Configuration Caching and Swapping

Suraj Sudhir, Suman Nath, and Seth Copen Goldstein

Carnegie Mellon University
{ssudhir,sknath,seth}@cs.cmu.edu

Abstract. Speedups of coupled processor-FPGA systems over traditional microprocessor systems are limited by the cost of hardware reconfiguration. In this paper we compare several new configuration caching algorithms that reduce the latency of reconfiguration. We also present a cache replacement strategy for a 3-level hierarchy. Using the techniques we present, total latency for loading the configurations is reduced, lowering the configurable overhead.

1 Introduction

Configurable computing systems can exhibit significant performance benefits over conventional microprocessors by mapping portions of executable code to a reconfigurable function unit (RFU). In such a system, native code sequences are replaced with configurations, which are loaded into the RFU using new instructions (rfuOps). In order to achieve speedups two requirements must be satisfied. First, a significant portion of the program must be mapped to the RFU and must execute significantly faster on the RFU as compared to native execution on the core. Second, the cost of loading the configurations onto the RFU must be small enough not to obviate the advantage of running on the RFU. In this paper we address the latter problem.

Some of the independent techniques researchers have proposed for reducing configuration overhead include configuration prefetching [14] , configuration compression [9] and configuration caching [7]. In this paper we describe an improved algorithm for RFU configuration caching and a new strategy for multilevel caching.

We divide configuration-caching algorithms into two classes, *penalty based* and *history-based* algorithms. Penalty-based algorithms evict configurations in the cache based on their size, distance of last occurrence and frequency of occurrence. The simple Least-Recently Used [7] algorithm, for example, evicts the configuration that was accessed furthest in the past. History-based algorithms evict rfuOps[1] based not only on their individual properties, but also their order of execution. In short, the one that is predicted to occur farthest in future is evicted.

[1] We use 'rfuOp' to refer to either the instruction that loads the configuration to be executed or the configuration itself

class	*size*	*cost*	*optional*
virtual memory	fixed	fixed	no
web caching	variable	variable	yes
VM w/ superpages	multiple	fixed	no
ideal config caching	variable	fixed	yes
required config caching	variable	fixed	no

Table 1. *Comparison between different caching problems*

In this paper, we propose an effective cache replacement algorithm and describe the performance of a realistic 3-level configuration-caching model. Our contributions are:

- We characterize the cache replacement problem in the context of reconfigurable computing systems and point out the theoretical complexity of achieving the optimal performance.
- We propose a lightweight history-based online algorithm that, based on simulation results, outperforms previous cache replacement algorithms.
- We extend the caching model to a three-level cache model and show how performance varies with defragmentation and the exclusion property.

The rest of the paper is organized as follows. In section 2, we describe the configuration caching problem. In section 3, we describe different FPGA models. 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.

2 Configuration Caching

There are significant differences between configuration caching and other caching problems (see Table 1) making previously developed techniques unsuitable to the configuration caching problem. Configuration caching is variable-size caching: The total latency depends not only on the number of times a configuration is loaded but also on its size. It is therefore possible that few loads of a very large configuration will be costlier than many loads of a smaller configuration. Thus, it might make more sense to keep larger configurations in cache longer and to consider both size and frequency when making eviction decisions.

While there are various systems that support variable-size pages, such as the superpage-based virtual memory systems in HP PA-RISC, Compaq Alpha and others [2], they only permit page sizes that are in multiples of a unit size. We call this *multiple page-sizes*, rather than variable page-sizes. Figure 1 shows how latency increases and is less predictable when page sizes are constrained to be a power of two.

Another variable but optional caching problem is web caching. Web caching is optional because the replacement algorithm may choose not to cache a page. This is called the Variable-size Variable-cost Optional Paging Problem [6, 13]. A reconfigurable system that optionally loads rfuOps is an *ideal* reconfigurable system. We call this problem an *optional caching problem.* In this paper we

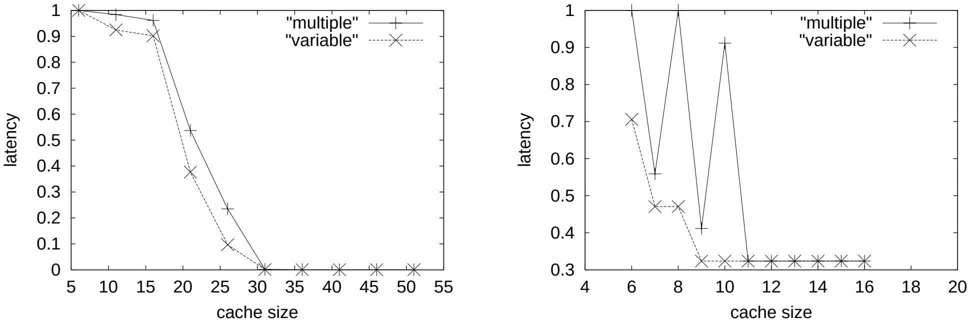

Fig. 1. *Comparison of the multiple-page size scheme which only allows page sizes that are a power of two, and the variable-page size scheme which allows pages of any size. On the left, gsm_d shows little difference in latency, whereas on the right, jpeg_e shows how latency can fluctuate for multiple page sizes.*

restrict ourselves to *required configuration caching*, i.e., all the rfuOps in the program must be loaded and executed.

In this paper we utilize a number of algorithms and conclusions developed for web caching. Besides the above differences between the two problems there are two more differences that affect the way techniques for web caching are extended to configuration caching. First, a web cache is much larger and can generally hold more pages making thrashing more likely in a configuration cache. Second, configuration cache replacement decisions must be made far more quickly than web cache replacement decisions. Thus, the configuration cache replacement algorithm needs to be implemented in hardware.

We begin our analysis with modified web caching algorithms LRU and Greedy-Dual as described in [4]. Both of these are penalty-based algorithms, i.e., they use past execution data in an order independent fashion to select a victim. We then propose a history-based technique which tracks the sequence in which rfuOps were executed to decide which rfuOp to evict at a given point of time.

3 FPGA Models

Different FPGA programming models have been proposed in literature. The three basic types are Single Context FPGAs, Multi-Context [10] FPGAs, and Partial Run-Time Reconfigurable (PRTR). According to [7] the PRTR FPGA has been found to be the best model and achieves speedups of more than 10 times that of single context FPGA models. However, even the basic PRTR model can suffer from thrashing caused by multiple configurations that are required frequently and must be loaded at the same address on the FPGA.

We describe our work in the context of two models based on the PRTR model, the *Location-independent* and the *Defragmentation* models[1]. In the Location-

[1] First proposed in [7], the Location-independent model is called the *Relocation* model and the Defragmentation model is called the *Relocation+Defragmentation* model

```
procedure SelectVictim( required_size)
 (min_cost, min_set)← (∞, nil)
 prefix ← nil
 last ← 1
 for i ← 1 to  rfu_count
     if (last > rfu_count – and required_size < size(prefix)) then
            break
    while (size(prefix) < required_size)
            if (defrag_mode  and rfuOp_last is not
                contiguous to rfuOp last-1) then
                    break
            prefix ← prefix ⋃ rfuOp_last
            last ← last +1
    endwhile
    if (cost(prefix) < min_cost) then
            min_cost = cost(prefix)
            min_set = prefix
            prefix ← prefix - rfuOp_i
    endif
 endfor
 return min_set
end procedure
```

Fig. 2. *SelectVictim algorithm to pick contiguous rfuOps. This algorithm returns the minimal set of contiguous rfuOps whose total size ≥ required size.*

independent model, a configuration can be dynamically allocated to any location of the chip at run time. However, once an rfuOp has been loaded it cannot be relocated. The Defragmentation model further improves chip area utilization by using a defragmenter to compact configurations on the fabric. It also permits rfuOps to be moved after being loaded. Including a defragmenter increases utilization of the fabric space, however, it also increases the total latency by adding the defragmentation cost.

4 Replacement Algorithms

In this section we describe cache replacement algorithms for different FPGA models. We break down the replacement algorithm into two phases. In the first phase a cost is computed for each configuration in the cache which is then used by phase two to determine which configurations to evict from the cache. All of the algorithms we present use the same phase two mechanism which we describe in Section 4.1. We then describe different cost computing algorithms in Section 4.2.

4.1 Victim Selection

The eviction policies for the Location-independent and the Defragmentation models are similar except that victims selected in the former must be physically contiguous on the fabric. Both models require that the set of rfuOps selected for eviction (*MINSET*) must have the minimum total eviction cost among all the rfuOps on the fabric. In the Location-independent model we call the set of evicted rfuOps the *contiguous MINSET*, where two rfuOps are contiguous if no other rfuOps lie physically between them in the fabric (though there may be a hole between them in which case it will just become part of the total freed space). Figure 2 shows the linear time procedure to find the MINSET. It maintains a sliding window of rfuOps, called *prefix*, from which it selects the victim. The operation *size(prefix)* and *cost(prefix)* are the total size and cost of all the rfuOps in the set *prefix*.

The complexity of this procedure is critical to the performance of the overall configuration caching algorithm. While an O(log n) hardware implementation based on parallel-prefix can be implemented, we use an alternate technique with O(1) complexity which is competitive with the actual SelectVictim algorithm. The minimum cost victim with enough space after it is selected even if there are rfuOps occupying that space. Using the simplified SelectVictim strategy increases the latency, on average, by 3%.

Victim Eviction Strategies When one or more victims need to be evicted, two possible strategies may be used. The first strategy, which we call *full eviction* is to evict entire rfuOps. The second strategy, *partial eviction*, evicts only as much of the victim(s) as necessary.

Partial eviction performs well when a partially evicted rfuOp is needed again before it has been fully evicted by subsequent SelectVictim operations. Since part of the rfuOp is already on fabric, the load latency is reduced because only the remainder of the rfuOp needs to be loaded.

It can be shown that the partial eviction algorithm is the same as the general fixed-size virtual-memory caching problem where the smallest unit of configuration is analogous to a page in a fixed-page size virtual-memory system. In the full eviction model, on the other hand, the problem of variable-sized pages means that optimality cannot be assured by a polynomial time algorithm.

While partial eviction appears to reduce the amount of excess eviction it is not very helpful because the excess space on fabric will likely be used up by some other rfuOp that is loaded in future. Furthermore, tracking the partially loaded rfuOps is complex.

4.2 Replacement Algorithms for PRTR FPGAs

.

Here we present both history- and penalty-based algorithms to compute the cost of the resident rfuOps. This cost is used by SelectVictim, described above, to make room for a new rfuOp.

History-based algorithms This algorithm tries to predict the future sequence of rfuOps based on recent history. It maintains a *Next* table where Next[i] is the

```
procedure
    HistoryBasedDecision(rfuOp_to_load)
  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
      return
   Make the chain R, Next[R],
      Next[Next[R]],...
  for each configuration C on fabric do
      C.cost ← -(distance of C on chain)
  endfor
   $S_i$ ← SelectVictim(size(rfuOp_to_load))
   Load rfuOp_to_load overwriting
      the configurations in $S_i$
end procedure
```

Fig. 3. *History-based Algorithm*

```
procedure
    PenaltyBasedDecision(rfuOp_to_load)
  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
   $S_i$ ← 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 ← C.cost−(FABRIC_SIZE
                         −C.Size)
      endfor
  rfuOp.cost ← LARGE_CONSTANT
end procedure
```

Fig. 4. *Penalty-based Algorithm*

rfuOp that last followed i. The evicted rfuOp is determined by following the Next pointers starting with the rfuOp being loaded, j, i.e., $j \rightarrow Next[j] \rightarrow Next[Next[j]] \ldots$ The rfuOp on the fabric that occurs furthest in this chain is predicted to occur furthest in future and will therefore be evicted.

Each rfuOp is assigned a cost which is negative of the smallest distance from j in the chain of next pointers. The rfuOp with the smallest cost is evicted. While the size of the rfuOp is not considered here, it is taken into account during phase two. As shown later, this algorithm works well despite the fact that it ignores the sizes of rfuOps in the chain. This agrees with [6].

Implementation of the algorithm in hardware requires a means to handle the chaining procedure. While a pointer jumping procedure would be expensive in hardware, it would execute with a complexity of O(log n). Here we describe an algorithm that predicts which rfuOp will occur furthest in the future without having to maintain or walk down a series of next links. The key idea is to keep information only about the RfuOps that are currently in the fabric. It maintains for each rfuOp, r,:

- FI(r) the first rfuOp to follow r that is in the fabric.
- FA(r) the last rfuOp to follow r that is in the fabric. FA(r) is only valid if r is resident in the fabric.

Using this information we can determine the rfuOp predicted to be needed farthest away in time when r is being loaded as: FA(FI(r)). For example, if the fabric currently holds rfuOps 1, 2, and 4 and we are about to load rfuOp 5, and the sequence of rfuOps upto this point is: ...,5,3,1,2,6,4, then FI(5) = 1 and FA(1) = 4.

To calculate FI and FA, we use two auxiliary items for each rfuOp: S(r), a virtual sequence number, and P(r), the rfuOp executed before r that is in the fabric. We also maintain a register, LastRfu, which holds the last rfuOp executed.

Before program execution, each rfuOp in the program is assigned a virtual sequence number such that S(r) = r. When an attempt is made to execute rfuOp R, one of three possibilities exists:

R **hits in the fabric:** We execute R and update the tables.

R **misses in the fabric and there is room for R:** We load R, execute R, and update the tables.

R **misses and we need to evict something:** We select FA(FI(R)) as the victim and invoke SelectVictim until there is room in the fabric. We load R, execute R, and update the tables.

Updating the tables requires that we update FI() for all the rfuOps in the program, FA() and P() for the rfuOps in the fabric, and S() for R.

FI**:** In parallel, for each rfuOp, i, in the program and $i \neq R$, if (S(i) < S(R) and S(R) < S(FI(i))), then set FI(i) = R. Finally, set FI(LastRfu) = R. In other words, if R occurs before FI(i) in the virtual sequence, make FI(R) = i.

FA**:** In parallel, for each rfuOp, i, in the fabric and $i \neq R$, if (S(i) < S(FA(i)) and S(FA(i)) < S(R)), then set FA(i) = R. Finally, set FA(R) = LastRfu. In other words, if R occurs after FA(i) in the virtual sequence, make FA(i) = R.

S**:** set S(R) = S(LastRfu) + 1 modulo some large number, e.g., 2^{16}.

P**:** If v is a victim, then in parallel, for each rfuOp, i, in the fabric and P(i) = v, P(i) = P(P(i)). This ensures that P(i) points to an rfuuOp still in the fabric.

If multiple victims need to be selected, they are found using the P entries. The update step can be done efficiently and is not on the critical path.

In all benchmarks, a significant portion of the rfuOp execution sequence constitutes a periodic pattern. This is because most of the speedup is achieved by implementing portions of a loop. In these cases the sequence information is as accurate as the algorithm in Figure 3. However, when the sequence is aperiodic the constant-time algorithm may differ from the algorithm in Figure 3. In some cases the simulation of the constant-time implementation proved to be better than that in Figure 3, in other cases the reverse was true. In all cases, the two are within 10% of each other and on average they are less than 2% of each other.

The strategy used by the history-based algorithm is not the same as most-recently used (MRU) unless the most recently accessed rfuOp is *always* the one that is accessed furthest in future. Consider the sequence of rfuOps 1 2 3 4 3 4 3 4...1 2 3 4 3 4..., with all rfuOps of size 1, and a fabric size of 3. The MRU algorithm will register evict 3 in order to load 4 and vice versa each time. On the other hand, the history-based algorithm does the following: the first time it loads 4 it will evict 3. Now FA(4) = 2, since FA(i) = P(LastRfu) when LastRfu is the victim. Also, FI(3) is 4. So when 3 is next loaded it will evict FA(FI(3)) = 2, avoiding the thrashing that happens with MRU. Furthermore, it can be seen that 2 really is the furthest in the sequence at this point, so the history-based algorithm makes a correct prediction, unlike the MRU, which on average increases latency by more than 160%.

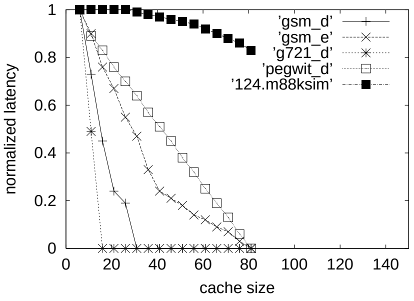

Fig. 5. *Effect of cache size for different benchmarks using the history-based algorithm*

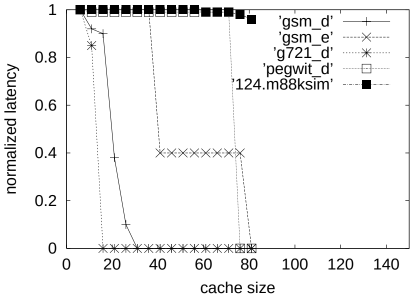

Fig. 6. *Effect of cache size for different benchmarks using the penalty-based algorithm*

Penalty Based Algorithm Existing solutions to the variable page-size replacement problem, as described in Section 2, fall under the category of penalty-based algorithms. The algorithm we chose to implement for comparison is a modified form of the Greedy-Dual Size algorithm in [4]. It is used in web cache replacement and is shown to outperform other widely used web cache replacement algorithms.

Our penalty-based algorithm assigns costs to each rfuOp currently in the fabric. Whenever a configuration is accessed, its cost is set to some large constant and the cost of the other rfuOps on fabric are reduced by (fabric_size - configuration_size). Intuitively this penalizes smaller configurations more than the larger configurations. During replacement, we look for a configuration that has the smallest cost (that has been penalized most).

5 Performance Results

5.1 Experimental Setup

We used the SUIF [12] compiler with custom passes to automatically extract rfuOps from the program. We instrument the rfuOp-enabled code to generate traces which are then simulated on an extension of SimpleScalar [3] to obtain our results. We ran two benchmark applications from the SPECInt95 [5] and ten from the MediaBench [8] suites.

5.2 Experimental Results

In this section we present and analyze the experimental results for the algorithms proposed in Section 4. We found that in most benchmarks the latency decreased in a roughly linear fashion with increasing fabric size. This was truer for the history-based algorithm than the penalty-based algorithm. Figure 5 shows how the cache size affects the performance for a representative sample of the benchmarks, using the history-based algorithm. The history-based algorithm scales

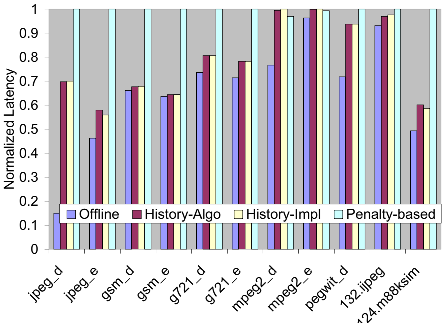

Fig. 7. *Performance of Replacement Algorithms*

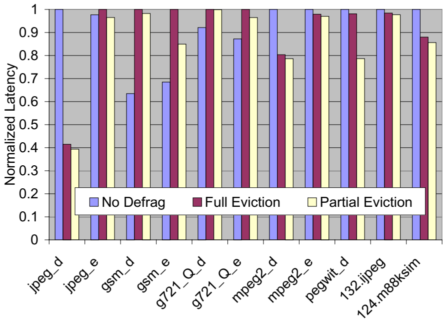

Fig. 8. *Effect of Defragmentation*

well with changing cache size. On the other hand, as Figure 6 shows, the penalty-based algorithm scales poorly with cache size. To eliminate the effect of differing working sets we simulate each benchmark with an RFU that can hold half of all the rfuOps for that particular benchmark.

Performance of Replacement Algorithms To see how well our online algorithms perform, we compared their performance to that of the best of two offline algorithms: Belady's algorithm [1] or an approximation algorithm presented in [7]. The results for a location-independent FPGA are shown in Figure 7. History-Algo and History-Impl refer respectively to the theoretical model (using next-chains) and our implemented version of the history based algorithm. We also implemented the basic LRU algorithm. However, its performance was found to be vastly inferior to even the penalty-based algorithm.

Simulation results show that the history-based algorithm is more effective than the penalty-based one, and is consistently competitive with offline algorithms. The main reason for this is that the penalty-based algorithm evicts the rfuOp that occurred furthest *before* the current point, while the history-based algorithm replaces the one that it estimates to occur furthest *after* the current point. In fact, we find that the history-based algorithm tends to make choices similar to that of Belady's offline algorithm.

Effect of Defragmentation In Figure 8 we compare the history-based algorithm on a location-independent fabric with partial and full eviction on a fabric which implements defragmentation. Since the defragmentation cost is small, it has been ignored in simulation. The results are ambiguous as sometimes the extra power of defragmentation leads to worse behavior.

Partial eviction performs slightly better than the full eviction model, except for cases like jpeg_d and pegwit_d where an rfuOp is not required soon after its eviction. However, the implementation cost of partial eviction outweighs its small advantage.

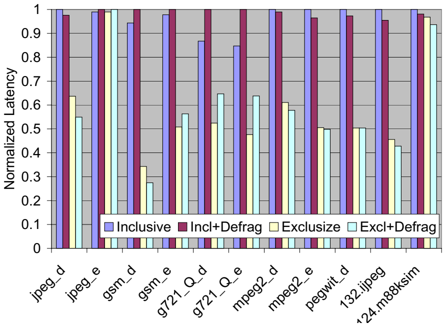

Fig. 9. *Performance of different three-level caching models*

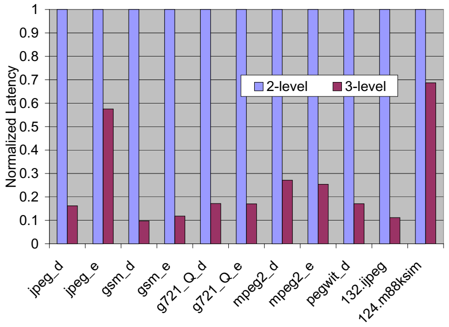

Fig. 10. *Comparison of different caching models*

Performance of Three-Level Model Here we show the effect of introducing a configuration cache for the RFU.[2] To model the latency we used a 20:1 ratio for loading the cache from memory as compared to loading the fabric from cache.

Figure 9 shows that making the fabric and configuration cache exclusive always improves performance. This comes at the cost of an additional buffer to hold rfuOps that are evicted from fabric before they are loaded into the configuration cache. This buffer is necessary to maintain proper serialization of the operation. The Defragmentation model performs better than the Location independent model.

Comparison of the effect of different models Figure 10 shows how performance improves significantly when we utilize the best three-level model, i.e., the one with the exclusion property, compared to the best two-level model. Most benchmarks showed a significant improvement in performance.

6 Conclusion

In this paper we have described algorithms for reducing reconfiguration overhead through effective replacement algorithms and configuration caching. We present an effective history-based algorithm with an efficient hardware implementation. We show that the added complexity of partial eviction does not yield significant performance improvement. Likewise, defragmentation increases the implementation complexity, but does not always improve performance.

There are still many avenues for further research. Our history-based algorithm considers only one preceding rfuOp, similar to a one bit prediction model in branch prediction. The algorithm can be extended to remember more than one preceding rfuOps, perhaps as a tree rather than a chain, and consider different possibilities before taking a replacement decision. Finally, we have not considered the case where the loading of the rfuOp into the fabric is optional.

[2] Space precludes describing our algorithms here, but a more complete description can be found in [11].

Acknowledgments

We want to thank the reviews for their many helpful comments. This work was supported in part by an NSF CAREER award and Intel corporation.

References

1. L. A. Belady. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal*, 5:78–101, 1966.
2. B.Jacob and T.Mudge. Virtual memory in contemporary microprocessors. In *Proceedings of IEEE MICRO*, volume 18, pages 60–75. IEEE, 1998.
3. Doug Burger, Todd M. Austin, and Steve Bennett. Evaluating future microprocessors: The simplescalar tool set. Technical Report CS-TR-1996-1308, University of Wisconsin-Madison, 1996.
4. P. Cao and S. Irani. Cost-aware www proxy caching. In *Proceedings of the 1997 USENIX Symposium on Internet Technology and Systems*, pages 193–206. USENIX, Dec. 1997.
5. http://www.spec.org/osg/cpu95. Specint95, 1995.
6. S. Irani. Page replacement with multi-size pages and applications to web caching. In *Proceedings of the 29th Symposium on the Theory of Computing*, pages 701–710, 1997.
7. S. Hauck K. C. Compton, Z. Li. Configuration caching techniques for fpga. In *IEEE Symposium on FPGAs for Custom Computing Machines.* IEEE, 2000.
8. W. H. Mangione-Smith M. Potkonjak, C. Lee. Mediabench: a tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of MICRO-30*, 1997.
9. E. J. Schwabe S. Hauck, Z. Li. Configuration compression for xilinx xc6200 fpga. In *IEEE Symposium on FPGAs for Custom Computing Machines.* IEEE, 1998.
10. A. Johnson S. Trimberger, D. Carberry and J. Wong. A time-multiplexed fpga. In *IEEE Symposium on FPGAs for Custom Computing Machines.* IEEE, 1997.
11. Sudhir, Nath, and Goldstein. Configuration caching and swapping. Technical report, CMU, 2001.
12. R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S.-W. Liao, C.-W. Tseng, M. Hall, M. Lam, and J. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. In *ACM SIGPLAN Notices*, volume 29, pages 31–37, December 1994.
13. N. Young. Online file caching. Technical Report PCS–TR97–320, Dartmouth College, 1998.
14. S. Hauck Z. Li. Configuration prefetch for single context reconfigurable coprocessors. In *International Symposium on Field-Programmable Gate Arrays*, pages 65–74, Feb. 1998.