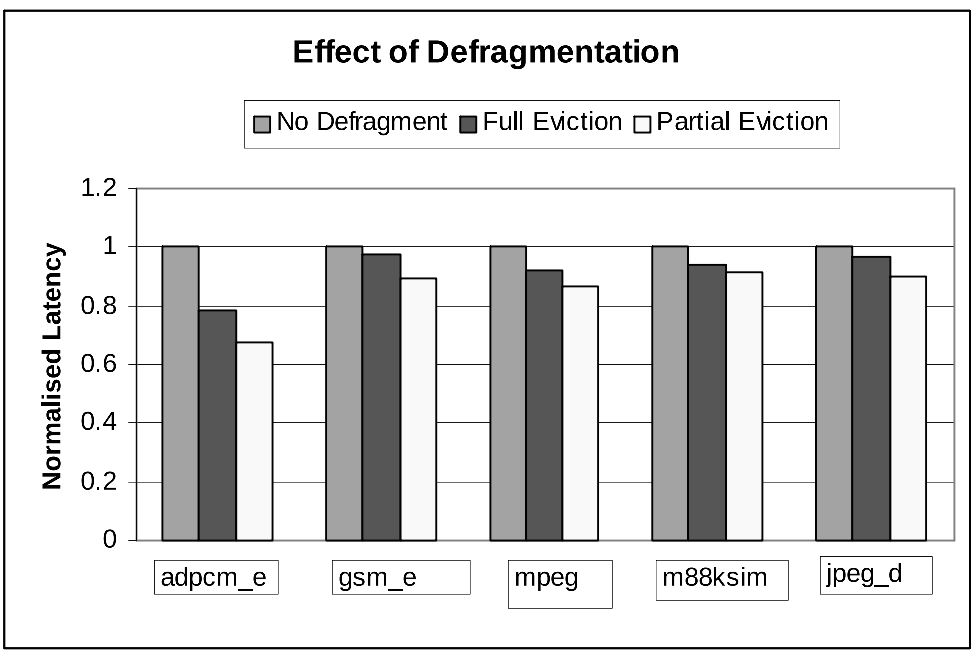
To see how good our online algorithms work, we used them with different benchmark programs and compared their performance with respect to the offline algorithm. The result is shown in Graph 1 (latencies have been normalized with respect to penalty oriented online algorithm). The model we consider here is the basic multi-context model (without any defragmentation). We used the penalty-based algorithm (widely used in web caching and also proposed to be used in configuration caching in [Hauck00]) and the history-based algorithm (proposed by us). Simulation result shows that general penalty based algorithm is not a good candidate in this context, rather our history based algorithms seem to be more effective. The reason why penalty based algorithm does not perform well can be explained as follows: penalty based algorithm performs well when it can always replace pages that incurred most penalty so far. But in this basic multi context model, it must select contiguous rfuOps (when more than one rfuOp needs to be replaced). Due to this extra constraints, it cannot always replace the rfuOp that incurred the most penalty so far (if its adjacent rfuOp is recently loaded), so the penalty cannot be used effectively. In contrary, the history-based algorithm don't make any assumption about size or adjacency during scoring. Simulation result shows it always performs better than the penalty based algorithm and its performance is consistently competitive with the offline algorithm performance. This indicates an avenue for studying aggressive history based online algorithms for efficient cache replacement strategies



Graph 1: Performance of Replacement Algorithms

5.2.2 Effect of Defragmentation

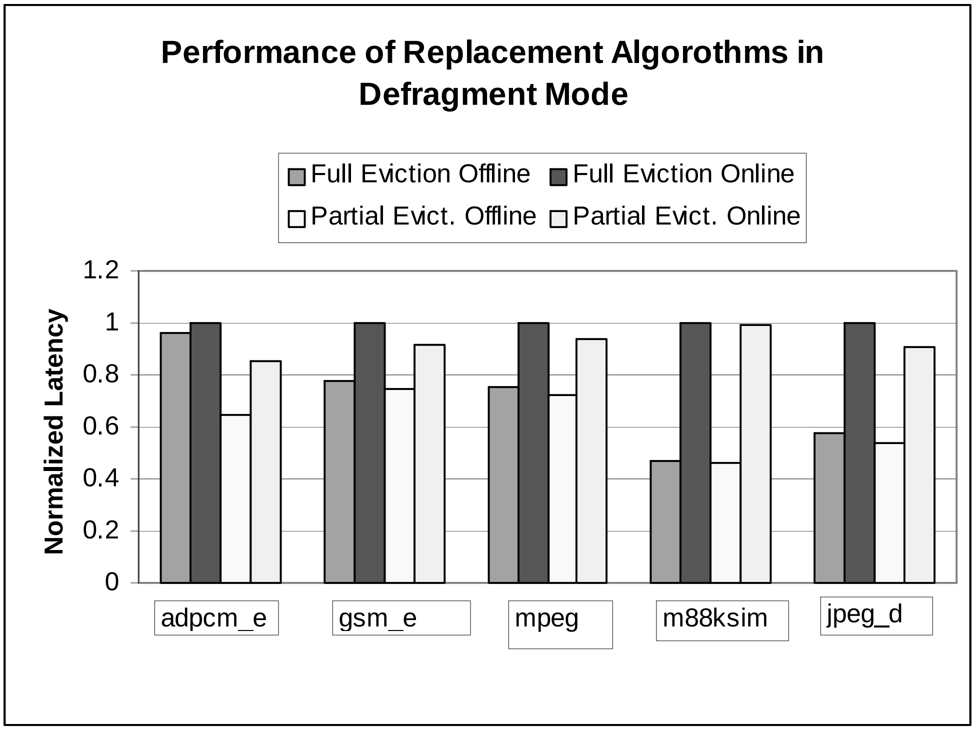
We here present the effect of introducing defragmentation on the fabric. We implement both the full and partial eviction model, implement the best online algorithm on those models and compare the performance with no defragmentation model. We consider the compaction cost to be the half of the cost to load rfuOps from configuration cache to fabric. (The compaction cost is in fact very small, and so it has been ignored in [Hauck00]). The result (latencies normalized to no defragment model) is shown in graph 2. As expected, defragmentation reduced miss penalty, because it makes better utilization of the fabric space. Also, partial eviction performs 3-13% better than the full eviction model.



Graph 2: Effect of Defragmentation

5.2.3 Performance of Replacement Algorithms in Defragment Mode

Graph 3 shows how online algorithms for the defragmentation model works, compared with the efficient online algorithms. The data are normalized with respect to the full eviction online algorithm performance. Partial eviction seems to be more efficient than the full eviction model, because this model allows leaving a portion of the RFUOP in the fabric, so if the evicted RFUOP is needed again, and part of it is on the fabric, only the rest of it can be loaded. It makes better utilization of the fabric too. But the gain may not be high if the sequence is such that a RFUOP is not referenced again soon after it is referenced once. In that case, even partial eviction will cause full eviction (may be by part, in different invocation of the algorithm) of the RFU before it is needed again (the situation is found in m88ksim in graph 3).



Graph 3: Replacement Algorithms in defrag mode

5.2.4 Performance of 3 level model

To model latency in 3 level hierarchy, we used a 20:1 ratio of delays for loading the cache from memory as compared to loading fabric from cache. This is consistent with real world situations. For example in the CMU Piperench chip loading the cache takes 4-5 PCI cycles whereas loading the fabric is generally done in a single 150Mhz Piperench clock cycle.

Graph 4 shows that making the fabric and configuration cache non-inclusive always improves performance (from 12 to 30%). Similarly, using defragmentation improves performance over the model not using defragmentation (from 10 to 60%). Simulation result shows there is no clear winner between the inclusion+defragmentation and exclusion+nodefragmentation models. Although most of the time, the former seems to be a good strategy, in some cases (like jpeg_d), defragmentation seems to be more important than making the model non-inclusive. But, in overall, the model without inclusion seems to perform well even without defragmentation.

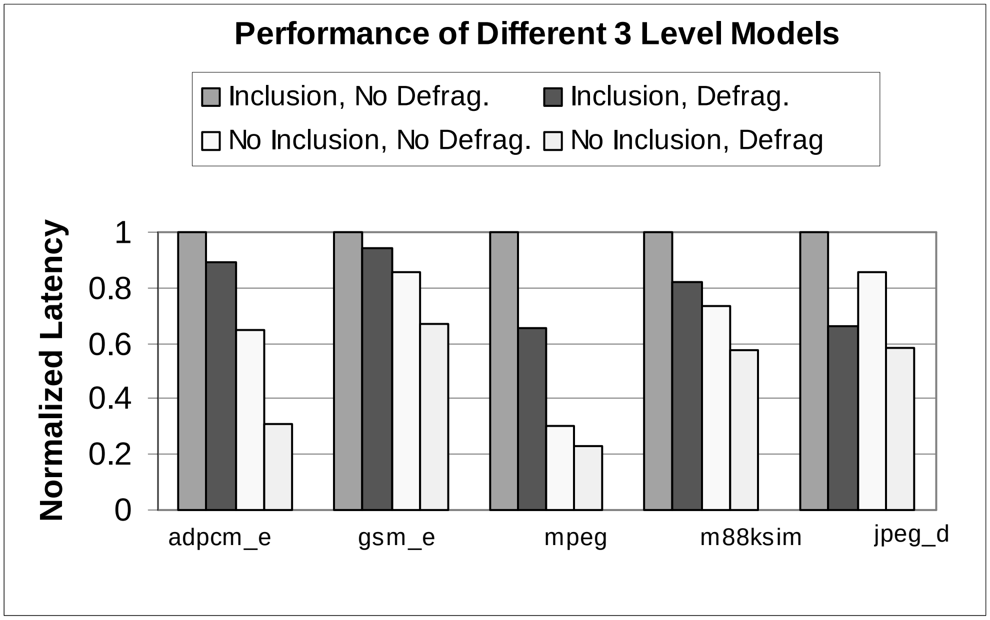
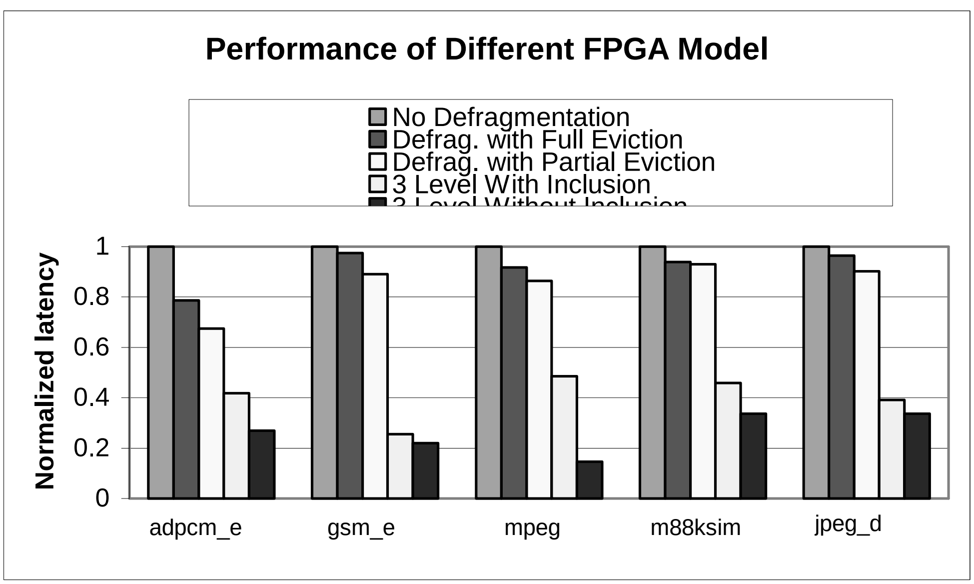


Figure 4: Performance of different 3-level models (Inclusion/not-inclusion, defragmentation/no defragmentation)

5.2.5 Comparison of the effect of different models

Figure 5 shows how performance improves as we move from basic multi-context FPGA model to defragmentation model to 3-level model. The data shown here has been normalized with respect to the data of 2 level, basic multi context data. For the 3 level models, we have not considered defragmentation. For all the models, we have taken the best performing online algorithms. For all the cases, we considered same fabric size, and for 2-level model, we considered the configuration cache to be (fabric size + half of the remaining required size). The graph shows that, 3-level model performs significantly better than 2-level model, even when the performance of 3-level model does not use defragmentation. Making the fabric and configuration cache non-inclusive improves the performance by 15-70%.



Graph 5: Performance of different models (basic multi-context, defragmentation and 3 level model)

We found from our simulation that with those benchmarks with a relatively smaller number of distinct rfuOps (such as adpcm_e or adpcm_d), the performance of the algorithm either didn't improve significantly or even became worse with defragmentation. However as the cache size increased this effect progressively disappears. For benchmarks that have a large number of distinct rfuOp calls, this effect is not visible and defragmentation generally provides uniformly significant improvement in performance.

Although it seems intuitive that, if we allow compaction using defragmentation using some history based replacement algorithm, miss penalty should decrease, since we are making better utilization of space here. But it can be shown that it is not always true. Sometimes, compaction can hurt performance, with small fabric. To show how this can happen, consider the rfuOp sequence 1 5 3 2 1 4 2 3, where size of rfuOp 4 is 2, and sizes of the rest rfuOps are 1. After the first 5 rfuOp, the fabric will contain the rfuOp s (2, 1, 3). For loading the rfuOp 4, we need to evict 2 pages (since its size 2). Any algorithm using compaction will select rfuOp 2 and 3 for eviction (since they happened before 1), and will experience 2 misses for next 2 rfuOp references. But, if compaction is not allowed, the algorithm would select (2,1) or (1,3) and would experience only one miss for the next two rfuOp references. The key here is that, in the two models, contents of fabric may be different over time, and so depending on the future reference, miss penalty may increase or decrease.

In summary, we found the following results from our experiments:

- History-based online algorithms show very good performance compared to conventional penalty based algorithms for cache replacement. Conventional LRU based algorithms performs terribly bad performance in such systems. Even a simple history-based algorithm gave much better performance than an efficient penalty based algorithm. More efficient history based algorithms can be designed to improve performance farther.
- Defragmentation generally improves performance except when the size of the cache is very small in which case it often leads to worse performance than without defragmentation.
- Partial eviction technique also significantly improves performance over full eviction in the case of defragmentation
- Three-level model performs better than two-level models.
- The exclusion technique increases performance of three-level caching model significantly over the usual inclusive three-level model.

6 Context Swapping

Current microprocessors are increasingly turning to new performance enhancing techniques like multitasking, multithreading and various other techniques. One aspect common to nearly all these paradigms is a multi-programmed environment where more than one process executes at once, with each process being assigned discrete units of processor time under the control of a process or thread scheduler.

To integrate a RFU with such a processor an effective means of swapping configurations in and out of the cache must be available. With such support, multiple processes can achieve speedups with the help of the RFU as they execute. In this case not only must the cache replacement overhead be overcome by the RFU speedup, but also the configuration swapping overhead. [Scalera98] describes some key elements for the successful implementation of context switching and an FPGA implementation that handles context switching. However this architecture is a special purpose device that does not envision coupling with a general-purpose microprocessor.

We implemented a simulation system where multiple processes can be executed at once, with a discrete time slice being assigned to each process. Any number of processes can be executed concurrently on the RFU with time multiplexing of the various rfuOps in a round-robin fashion. Performance was evaluated by considering the context-switching overhead as a fraction of the overall configuration cache latency for all the processes that were executed, as well as by measuring throughput of the program.

The main problem presented by context switching is the efficient saving and restoration of the state of each process in the RFU upon a switch. A simple if pessimistic method is to completely flush the contents of the fabric and the cache and load the new context. Alternatively on a switch we can save the state but not flush the cache or fabric, and instead let the new context remove rfuOps of the previous context from the cache as it executes and loads its rfuOps into the fabric. This requires additional identification or tagging of the rfuOps in the cache to indicate which process they belong to, so that there is no conflict among them during a switch.

7 Performance of Context Swapping

In this section we show the performance of configuration swapping in relation to the latency due to configuration replacement and as a fraction of the total execution time of the difference processes. We provide results for different number of processes as well as for different time slices for each rfuOp to execute.

Context swapping gives good performance where the sequence of rfuOp calls is not so short as to prevent the speedup from exceeding the total latency due to rfuOp configuration as well as context switching.

Figure 6 shows how overhead due to swapping decreases as we give more time to each process. Since processes are switched less often the latency would decrease. However switching less often can also reduce the response time, since each process occupies processor resources longer.

The number of processes running concurrently also affects performance. We define throughput here as the number of rfuOps executed per unit time. Increasing the number of processes should increase the throughput of the program initially, since more processes execute in the RFU so that it is inactive for a less portion of the total execution time. However increasing the number of processes also increases the overhead of switching for a fixed time slice for all

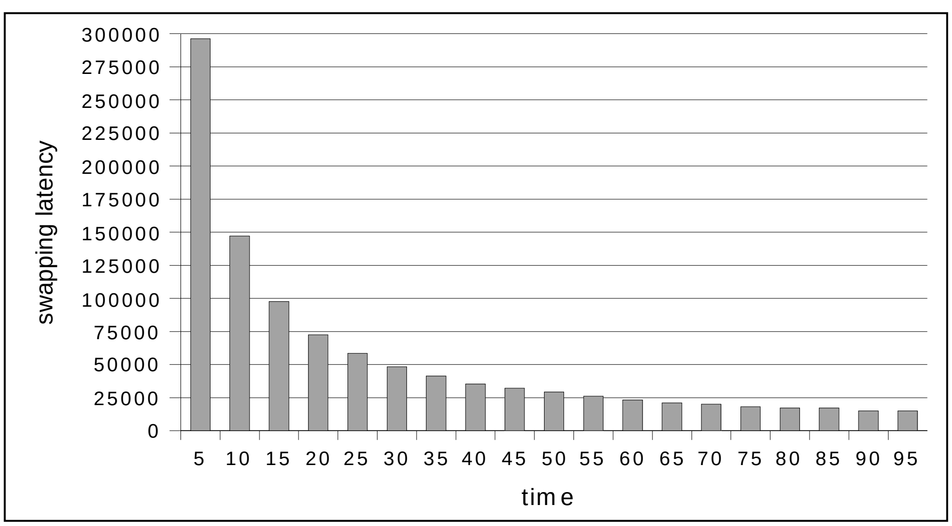


Figure 6: Effect of time slice on latency.

processes. As a result the RFU will pass its point of maximum utilization and then start *thrashing* since the throughput and speedup will be nullified by the overhead due to process switching.

Figure 7 presents the above result. We executed an increasing number of benchmarks

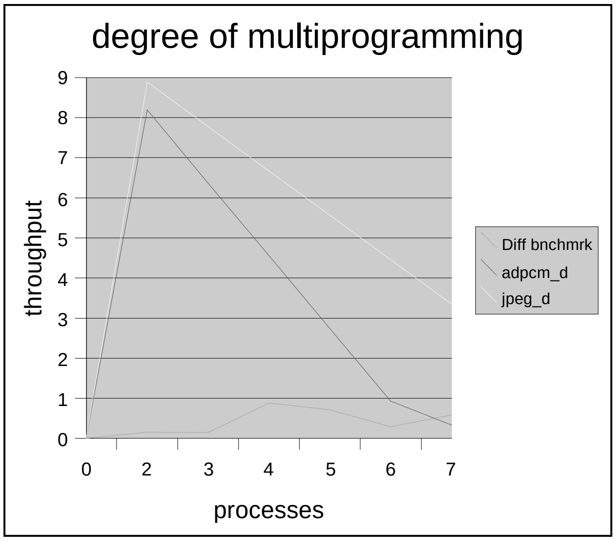


Figure 7: Degree of multiprogramming. Effect of multiple processes on throughput

For multiple processes that execute the same benchmark the increase in the overhead is linear which is indicated by the straight line showing decrease in throughput. For different benchmarks running as multiple processes, the throughput first increases and then decreases due to increasing swapping overhead coupled with configuration caching latency. Towards the end there is a small increase in throughput mainly as a result of the wide difference in the nature of different benchmarks and the types of rfuOps they generate as well as the rfuOp sequence.

Another concern with multitasking is that with increasing fabric and cache sizes the amount of data to be saved and restored on each context switch will increase, to the point that unless there is an effective tradeoff between the cache/fabric sizes, time assigned to each process and cache replacement latency with respect to the speedup obtained by the RFU for each process, the advantage gained from the speedup may be lost due to the combined overhead of configuration caching and swapping.

We can make the following conclusion from our experimental result in multitasking environment.

- Effectively multitasking processes that utilize the RFU requires a tradeoff between the number of processes and time assigned to each process in order to ensure that the speedup contained on individual processes is not nullified by their configuration caching and swapping latencies in a multitasking environment.
- It is possible that an increasing number of processes need not reduce the throughput (measured in terms of number of rfuOp configurations) with increasing number of processes. Throughput here depends upon the rfuOp size and latency(cost) too as a result of which it is possible that throughput can increase as degree of multiprogramming increases.

8 Conclusion

In this paper we have presented techniques, algorithms and hardware support for reducing reconfiguration costs by caching them on the fabric or configuration cache. We have considered the most versatile FPGA models, extended them to 3-level hierarchical models, and presented algorithms to carefully manage the configurations present on the reconfigurable hardware throughout program execution and context switching. Through extensive simulation, we have shown our techniques and algorithms to be quite effective, and reconfigurable systems can efficiently be extended to multitasking environment with those techniques.

However, there are still avenues to pursue. Our history-based algorithm considers only one preceding rfuOp. This resembles one bit prediction model in branch prediction. The algorithm can be extended to remember more than one preceding rfuOps, and consider all of those to take replacement decision. We also have not considered the case where the loading of the rfuOp into the fabric is optional. Our model always loads the rfuOp to fabric before execution. It would be interesting to explore the possibility of making the load to be optional, based on feasibility of loading the rfuOp and its interaction with other rfuOps in the fabric. In multitasking it is possible to select and schedule processes at runtime such that the throughput can be increased or maintained even with an increasing number of processes. This provides possibilities for further study.

References

[Abrams95] M. Abrams, C. R. Standbridge, G. Abdulla, S. Williams and E. A. Fox, Caching Proxies: Limitations and Potentials, *WWW-4*, *Boston Conference*, December, 1995.

[August98] D. L. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B. C. Cheng, P. R. Eaton, Q. B. Olaniran, W. M. Hwu, Integrated Predicating and Speculation in the IMPACT EPIC Architecture, *ISCA* 1998.

[Bednara00] M. Bednara, O. Beyer, J. Tiech, R. Wanka, Hardware-Supported Sorting: Design and Tradeoff Analysis, *System Design Automation* '00.

[Belady66] L. A. Belady, A Study of Replacement Algorithms for Virtual Storage Computers, *IBM Systems Journal*, vol. 5, pages 78-101, 1966.

[Budiu00] Mihai Budiu, Seth Copen Goldstein, Kip Walker, Majd Sakr BitValueInference: Detecting and Exploiting Narrow Bitwidth Computations, *Europar* 2000.

[Cao97] Pei Cao and Sandy Irani, Cost-Aware WWW Proxy Caching Algorithms, *Proceedings of the 1997 USENIX Symposium on Internet Technology and Systems*, pp. 193-206, Dec 1997.

[Cohen99] E. Cohen and H. Kaplan, LP-Based Analysis of Greedy-Dual-Size.. In *Proc. 10th ACM-SIAM Symposium on Discrete Algorithms.* ACM-SIAM, 1999.

[Compton00] S. Hauck, Z. Li, K. C. Compton, Configuration Caching Techniques for FPGA, *IEEE Symposium on FPGAs for Custom Computing Machines*, 2000.

[Hauck98] S. Hauck, Z. Li, E. J. Schwabe, Configuration Compression for Xilinx XC6200 FPGA, *IEEE Symposium on FPGAs for Custom Computing Machines,* 1998.

[Hauck00] S. Hauck, Z. Li, Improved Configuration Prefetch for Single Context Reconfigurable Coprocessors, submitted to *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

[Irani97] Sandy Irani, Page Replacement with Multi-size Pages and Applications to Web Caching, *Proceedings of the 29th Symposium on the Theory of Computing*, 1997, pages 701-710.

[Loren97] P. Lorrenzetti, L. Rizzo and L. Vicisano, Replacement Policies for a Proxy Cache. http://www.iet.unipi.it/luigi/research.html

[Potkonjak97] M. Potkonjak, C. Lee, W. H. Mangione Smith, Mediabench: a tool for evaluating and synthesizing multimedia and communications systems, *MICRO*-30, 1997.

[Scalera98] S. M. Scalera, J. J. Murray, S. Lease, A Mathematical Benefit Analysis of Context Switching Reconfigurable Computing, *Reconfigurable Architecture Workstation Symposium*, 1998.

[Trimberger97] S. Trimberger, D. Carberry, A. Johnson, J. Wong, A time-multiplexed FPGA, *IEEE Symposium on FPGAs for Custom Computing Machines*, 1997.

[Williams96] S. Williams, M. Abrams, C. R. Standbridge, G. Abdulla and E. A. Fox, Removal Policies in Network Caches for World Wide Web documents, *Proceedings of the ACM Sigcomm96*, August, 1996, Stanford University.

[Wooster97] R. Wooster and M. Abrams, Proxy Caching the Estimates Page Load Delays. In the *6th International World Wide Web Conference*, April 7-11, 1997, Santa Clara, CA. http://www6.nttlabs.com/HyperNews/get/PAPER250.html

[Young98] N.E.Young, Online File Caching, *Technical Report PCS-TR97-320*, Department of Computer Science, Dartmouth College.