ISSN No. 0976-5697

Volume 3, No. 1, Jan-Feb 2012

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

A Smart Cache Designed for Embedded Applications

Afrin Naz*
Department of Computer Science and Information Systems
West Virginia University Institute of Technology
Montgomery, USA
afrin.naz@mail.wvu.edu

Krishna M. Kavi
Department of Computer Science and Engineering
University Of North Texas
Denton, USA
Kavi@cs.unt.edu

Abstract: In this paper, we extend our previous investigation of split array and scalar data caches to embedded systems. More specifically we explore reconfigurable data caches where L-1 data caches are optimally partitioned into scalar caches augmented with victim caches and array caches. We do not change cache block size or set-associativities, making it easier to reconfigure cache banks. We also evaluate how any unused portions of cache resources can be used as prefetch buffers and branch target buffers to further improve the performance of applications. Since embedded systems require very careful management of available resources, our approach to configuring L-1 caches can lead to better performance and better energy savings.

Keywords: Reconfigurability, Embedded systems, Cache Memories, Split Caches, Spatial Locality, Temporal Locality, Prefetching.

I. INTRODUCTION

For embedded applications, it is necessary to provide the required performance within specified size and power budgets. Studies have found that the on-chip cache is responsible for 50% of the power consumed by an embedded processor [17]. Therefore, it is worthwhile investigating new cache organizations to address both performance and power requirements of embedded applications. In this paper we explore how to design reconfigurable caches that achieve high performance for embedded applications while remaining both energy and area efficient.

For the last two decades computer architects have proposed various cache-control mechanisms and novel cache architectures that detect program access patterns and fine-tune cache policies to improve both the overall cache use and data localities for desktop applications. Major cache optimization techniques (to improve either or both miss rate and miss penalty) include increasing block size and cache size, increasing associativity, complementing the regular cache with victim cache, prefetching data, including additional cache hierarchies. Since for embedded applications it is necessary to provide the required performance within specified size and power budgets, most of these techniques often are not implemented. In our previous work [20] we have studied each of these different cache-control mechanisms and performed comprehensive evaluation of our proposed partitioned caches. Our results demonstrated that split-caches can outperform all of these conventional cache optimization techniques. In this paper we adapt and further extend these studies for embedded systems, with the primary goal of energy savings while maintaining execution performance, yet using significantly smaller data caches. In addition to partitioning data caches into array (or stream) and scalar caches, we investigate how the split caches can be optimally reconfigured for each application. Our studies show significant savings in power and cache capacities. By using these saved area and power for other architectural features to implement different cache optimization techniques, additional performance gains can be achieved for embedded applications.

We assume that caches can be designed to permit reconfigurability [10]. Previous studies investigated configuring block sizes and set-associativities. In this paper we only explore configuring caches by changing cache sizes, without changing associativity or block sizes. The reconfigurability is achieved by using a configuration vector that can be loaded with a new configuration before an application starts executing. The optimal cache sizes are explored off-line by searching through possible configurations. Our studies show that for L-1 cache system, reconfigurable caches consisting of an instruction cache with prefetching and split data caches (scalar data cache augmented with victim cache, and a separate array data cache) are effective for embedded systems. With such a L-1 cache organization for embedded applications, our results show significant reductions in the number of cache misses, translating into reduced cache access times; reductions in required cache capacities, power consumptions and reduction in the number of execution cycles. This is primarily because we used separate caches which eliminate conflict among different data type that exhibit divergent access behaviors. Since lower miss rates at L-1 reduce the number of times one needs to access L-2 cache, we can reduce the size of L-2 cache. This saved area can be used for other purposes or further power reductions can be achieved by partially or completely shutting down L-2 caches. The energy savings result from the reduced number of cache misses, which in turn reduces the number of trips to higher levels of memories, often crossing chip boundaries.

The key contributions of this work are as follows.

While partitioned caches have been explored previously, those studies relied on dynamically detecting spatial and temporal localities and directing cache accesses to different cache partitions. Dynamic detection requires additional hardware. Also the identification of different locality types requires observation of hundreds or even thousands of

memory accesses, and this leads to delays in adapting the caches for different locality types. In our research we use compile time analyses to detect array (or stream) and scalar accesses. This can be achieved by using different memory access instructions for each data type (e.g., Array_Load/Array_store and Load/Store). In our design, we first address the problem of improving L-1 data cache performance for embedded systems through the use of separate array and scalar data caches. We extend our architecture by augmenting the scalar cache with a victim cache [6], and augmenting instruction caches with prefetch buffers. We exhaustively explore optimal array, scalar and instruction cache sizes for each application, to achieve desired levels of energy savings while maintaining required performance levels. The analysis to identify optimal cache configurations is conducted off-line. These analyses are used to customize caches for each application by setting configuration vectors. Our reconfigurability leads to significant savings in cache capacities.

Prefetching is not popular for embedded applications as such techniques require additional buffers to store prefetched instructions (or data). We show that some of the unused cache portions resulting from our optimally reconfigured L-1 and L-2 caches can be used for instruction prefetching, thus eliminating the need for additional buffer space. Similarly, while branch prediction and branch target buffers have been used for desktop computer systems, such techniques add to the hardware complexity and lead to additional power consumption. By using the unused portion of caches resulting from our reconfigurable caches, we can explore branch prediction and branch target buffers without requiring additional buffer space. Finally we also explored the energy savings if the unused cache partitions are shut down.

The rest of the paper is organized as follows. Section 2 describes the architectural design of our reconfigurable cache. In section 3 we describe the benchmarks and simulation environment used in our study. In section 4 we evaluate our reconfigurable L-1 and L-2 caches. In section 5 we evaluate two different possible uses of silicon area savings resulting from our cache organizations. In section 6 we provide a survey of related work. Finally in section 7 we present our conclusions and possible future research.

II. ARCHITECTURAL DESIGN OF RECONFIGURABLE CACHE

Figure 1 shows our reconfigurable split data cache architecture, with L-1 array and scalar data caches, victim cache with scalar data cache, the L-1 instruction cache augmented by a small prefetching buffer and L-2 instruction and data caches. For our purpose of the experiments in this study, we marked traces as array accesses and scalar accesses. We identified array references by assuming that such references involve some form of indexing. While we cannot assure that all array data items are captured by our method, our analyses for selected sample programs show that most of the array data items (better than 99%) have been correctly identified. In an actual implementation of split caches, compiler will allocate data items to array and scalar cache portions, and use different memory access instructions for each portion (viz., array_load/array_store and load/store).

In order to implement reconfigurable caches, only a small amount of additional logic is required. Additional wiring is also necessary from the cache to the processor for directing data to/from the various partitions. The most challenging part in designing a reconfigurable cache is the implementation of a mechanism to divide the cache into different (variable sized) partitions and designing an addressing scheme that can address any partition. Ranganathan *et al* in [10] have already proposed two partitioning and addressing schemes: "Associativity based partitioning" and "Overlapped wide-tag partitioning". In our design we use "Overlapped wide-tag partitioning" scheme. In this scheme, the key challenge is to devise a mechanism so that the size of the tag array can be dynamically changed with the size of partitions (since the number of bits in a tag and index fields of the address will vary based on the size of the partition). We restrict the size of each partition to a power of 2 and support a limited number of possible configurations (usually two or three). A reconfigurable cache with N partitions must accept N addresses and generate N hit/miss signals. In order to track the number and sizes of the partitions and control hit/miss signals, a special hardware register (viz., configuration vector) is needed. This register will be a part of the processor state [10].

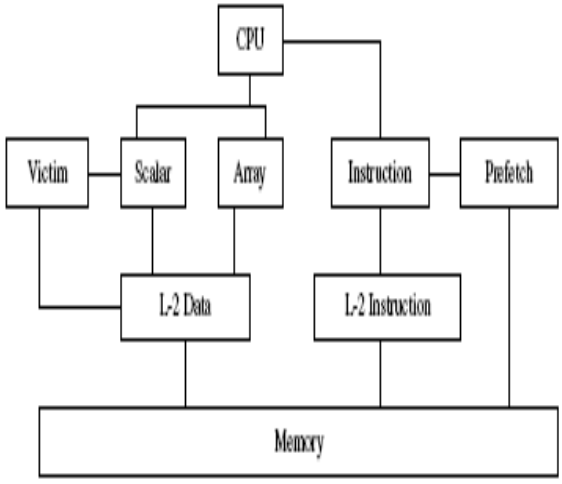

Figure. 1. Reconfigurable split cache organization

A reconfigurable cache can be used in different ways. The best configuration for an application can be determined by executing applications with different configurations. Software profiling tools can be used to identify portions of code that exhibit different cache behaviors. Reconfiguration can also be implemented dynamically with appropriate hardware profiling and an automatic cache tuner. However this requires additional hardware to profile applications. For L1 level we perform exhaustive search to find the best configuration offline. For L2 we explored offline to find configurations that reduce the silicon area. For both cases the configuration vector is set appropriately to customize caches for each application.

III. EXPERIMENTAL METHODS

In this section we describe the experimental framework and the benchmarks used for this study. We used benchmark programs from the MiBench suite [5]. MiBench includes benchmarks from several representative embedded application domains: (1) Automotive and Industrial Control, (2) Office Automation, (3) Networking, (4) Security, (5) Consumer and (6) Telecommunications. For our study we included selected programs from these application domains. The descriptions of the benchmarks used in our studies are listed in Table 1.

Table 1: Descriptions of benchmarks

Benchmark	***Class***	***Description***	***% of load/store***	***Name in Figure***
bit counts	Auto./Industrial	Test bit manipulation	10	bc
Qsort	Auto./Industrial	quick sort	52	qs
dijkstra	Network	Shortest path problem	34.8	dj
blowfish	Network	Encription/decription	29	bf
Sha	Security	Secure Hash Algorithm	19	sh
rijndael	Security	Encryption Standard	34	ri
string search	Office	Search mechanism	25	ss
Adpcm	Telecomm	Variation of PCM standard	7	ad
CRC32	Telecomm	Redundency check	36	cr
FFT	Telecomm	Fast Fourier Transform	23	ff

Table 2: Default parameters defined by SimpleScalar

Fetch queue size	4	LSQ	8
Fetch speed	1	FUs	alu:4, mult:1, memport:2, fpalu:4,
Decode, width	4	Memory latency	18 cycles for first chunk, 2 thereafter
Issue width	4 out-of-order	Memory width	8 bytes
Commit width	4	Instruction TLB	16-way, 4096 byte page, 4-way, LRU, 30 cycle miss penalty
RUU (window) size	16	Data TLB	32-way, 4096 byte page, 4-way, LRU, 30 cycle miss penalty

Our experimental environment is built on the SimpleScalar (version 3.0d) simulation tool set [3], modeling an out-of-order speculative processor with a two-level cache hierarchy. We rely on default parameters used by SimpleScalar and are shown in Table 2. The base cache system, against which we compare our designs, uses an 8k byte L-1 instruction cache, an 8k byte L-1 data cache, a 32k byte L-2 instruction cache and a 32k byte L-2 data cache. We used a modified CACTI [16] timing model to obtain area, access time and power overheads incurred by reconfigurable caches. Our analyses do account for additional hardware needed for reconfiguration.

IV. EVALUATION OF RECONFIGURABLE SPLIT CACHE

In the following two sections we describe our strategies for reducing power consumption while maintaining performance of caches.

A. Results with L-1 Data and Instruction Caches:

We believe that the main problem with data cache is the negative interaction between two different locality types - temporal and spatial localities, exhibited by different data types. To solve this problem, for L-1 data cache, we use separate scalar and array (or stream) caches, and augment direct mapped scalar caches with a small victim cache. As noted previously, in a real system, compiler will assign data to array and scalar cache portions, and use separate instructions to access these portions (e.g., load/store and array_load/array_store). In addition, with reconfigurability we permit varying the sizes of scalar and array caches for each application. We augment our L-1 instruction cache with a small buffer to permit for effective prefetching of instructions. Even with the additional power needed for prefetching, we show significant reductions in total power consumed by all our caches (by 47% on average). The three series in Figure 2 represent percentage reductions in power, chip area and access times for L-1 instruction and data caches respectively.

In order to obtain these results, we exhaustively searched for optimal cache sizes for each cache structure (array, scalar and instruction cache). In this figure we also show the average power, area and cache access time across all the benchmarks used in our experiments (last series). As can be seen, for instruction cache, on average we achieve reductions of 47% in power, 95% in area and 37% in cache access times. Here it should be mentioned that for benchmark “ss” we did not achieve any reductions in power or access times. For data caches, on average we show more than 50% reduction in both power and area consumption. For each of the benchmarks we also achieve reduction in cache access times. However, considering the worst case such as “qs” and “bf”, less than 10% access time reduction was achieved. In exploring optimal configuration, we varied only cache-size (not line-size or associativity)—we start from smaller to larger sizes (from 256 to 8K bytes). At L-1, for each benchmark we exhaustively explored all cache size combinations for array, scalar and instruction caches to find the best configuration those results in optimal power, area and access times. In Table 3 we provide the optimum configurations for each benchmark.

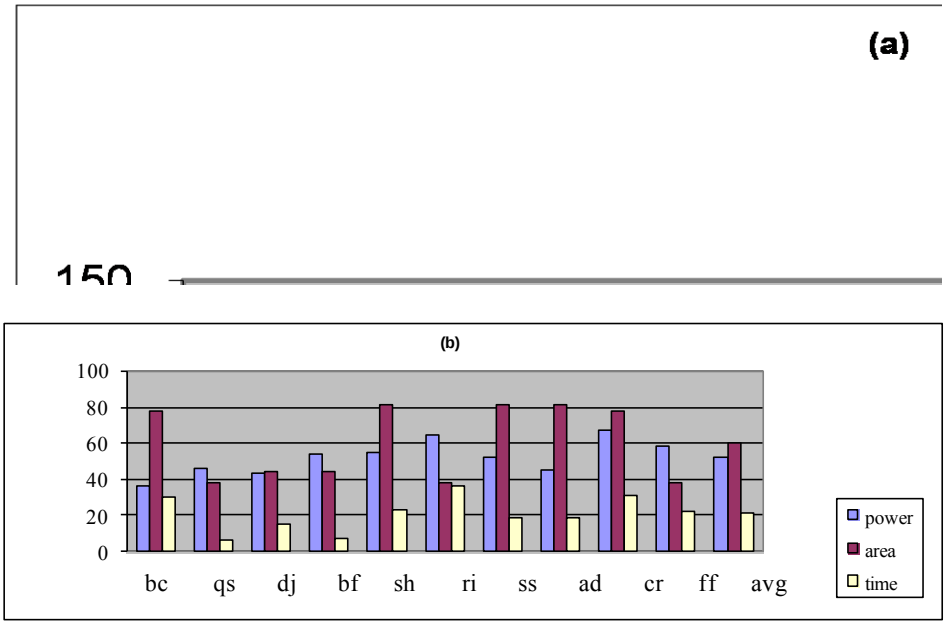

Figure. 2. Percentage reduction of power, area and cache access time for L-1 (a) instruction and (b) data caches

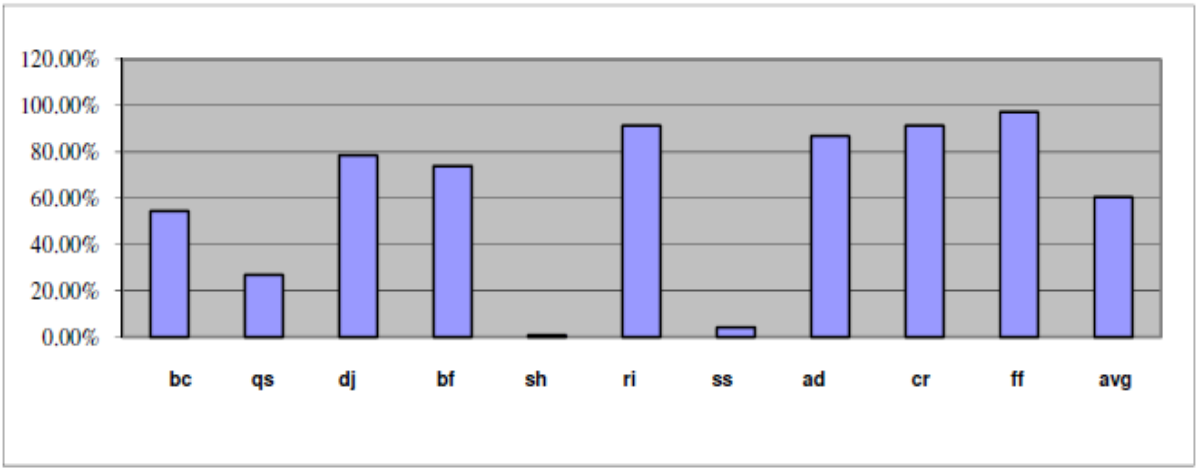

Figure. 3. Percentage reduction of number of access in L-2 caches

Table 3: Cache configurations yielding lowest power, area and cache access time

Benchmark	***L-1Instruction cache***	***Array cache***	***Scalar cache***	***L-2 Instruction cache***	***L-2 Data Cache***
bit counts	256 bytes	512 bytes	512 bytes	2k	2k
qsort	256 bytes	1k	4k	2k	32k
dijkstra	1k	512 bytes	4k	4k	8k
blowfish	1k	512 bytes	4k	2k	8k
sha	256 bytes	512 bytes	1k	1k	8k
rijndael	512 bytes	1k	4k	4k	32k
stringsearch	256 bytes	512 bytes	1k	1k	16k
adpcm	256 bytes	1k	512 bytes	1k	4k
CRC32	256 bytes	512 bytes	512 bytes	4k	2k
FFT	1k	1k	4k	4k	16k

B. Results with L-2 Data and Instruction Caches:

Unlike L-1 caches where cache behavior is mainly controlled by locality types, for L-2 cache, the main concern is the number of misses from L-1 caches. For most of the benchmarks (except "sh", "qs" and "ss"), our L-1 caches achieved excellent reductions in the number of cache misses, resulting in fewer visits to L-2 caches. For benchmark "ff" we were able to achieve as much as 96% reduction in the number of misses. In Figure 3 we show the percentage reduction in the total number of L-2 accesses (which is the number of misses in L-1 caches) compared to those in the base cache system. This implies that we can reduce the size of L-2 caches, yet maintain the desired level of performance and the size of L-2 caches must be configured based on each application. Since the number of access to L-2 caches is small, we did not see a need for split array and scalar L-2 data caches. We also felt that it is unnecessary to perform an exhaustive search of all possible L-2 cache configurations (as done for L-1 cache, see Figure 2 and Table 3). We only explored configurations that reduce the silicon area needed for L-2 caches. We start with a very small L-2 caches, and continuously increase the sizes of the caches until no further reductions in misses are achieved (compared to the base configuration of 32KB L-2 instruction and data caches). Since both cache access time and power consumption is determined by the number of misses, this method allows us to find the smallest cache sufficient to meet performance requirements without increasing power consumption.

The three series in Figure 4 represent the percentage reductions in area, access time and power for L-2 instruction (a) and data (b) caches respectively. In this figure we also show the average area, cache access time and power across all the benchmarks used in our experiments (last series in the figures). As can be seen, for instruction cache, on average we achieve more than 80% reductions in power, in area and in cache access times. At the same time we have achieved significant improvement for each benchmark. However for data caches we can observe a different situation. Although on average we have achieved more than 50% reductions in power, in area and in cache access times, for some benchmarks we did not achieve any improvement. As we can see from Figure 4(b) for benchmark "qs" we did not achieve any reductions in power or area consumptions. Similarly for benchmark "ri" we did not achieve any reductions in area consumptions.

Our goal is not only to reduce silicon area, cache access time and power consumption, but also to confirm that there is no degradation in overall performance. In Figure 5 we compare the execution cycles (not just memory access times but actual execution times of the applications) of the selected benchmarks of our proposed cache systems (with optimal configurations for various L-1 and L-2 structures as outlined previously) with that of base cache systems. In this figure we also show the average reduction across all the benchmarks used in our experiments (last series). As for benchmarks "bc" and "ad" have less than 10% load and store (Table 1) instructions, for both we did not achieve any reductions in numbers of execution cycles.

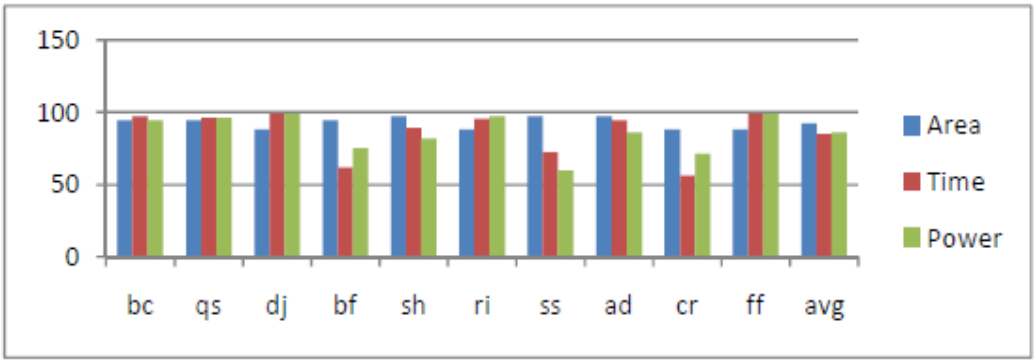

Figure.4 (a). Percentage reduction of area, power and cache access time for L- 2 instruction caches

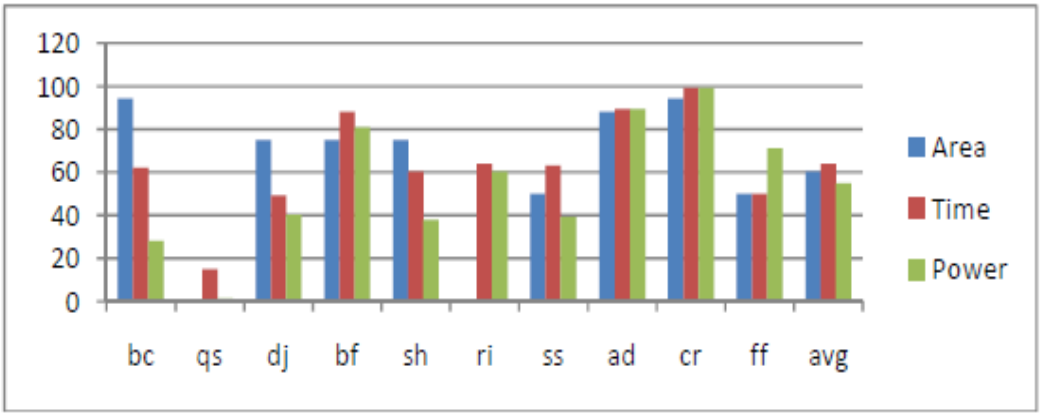

Figure 4(b). Percentage reduction of area, power and cache access time for L-2 data caches

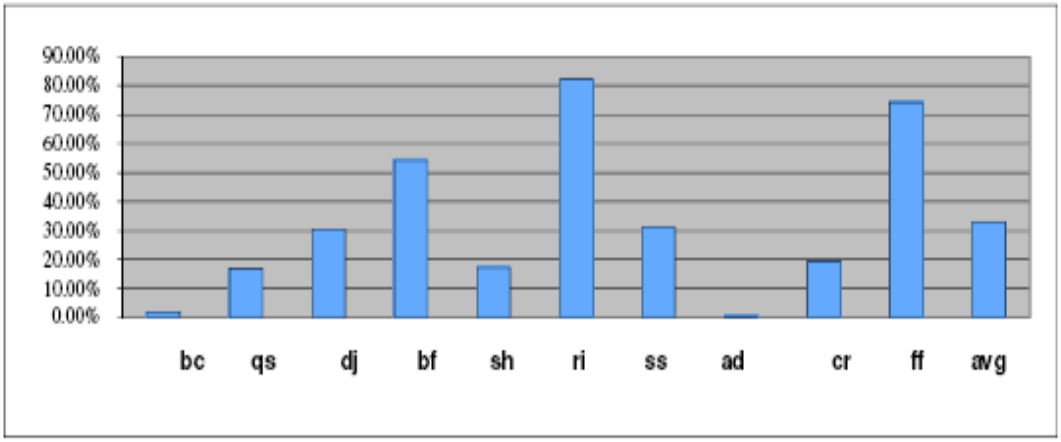

Figure. 5. Percentage reduction of execution cycles

V. ACHIEVING FURTHER IMPROVEMENT

When provided with larger caches than needed for an application, we can either disable unused sub-arrays to save power or use the sub-arrays for purposes other than traditional caching, so that the overall execution performance of an application can be further improved. In the following two sections we evaluated both options.

A. Utilization of the Unused Areas:

Techniques such as hardware prefetching, instruction reuse, value prediction and branch prediction have been used effectively in desktop applications. However, these techniques require additional space for implementing look-up tables or buffers (viz., prefetch buffers, trace caches, branch target buffers). And the performance gains from these techniques are proportional to the sizes of these tables [12]. Because of the additional tables needed, these techniques are often viewed as inappropriate for embedded systems [19]. Since we show reductions in cache sizes in our designs, these savings may be used to implement buffers or look-up tables to implement prefetching or branch prediction ideas.

a) Hardware and software Prefetching: Prefetching or exploiting the overlap of processor computations with data accesses has proven effective in tolerating long memory latencies [2, 9]. Successful prefetching can reduce miss rates, but scheduling the prefetching requests is still a challenge. Prefetching too far ahead not only wastes the embedded system's valuable power but may also cause cache pollution, since the prefetched data may displace data that will be used prior to the prefetched data. This in turn leads to additional misses and wasted energy. On the other hand prefetching too late will not hide the latency. In our reconfigurable cache we can use unused cache areas as prefetch buffers to avoid cache pollution. We use prefetching for both array data items and instructions at L-1 cache level. The prefetching areas can be implemented in cache arrays with minor hardware changes.

Figure 6 shows the percentage improvement in power consumptions and cache access time when using prefetching at L-1 level for both array and L-1 instruction caches, compared to the base cache system. As can be seen, for all the benchmarks there is a significant reduction in cache access times and power consumption. The data in Figure 6 accounts for the additional power needed for prefetching. In Figure 7 we present the percentage improvement in terms of execution cycles of an application using prefetching (along with our scalar, victim and array caches) when compared to the base cache system. As we can see for benchmark "ri" we obtain as much as 85% reduction in number of execution cycles. The average reduction in execution cycles is 47%. Again for benchmarks "bc" and "ad" we did not achieve any reductions in numbers of execution cycles as both have less than 10% load and store instructions (Table 1).

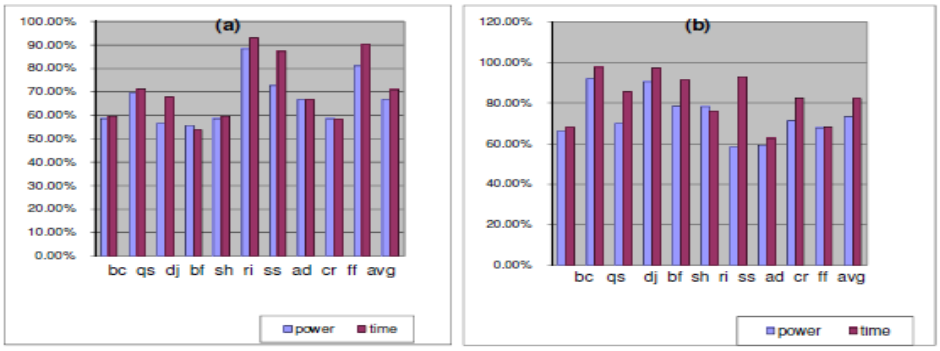

Figure. 6. Percentage of power and access time reduction with prefetching for (a) instruction and (b) data caches

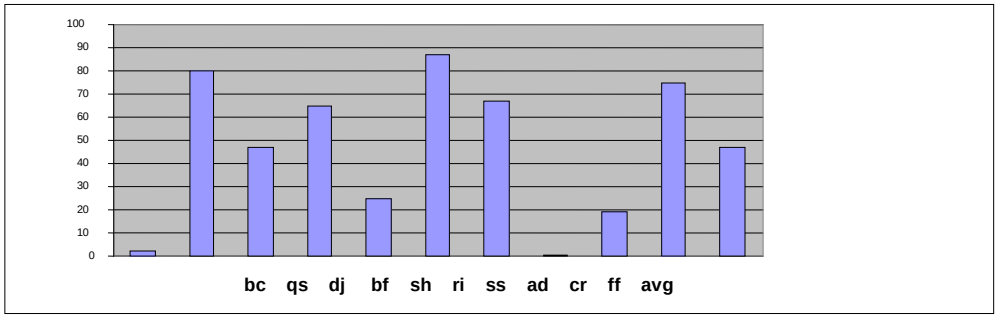

Figure. 7. Percentage reduction of execution cycles after implementing prefetching

b) Hardware optimization techniques with branch prediction tables: Modern processors utilize speculative execution of instructions using branch prediction, instruction reuse (or value prediction) and function reuse technique to improve performance [13]. It has been found that many instructions and functions, are repeatedly executed with the same inputs, producing the same outputs [13]. These behaviors can be exploited to reduce the number of instructions/functions executed dynamically as follows: by buffering the previous results of instructions and functions, future dynamic instances of the same instructions (or functions) can use the results from previous executions by looking up the buffered information [13]. For branch instructions, branch decisions are correlated and can be predicted. Branch predictions along with branch target buffers have been used to eliminate pipeline stalls and improve instruction level parallelism (ILP). Until recently these optimization techniques have been studied for embedded applications because all such techniques require additional hardware that lead to additional silicon area and increased power budgets. Since we can save some cache resources using our reconfigurable designs, the saved space can be used to build needed look-up tables to implement these techniques in embedded systems. In this section we compare the percentage improvement in the number of execution cycles for each application using branch prediction when compared to the base cache system without branch prediction. In this study we used combined prediction with both bimodal predictor and 2-level adaptive predictor. The table size for bimodal predictor is 2048 and for 2-level predictor is 1024 with a history width of 8. The meta-table size of combined predictor is 1024. For all of the applications, we achieve enough space from L-2 instruction and data caches to accommodate space for these predictors (see Figure 4). Figure 8 shows the percentage improvement in number of execution cycles for each benchmarks using branch prediction when compared to the base cache system without branch prediction. For loop intensive benchmark "ff" we achieved 75 % reduction in execution cycle, since for such applications branch prediction can be very accurate.

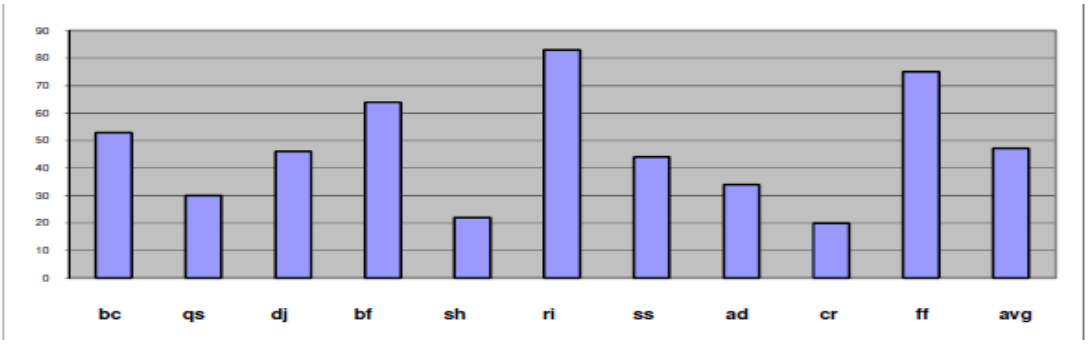

Figure. 8. Percentage reduction of execution cycle after implementing branch prediction

B. Shut Down Portions of Level two Instruction and Data Caches:

The most important concern for the designers of any embedded system is the power consumed by applications. As our proposed design for L-1 instruction and data caches result in reductions in the number of misses, translating into fewer accesses in L-2 caches, we may shut down unused portions of L-2 cache. In this section we explore the power savings from such shut downs. This requires us to model both static and dynamic power consumed by cache memories. In previous sections we only accounted for dynamic power since all cache portions are left active (not shut down). In Figure 9 we show the percentage reduction in total power consumption (both dynamic and static) for (a) instruction and (b) data caches. In each figure we show the power reductions with and without prefetching. Here for prefetch buffer we are using area saved from L-1 instruction and data caches (and not from L-2 cache savings). It should be mentioned that although prefetch consumes additional power, the benefits achieved in terms of reduced cache misses outweighs the extra cache and hardware needed for prefetch.

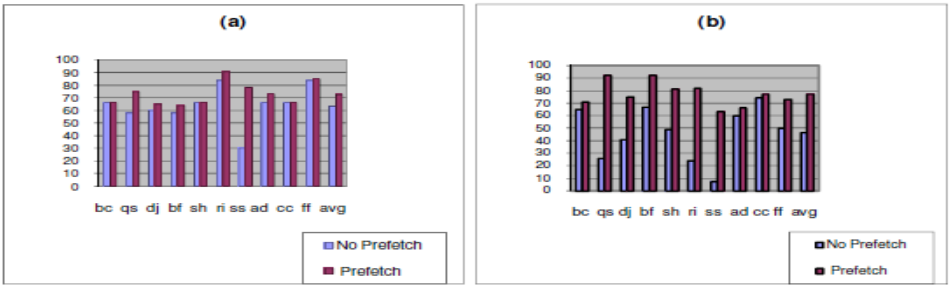

Figure. 9. Percentage of dynamic and static power reduction without and with prefetching for (a) instruction and (b) data caches

VI. PREVIOUS WORK

In [10] the authors proposed a reconfigurable data cache architecture for general purpose processors, focusing on one particular option of using the saved silicon area; namely for "instruction reuse". In [1] authors proposed "selective cache ways" to selectively disable portions of a data cache, trading off performance with power. In our research we provided a detailed analysis of silicon area savings, reduction in execution cycles and power consumed when our reconfigurable cache structures are used. We also perform detailed analyses of achieving additional performance improvement by using saved silicon area for prefetching and branch prediction. We included reconfigurability for both L-1 and L-2 caches. Work by Zhang *et al* [17-19] is closely related to our research, as they evaluate reconfigurable unified data caches for embedded applications. Later in [4] the authors analyzed the possibilities of reconfigurability of L-2 caches. Unlike their work, we do not see set-associativity as an important reconfigurable design parameter. In our design, both our array and scalar caches are designed as direct mapped caches, and we use victim caches to reduce conflict misses of scalar data. Several studies have been reported on split data caches [14, 12, 8, 15] but none of these studies explored reconfigurability. Moreover, previous split cache investigations relied on dynamic detection of locality types (spatial versus temporal). We use compile time analysis to identify array and scalar accesses.

VII. CONCLUSION

In this paper we introduced a novel cache architecture for embedded microprocessor platforms. When using our proposed caches for embedded applications, our results show excellent reductions in both memory size and memory access times, translating into reduced power consumption and improved overall execution times. Our cache architecture reduces the cache area by as much as 95% for L-1 instruction and 67% for L-1 data caches, access times by as much as 72% for L-1 instruction and 36%, for L-1 data caches and power consumption by as much as 75% for L-1 instruction and 67% L-1 data caches respectively when compared with an 8k byte L-1 instruction and 8k byte L-1 data caches. These reductions can be profound when working with small L-1 caches often found in embedded systems. For L-2 instruction cache we achieved on average 50% improvement in power and more than 80% reduction in access times. Whereas for L-2 data cache the average improvement is 50% in power and more than 60% in access times. We also show that the saving in cache sizes resulting from our designs can be used for other processor features including instruction and data prefetching, branch prediction buffers. We evaluate the potential benefits of such techniques for embedded applications. We also explored the energy savings if the unused cache partitions are shut down.

VIII. ACKNOWLEDGMENT

This paper was partially supported by Net-Centric IUCRC.

VIII. REFERENCES

[1]. H. Albonesi, "Selective Cache Ways: On-Demand Cache Resource Allocation," Journal of Instruction Level Parallelism, 2000.

[2]. J. L. Baer, and T. F.Chen, "An effective on –chip preloading scheme to reduce data access penalty." Proceedings of the Supercomputing, 1991, pp.176-186.

[3]. D. Burger, and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0", Tech. Rep. CS-1342, University of Wisconsin-Madison, 1997.

[4]. A. Gordon-Ross, F. Vahid, and N. Dutt, "Automatic tuning of two-level caches to embedded applications", Design Automation and Test in Europe Conference (DATE), 2004.

[5]. M. Guthaus, J. Ringenberg, T. Austin, T. Mudge, R. Brown, "MiBench: A free, commercially representative embedded benchmark suite", Proceedings of the IEEE 4th Annual Workshop on Workload Characterization, 2001.

[6]. N. P. Jouppi, "Improving direct-mapped cache performance by the Addition of a small fully associative cache and prefetch buffers", Proceedings of the 17th ISCA, 1990, pp. 364-373.

[7]. M.B. Kamble, and K. Ghose, "Energy-efficiency of VLSI caches: a comparative study", Proceedings of Tenth International Conference on VLSI Design, 1997, pp.261-267.

[8]. J. H. Lee, S. D. Kim and C. Weems, "Application adaptive intelligent cache memory system", ACM Transactions on Embedded Computing Systems, Vol. 1, Issue. 1, 2002, pp. 56-78.

[9]. C. K. Luk, and T. Mowry, "Compiler based prefetching for recursive data structures", Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, 1996, pp. 222-233.

[10]. P. Ranganathan, S. Adve, and N. P. Jouppi, "Reconfigurable Caches and their Application to Media Processing," Int. Symposium. on Computer Architecture. 2000.

[11]. F. J. Sanchez, A. Gonzalez, and M. Valero, "Software management of selective and dual data caches," IEEE TCCA Newsletters, 1997, pp. 3-10.

[12]. Y. Sazeides, and J. E. Smith, "The predictability of Data values," Proceedings of the 30th Annual International Conference on Microarchitecture, 1997, pp 248-258.

[13]. A. Sodani, and G. Sohi, "Dynamic Instruction Reuse," Proceedings of 24th Annual International Symposium on Computer Architecture, 1997, pp.194 - 205.

[14]. M. Tomasko, S. Hadjiyiannis, and W. A. Najjar, "Experimental evaluation of array and scalar caches," IEEE TCCA Newslatters, 1997, pp. 11-16.

[15]. O. S. Unsal, I. Koren, C. M. Krishna, C. A. Moritz, "The Minimax Cache: An Energy-Efficient Framework for Media Processors," 8th International Symposium on High-Performance Computer Architecture, 2002, pp. 131-140.

[16]. S. J. E. Wilton, and N. P. Jouppi, "CACTI: an enhanced cache access and cycle time model," IEEE Journal of Solid-State Circuits, Volume: 31 Issue: 5, 1996, pp.677 -688.

[17]. C. Zhang, F. Vahid, and W. Najjar, "Energy benefits of a configurable line size cache for embedded systems," IEEE International Symposium on VLSI Design, 2003.

[18]. C. Zhang, F. Vahid, and W. Najjar, "A highly configurable cache architecture for embedded systems," Proceedings of 30th Annual International Symposium on Computer Architecture, 2003, pp.136 -146.

[19]. C. Zhang, and F. Vahid, "Using a victim buffer in an application-specific memory hierarchy," Design Automation and Test in Europe Conference (DATE), 2004, pp. 220-225.

[20]. A. Naz, K. Kavi, W. Li and P. H. Sweany, "Tiny split data caches make big performance impact for embedded applications," Journal of Embedded Computing (Special Issue on Embedded Single-Chip Multi-core Architectures from System Design to Application Support), Vol.2, No.2, 2006, pp 207-219.