SPLIT ARRAY AND SCALAR DATA CACHES: A COMPREHENSIVE STUDY OF DATA CACHE ORGANIZATION

Afrin Naz, B.Sc., M.S.

Dissertation Prepared for the Degree of

DOCTOR OF PHILOSOPHY

UNIVERSITY OF NORTH TEXAS

August 2007

Naz, Afrin. Split array and scalar data cache: A comprehensive study of data cache organization. Doctor of Philosophy (Computer Science) August 2007, 142 pp, 16 tables, 50 figures, references, 84 titles.

Existing cache organization suffers from the inability to distinguish different types of localities, and non-selectively cache all data rather than making any attempt to take special advantage of the locality type. This causes unnecessary movement of data among the levels of the memory hierarchy and increases in miss ratio. In this dissertation I propose a split data cache architecture that will group memory accesses as scalar or array references according to their inherent locality and will subsequently map each group to a dedicated cache partition. In this system, because scalar and array references will no longer negatively affect each other, cache-interference is diminished, delivering better performance. Further improvement is achieved by the introduction of victim cache, prefetching, data flattening and reconfigurability to tune the array and scalar caches for specific application.

The most significant contribution of my work is the introduction of novel cache architecture for embedded microprocessor platforms. My proposed cache architecture uses reconfigurability coupled with split data caches to reduce area and power consumed by cache memories while retaining performance gains. My results show excellent reductions in both memory size and memory access times, translating into reduced power consumption. Since there was a huge reduction in miss rates at L-1 caches, further power reduction is achieved by partially or completely shutting down L-2 data or L-2 instruction

caches. The saving in cache sizes resulting from these designs can be used for other processor activities including instruction and data prefetching, branch-prediction buffers. The potential benefits of such techniques for embedded applications have been evaluated in my work.

I also explore how my cache organization performs for non-numeric data structures. I propose a novel idea called “Data flattening” which is a profile based memory allocation technique to compress sparsely scattered pointer data into regular contiguous memory locations and explore the potentials of my proposed Spit cache organization for data treated with data flattening method.

ACKNOWLEDGEMENTS

My gratitude and thank, first and foremost, belongs to Almighty Allah Subhanatala, who has given me this life, wisdom and strength.

There are a number of people who have been important in the completion of my dissertation, both academically and personally. This work would not have been possible without them. I owe them a great deal.

I thank my Prophet Muhammad (Peace be upon him.), Hazrat Abu Bakar Siddiq, Khwaja Bahauddin Naqshband, Shaykh Abd Allah al-Faiz ad-Daghestani, Shaykh Nazim Haqqani and finally my Shaykh Hisham Kabbani to enlighten my heart with their blessings.

Dr. K. Kavi has been my faculty advisor during my doctoral research in University of North Texas. He has taught me more about Computer Science than anyone else. His professional and personal standards would be an example to any student. It has been a privilege to work with him. My very best thanks to him, whom I owe a depth of gratitude. I am indebted to Dr. P. Sweany and his wife Dr. M.F. Sweany for all of their support and encouragement. Without their help I would not be able to finish writing of this dissertation. I also give my thanks to Dr. R. Brazile, Dr. S. Tate and Dr. M. Rezaei for devoting their time and support. I would like to thank the staff of Computer Science and Engineering Department of UNT for all the times they patiently took care of administrative matters of my PhD.

I would like to thank my husband, Shahed, for his love, support, patience that he has given me for last ten years. He is the one who made me believe that I can finish my Ph.D. degree, he took care of my children when I needed to work, he gave me encouragement when I lost my hope, he finished the editing of this dissertation when I fall into sleep. I also want to thank my son Nazim and daughter Ameera for all the sacrifice they had made since they are born. I used to keep Nazim in my laboratory since he was three weeks old. Today, Ameera's birthday is not being celebrated only because I need to submit my dissertation tomorrow.

Today I am so happy that I have fulfilled the promise I made to my mom, Prof Razia Begum eleven years ago that someday I will attain a PhD degree like my dad, Dr. Manzural Hosain. I know both of my parents are happy for me. I also like to thank my brother, Dr. Nazmul Hosain and other relatives to provide support during this five and half years journey.

My Ph.D. is dedicated to my spiritual leader Shaykh Muhammad Hisham Kabbani.

May this work be a glory to Almighty Allah Subhanatala.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Although caching dates back to Von Neumanns' classic 1946 paper that laid the foundation for modern practical computing, it became vital to the performance of a processor since the beginning of 1990's, as the gap between the processor cycle and memory latency times increased dramatically. Figure 1.1 shows this famous "memory wall" problem; how fast Central Processing Unit (CPU) can process data is no longer a

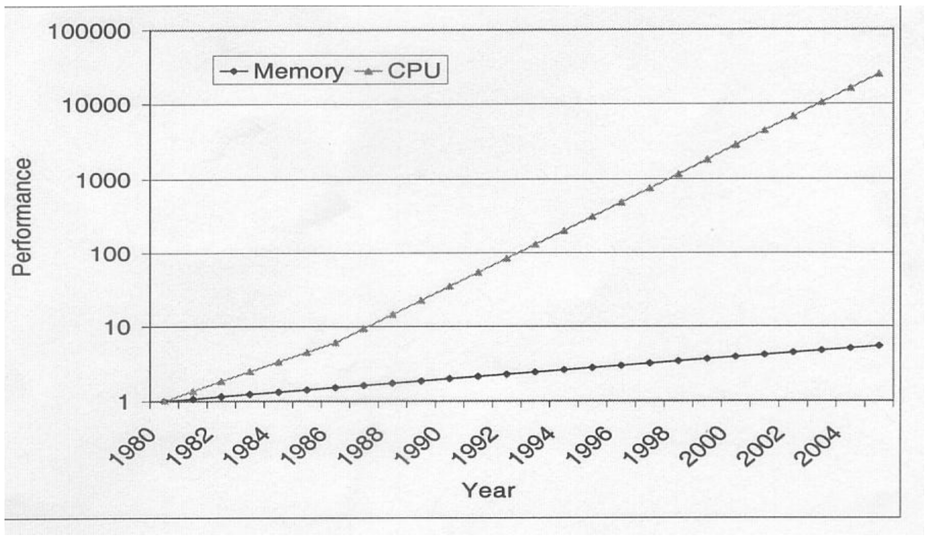


Figure 1.1 CPU-Memory Speed Gap (Note that the y axis is log scaled.) [31]

problem, rather the main issue of concern in todays' computer system is how fast data can be provided to CPU. The problem is even more complicated in pointer intensive applications, which do not have regularity in data stream as scientific applications and for embedded applications, which place requirements along a number of dimensions including tighter constraints on functionality and on implementation.

As the CPU speed has outstripped the rest of the system by many orders of magnitude, more advanced techniques are needed to improve the performance of cache. This makes cache design one of the most important research topic in computer architecture field. According to Hennessy *et al.* [31] a bibliographic search for the years 1989-2001 revealed more than 5000 research papers on the subject of caches. Like most of these works, my research is also motivated by the "memory wall" problem. While searching for solutions, I have noticed a major drawback in conventional cache designs. Conventional caches imply no separation of data based on the nature of the characteristics (locality) exhibited by different data references and try to compromise between these data by sending them all to a unified data cache. I believe, that while transistors are plentiful in current Very-Large-Scale Integration (VLSI) designs, it is useful to allocate more resources to allow intelligent control over memory latency reducing techniques and that it is better to implement multiple smaller dedicated caches because these can be accessed relatively quickly. Being inspired by this realization, I propose "Split data caches." In my framework, the compiler will separate data references according to their inherent locality type and send them to appropriate cache. In this study I also show that when carefully designed, combination of different cache optimization techniques (Viz., prefetching, victim caching, data flattening and reconfigurability.) overcomes each other's deficiency and allows each to provide optimum benefit.

In this dissertation I show that even very small data caches, when split to serve data streams exhibiting temporal and spatial localities and augmented with smarter techniques like prefetching, victim caching, data flattening and reconfigurability, can improve performance of scientific, embedded and pointer intensive applications without consuming excessive silicon real estate or power.

1.1 Proposed Cache Organization

First I describe my proposed cache organization in a step by step manner followed by the presentation of my final cache design.

1.1.1 Scalar Cache and Array Cache

Programs exhibit two types of localities, temporal and spatial. Temporal locality implies that, once an instruction or datum is accessed, a high probability exists that it will be accessed again in the near future, and less likely to do so as the time passes. Scalar data items exhibit this behavior. Spatial locality implies that when an instruction or datum is accessed, nearby instructions or data will likely be accessed in the near future. Array data items have spatial locality. I propose a split cache architecture that will group memory accesses as scalar or array references according to their inherent locality and each group will be subsequently mapped to a dedicated cache partition. The "array cache" is a direct mapped cache with larger block sizes to exploit spatial localities more aggressively by prefetching multiple neighboring small blocks on a cache miss. The "scalar cache" is a set associative cache with smaller block sizes to exploit temporal locality.

1.1.2 Integration of Split Caches with Victim Cache and Stream Buffer

Next I propose the inclusion of a victim cache and a stream buffer together with my split cache architecture to further improve my cache designs. Victim caches are based on the fact that reducing the cache misses due to line conflicts for data exhibiting temporal locality provides an effective way to improve cache performance. Thus Victim caches can augment a direct mapped scalar cache without having to use a set associative cache. Stream buffers tend to eliminate cold misses and prefetch data exhibiting spatial locality. Thus stream buffers can improve array cache performance.

1.1.3 Reconfigurability

Then I try to reduce unnecessary area, power and time consumption by introducing reconfigurability. Reconfigurability is a design alternative that provides flexibility in the system so that resource constraints like area, power and performance can be balanced based on application needs. Ideally, the split cache organization should be dynamically reconfigurable to meet the application requirements. Depending upon the types of locality of data my system can partially shutdown a specific portion of the system. For example if the application has more scalar data then prefetch buffers and even the entire array cache may be shut-down to save power. Or the array cache can be dynamically reconfigured to increase the size of the scalar cache. When the unused cache areas are sufficiently large, these areas may be utilized for other processor activities such as larger branch prediction tables or prefetch buffers. By setting a few bits in a configuration register, the cache can be configured in software for optimum sizes for each of my array and scalar caches in Level one (L-1) and Level two (L-2) levels to utilize the unused area for other processor activities or to shut down. For both cases, the reconfiguration leads to only a small overhead in terms of time, power, silicon area and hardware complexity.

1.1.4 Data Flattening

Finally I try to explore how my proposed cache organization performs for non numeric data or non-array data structures. I propose a novel idea called “data flattening” which is a profile based memory allocation technique to compress sparsely scattered pointer data into regular contiguous memory locations (Like array data.). Then I explore the potentials of my proposed spit cache organization for pointer data treated with data flattening method.

1.1.5 Final Scheme

In Figure 1.2 I show the design of my final scheme.

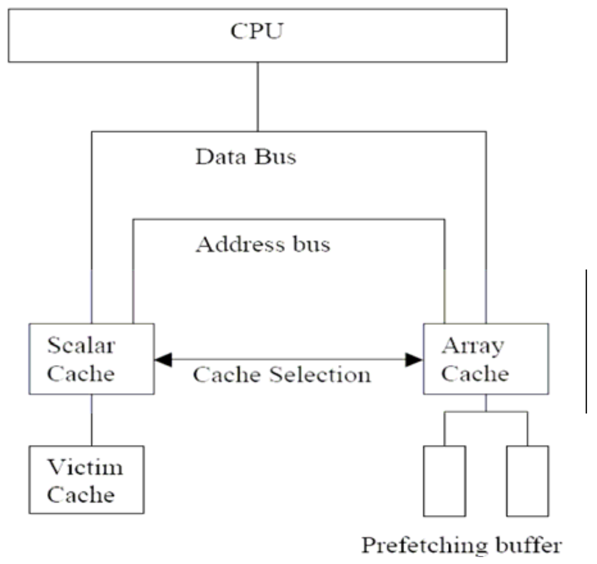


Figure 1.2: Proposed Cache Organization

While other research groups reported on some aspects of cache organizations used in my research, the clear benefits of the combined approach to aggressively improve cache utilization for scientific, pointer intensive and embedded applications by combining split caching, victim caching, prefetching, data flattening and reconfigurability have, to my knowledge, not been studied in the literature.

1.2 Different Application Domains

In order to fully explore the problems in different application fields and describe my solutions, I first study existing cache memories for two classes of application domains: “desktop applications” and “embedded applications.” The problems with desktop applications are more general as they are also applicable to embedded applications. Whereas for embedded applications the situation is more complicated because on top of these general problems (Inherited from desktop applications.), they have their own set of constraints. Finally I explore “pointer intensive applications” because both “desktop applications” and “embedded applications” utilize pointer based variables in their codes. Hence the problems with pointer intensive applications are present in any member of either desktop or embedded applications which have intensive non-numeric (pointer) data.

1.2.1 Desktop Applications

The success of cache memories has been explained by the property of *locality of reference* [74]. Caches exploit locality to shorten the effective access time to data, thereby reducing the cost of accessing main memory. To improve cache performance, modern processors rely on split cache architecture, at least on the first cache level, with separate instruction and data caches, since instruction and data streams exhibit different types of localities. The locality within the data address stream is also not uniform. However it is not common to see a separation of data caches based on access localities exhibited. Since not all data items exhibit both spatial and temporal localities, using a common data cache is inefficient at adapting to spatial and temporal localities. This traditional method of using a single data cache causes unnecessary movement of data among the levels of the memory hierarchy, causing significant interference between unrelated data inside the cache, causing cache pollution (Viz., untimely removal of needed data.) and unnecessary increases in miss ratio, memory access time and memory bandwidth.

Hence it seems worthwhile to explore separate caches to deal with the two types of localities in data intensive applications. In this work, I have simulated a direct mapped array cache with large block sizes for stream and array references exhibiting spatial locality in order to permit prefetching and reduce compulsory misses. I use a set associative scalar cache with more numbers of smaller blocks to overcome high conflict misses. I have experimentally studied the impacts of split data caches using Standard Performance Evaluation Corporation (SPEC) floating point 2000 benchmarks. My results have shown that the split cache organization achieves lower miss rates and shorter average access times even when the combined size of array and scalar caches is roughly one quarter the size of a unified data cache.

This success has encouraged me to extend my research to explore other ideas for further improvement. I have investigated the interaction between three established methods, split caches, victim cache and stream buffer for improving hit rates in the first level of memory hierarchy. In my proposed cache organization, victim cache nicely complements the direct mapped scalar cache in terms of minimizing conflict misses, while stream buffers add to my array cache with prefetching capabilities. I show that the inclusion of a victim cache and a stream buffer together with partitioned cache architectures provides an effective solution for alleviating existing problems in cache designs and enhancing the effective cache memory space for a given cache size and cost.

1.2.2 Embedded Applications

Challenges to the design of processing elements for embedded applications are more stringent than those for desktop applications. Desktop systems afford greater design flexibility for cache memories in terms of cache sizes, associativities, block sizes and multi-level caches. Where as, the limitations on the physical size, real-time predictability, energy budget have made caches less common for embedded systems [35]. Because, caches have demonstrated their usefulness in a wide-range of applications, I feel that it is worthwhile investigating new cache organizations to address both performance and power requirements for embedded applications. While performing design space explorations of embedded systems, I have made four key observations.

Observation 1: Extreme cost sensitivity requires that designers of embedded systems pay more attention to physical size and energy requirements than to performance requirements. For almost all battery-operated systems, reducing total energy consumed is critical. Studies have shown that on-chip caches are responsible for 50% of an embedded processor's total power dissipation [82, 83] and, thus, savings in cache memory power can be significant in overall power savings.

Observation 2: For desktop systems, designers typically emphasize speed or throughput, based on average-case behaviors. In contrast, designers of real-time embedded systems emphasize system's accuracy, predictability, and reliability, all related to the system's worst-case behavior. When a real-time embedded system controls critical equipment, execution time variability becomes completely unacceptable. Traditional hardware-managed caches prove unsuitable for embedded systems because they cannot accurately predict the access times to data.

Observation 3: Traditionally, designers of embedded systems have viewed complementing direct-mapped cache with a victim cache as an inappropriate choice [84], because fully associative victim caches consume additional energy and silicon area. For similar reasons, prefetching is considered ill suited for embedded systems [35]. Other performance optimization techniques such as instruction reuse (Using trace caches.), branch predictions are also not implemented in embedded systems as they require additional hardware for implementing look-up tables, which lead to increased area and power requirements.

Observation 4: The manufacturer typically selects a fixed cache architecture as a compromise across several applications. For embedded systems, since each architectural feature must be customized for each application, this "one-size-fits-all" fixed cache design philosophy is not adequate, as this will lead to suboptimal performance and/or power consumption profiles. Thus it is necessary to provide for reconfigurable caches whereby the size of array and scalar caches, their block-sizes, associativities, as well as the sizes of prefetch buffers and victim caches can be chosen for each application.

I claim that my approach that brings together various cache alternatives addresses the challenges faced by embedded system designers. Since my design reduces the effective memory access times, they can be useful even in real-time systems with strict-deadlines. At the same time, because my split cache design leads to smaller cache sizes (To achieve the same level of performance as a unified data cache.), with reconfigurability, one can either shut down portions of cache to save energy, or use the space to improve system performance by converting the unused portions of caches into trace caches (For instruction reuse or data predictions.) or branch prediction buffers.

1.2.3 Pointer-intensive Applications

It is very obvious that my proposed split cache organization will perform better for numeric applications that utilize arrays and streams. But I wanted to explore the benefits of my proposed cache organization for non-numeric applications with pointer data. Pointer data structures contain dynamically allocated data elements that are linked together. In many applications such structures provide for elegant algorithms since the number of elements can grow without a predetermined limit. However, the dynamic allocation cannot assure that consecutive nodes occupy consecutive areas in memory, thus making it difficult to prefetch linked data elements. The lack of (spatial) localities also makes traditional cache designs ineffective. This led computer system designers to investigate new ways for tolerating memory latencies in accessing linked data elements. In my research, I propose to “flatten” the linked data structures into a linear data structure similar to arrays and streams so that I can take advantage of my split data caches described earlier.

Data placement is the method to pack sparsely scattered data into adjacent memory locations to take advantage of spatial locality and prefetching. Other researchers have proposed innovative data placement techniques to improve localities of non-numeric (Or non-array.) data [15, 22]. However they have been proposed in the context of a conventional data cache design and not in the context of a reconfigurable split data cache design. In my context "data flattening" is a profile based memory allocation technique which compresses sparsely scattered pointer data into regular contiguous memory locations (Like array data.). I believe since the non-numeric pointer data will exhibit similar pattern after being flattened into a linear structure my cache organization will prove to be useful for a wide-variety of scientific, non-scientific and embedded applications.

The rest of the dissertation is organized as follows:

In Chapter 2 I present a description of designs and deficiencies of conventional cache memories. I provide a survey on related work on Chapter 3. In Chapter 4 I describe the experimental environment used in my study, including performance metrics, benchmarks and tools used in my study. I begin Chapter 5 with the initial evaluation of split array and scalar data caches and show their uniform superiority over the conventional unified data cache design across all of the benchmarks. For example, a 4 kilobytes 32 bytes block sized scalar cache and 2 kilobytes 128 bytes block sized array cache show 43.41%, 24.14%, 11.76% and 43.33% improvements over a 16 kilobytes 64 bytes block sized unified scalar cache for ar, eq, am and me benchmarks respectively. Then I demonstrate how three inherently different approaches, split cache, victim cache and stream buffer could be combined and make to work together to further improve data access times. For example a 4 kilobytes scalar cache (With 256 bytes victim cache.) and 4 kilobytes array cache (With 512 bytes stream buffer.) on average show 55% improvement over a 16 kilobytes unified data cache for the benchmark set. In Chapter 6 I explore how to design small caches (Including prefetching and victim caching.) that achieve high performance for embedded applications. I also performed a comprehensive evaluation of my proposed methods by comparing with a variety of techniques that have traditionally been applied to improve cache systems. My cache architecture reduced the overall cache size by 43%, access time by 37% and energy consumption by 63% when compared with a unified 2-way set associative cache. In Chapter 7 I explore how reconfigurability can be used to design caches for embedded applications. My cache architecture reduces the cache area by as much as 78%, access times by as much as 36% and power consumption by as much as 67% respectively when compared with an 8 kilobytes L-1 unified data caches. Then I explored how unused cache portions can be used for different optimization methods or how power reduction can be achieved by partial/complete shutdown of L-2 caches. I evaluate my novel “data flattening” method in Chapter 8. My split data caches with flattened data provide excellent improvement for all of the benchmarks; on average 56%, 73% and 43% reduction in access time, power consumption and performance respectively, when compared with original code (Without data flattening.) using unified data cache. Finally in Chapter 9 I draw my conclusion with a brief synopsis of future work.

CHAPTER 2

DESIGN AND DEFICIENCIES OF CONVENTIONAL CACHE MEMORY

I begin this chapter with a description of design and deficiencies of conventional cache memories. Next I present the traditional cache optimization techniques used for desktop applications, followed by a brief description of comprehensive evaluation of my proposed design, which includes all of these traditional techniques. Finally I point out the challenges involved with cache designing for embedded microprocessors.

2.1 Cache and Locality of Reference

In modern systems cache has become the basic and inevitable mechanism for effectively reducing memory access latencies and for improving overall system performance. Caches are typically placed between a large, relatively slow and inexpensive source of information (The lower level of memory.) and a much faster consumer of that information, the processor. The success of cache memories has been explained by the property of locality of reference [74], which is a property exhibited by most programs. The property of locality has two aspects: temporal and spatial. Temporal locality implies that, once a location is referenced, there is a high probability that it will be referenced again soon, and less likely to do so as the time passes; spatial locality implies that when an instruction or datum is accessed it is very likely that nearby instructions or data will be accessed soon. Since cache buffers recently used segments of information, the property of locality implies that needed information is also likely to be found in the cache. A cache exploits this property to improve the effective access time to data and reducing the cost of accessing main memory.

2.2 Deficiencies of Conventional Cache Designs

As the Central Processing Unit (CPU) speed has outstripped the rest of the system by many orders of magnitude and the memory latency problem has continued to grow, some deficiencies of conventional caching have become evident. Conventional caches imply no separation of data based on the nature of the locality exhibited by different data references. As a result all memory references are handled in a uniform manner – whenever a reference misses, a new block is brought into cache at the expense of replacing another block. Because not all data items exhibit both spatial and temporal localities, this simple-minded treatment to the references makes the data cache inefficient at adapting to the two types of localities. Generally, caches exploit temporal locality by retaining recently referenced data for a long time, and spatial locality by fetching multiple neighboring words as a cache block on a cache miss. If a data item exhibits no temporal locality, bringing it into the cache is useless. Likewise if no spatial locality is exhibited by data items, bringing an entire cache block leads to wastage. Thus traditional treatment of cache misses not only causes unnecessary movement of data between the various levels of the memory hierarchy, it may lead to premature displacement of blocks that are likely to be re-referenced (A phenomenon known as cache pollution.). This can become costly if the newly loaded data tends to be non-temporal. In any case, an unnecessary increase in miss ratios, memory access times and wasted memory bandwidths occur.

References can be easily divided into two groups according to the types of localities exhibited by the program – the scalar and array (stream) references. Conventional cache techniques perform acceptably for general-purpose scalar references with high temporal locality. But such is not true for array references, which do not reuse data soon or often enough to derive much benefit from caching. Because arrays and streams exhibit only spatial localities and the data sizes are often very large for caches, computations with streams and array access patterns cause mostly compulsory misses (Rather than conflict misses.) and perform extremely poorly in terms of cache behavior.

2.3 Conventional Approaches to Improve Cache Design

For two decades computer architects have proposed smart cache-control mechanisms and novel cache architectures that detect program access patterns and fine-tune cache policies to improve both overall cache use and data localities for desktop applications. Major cache optimization techniques (To improve either or both miss rate and miss penalty.) are generally categorized as: (1) increasing block size and cache size, (2) increasing associativity, (3) cache probing, (4) supplementing the regular cache with victim cache, (5) prefetching data, or (6) including additional cache hierarchy.

2.3.1 Increasing Block Size and Cache Size

The simplest way to reduce miss rate is to use large block sizes; large blocks reduce cold misses and aid in data prefetching. Unfortunately, larger blocks increase miss penalty, which may outweigh the benefits of reduced miss rates. Actually, increasing block size without increasing the total cache size increases other types of misses because increasing only block size (Not cache size.) reduces the number of lines, leading to an increase in conflict and capacity misses [31]. Current desktop computers have used larger caches with larger block sizes for off-chip caches. Some of these caches are as large as the main memories of a decade ago [31].

2.3.2 Increasing Associativity

Another common technique for reducing miss rates is increasing associativity. Caches with higher associativity (4- to 8-way associativity.) have become common in both desktop and server systems. Unfortunately, the design of a first-level cache always involves fundamental tradeoffs between miss rates and access times. Direct-mapped caches have proven to be simpler, easier to design and require less silicon area than caches with higher associativities. The main disadvantage of a direct-mapped cache is the high conflict miss rate—conflict misses typically account for 40% of all direct-mapped cache misses [39]. Conversely for caches with higher associativity, the main advantage becomes lower miss rate, but such caches have higher access times as they require associative searches of sets and multiplexing of the appropriate data words to the processor.

2.3.3 Probe Caches or Modified Set-associative Caches

Recently researchers have proposed Probe caches (Modified 2 way set-associative cache.) that result in miss rates like a 2-way set associative cache, but hit-times like a direct mapped cache [1, 2, 14, 17, 75]. These “sequential search” of set associative caches or probe caches are based on the key observation that associativity is needed only for conflicting blocks and should not be provided at the expense of higher hit latencies for all accesses. In most of these schemes, a traditional direct-mapped cache is conceptually partitioned into sets with 2 blocks per set. Cache access is sequential; first one block is probed, and if the tag doesn’t match the second block in the set is probed. In order to achieve fast access different probe caches use different data structures, ranging from simple hash/mru bit, to complicated way prediction mechanisms [1, 2, 14, 17, 75]. The examples of probe cache include Hash rehash cache [1], Column associative cache [2], MRU cache [17, 75]. Hash rehash cache (HR cache) uses fixed probe order with different hash functions to search the cache [1]. The Column associative cache (CA cache) improves on Hash rehash cache by associating rehash information with each block and improving the replacement algorithm, which reduces the number of second probes required [2]. MRU cache uses a dynamic probe ordering based on most recently used information [17, 75].

2.3.4 Supplementing Cache with Victim Cache

A Victim cache is a fully associative cache, with typically 4 to 16 cache lines that reside between a direct-mapped Level one (L-1) cache and the next level of memory [39]. On a main cache miss, the victim cache is checked before going to the next level of memory. If the address hits in the victim cache the desired data is returned to the CPU and swapped (Or promoted.) with the data currently occupying the primary cache. Upon a miss in victim cache, the next level of memory is accessed and the arriving data is placed in the primary cache, moving the current data to victim cache. In this case an element from victim caches has to be removed (Or written back to next level of memory.) to make room for the newly victimized data.

While set-associative caches, with fewer conflict misses, offer lower miss rates than direct-mapped caches, they cost more and incur longer access times on a hit. Victim caches, in contrast, reduce conflict misses of direct-mapped caches without affecting its fast hit access times. Because victim caches are fully associative (Albeit small.), they can hold many blocks simultaneously that might otherwise conflict in direct-mapped cache. If most of the conflicting blocks can fit in victim cache, both the miss rates and the average access times improve.

2.3.5 General Prefetching

As another technique to improve efficiency, prefetching or exploiting the overlap of processor computations with data access has proven to be effective in tolerating large memory latencies in desktop systems [8, 52]. Although increasing line size presents the simplest way of prefetching, line sizes cannot be made arbitrarily large without both increasing miss rates and greatly increasing the amount of data transferred on cache misses (Thus increasing miss penalties.) [31]. Prefetching can be either hardware or software based [8], [52]. Hardware-based prefetching [8] requires additional hardware connected to the cache. Software prefetching [52] relies on compiler technology to insert explicit prefetch instructions. Typically, on a miss the processor fetches additional blocks along with the requested block. The processor places the requested block in the primary cache and places the prefetched blocks either in the primary cache or in an external buffer. On a future reference, if the processor locates the requested block in the buffer, the original cache request to next level of memory is cancelled, the processor reads the block from buffer and issues the next prefetch request. For data with spatial locality, prefetching is beneficial.

Stream buffer is a fully associative, First In First Out (FIFO) buffer with 4 or 5 entries specially designed to support direct-mapped cache through hardware based prefetching [39]. A miss induces fetching of the missed block along with successive blocks stored in the buffer rather than the cache. My intent is to use the stream buffer for prefetched blocks and avoid cache pollution.

2.3.6 Multilevel Caches

Inclusion of additional cache hierarchy provides a common technique to improve performance of desktop applications. Adding a second level of cache between the original cache and the main memory improves access times. The L-1 cache can remain small enough to match the clock cycle time of fast CPU, while the L-2 cache can become large enough to capture most accesses, thereby lessening the access time. Some recent architectures contain a third level cache, whereby an even larger cache is placed between the L-2 cache and the main memory.

2.4 Comprehensive Evaluation

I have provided a survey of most popular traditional approaches (Described in section 2.3.) that have been used to improve the cache performance of desktop applications and have selected one or more representatives of each technique. In this research I employed a simulation environment representing each of these traditional techniques and performed a comprehensive evaluation of my proposed methods by comparing them with each of these traditional cache organizations. Results of this comprehensive evaluation are described in chapter 6.

2.5 Issues in Designing Caches for Embedded Systems

Unfortunately the optimization methods described in section 2.3 (Which work well for desktop application.) are not suitable for embedded applications. For desktop applications, the simplest way to improve performance is to increase the cache size and block size. Whereas, embedded systems cannot include large caches or arbitrarily large block sizes because of the physical size and power constraints. Caches with higher (Or modified.) associativity remain unpopular for embedded systems mostly because higher associativity leads to high power consumption [67, 82]. Thus most cache design efforts have concentrated on optimizing direct-mapped cache organizations. Although victim caches commonly appear in desktop computing, designers have viewed supplementing direct-mapped cache with a victim cache as an inappropriate choice for embedded systems [84], because fully associative victim caches require additional energy and silicon area. Similarly, although successful prefetching can reduce miss rates, any unnecessary prefetching not only wastes the embedded system's valuable power but may also cause cache pollution which leads to additional misses and wasted energy. Additional level of cache hierarchy is also not popular in embedded applications [35]. Because the requirements of cache design in embedded systems differs from that in desktops, I propose reconfigurable split data caches for embedded applications, which will be described in later chapters.

CHAPTER 3

RELATED WORK

In this chapter I provide a survey of related work. In later chapters (Chapter 5-8.) when I describe different stages of my work, I try to distinguish my work from the related work described here.

3.1 Cache in Desktop Applications

I divide this chapter in to two sections. First I describe the related work with data cache splitting, which is followed by the description of the related work with victim cache and stream buffer.

3.1.1 Related Work in Data Cache Splitting

Supplementing the cache with a small extra module to exploit temporal and spatial localities was first proposed by Jouppi [39]. Subsequent to Jouppi's work, two approaches emerged; the first approach retains Jouppi's original idea and supplements the regular cache with a small buffer for prefetching all data items regardless of the nature of locality exhibited by the data; the second approach is real cache partitioning to exploit data localities exhibited by different data types. Partitioning of the cache can be either static or dynamic.

The most extensive and prominent work belonging to the first trend is done by Mckee *et al*. [54]. They designed a stream memory controller (SMC), a combination of a small buffer and an intelligent scheduling unit for supporting the regular cache. When the program enters a loop that accesses one or more streams, compiler-generated code provides the scheduling unit with the base addresses, the number of elements, and the strides for any streams accessed in the loop body. A Memory Scheduling Unit (MSU) uses this information to reorder the requests so that even though the processor still issues requests to the Stream Buffer Unit (SBU) in the natural order, the order in which associated requests are made to memory will maximize the use of its bandwidth. Because the stream accesses no longer affects the cache, the cache can be designed more optimally for the remaining requests.

Sanchez *et al.* [71] propose a dual data cache, composed of two modules. The temporal module is a fully associative buffer, built to exploit just temporal locality and spatial module, is a direct mapped cache targeted to exploit spatial locality. The temporal module has only 16 short blocks (Each 8 bytes.) and the spatial module has larger blocks (32 bytes per block.). At compile time, memory instructions are tagged as *bypass* (Data that do not exhibit any type of locality.), spatial, or temporal. On a miss both modules are checked in parallel to find the required data. References tagged as bypass are sent to directly to the Central Processing Unit (CPU) rather than bringing them into cache. If the reference receives a spatial or temporal tag, a new block is brought into the module indicated by the tag. Previously they proposed a similar architecture [28], where rather than using compile time annotations, they tagged memory references at execution time using an additional hardware unit called *locality prediction table*.

Tomasko *et al.* [78] reported on a preliminary experimental evaluation of architecture with separate array and scalar caches to exploit the potential benefits to a cache organization based on a specific type of locality. In their experiment they assumed a model where the tagging of data as array or scalar to be allocated in one or the other caches would be done statically by compiler and the model does not assume any extensive analysis of references to determine the nature of the locality of access; rather it allocates the data only on the basis of the data type declaration.

Split Temporal/Spatial (STS) cache proposed by Milutinovic *et al.* [56] differs slightly from other proposed split caches I have already discussed. Because for "temporal" data hierarchy is needed (To reduce miss penalty for subsequent misses.) but fetching the entire block is unnecessary, the temporal part is organized as a two level hierarchy with one word block size. The spatial part is one-level with four 32-bit words with a hardware implemented prefetching mechanism. The STS cache contains four variants STS1, STS2, STS4a STS4b, each with same sized temporal module but larger spatial modules. Spatial and non spatial modules are 4-way set associative with Least Recently Used (LRU) replacement. Initially all data blocks are regarded as "spatial," a data block may be changed to "temporal" and re-allocated through optimization of relevant parameters (Using different counters to detect locality.) during profiling or during runtime by means of a monitoring hardware unit.

In order to avoid the problem of determining counter thresholds present in STS method and also the problem of complicated memory hierarchies for each module, Milutinovic *et al.* [57] proposed a simple method of detecting useful spatial locality which is tested by incorporating it into a new split cache design, called the Split Spatial/Non-Spatial (SS/NS) cache. For detecting different types of locality the design used a flag-based method, requiring fewer cache bits than counter implementation of STS. For exploiting the locality the system has two separate modules with same associativity and equal hierarchy. Prefetching is used if spatial locality is too large to be exploited by larger cache block.

The Hewlett Packard-7200 Assist cache [46, 68] design tries to avoid both cache conflict and cache pollution due to prefetching. The primary direct-mapped cache is coupled with a small fully associative buffer (the Assist buffer), with a one-cycle lookup in both units. The direct-mapped primary cache and the buffer units are designed with equal sized blocks. Until a block is identified as temporal, if it is requested either by a cache miss or a prefetch, the block is first loaded into the Assist buffer. It is promoted into the direct-mapped unit only when it exhibits temporal reuse. Spatial-only data, especially array data, may bypass the direct-mapped cache entirely, moving back to memory in First In First Out (FIFO) fashion from the Assist buffer. In this system dynamic associativity is provided by allowing up to N+1 conflicting blocks, which belong to the same direct-mapped set, to co-exist in the cache simultaneously, where N is the number of block entries in the Assist buffer. Since only a uni-directional communication exists between the direct-mapped unit and the Assist buffer no swapping between the two units is allowed.

The Non-Temporal Streaming (NTS) cache proposed by Rivers and Davidson [69] dynamically detects temporal (T) and non-temporal (NT) data and cache them separately. The NTS cache consists of a Data Storing Unit (DSU) which is a conventional direct mapped cache supplemented with a small fully-associative buffer (NT buffer) and a Non-Temporal Detection Unit (NTDU), which is a hardware bit-map structure attached to the main cache in order to monitor the reuse behavior of the blocks. Cache block size is uniform across the direct-mapped cache primary cache, the buffer units and the next level of memory. The strategies adopted in the NTS cache for detecting and caching temporal and non-temporal references are very similar to those implemented in Hewlett Packard - 7200 Assist cache [46, 68].

Lee *et al*. [49] have proposed a cache system called Selective Temporal and Aggressive Spatial (STAS) cache. Although they claimed to have two separate caches for temporal and spatial references with different block sizes, the module for spatial locality is actually a buffer rather than a cache. In this system, on every memory access, both modules are accessed simultaneously. If a miss occurs at both places the block is brought to the spatial buffer. However a write back of dirty block in spatial buffer cannot occur directly - the dirty block is always placed in the direct mapped cache before being replaced. Later Lee *et al*. [50] extended the STAS cache into another cache structure Selective Mode Intelligent (SMI) cache which consists of three parts: a direct-mapped cache with a small block size, a fully associative spatial buffer with large block size and a hardware prefetching unit.

3.1.2 Related Work with Victim Cache and Stream Buffer

Albera and Bahar [4] combined software code placement and associative-buffer solutions for high performance processors and showed that the buffer can improve performance even more after code layout optimization is applied than when used without the code optimization. In a later study Bahr *et al.* compared the use of victim caches to more traditional techniques and showed that use of a victim cache is usually a better choice for both power and performance [9].

Espasa and Valero [25] considered the usefulness of adding a victim cache next to the register level of a vector processor and show that such placement can provide speedups by allowing a good tolerance of large memory latencies. Hormdee *et al.* proposed an architecture of a self-timed victim cache with a forwarding mechanism suitable for use within an asynchronous environment [33].

Except for the addition of the non-swapping option, no other extension to Jouppi's original victim cache [39] was implemented by any of these above-mentioned studies. Bahar *et al.* [9] tried to add some extra flavor in their "penalty buffer" but failed to gain much improvement. Only one group, Stiliadis *et al.* proposed an improvement of victim caching called "selective victim caching" [77]. In this method a prediction scheme based on each block's past history of utilization is used to selectively place a block either in the main cache or victim cache on a cache miss in either cache and to decide whether to perform swap or not in the case of victim hit.

As described in section 3.1.1, the most extensive and prominent work with stream buffers is that of McKee *et al.* [54]. Their designed SMC is a combination of a stream buffer and an intelligent scheduling unit for supporting regular cache. Palacharla and Kessler [65] have proposed the use of multiple stream buffers to replace big secondary cache.

To date, no study has combined the implementation of victim cache and stream buffer with separated data cache approach. Johnson *et al.* [37] proposed a method where a single 4–way set associative buffer is used to serve the function of both victim cache and stream buffer on groups of data that have been differentiated based upon the reuse behavior.

Johnson *et al.* [37] presented a method to improve the efficiency of cache by bypassing data that is expected to have little reuse in cache and allowing more frequently accessed data to remain cached longer. The bypassing choices are made by a Memory Address Table (MAT), which analyzes the usage patterns of the memory locations accessed. In order to characterize memory locations they introduce the notion of macro-block, which is a group of statically defined blocks of memory with uniform size (1 kilobytes). They used a direct mapped 16 kilobytes Level one (L-1) data cache and 256 kilobytes Level two (L-2) data cache with fully associative buffers of 8 and 256 entries respectively, which hold bypassing data and are accessed in the same manner as a victim cache. Since fetching the entire cache block for bypassed data with little spatial locality will cause cache pollution and extra traffic, they used small lines (Equal to the element size.) for the buffers and optionally filled in consecutive blocks when spatial locality is detected. As we can see they are using a single buffer to serve the purpose of both a victim cache (For scalar data.) and a prefetch buffer (For stream data.). In a later study [38] they extended their scheme by adding an extra structure Spatial Locality Detection Table (SLDT) and extra counter for each MAT entry to detect spatial locality so that the system can adapt to varying spatial locality by dynamically adjusting the amount of data fetched on a cache miss.

3.2 Caches for Embedded Systems

Exploitation of various cache parameters such as associativity and block sizes offer the most common approaches to improve cache performance for desktop systems [74]. Following this approach, Givargis, *et al.* explored the effects of cache size, block size and associativity on the cache performance in embedded systems [27]. For desktop applications, other cache performance improvement techniques include augmentation of a cache with additional structures and hardware prefetching. Zhang *et al*. [84] explored the role of victim caches in embedded systems. They concluded that because of its high energy requirement, victim caches do not offer a good option for embedded systems. Data prefetching, has long been known to significantly decrease cache miss latency and both hardware and software prefetching approaches have been studied for desktop applications [8, 52]. However, these techniques have not been investigated for embedded systems because designers believe that such structures add to the embedded processors' energy requirements [31]. Another common practice in desktop application includes additional cache hierarchies [10], Ross-Gordon *et al.* has evaluated L-2 caches for embedded systems [29]. In the arena of embedded processors, static or dynamic cache partitioning has been investigated. Unsal *et al.* [79] proposes a minimax cache which has a 8 kilobytes 2-way set associative cache for non scalar data while the scalar data are directed to a 512 bytes fully associative minicache. The system also has a secondary cache and the block size is the same across the buffer and caches. Intel's [34] StrongARM SA-1110, a low-power processor for embedded system, has an 8 kilobytes data cache with 32-way set associativity and 512 bytes fully associative mini data cache to enhance caching performance when dealing with temporal references. This system does not contain a L-2 cache.

3.3 Reconfigurable Caches

Albonesi *et al.* [5] propose reconfigurable unified data caches for desktop systems, where as Ranganathan *et al.* [67] and Zhang *et al*., [82, 83] propose reconfigurable unified data caches for embedded systems. In their customizable reconfigurable unified data cache, Albonesi *et al.,* [5] selectively shut down portions of cache to achieve power reduction. The unified data reconfigurable cache proposed by Ranganathan *et al*. [67] allows the cache array to be divided dynamically into two or more partitions that can be utilized by the processor for various purposes. Zhang *et al.* [82, 83] proposed a reconfigurable unified data cache for embedded applications. They emphasize more on associativity as an important reconfigurable design parameter. In a later work [29] they also analyzed the possibilities of reconfigurability with L-2 cache.

3.4 Data Flattening

As no study has combined the implementation of restructuring computation or data layout with separated data cache approach, in this section I generally discuss the related work done with computation and data reconstruction. Both approaches of "restructuring computation" and "restructuring data" have received considerable attention in the past. Research with "restructuring computation" attack the processor-memory gap by reordering computations to increase spatial and temporal locality [16, 81]. Carr *et al*. [16] use a simple model of spatial and temporal reuse of cache lines to select compound loop transformations. Wolf and Lam [81] develop a loop-transformation theory, based on uni-modular matrix transformations, and use a heuristic to select the best combination of loop transformations.

Over the years, researchers have also proposed data placement optimization techniques to employ “restructuring data.” Early work on improving the spatial locality on data streams with pointer data structures occurs in the context of LISP [19, 47, 58] and aimed to increase page reference density. Clustering has been used to improve virtual memory performance of LISP systems [47, 58] by reorganizing data structures during memory allocation or garbage collection. Database researchers also used clustering [11, 12] and compression [23] to improve virtual memory performance. Recently Seidl and Zorn [73] and Calder *et al.* [15] have shifted focus to cache-conscious allocation and added profile feedback to this process. Seidl and Zorn [73] combined profiling with a variety of different information sources present at the time of object allocation to predict an object’s reference frequency and lifetime. They show that program references to heap objects are highly predictable. Calder *et al.* [15] applied placement techniques developed for instruction caches and data caches. They use a compiler directed approach that creates an address placement for the stack (Local variables.), global variables, heap objects, and constants in order to reduce data cache misses.

Concurrently, Chilimbi *et al.* developed several techniques for optimizing heap data placement. In [20] they describe a data placement optimization for tree-like structures. Their approach is semiautomatic, permitting more aggressive optimizations, such as the splitting of structure variables. In [21] they extend this approach to support on-line profiling and data placement for Cecil, an object-oriented language with generational garbage collection. They collect low-overhead, real-time profiling information about data access patterns and applied a new copying algorithm that uses this information to produce a cache-conscious object layout.

Luk and Mowry [52] present a case for history-pointer prefetching, which augments linked structure nodes with prefetching pointer fields, and data-linearization, in which pointer data structures are programmatically laid out at runtime to allow sequential prefetch machinery to capture their traversal.

In this chapter I have included discussion of research related to my work. I will further discuss these works as I introduce my technique in later chapters. As stated previously, the literature is rich with cache design techniques. I do not claim to have included all relevant research in this chapter.

CHAPTER 4

EXPERIMENTAL ENVIRONMENT

In this chapter I describe the experimental environment used in this study. First I define performance metrics and power models used in my studies. Then I investigate the interactions between different parameters. Finally I describe the benchmarks and the simulation environment of my study.

4.1 Performance Metrics

First I define performance metrics and power models used in my studies.

4.1.1 Cache Miss Rate

Miss rate is the percentage of cache accesses that are not found in the cache. Because a reduction of the miss rate leads to time and power savings, I include miss rate as a performance metric. Unfortunately, miss rate does not always depict the true picture. For example, although 2-way set associative, MRU [17, 75] and Column associative [2] cache will result in the same miss rates, each of these organizations has different access times and power consumptions. Although reducing miss rate can lead to improved access times, miss penalties may also affect access times. Additionally, energy consumed by an application is also affected by miss rates. Because I use an array cache and a scalar cache, I can represent the effective (Or combined.) miss rate as

*Effective miss rate = Array miss rate * (Number of Array references/Number of total references) + Scalar miss rate * (Number of Scalarref rerences/Number of total references)*

I use this formula to calculate effective miss rates in my experiments, and compare this effective miss rate with miss rates of unified data caches.

4.1.2 Cache Access Time

Cache access time is the average number of cycles required to successfully access a referenced address. I use CACTI [80] using a 0.8 micron technology to compute access times for cache hits. The equations for different cache configurations are included in Table 4.1. This metric proves useful in evaluating the performance of a cache scheme because, although a particular cache design may demonstrate lower miss rates, the lower miss rates may have been achieved at the expense of the hit access times. For example, a cache with higher associativity can have lower miss rates than a direct-mapped cache, but an associative cache will have longer access times. (All the terms I use in Table 4.1 are defined in Table 4.3.)

Table 4.1. Timing Equations Used to Compare Performance

Cache name	Equation to compute the Access Time
Direct mapped	((Hit * HAT) + (Miss * (FTM + OAT)))
Set associative	((Hit * HAT) + (Miss * (FTM + OAT)))
Hash rehash	((Prob 0 Hit * prob0 HAT) + (prob 1 Hit * (prob0 FTM + prob1HAT + prob1SW)) + (Miss * (prob1 FTM + MSW + OAT)))
Column associative	((Prob 0 Hit * prob0 HAT) + (prob 1 Hit * (prob0 FTM + prob1HAT + prob1SW)) + (Miss * (prob0 FTM + FTM + MSW + OAT)))
MRU	((Prob 0 Hit * (prob0 HAT + MI)) + (prob 1 Hit * (prob0 FTM + prob1HAT)) + (Miss * (prob1 FTM + OAT)))
Victim	((Hit * HAT) + ((Victim cache Hit * (FTM + Victim cacheHAT + Victim cacheSW)) + (Miss * (FTM + MSW + OAT)))
Prefetching	((Hit * HAT) + (Miss * (FTM + OAT+ EWP)))
Array/scalar cache	((Hit * HAT) + (Miss * (FTM + OAT)))
Array- stream buffer	((Hit * HAT) + (Stream bufferHit * (FTM + Stream buffer HAT)) + (Miss * (FTM + OAT+replacement cost + (4*EWP))))
Scalar- victim cache	((Hit * HAT) + ((Victim cache Hit * (FTM + Victim cacheHAT) + (Miss * (FTM + MSW + OAT)))

4.1.3 Area Analysis

As embedded systems designers are interested not only in the performance but also in better use of silicon area, my performance evaluation also includes silicon area consumption of cache systems. I use CACTI [80] for computing silicon areas needed by caches.

4.1.4 Static and Dynamic Power Consumption

To complete my experimental evaluations, in my calculations I also include the amount of energy consumed by cache systems as an application executes. In Complementary Metal–Oxide–Semiconductor (CMOS) circuits, the major sources of power consumption are dynamic and static power. Dynamic power dissipation is due to logic switching current and the charging and discharging of the load capacitances, whereas static power dissipation arises with leakage current. In this study, to evaluate dynamic power consumption, I include the power consumed due to cache misses and off-chip accesses. My model uses the following general equations to compute the dynamic power consumed by a cache.

Dynamic_power = Hit * power_hit + Miss * power_miss

power_miss = OPC + PCW + FTM

I obtained values for *hits* and *misses* for each cache type by executing the selected benchmarks using corresponding cache simulators. *Power _hit* is the power consumed to access the cache (Computed using CACTI [80].). Here it should be mentioned that different cache structures possess different *power_hit* values based on the cache type, size and hit type of each access. For example, in the case of HR [1] or CA [2] cache, when a hit occurs on the second probe, additional swapping energy is needed. In Table 4.2 I describe the equations used to compute *power* for specific cache types. The *PCW* is the power consumed to write an entire line to the cache, which is computed using CACTI [80]. OPC, the power needed for off-chip access, I calculate as $0.5 * Vdd^2 * (0.5 * W_{data} + W_{addr})) * 20pF$ [42], where W_{data} and W_{addr} are the number of bits for both the data sent/returned and the address sent to the next level of memory on a miss. The last term is the load capacitance for off-chip destinations. For any miss the FTM (First Time Miss.) includes the overhead for searching in cache . (All the terms used in Table 4.2 are defined in Table 4.3.)

For static power estimation, I (Using a 0.13 micron technology.) include 30% of Level one (L-1) cache power as leakage power as reported by Agarwal, *et al.* [1].

Table 4.2. Power Consumption Equations Used to Compare Performance

Cache name	Equation to compute the Power Consumption
Direct mapped	((Hit * PCR) + (Miss * (FTM + OPC +PCW)))
Set associative	((Hit * PCR) + (Miss * (FTM + OPC +PCW)))
Hash rehash	((Prob0Hit* prob0 PCR)+(prob1Hit*(prob0FTM + prob1 PCR + prob1SWPC)) + (Miss*(pro1FTM+MSWPC+OPC+PCW)))
Column associative	((Prob0Hit* prob0 PCR)+(prob1Hit*(prob0FTM + prob1 PCR + prob1SWPC)) + (Miss*(prob0FTM+MSWPC+OPC+PCW)))
MRU	((probe 0 Hit * (prob0 HPC + MI)) + (prob 1 Hit * (prob0 FTM + prob1HPC)) + (Miss * (prob1 FTM + OPC + PCW)))
Victim cache	((Hit*HPC)+ ((Victim cache Hit* (FTM+Victim cacheHAC+Victim cacheSWPC))+(Miss*(FTM + MSW + OPC +PCW)))
Prefetching cache	((Hit * HPC) + (Miss * (FTM + OPC +PCW + EWP PC)))
Array/scalar cache	((Hit * HPC) + (Miss * (FTM + OPC+PCW)))
Array-streamBuffer	((Hit * HPC) + (SBHit * (FTM + SB HPC)) + (Miss * (FTM + OPC+ PCW + (4*EWP PC))))
Scalar- victim cache	((Hit * HPC) + ((Victim cache Hit * (FTM + Victim cacheHPC) + (Miss * (FTM + OPC+PCW)))

Table 4.3. Definition of Terms Used in Timing and Power Consumption Equations

Abbreviation	Definition
HAT	Time to read a cache line in case of hit
FTM	Time overhead to search in case of miss
OAT	Time to get data from next memory level
MSW	Time need to swap line for HR/CA or victim cache miss
EWP	Time need to prefetch for stream buffer or prefetching cache
PCR	Power Consumption to read a cache line in case of hit
PCW	Power Consumption to write a cache line in case of miss
OPC	Power Consumption of getting data from next memory level

4.1.5 Execution Cycles

Experiments with data cache design involve only load and store instructions. Hence, cache access time only reflects the time needed to access cache for load and store instructions. However, many instructions reside in a program. Load/store instructions comprise only a portion of these instructions. For instance, if for a benchmark on average, 25% of the instructions are load and store, time analysis with only cache access time does not provide the whole picture of execution cycles. For example, if I show an average 80% improvement in cache access time for this benchmark, the actual improvement of execution time for the whole program is only 20%. For this reason, I include execution cycles as a performance matric to demonstrate the actual speed up of the entire program.

4.2 Influence of Different Parameters

In this section I discuss the impact of different parameters on my proposed cache system.

4.2.1 Interaction among Area, Performance and Power Consumption

Let me first analyze the interaction between area and power consumption and then the interaction between performance and power. Then I explore interactions among all

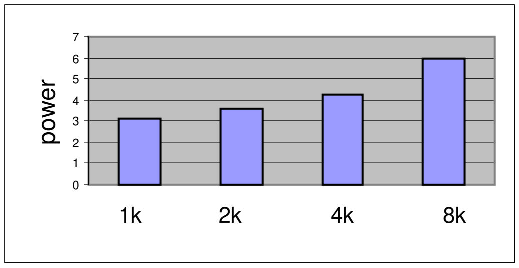


Figure 4.1: Increase in Power Consumption as Cache Size Increases

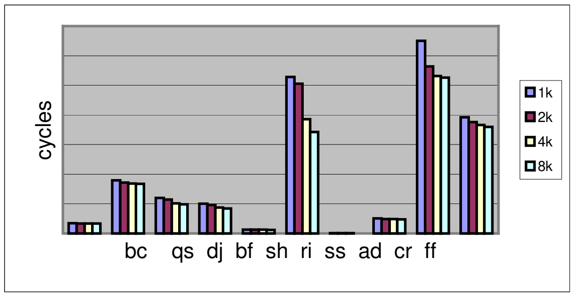
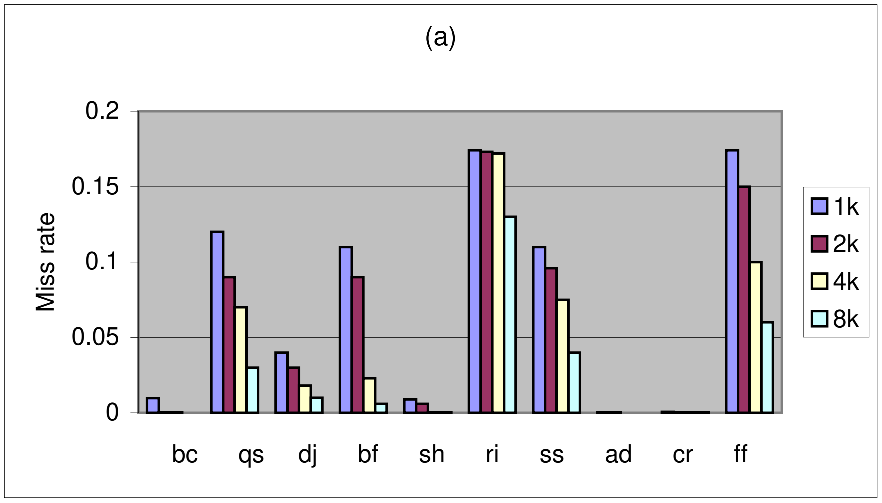


Figure 4.2: Decrease in Number of Cycles as Cache Size Increases

three metrics. The individual impact of area or performance on power consumption is straightforward. A smaller cache is more energy efficient; a faster cache potentially reduces overall energy consumption, by completing execution in a shorter time. In Figure 4.1 I show the increase in cache power consumption as size of the cache increases. In Figure 4.2 I show the decrease in execution cycles for each benchmark as cache size increases.



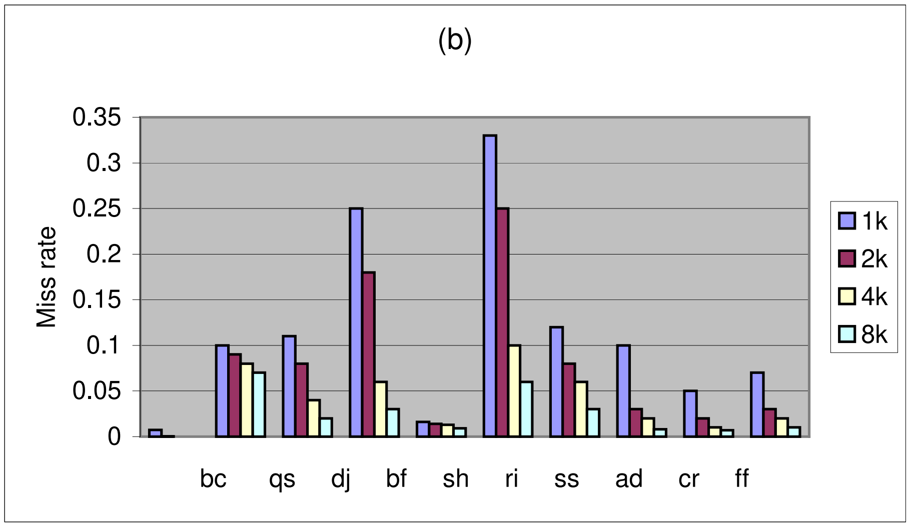


Figure 4.3: Instruction (a) and Data (b) Cache Miss Rates for Increasing Cache Size

In Figure 4.3, I show the reduction in miss rates with increasing cache size for both instruction (a) and data (b) caches. For several benchmarks (ad, cr, bc in Figure 4.3(a) and bc in Figure 4.3(b).), miss rates are too small (Compared to other benchmarks.) to be visible in the Figure.

Evaluation of the three parameters (Area, power, and performance.) together becomes complicated. Size and performance are usually conflicting requirements. In Figure 4.4 I show the increase in cache power consumption as the size of the cache increases. I also show the average decrease in execution cycles (In 100 millions.) and the average miss rate for my benchmarks as the cache size increases.

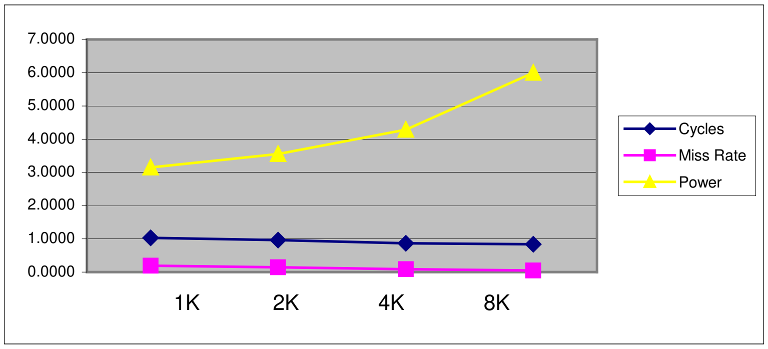


Figure 4.4: Changes in Power Consumption, Average Execution Cycles (In 100 millions.) and Miss Rate as Cache Size Increases

For most applications reducing cache size will result in more cache misses which in turn causes more visits to main memory—resulting in increased time consumption and in higher power consumption. Thus, although a larger cache requires more power per access, that extra power may be compensated for by reductions in execution time and power consumption as there are fewer cache misses than may occur when using a smaller cache. It should also be mentioned that although overall reduction in execution cycles results in a proportional decrease in the amount of power consumed by an application, improving cache performance will more profoundly impact energy savings, because cache is responsible for 50-80% of processor's total power consumption [82, 83]. Hence, the need for tuning the cache size and miss rate for each particular application to reduce overall power consumption motivates the need for a reconfigurable partitioned cache.

4.2.2 Influence of Associativity

Higher associativity in both data and instruction caches is identified as the most important reconfigurable parameter by Zhang *et al.* [82, 83]. However, when the data cache is split (In this case into array and scalar caches.), associativity is no longer a significant reconfigurable parameter. At L-1 cache (My primary concern.), it is important to maintain a balance between miss rates and access times. I believe in my design, direct-mapped caches provide for such a balance as my split cache organizations eliminate conflicts between different classes of data (Namely, arrays and scalars.). Additionally, because I provide for a small victim cache with the direct-mapped scalar cache, my split cache organizations further reduce the miss rates, without having to resort to higher associativities. For instruction cache, I believe that cold misses are more problematic to performance than conflict misses, and I use a small pre-fetch buffer to reduce cold misses.

4.3 Benchmarks

I believe the biggest advantage of my proposed cache system is its efficiency over a wide ranged of applications. Being equipped with a special array cache, my split cache design should be beneficial for scientific applications and should show significant improvement for Standard Performance Evaluation Corporation (SPEC) floating point benchmarks. For data flattening (Designed for pointer-intensive applications.) I use pointer-intensive Olden benchmarks. However, my cache system for embedded applications also show significant benefits. To evaluate my proposed cache for embedded systems, I have used benchmarks from Mi-bench benchmark suit.

4.3.1 Scientific Applications: SPEC Floating Point Benchmarks

At the initial phase of my work, I evaluate the proposed cache architecture for the SPEC floating point 2000 benchmarks[32]: ar, am, me and eq. Descriptions of the SPEC benchmarks appear in Table 4.4. Each program is written in C. I used gcc compiler version 2.3. In my study, I used the exact benchmark codes and did not modify the codes to efficiently use split cache (Such as reordering array references or including prefetching hints.). The percentage of array references ranged from a low of 6.58% in mesa to a high of 26.92% in ar.

Table 4.4: Descriptions of SPEC Floating Point Benchmarks

Benchmark name	Description	Name In figure
179.art	Image Recognition/Neural networks	ar
188.ammp	Computational Chemistry	am
183.equake	SeismicWavePropagation Simulation	eq
177.mesa	3-D Graphics Library	me
172.mgrid	Multi-grid Solver: 3D Potential Field	mg
191.fma3d	Finite-element Crash Simulation	fm
200.sixtrack	Nuclear Physics Accelerator Design	sx
173.applu	ParabolicPartialDifferentialEquation	ap

To collect data, I traced complete program runs and did not limit my traces to array and in-loop references. Non-array references, especially scalar and stack variables, contribute most to temporal reuse and are the main victims of cache pollution. Therefore, excluding non-array references from traces will not provide the true picture of a program's memory reference behavior. In a later experiment I included additional benchmarks from SPEC floating point Benchmark suite, fm, mg, ap and sx [32]. The number of instructions executed by each application ranged from 1 billion to 129 billion.

4.3.2 Embedded Applications: Mi-bench Benchmarks

In my experiments I used the benchmarks of Mi-Bench suite [30] as representative of embedded applications. To cover a wide range of applications, I included benchmarks from (1) Automotive (2) Office Automation, (3) Networking, (4) Security, (5) Telecommunications and (6) Consumer groups. I selected only those benchmarks that I could compile on a Simplescalar system. The descriptions of the benchmarks used in my studies are listed in Table 4.5.

Table 4.5: Descriptions of Mi Bench Benchmarks

Benchmark	Description	% of load/store	Name in fig
bit counts	Test bit manipulation	11	bc
qsort	Quick sort algorithm	52	qs
dijkstra	Shortest path problem	34.8	dj
blowfish	Encription/decription	29	bf
sha	Secure Hash Algorithm	19	sh
ri	Encryption Standard	34	ri
string search	Search mechanism	25	ss
adpcm	Variation of PCM standard	7	ad
CRC32	Redundancy check	36	cr
FFT	Fast Fourier Transform	23	ff

Because the performance of my system depends on the percentage of memory references application issues, I also include the load/store percentages for each benchmark. As can be seen, I included very memory intensive benchmarks, benchmarks that are moderately memory intensive, and some that execute very few load/store instructions.

4.3.3 Pointer-intensive Applications: Olden Benchmarks

I attempt to evaluate my data flattening method by running pointer-intensive applications selected from different Benchmark suits. To select benchmarks exhibiting wide-ranging memory allocation behaviors, I chose three benchmarks from the Olden pointer-intensive benchmark suite [70], three from SPEC integer 2000 benchmark suite [32], one from the memory pointer benchmark suite [70] and one from Mi-bench suite [30]. The benchmarks from Olden benchmark suite [70], SPEC integer benchmark suite [32] and memory pointer benchmark suite [70] contain pointer-intensive benchmarks and used by memory allocation researcher. Because I also want to evaluate my data flattening approach for embedded applications, I selected one pointer-intensive benchmark **qs** from the Mi-bench suite [30]. A summary of the benchmarks is shown in Table 4.6.

Table 4.6: Descriptions of Pointer-intensive Benchmarks

Benchmark name	Suit name	Description of Benchmark	% of load/store	Name in fig
129.compress	SPEC integer	Compress and decompress file in memory	8.3	co
132.ijpeg		Image compression/decompression on in-memory images based on the JPEG facilities	18.6	ij
vortex		Object-oriented database	32.1	vo
qsort	Mi-bench	Quick Sort	52	qs
cfrac	Memory	Factoring numbers	27.8	cf
treeadd	Olden	Summing values in a tree	20.6	tr
BH		Barnes-Hut's N-body force calculation algorithm	29.1	bh
voronoi		Graphics utilities routine	14.3	vo

4.4 Simulation

For my work, I marked traces as array accesses and scalar accesses. I identified array references by assuming that such references involve some form of indexing. While I cannot ensure that my method captured all array data items, my analyses for selected sample programs showed that my approach correctly identified better than 99% of the array data items. In an actual implementation of split caches, compile time analyses can be used to separate array and scalar data references. By using different instructions (e.g., Array_Load and Array_Store, Load and Store.) data can be directed to array and scalar caches. This capability eliminates the complex hardware needed to detect and direct spatial and temporal localities to split data cache partitions (As has been proposed in [28, 46, 66, 69].). Arul, *et al.* and Kavi, *et al.* both use a similar approach in dataflow architectures [7, 43].

To compare cache design alternatives I developed a suite of simulators. The experiments shown in chapter 5 and 6 were performed using the ATOM instrumentation and analysis tool [26] on Hewlett Packard alpha Dec processor. The experiments discussed in chapters 7 and 8 were done using the Simplescalar simulator [13].

4.4.1 Simulation with ATOM Tools

In the initial phase of my work, I used trace-driven simulation as my evaluation methodology. The executables of the benchmarks are instrumented using ATOM, a performance measurement tool [26]. ATOM instrumentation routine produces a new executable file a.out.atom. When a.out.atom is executed in the same manner and same input as the original program, a highly compressed trace file of every load and store reference the program made is produced. This trace file is fed to the ATOM's analysis routine to simulate different cache organizations for split array and scalar cache. The routine generates number of hits, misses, and other relevant statistics for the program. I mark traces as array accesses and scalar accesses. In this study, I identified array references by assuming that such references involve some form of indexing. ATOM analysis program tracked indexes so that any data item that used an index was marked as an array reference.

4.4.2 Simplescalar Simulator

In a later phase of my work, I use an execution-driven simulator. I designed a cache simulator as an extension of the Simplescalar simulator [13]. This Simplescalar simulator, a cycle-by-cycle simulation written in C, includes an out-of-order speculative processor, memory, and peripherals. To collect data, I replaced the cache memory of Simplescalar with my split, multi-level cache hierarchy. This change offered a tunable cache size for reduced power, time and area consumption.

These experimental approaches are similar to those used by other researchers. I was unable to compile all Mi-Bench and SPEC on my system. This is primarily because ATOM tools are no longer maintained and Simplescalar simulator do not compile some of the Mi-Bench benchmarks.

CHAPTER 5

SPLIT DATA CACHE: AN APPROACH FOR SCIENTIFIC APPLICATIONS

In this chapter, I explore a cache organization which provides architectural support for distinguishing between memory references that exhibit spatial and temporal locality and mapping these memory references to separate caches.

5.1 Introduction

In prior work, I explored use of a separate cache for I-structure memories within the context of dataflow-based, multithreaded systems [7, 44]. I-structure memories in dataflow systems store arrays and other indexed or stream data items. In this chapter I show that using separate (data) caches for indexed or array data and scalar data items in conventional architecture can lead to substantial improvements in terms of cache misses. In addition, such a separation allows for cache design that can be tailored to meet the properties exhibited by different data items.

Selections of a proper block size or associativity to maximize performance while staying within the cost constraints are the hardest choices in designing cache memories. By partitioning the cache as I propose, my cache system can implement configurations that exploit cache parameters more selectively and effectively. Array cache, a direct-mapped cache, uses large block sizes to exploit spatial localities more aggressively by prefetching multiple neighboring small blocks on a cache miss. Scalar cache, either a 2-way or 4-way set associative cache, uses small block sizes to exploit temporal locality. A combination of block sizes and associativities together with a partitioned cache architecture provides an effective solution for the existing problems in cache designs.

Because significant amounts of compulsory and conflict misses are avoided, each cache's size (Whether array or scalar.) as well as the combined cache capacity can thus be reduced. According to my simulation results, a partitioned, 4 kilobytes scalar cache with streams (Or arrays) mapped to a 2 kilobytes array cache can be more efficient than a 16 kilobytes, unified data cache. Even if the program displays only a small percentage of scalar references (As in the case of scientific applications.) or very few arrays or streams, I feel that it is better to use separate scalar and stream caches.

The success of my proposed split data caches encourages me to extend my research to achieve further improvement. To that end, I investigated the interaction between three established methods for improving hit rates in the memory hierarchy's first level: split caches, victim caches, and stream buffers. I show that the inclusion of a victim cache and a stream buffer together with partitioned cache architectures provides a solution for alleviating existing cache design problems and for enhancing the effective cache memory space for a given cache size and cost.

In his paper, Jouppi [39] proposes both victim caches and stream buffers. Victim caches are based on the fact that reducing the cache misses due to line conflicts for data exhibiting temporal locality is an effective way to improve cache performance, whereas stream buffers are oriented towards eliminating cold misses coming from the portion of the code exhibiting spatial locality. Each approach (Split caches, victim caches, and stream buffers.) possesses valuable strengths. Each works well for the patterns it is designed for.

To date, no split cache has considered the existence of a victim cache or a stream buffer and their interaction with data references. Similarly, a victim cache or stream buffer does not normally consider what optimizations have already been incorporated by locality enhancing split cache techniques. In this chapter I combine these techniques and study their interaction. Here, I propose an integrated scheme that partitions a program into regions, each with its own locality type. My approach then sends the partitioned memory references to appropriate caches and, finally, selectively applies either a victim cache for program regions exhibiting temporal locality or a stream buffer for regions with spatial locality, to further enhance the split cache organization.

5.2 Design of Split Data Caches

My proposed architecture tags memory accesses as either scalar or array references according to their inherent locality and subsequently maps each group to a dedicated

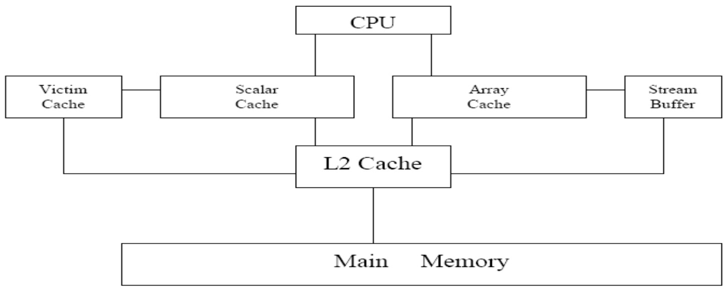


Figure 5.1: Split Data Cache Organization

cache partition equipped with architectural constructs built to exploit that particular locality type. Figure 5.1 provides a block diagram of my spilt data cache. Figure 5.1 also shows the victim cache attached with scalar cache and the array cache augmented with stream buffer.

5.3 Related Work

To contrast my approach with the designs summarized in section 3.1.1, I propose a very simple design by providing two separate caches, named array and scalar. These caches employ individual design parameters optimized to meet the specific data type's needs. For instance, scalar cache will exploit temporal locality for some data items, while the array cache will exploit spatial locality exhibited by other types of data items. Among the approaches mentioned in section 3.1.1, SMC [54], Dual data cache [71], Assist cache [46, 68], Non-Temporal Streaming (NTS) [69], Selective Temporal and Aggressive Spatial (STAS) [49], Selective Mode Intelligent (SMI) [48] use multiple stream buffers to supplement a single data cache. Split Temporal/Spatial (STS) [56], Split Spatial/Non-Spatial (SS/NS) [57], Array/Scalar [78], Minimax cache [79], and StrongARM [34] are real split cache architectures. My design allows one to build correspondingly simple hardware controllers. I believe that rather than using a multiple-streamed First In First Out (FIFO) buffer it is more practical to use a cheaper, faster, and well-established architectural construct such as cache. The performance of the split caches can be improved with compile time analysis and direct memory accesses to appropriate cache.

In addition to taking advantage of three approaches, my architecture permits the use of different block sizes and different associativities within a single Central Processing Unit (CPU) design. STS [56], SS/NS [57], and Array/Scalar [78] cache systems use the same associativity with different block sizes. Victim [39], Assist [46, 68], NTS [69] cache systems, as well as the Minimax [79], and Intel StrongARM SA-1110 [34] cache systems use different associativities but the same block size. None of these designs permit variations of blocks sizes and associativies in a truly split cache model. For stream references with spatial locality, which causes more compulsory misses, I use a direct-mapped array cache with large block sizes. By doing so, I benefit from prefetching. For a scalar reference, which causes more conflict misses, I use a 2-way set associative cache with small block sizes and more blocks in cache. In this way, I avoid the high conflict and the thrashing effect associated with direct-mapped caches.

I compare my work with the most closely related work by Johnson and Hwu [37]. The first difference between their work and mine is that rather than using locality types Johnson and Hwu employ the "reuse" behavior of data as a metric for data separation. Because the Memory Address Table (MAT) retains the reuse pattern for all data in a program, it is possible for an array element to have higher reuse count at some point during execution than scalar data has. In that case, Johnson and Hwu's MAT scheme bypasses the data which, in any case, may have had a few hits before being displaced from the cache. Therefore, not caching that data will not incur more than one additional miss, whereas not displacing the more frequently accessed data removes only one miss. Second, after identifying data as scalar or array, I cache both in separate caches. Johnson and Hwu implemented bypassing for data with history of low use. The third and the most significant difference is the number and types of architectural constructs. I not only use two separate caches for two data types, I also implement two additional structures: victim cache and stream buffers. This allowed me to tune the amount of data cached and fetched, and to fully exploit the victim cache's functionality. I can reduce conflict misses of scalar data by holding data longer. In addition, I use the stream buffer to reduce the cold miss of array data by prefetching. In Johnson and Hwu [37] using 8 lines of buffer, as both victim cache and prefetch buffer will negatively affect each others performance.

5.4 Experimental Methods

The cache architecture proposed in my initial work (Section 5.5.) has been evaluated for the following Standard Performance Evaluation Corporation (SPEC) floating point 2000 benchmarks, ar, am, me and eq [32]. My integrated approach (Section 5.6.) has been evaluated for the following SPEC floating point 2000 benchmarks, ar, am, me, eq, fm, mg, ap and sx [32]. I used trace-driven simulation as my evaluation methodology. The benchmark executables are instrumented using ATOM tools. To evaluate the optimal split cache configuration, I examined a variety of array and scalar cache sizes, block sizes, and associativity. In these experiments I simulated three cache sizes for array cache (1, 2 and 4 kilobytes.) and five cache sizes for scalar cache (4, 8, 16, 32 and 64 kilobytes.). Block sizes ranged from 32 bytes to 128 bytes for both array and scalar caches. I choose two common approaches, a direct-mapped cache and a 2-way set associative cache for both array and scalar cache. For 2-way set associativity I used both Least Recently Used (LRU) and random replacement policies. I also simulated a conventional cache with corresponding configurations for comparison with my experimental results. Table 5.1 presents the configurations for the memory hierarchy I implemented in this study.

Table 5.1. Configurations of Memory hierarchy for Array, Scalar, Victim Caches and Stream Buffer

Scalar cache configuration	4k, Directmapped, 64bytes block
Access time of Scalar cache	1 cycle
Number of lines in victim cache	8 lines, non swapping
Victim cache associativity	Fully associative
Replacement Policy	LRU
Victim cache block size	64-bytes
Access time of Victim cache	1 cycle
Array cache configuration	4k, Direct mapped, 64bytes block
Access time of array cache	1 cycle
Number of stream buffer	2
Number of lines in stream buffer	8
Stream buffer block size	64 bytes
Access time of stream buffer	1 cycle
Level two (L-2) cache configuration	256k, Directmap, 64bytes block
Access time of L-2 cache	10 cycle

5.5 Initial Evaluation of Split Data Caches

In the initial phase of my experiments, I was more concerned about the selection of different cache parameters. I will begin this section discussing about the selection of optimum parameters. I follow this discussion with description of achieved results with cache splitting.

5.5.1 Finding Optimum Associativity, Block Size and Cache Size

In my proposed cache system, I believe, not only both caches will be designed more optimally according to their specific needs but other issues and concerns that arise when designing cache would be simplified, such as selection of associativity, cache block size, and cache capacity. Cache designers always need to find a compromise between parameters. In the following subsections I discuss these issues in general cache design and envisage how my cache can be tailored to find an optimum for each cache parameter to meet the properties exhibited by different data items.

5.5.1.1 Cache Block Size

The selection of block size depends on both the latency and the bandwidth of the lower-level memory [31]. It is possible to achieve higher memory bandwidths on modern memories that are supported by technologies such as wider buses, multiple banks, additional pins, integrated circuit properties of Dynamic Random Access Memory (DRAM) and Synchronous DRAM [54]. However, high memory latency remains an issue that must be addressed. Although high latency and high bandwidth both encourage larger block sizes because the cache gets many more bytes per miss for a small increase in miss penalty, not all applications benefit from access to larger blocks. Increasing block size to reduce the memory latency's impact implies use of prefetching of data for applications exhibiting greater spatial localities, such as applications using streams. On the other hand, applications exhibiting very little spatial but large temporal localities (As exhibited by scalar data items.) cannot benefit from prefetching or larger cache blocks. In fact, for scalar references, it is better to have smaller cache block sizes and more cache lines to eliminate conflict misses and even capacity misses when smaller caches are used [31]. My work is motivated by the observation that it is not possible to design a single cache that works well for the types of localities and data types present in programs. I propose separate data caches designed with different block sizes to meet the needs of different data types.

5.5.1.2 Cache Capacity

Increasing cache size will obviously reduce capacity misses; however, as Hennessy and Patterson note [31], as cache size increases, a capacity miss becomes a conflict miss.

In addition, when the number of capacity misses is small, increasing cache capacity will not benefit the application. Jouppi, *et al.* [39] report increasing cache capacity actually increases the significance of cold-start (Or compulsory) misses. Also, these misses will more likely be sequential. This is particularly the case with stream data types. A larger cache can benefit applications that access several sets of data but not benefit applications that access a single stream [39]. Similar results have been reported for media processing workloads [67].

5.5.1.3 Associativity

For a cache of given size, set-associativity is dictated by a number of criteria, that include implementation cost, access time (Both on hit and miss.) and miss rate. Direct-mapped caches are simpler and easier to design and require less silicon area than set associative caches. However, direct-mapped caches also have higher miss rates. Conversely, for caches with higher associativity the main advantage is a lower miss rate, but such caches are more expensive and incur longer access times on hit. The goal of a computer architect is to maximize performance while staying within the cost and power constraints. A more desirable cache design would reduce the conflict miss rate to the same extent as a set associative cache, but at the same time, the design would maintain the critical hit access path of the direct-mapped cache. Because of a lack of temporal locality, stream references cause more compulsory misses than conflict misses. Direct-mapping offers a better option for an array cache. For a scalar cache, increasing associativity leads to a reduction of conflict misses and exploitation of temporal locality.

5.5.2 Empirical Results with Split Data Caches

By changing block size, cache capacity, and associativity, designers have attempted to obtain the best configurations for array and scalar caches. My work continues these efforts. The next three subsections present my experimental results. In this section I compare the effective miss rate of split cache against that of conventional unified cache (For both stream and scalar data types.) to support my view that completely separating array and scalar data items can be a key to boosting cache performance.

5.5.2.1 Selection of Block Size

In a fixed-sized cache, increasing block size results in decreased number of lines. Consequently, when using a single data cache, it is impossible to achieve a balance between block size and number of lines.

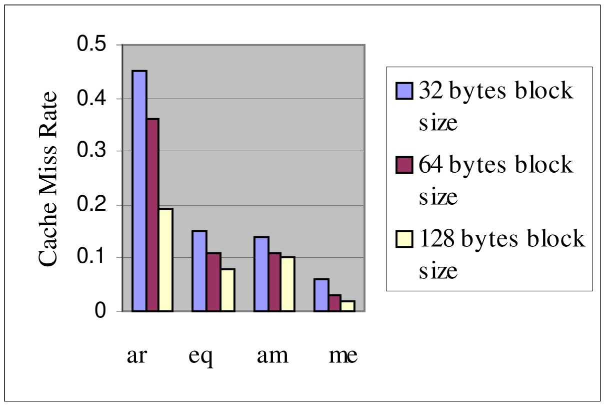


Figure. 5.2. Decrease in Cache Miss Rate with Increase in Block Size of 4 Kilobytes Array Cache

Figure 5.2 shows the decrease in miss rate with increasing block sizes in a 4 kilobytes array cache. As shown in Figure 5.3, for the benchmark me (Testing scalar cache.), increasing block size actually caused an increase in miss rate. Similar results have been found for the other three benchmarks. Consequently, in my proposed architecture I take advantage of both techniques—using larger cache blocks for array caches and using smaller block sizes for scalar cache.

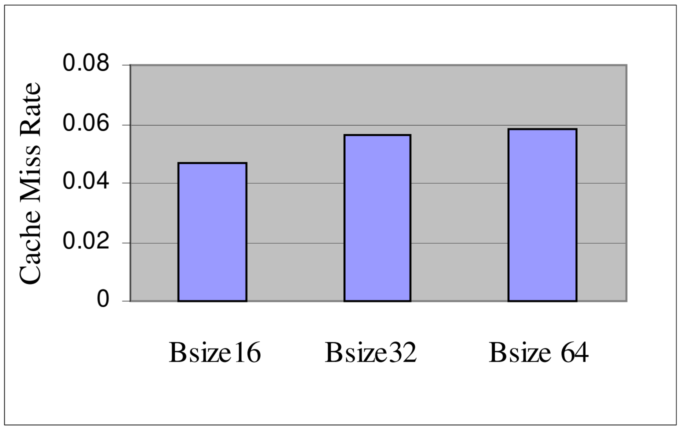


Figure. 5.3. Increase in Cache Miss Rate with Increase in Block Size of Scalar Cache for Benchmark 177 Mesa

5.5.2.2 Selection of Cache Size

As mentioned in section 5.5.1.2, an important criterion for selecting cache size is the frequency of capacity misses. I expected that when separate scalar and array caches

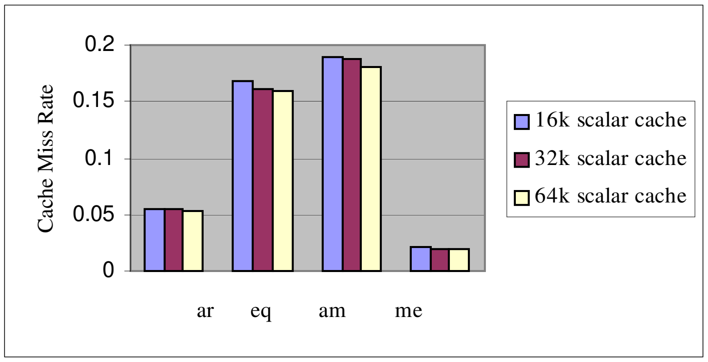


Figure. 5.4. Changes in Cache Miss Rate with Increase in Cache Size of Scalar Cache

were used, the scalar cache could be small (Say a 4 kilobytes level one.) because the number of capacity misses was small with scalar data items. As Figure 5.4 shows, almost no improvement was achieved for scalar cache even when cache size was doubled or quadrupled. For this reason, Idecided to use a 4 kilobytes or 8 kilobytes scalar cache.

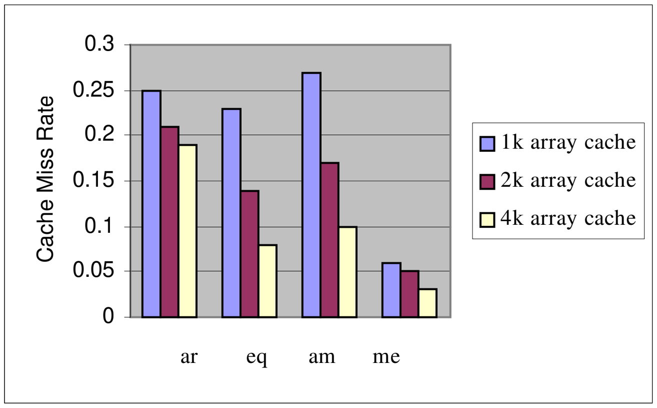


Figure. 5.5. Changes in Cache Miss Rate with Increase in Cache Size of Array Cache

Figure 5.5 shows that for array cache increasing cache size with increasing block size reduced the miss rate. But, I did not repeat my experiments with larger array caches than 4 kilobytes. Jouppi [39] and Ranganathan, *et al.* [67] have already demonstrated that for stream references miss rate increases with cache size unless even larger block sizes are used.

5.5.2.3 Selection of Associativity

Several experiments were performed to determine the optimum associativity for each cache type. The cache miss rates for each benchmark were then plotted. From Figure 5.6 we can see that for array cache, except for **me**, increasing associativity was not worth its cost. This observation is consistent with my initial expectations.

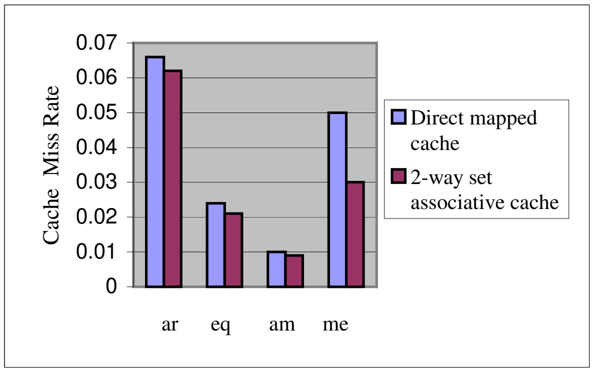


Figure. 5.6. Changes in Cache Miss Rate for Array Cache with Different Associativity

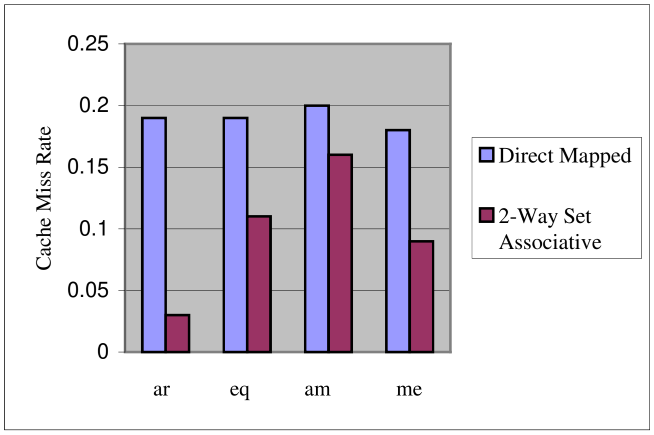


Figure. 5.7. Changes in Cache Miss Rate for Scalar Cache with Different Associativity

In my test suite, the percentage of capacity misses was very low and after removing streamed references for my scalar cache, conflict misses were my main concern. For the scalar cache, while the lack of capacity misses first directed me to use a small scalar cache, this decision and the higher conflict miss rate that resulted then convinced me to select 2-way set associativity. It is obvious from Figure 5.7 that for scalar cache increasing associativity reduces the miss rates.

5.5.2.4 Comparison of Split Array and Scalar Data Cache with Conventional Unified Data Cache

After evaluating my results to determine optimal configurations for both array and scalar caches, I compared weighted effective miss rates for array and scalar caches for the four benchmarks against the miss rate of a unified, 16 kilobytes data cache. Figure 5.8 shows the effectiveness of cache splitting across the benchmark suite. The split array and scalar

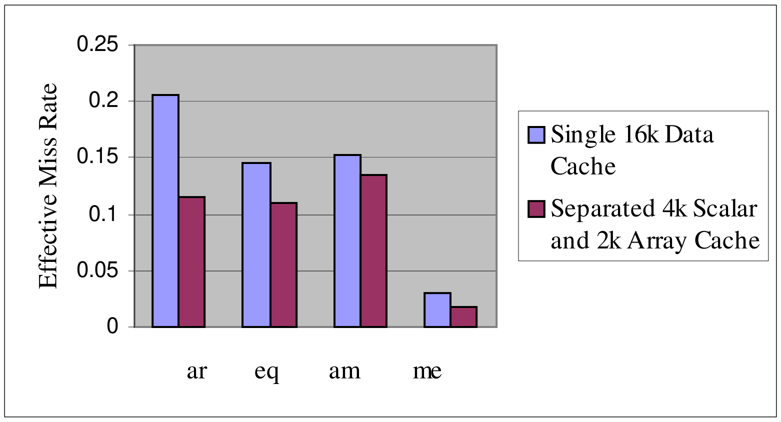


Figure. 5.8. Reduction in Effective Miss Rate with Split Array and Scalar Caches

cache demonstrate uniform superiority over the conventional unified data cache design for each of the benchmarks. For 4 kilobytes, 32 bytes block sized scalar cache and 2 kilobytes, 128 bytes block sized array cache, I achieved 43.41%, 24.14%, 11.76% and 43.33% improvement over a 16 kilobytes, 64 bytes block sized unified scalar cache for the ar, eq, am and me benchmarks. Some of these results were published in [59].

5.6 Split Data Caches Integrated with Victim Cache and Stream Buffer: An Approach for Further Improvement

Now I will describe my integrated approach and its performance.

5.6.1 Victim Cache, Stream Buffer and the Integrated Approach

As my approach combines three techniques within a single framework, I will first review each of these techniques. I follow this overview by describing my integrated approach in 5.6.1.3. In 5.6.2, I present the results I achieved when implementing an integrated approach.

5.6.1.1 Functionality of Victim Cache

The design of a Level one (L-1) cache always tries to establish a balance between miss rate and access time. Addition of a victim cache to a direct-mapped cache can ease this problem by reducing the conflict miss rate to the same extent as a set associative cache while maintaining the critical hit access path of a direct-mapped cache. Victim cache temporarily holds data evicted from the cache and, because of its full associativity, it can simultaneously hold many blocks that would conflict in direct-mapped cache. If the number of conflicting blocks are small enough to fit in victim cache, both the miss rate to the next memory level and the average access time will be improved due to relatively low miss penalty for fetching from victim cache.

5.6.1.2 Functionality of Stream Buffer

Although increasing line size provides the simplest means of prefetching, Jouppi [39] argues that line sizes cannot be made arbitrarily large. Other conventional prefetching methods also have deficiencies [8, 52]. The stream buffer not only mitigates traditional problems with larger cache lines and extensive prefetching; a stream buffer functions more effectively than other investigated prefetch techniques [39]. Because in a cache with Stream buffer, the successive blocks are stored in the buffer rather than in the cache, stream buffer can avoid premature displacement of data (Cache pollution). However the biggest problem with stream buffers is that non-spatial data needs to be flushed when detected. Jouppi's investigation did not explore the stream buffer only for data with spatial localities (Such as streams.), rather the buffer was used for all data items.

5.6.1.3 Functionality of the Integrated Approach

So, can I design a combined approach that provides better performance than either applying only one or applying each independently? Until now there has not been significant research investigating the interaction among split caches, victim cache, and stream buffers. In section 5.5 I already have shown that using separate (data) caches for indexed or array data and for scalar data items can lead to substantial improvements in terms of cache misses. Although victim caches and stream buffers can reduce miss rates in L-1 cache, the reduction achieved depends on the cache configuration as well as the data reference types. Now, we will see how a separation of caches can benefit from victim cache and stream buffer.

A conflict miss occurs when data with temporal locality is referenced multiple times but is replaced by another data item in between the references. Victim caching is based on the principle of temporal locality and provides dynamic associativity by allowing up to N+1 conflicting blocks that belong to the same direct-mapped set to co-exist in caches simultaneously, where N is the number of block entries in the victim cache. In his original paper, Jouppi [39] implemented a victim cache for a unified data cache. As a result, array or stream elements remove scalar data from the victim cache causing expensive victim cache pollution. In my work, as I remove the array references from the scalar cache, the victim cache not only has to deal with fewer references (Victim cache is reported to work better for small cache [39].) but also without being polluted by stream references. The reduced cost of using small victim cache with direct-mapped data cache outweighs the performance gains of having a cache with large associativity.

A cold miss occurs when stream or array data are traversed linearly by using the elements only once or very few times during traversals. Stream buffers exploit spatial locality and perform prefetching for stream or array data. Jouppi's analysis [39] also included the stream buffer for a unified data cache. For that reason every time scalar data detected, the buffer was flushed. In my study because I am removing the contaminating scalar data from array cache the performance can increase significantly by augmenting array cache with stream buffers. In this study, I implemented the promising aspects of victim cache in keeping conflicting blocks to satisfy the requirements of scalar cache and the prefetching ability of stream buffer was included with the array cache to exploit its capability of streaming data.

5.6.2 Empirical Results with Integrated Approach

To test my hypotheses about the effectiveness of an integrated approach I completed a series of simulations. The next three subsections present the selection of cache organizations in the same order as these parameters were described in section 5.6.1. I also compare the effective miss rate of my cache against that of conventional unified cache. The results support my view that a complete separation of array and scalar data items with victim cache and stream buffer can be a key to boosting cache performance.

5.6.2.1. Results with Victim Cache

In Table 5.2 I compare the miss rates of a 4 kilobytes direct-mapped scalar cache without victim cache, a 4 kilobytes 2-way set-associative scalar cache, and a 4 kilobytes direct-mapped scalar cache with victim cache (256 bytes) respectively. Figure 5.9 presents the percentage reduction in access time by switching from 2-way set associative to direct-mapped scalar cache with victim cache. From Table 5.2 we can see that direct-mapped scalar cache with victim cache does not always results in reduction in cache miss rate (For benchmarks eq, me, fm.). However for each of the benchmark my scalar cache with non-swapping victim cache provides huge reduction in cache access time. From this example we can see that access time is more useful than miss rate in evaluating the performance of a cache scheme. In this case although a 2 way set-associative cache design demonstrate lower miss rate, this lower miss rate is achieved at the expense of the longer hit access times. Where as the small victim cache allowed a significant reduction in access times for scalar data items. Given that access time is a better metric of cache performance than miss rate, my experiments show the significant benefit possible with a victim cache.

Table. 5.2. Comparison of Miss Rates of Direct-mapped Scalar Cache without Victim Cache, 2-way Set-associative Scalar Cache, and Direct-mapped Scalar Cache with Victim Cache

Benchmark name	Direct-mapped scalar cache without victim cache	2 way associative scalar cache	Direct-mapped scalar cache with victim cache
ar	0.159	0.03	0.03
eq	0.172	0.11	0.12
am	0.171	0.16	0.16
me	0.095	0.06	0.07
mg	0.82	0.82	0.81
ap	0.139	0.12	0.12
fm	0.075	0.05	0.06
sx	0.11	0.113	0.1

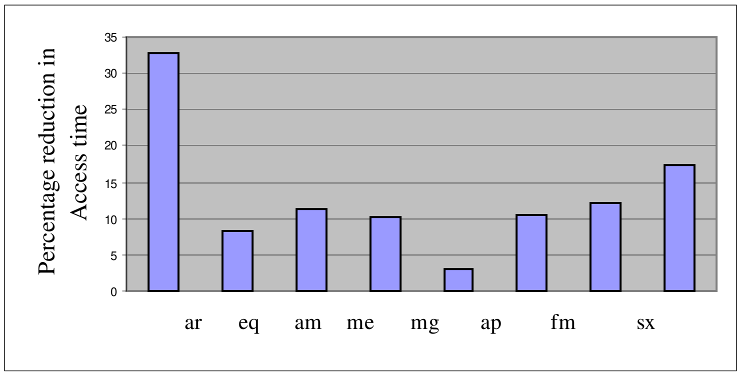


Figure.5.9. Percentage Reduction in Access Time by Switching from 2-way Set-associative to Direct-mapped Scalar Cache with Victim Cache

5.6.2.2. Results with Stream Buffer

To evaluate the benefit of stream buffers with the array cache, I used multiple (2) stream buffers of 8 elements. The cache miss rates of a 4 kilobytes array cache without stream buffer and a 4 kilobytes array cache with stream buffers are plotted in Table 5.3. Figure 5.10 shows the percentage reduction in access time by addition of stream buffers. From Table 5.3 and Figure 5.10 I can see that for each benchmark, addition of stream buffers with direct-mapped array cache results in huge reduction in cache miss rate and cache access time respectively.

Table. 5.3 Comparison of Miss Rate of a 4 Kilobytes Array Cache without Stream Buffer and a 4 Kilobytes Array Cache with Stream Buffer

Benchmark name	4k Direct-mapped array cache without stream buffer	4k Direct-mapped array cache with stream buffer
ar	0.357	0.034
eq	0.108	0.017
am	0.111	0.023
me	0.05	0.01
mg	0.37	0.045
ap	0.125	0.015
rm	0.052	0.01
sx	0.135	0.018

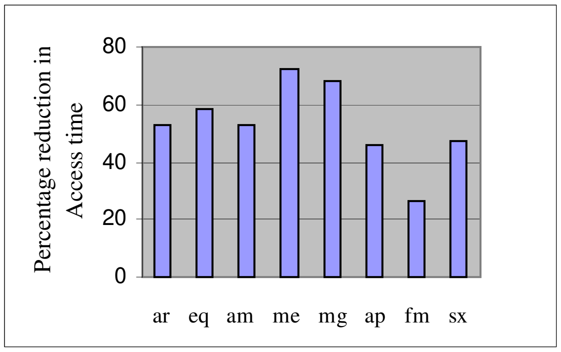


Figure. 5.10. Percentage reduction in Access Time by Addition of Stream Buffers

5.6.2.3 Results of combining Victim Cache, Stream Buffers, and Split Caches

After evaluating to determine optimal configurations for victim cache and stream buffers, I compared the weighted effective miss rate for array and scalar caches against the miss rate of

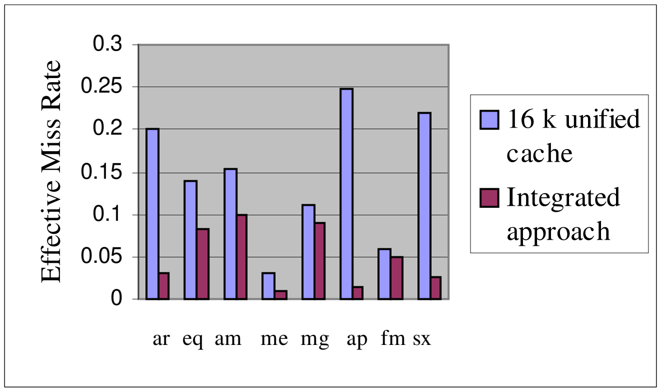


Figure. 5.11. Reduction in Effective Miss Rate with Integrated Approach

a unified 16 kilobytes data Cache. The results are shown in Figure 5.11. For a 4 kilobytes, 64 bytes block sized scalar cache with 256 bytes Victim cache and a 4 kilobytes, 64 bytes block sized array cache with 512 bytes stream buffer on average, my integrated approach achieved a 55% improvement over a 16 kilobytes, 64 bytes block sized unified scalar cache for the benchmark set. Some of these results were published in [60]. Based on these results, I believe my integrated approach demonstrates a uniform superiority over conventional unified data cache design across all of the benchmarks I tested.

5.7 Conclusions

The widening gap between processor and memory speeds makes data locality optimization an important issue when designing modern cache systems. Computer architects focus on optimizing data cache locality using intelligent cache management mechanisms. In this chapter, first I presented the evaluation of a split array and scalar data caches. Then I investigated the interaction between three established methods: split cache, victim cache and stream buffer and proposed a strategy to optimize cache locality for scientific applications. Simulation results showed that my technique improved miss rates with respect to the base configuration, even while using a smaller, combined cache footprint. My investigations further demonstrated how designers can combine three inherently different approaches to work together to further improve data localities.

CHAPTER 6

PROSPECTIVE OF PROPOSED CACHE IN THE ARENA OF EMBEDDED SYSTEMS

So far I have evaluated my proposed caches only for desktop applications. More recently investigators have begun to focus on the use of cache memories for improving embedded systems' performance. Because my cache design offers a reduction in cache size, I believe it will be equally beneficial for embedded applications. As power consumption is of paramount importance for embedded applications, I also include power consumed by an application as a performance metric.

6.1 Introduction

Computing elements can be found embedded in almost every device and gadget we use. Challenges to the design of processing elements for embedded applications are more stringent than those for desktop applications. Embedded applications place requirements along a number of dimensions including tighter constraints on functionality and on implementation. Not only must the application's functionality be correct, it often must meet strict timing constraints and be designed to function within limited resources such as memory size, available power, and allowable weight. Unfortunately, physical size, real-time predictability, and small energy budgets have made caches less commonly used for embedded systems [35]. So, I feel that it is worthwhile investigating new cache organizations to address both the performance and the power requirements of embedded systems. In this chapter I explore how to design small caches that achieve high performance for embedded applications while remaining both energy and area efficient.

In chapter 5 I proposed implementing split data caches for desktop applications by employing an array cache and a scalar cache. My results in those studies have shown that a split cache organization achieves lower miss rates and shorter average access times—even when the combined size of array and scalar caches is roughly one quarter the size of the unified data cache used in my comparisons. In this chapter I report on my investigations into whether split caches can also benefit embedded applications. In this research I hypothesized that these benefits would be evident particularly when working with small Level one (L-1) caches often found in embedded systems.

This work makes several significant contributions. First, leveraging my previous studies of split data caches for scientific applications, I evaluated split data caches for applications often encountered in embedded systems. Second, I evaluated my integrated cache architecture that used split data caches with a victim cache and stream buffers to further reduce (silicon) area, access time, and dynamic power consumed by cache memories. When my augmented split caches are used for embedded applications, results showed excellent reductions in both memory size and memory access time, translating into reduced power consumption. My cache architecture (Scalar cache 512 bytes, array cache 4096 bytes, victim cache 256 bytes and stream buffer 128 bytes.) reduces the overall cache size by 80%, access time on average by 39%, and energy consumption on average by 26% when compared with an 8 kilobytes unified direct-mapped cache with a 32 kilobytes Level two (L-2) cache.

Finally, I performed a comprehensive evaluation of my proposed methods. I survey a variety of techniques that have traditionally been applied to improve cache systems for desktop applications (Described in section 2.3.). I employed a simulation environment and developed a power evaluation model. I examined direct-mapped caches, conventional set-associative caches, three different probe caches (Hash rehash [1], Column associative [2] and MRU [17, 75].), direct-mapped caches with prefetching and direct-mapped caches augmented with victim cache as representatives of traditional approaches. Following the same organization in as I used in section 6.2, first I present my evaluation of split caching with all these traditional cache organizations. Next I investigate the inclusion of a victim cache and a stream buffer together with split data cache architecture. I compare the miss rate, access time, area and power consumption of proposed integrated cache against that of different conventional cache architectures in the context of Mi-bench benchmarks. For all of the benchmarks except one, my split data caches deliver higher performance than the alternate organizations. My cache architecture reduced the overall cache size by 43%, access time by 37% and energy consumption by 63% when compared with a unified 2-way set-associative cache. In my study the reduced costs of using victim cache with tiny scalar cache and stream buffer with array cache allow to achieve up to 33% reduction in power consumption when compared with a traditional unified caches with victim cache and 80% when compared with a traditional unified caches with prefetching for embedded applications. Some of these results were published in [61, 63].

6.2 Proposed Cache Organizations

In this chapter I propose a split data cache organization for embedded systems. I believe that cache splitting is a step in the right direction because it plays a role in achieving cost-performance goals for embedded systems. This cache organization allows to exploit temporal and spatial localities to improve performance, minimize the memory footprint, and to lower energy consumption. As I discuss in the next sections, to prove this claim, I first evaluate the effectiveness of a split data cache employing separate array and scalar caches. The design of my work in this area are reported in 6.2.1. Next I investigate the integration of victim caches and stream buffers to further evaluate the efficacy of my split cache design for embedded systems. This investigation is discussed in 6.2.2.

6.2.1 Evaluation of Split Data Caches for Embedded Systems

As noted in the previous chapter, I found split data caches to provide significant improvements in the performance of desktop applications. In this series of studies I extend my case for split data caches to embedded applications. To evaluate the efficacy of such caches for this purpose, I used a simulation environment and modeled various major cache techniques as I described in section 2.3. This allowed me to compare my split data cache organization's results with these achieved using alternate organizations. (Details of these comparisons are included in Section 6.5.1.) I compared my cache design with four conventional organizations: a direct-mapped cache, a 2-way set-associative cache, a cache supported by prefetching, and a direct-mapped cache augmented by a victim cache. For prefetching, I used a simple prefetching scheme where every miss induced prefetching of the next two blocks. The victim cache was an 8-line, fully associative cache with swapping of cache lines between the primary and victim caches on a miss. For each of these cache configurations I measured miss rates, access times, and power consumption.

6.2.2 Generalization of Split Caches with Victim Cache and Stream Buffer

In my next set of experiments, I augmented my split cache design with a small victim cache and small stream buffers. My goal was to determine the applicability of victim caches and data prefetching to embedded systems. I compared my cache designs with set-associative caches, with hardware prefetching where every miss induced prefetching of the next two blocks, and with victim cache, augmenting a unified data cache. In this chapter I also compared my organization, which does not use L-2 cache, with a direct-mapped cache supported by L-2 cache. Other alternate cache organizations (2-way set-associative cache and unified data cache augmented with victim cache or prefetch.) did not include L-2 cache. I compared the designs for average access times, silicon areas and energy consumptions.

6.3 Related Work

My proposed integrated cache differs from the split caches for embedded systems (Described in section 3.2.) in two ways. First, my proposed cache is augmented by a victim cache and stream buffers while the others are not. Second, unlike the other reported studies (Except minimax cache.), I performed analyses not only with miss rates but also on power consumption and access times.

6.4 Experimental Methods

In experimental evaluations of my proposed integrated approach for embedded applications, I use benchmarks from the Mi-Bench suite.

Table 6.1: Memory configurations for Cache Designs Used in Comprehensive Evaluation

Scalar cache configuration		Size – 512 bytes, Block size – 32 bytes, Associativity – Direct mapped
Array cache configuration		Size – 4096 bytes, Block size – 32 bytes, Associativity – Direct mapped
Scalar Victim cache configuration		Size – 256 bytes, Block size – 32 bytes, Associativity - Fully associative, Replacement policy – LRU, non swapping
Stream buffer configuration		Total Size – 128 bytes, Block size – 32 bytes, Number of stream buffer - 2
Direct-mapped cache configuration		Size - 8192 bytes, Block size – 32 bytes, Associativity – Direct mapped
2-way set-associative configuration		Size - 8192 bytes, Block size – 32 bytes, Replacement policy – LRU
Hash rehash cache configuration		Size – 8192 bytes, Block size – 32 bytes
Column associative cache configuration		Size – 8192 bytes, Block size – 32 bytes
MRU cache configuration		Size – 8192 bytes, Block size – 32 bytes
Victim cache configuration	Main cache	Size – 8192 bytes, Block size – 32 bytes, Associativity – Direct mapped
	Victim cache	Size – 256, bytes, Block size – 32 bytes, Associativity - Fully associative, Replacement policy – LRU, swapping
Prefetching cache configuration		Size – 8192 bytes, Block size - 32bytes, Associativity – Direct-mapped Prefetches 2 lines

To compare the various cache design alternatives I developed a suite of simulators, with ATOM instrumentation and analysis tools. Table 6.1 lists the architectural parameters for each cache configuration used in my studies.

6.5 Empirical Results

The next two sections present the results of my study. In the section 6.5.1 I show the results for my split data cache. Section 6.5.2 shows the results for my split caches augmented by a victim cache and stream buffers.

6.5.1 Results with Split Data Caches for Embedded Systems

I compare my split cache organization (With an array and a scalar cache.) with a direct-mapped cache, with a conventional 2-way set-associative cache, with a direct-mapped cache that uses prefetching and with a direct-mapped cache augmented by a victim cache for cache miss rate, cache access time and power consumption.

Table. 6.2. Comparison of the Miss Rate of Split Cache with Traditional Cache Architectures

Name of Benchmark	Direct Mapped Cache	2 way Set-Associative Cache	Direct Mapped with Victim Cache	Direct Mapped with Prefetching	Split Cache
art	0.205	0.175	0.177	0.187	0.116
ammp	0.11	0.09	0.103	0.136	0.09
mesa	0.03	0.02	0.0229	0.027	0.017
bf	0.167	0.151	0.159	0.21	0.16
bc	0.0001	0.00007	0.00009	0.0001	0.0001
cj	0.11	0.097	0.112	0.165	0.1

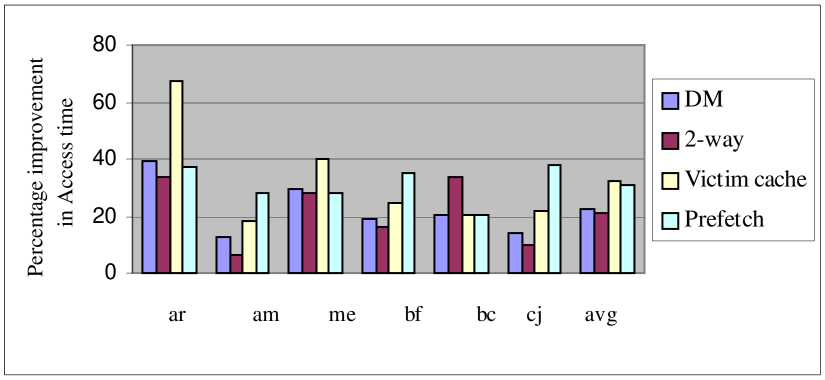


Figure. 6.1 Percentage Improvement in Access Time by Using the Split Data Caches

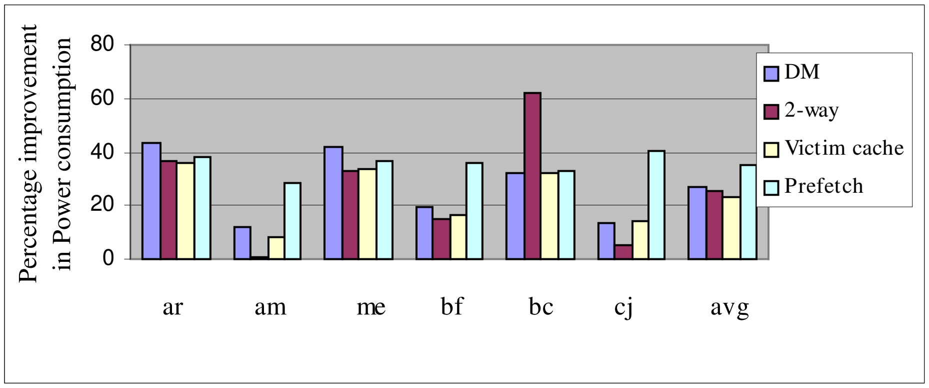


Figure. 6.2. Percentage Improvement in Power Consumption by Using the Split Data Caches

In Table 6.2 I compare the weighted effective miss rates for my split cache method with the miss rates of the studied conventional cache architectures. In Figures 6.1 and 6.2 I compare my split data cache organization with other cache organizations in terms of percentage improvement of access times and power consumptions. In each figure, I also include the average for the selected benchmarks. Although from Table 6 I can see that my split cache does not result in significant reduction in cache miss rate, both Figure 1 and 2 indicate that the split data cache organization led to a significant reduction in access time and in power consumption. As access time and power consumption are better metrics of cache performance than miss rate, my experiments showed the significant benefits. For example, for benchmark ar my split cache showed a more than 60% reduction in access time when compared with a direct-mapped cache augmented with a victim cache. For benchmark bc my split data cache showed a more than 60% reduction in power consumed when compared with a 2-way set-associative cache. The average across all benchmarks indicated a 23% reduction in access times when compared to direct-mapped cache, 21% when compared to 2-way set-associative cache, 32% when compared to direct-mapped cache with a victim cache, and 31% when compared to direct-mapped cache with prefetching. Likewise, in terms of power consumption, my cache offered reductions of 27% when compared to direct-mapped cache, 26% when compared to 2-way set-associative cache, 23% when compared to direct-mapped cache with a victim cache and 35% when compared to direct-mapped cache with prefetching.

6.5.2. Results with Generalization of Split Caches with Victim Cache and Stream Buffer

In this section I present the experimental results achieved using my split caches augmented by a victim cache and stream buffers. Because access time, area and power consumption are more important than the miss rates for embedded systems (And because I am actually using the miss rate to compute both access time and power consumption.), in this section my evaluation compares cache designs for three metrics: cache area, access time, and power consumption.

In Figure 6.3, I show the percentage improvement in silicon area (Reduction in area.) achieved by my cache organization when compared to the area needed by other cache organizations. It should be mentioned that the silicon area required depends solely on the

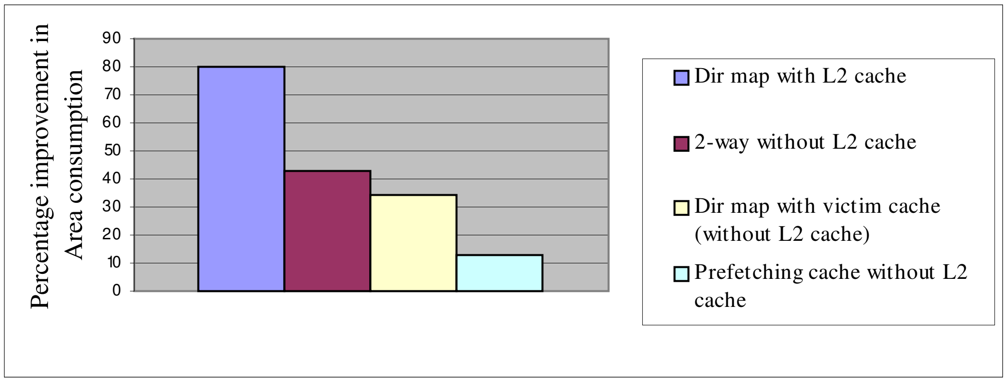


Figure. 6.3. Percentage Improvement in Area Consumption by Using the Integrated Approach

cache design; it does not depend on the actual application. The Y-axis shows the percentage improvement (i.e., reduction in silicon area.) my cache design exhibited. For example, my cache organization shows an 80% reduction in the area required when compared to a unified, 8 kilobytes, 32 bytes block sized direct-mapped cache with a 32 kilobytes, L-2 cache and a 43% reduction when compared to a unified, 8 kilobytes, 32 bytes block sized 2-way,

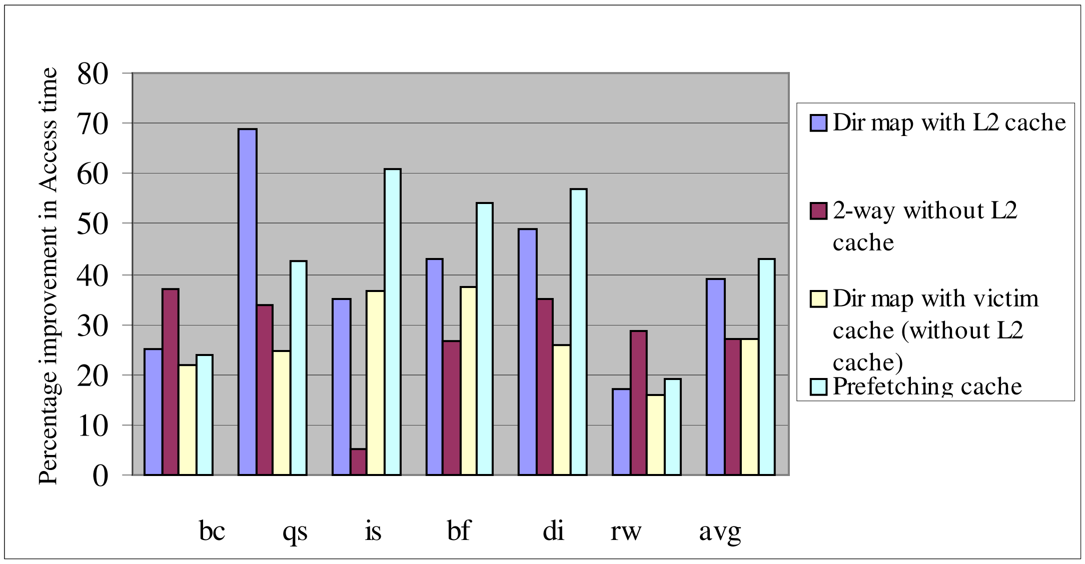


Figure. 6.4. Percentage Improvement in Access Time by Using the Integrated Approach

set-associative cache without an L-2 cache. It should be noted that my cache organization does not include an L-2 cache. As Figure 6.3 clearly shows, my cache architecture requires a smaller silicon area when compared to the area used by the other cache organizations.

In Figure 6.4 I compare my cache organization with other cache organizations in term of access times. Here I show the percentage reduction in access times resulting from my cache system when compared to the access times for alternate organizations. My organization consistently had faster access times across all of the benchmarks than the access times of the other cache designs. For example, for benchmark **qs** my split cache achieved a more than 69% reduction in access times when compared to an 8 kilobytes, 32 bytes block sized direct-mapped cache using a 32 kilobytes, L-2 cache. In this figure I also report the average reductions achieved for all the tested benchmarks. My cache design improved access time on average by 39% when compared with a direct-mapped cache with L-2 cache; 27% when compared to a 2-way, set-associative cache; 27% when compared to a direct-mapped cache with a victim cache, and 43% when compared to a direct-mapped cache with prefetching.

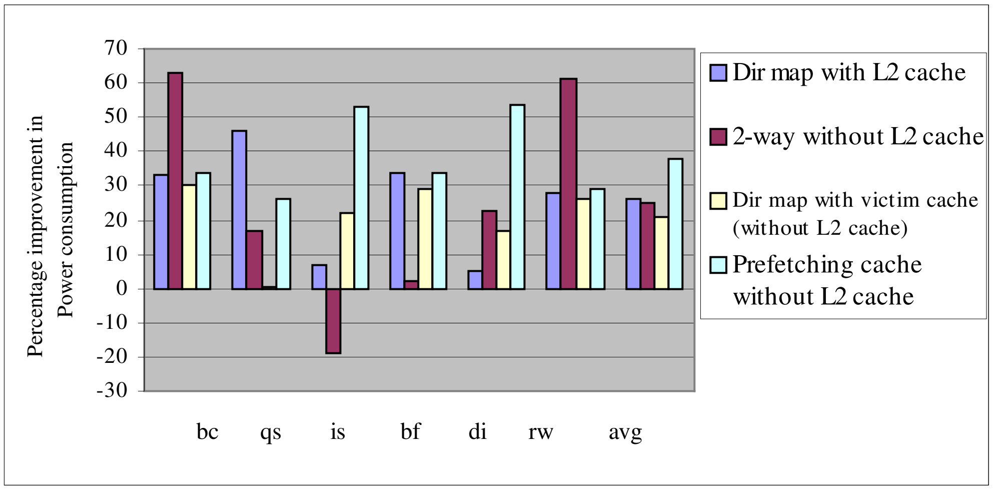


Figure. 6.5. Percentage Improvement in Power Consumption by Using the Integrated Approach

The percentage improvements in power consumption achieved by my design when compared to other cache organizations are plotted in Figure 6.5. In this experiment, my split data caches resulted in significant energy savings for all benchmarks except **is**. This benchmark is a spelling checker that contains more scalar data items than array data. As a result, my 512 bytes scalar cache was too small to accommodate the benchmark's needs. This size problem led to more cache misses that had to be satisfied by longer access paths to main memory. These longer access paths in turns led to higher energy consumption. Note that I do not employ L-2 cache; thus, L-1 misses must be satisfied by accesses to main memory. It would have been beneficial if the space unused by the array cache were reconfigured to support scalar data for this application. I will discuss such options in next chapter.

For streaming benchmarks ra and bc my approach achieved a more than 60% energy reduction when compared with a conventional 8 kilobytes, 32 bytes block sized 2-way, set-associative cache configuration. The average reductions for all benchmarks I used are also included in the figure. The average across the benchmarks indicates that my cache reduced power consumption by 25% when compared to a direct-mapped cache with an L-2 cache; 24% when compared to a 2-way, set-associative cache; 21% when compared to a direct-mapped cache with a victim cache; and 38% when compared to a direct-mapped cache with prefetching.

6.6 Comprehensive Evaluation of Proposed Caches

In this section I present complete results of my experiments. For the comprehensive evaluation of my proposed approach, I used a simulation environment and modeled all major cache techniques as I described in section 2.3. I compared my cache design with direct-mapped cache, with a 2-way set-associative cache, with a Hash rehash cache[1], with a Column associative cache [2], with a MRU cache [17, 75], with a cache supported by prefetching, and a direct-mapped cache augmented by a victim cache. I did not include probe caches with way-prediction mechanisms because of their higher energy requirements (Due to hardware complexity.). For prefetching I used a simple prefetching scheme where every miss results in prefetching of the next block. The victim cache was an 8-line, fully associative cache with swapping. My evaluation of split caching compares the cache designs for three metrics: the miss rate, the access time and the power consumption. I have found that my split cache method offered improvements for all three metrics when compared to other cache architectures. The results support my view that a complete separation of array and scalar data caches can decidely boost cache performance in the embedded systems.

Table 6.3: Comparison of the Miss Rate of Split Cache with Traditional Cache Architectures

Name of Benchmark	Direct Mapped Cache	2 Way Set-Associative Cache	HashRe hash Cache	Column Associative Cache	MRU Cache	Direct Mapped with Victim Cache	Direct Mapped with Prefetching	Split Cache
ar	0.205	0.175	0.184	0.175	0.175	0.177	0.187	0.116
am	0.11	0.09	0.11	0.09	0.09	0.103	0.136	0.09
me	0.03	0.02	0.026	0.02	0.02	0.0229	0.027	0.017
bf	0.167	0.151	0.154	0.151	0.151	0.159	0.21	0.15
bc	0.0001	0.00007	0.00009	0.00007	0.00007	0.00009	0.0001	0.0001
cj	0.11	0.097	0.101	0.097	0.097	0.112	0.165	0.09

In Table 6.3 I compare the weighted effective miss rates for my split cache method with the miss rates of the other studied cache organizations. Where as in Figures 6.6 and 6.7 I compare my split data caches with other cache organizations in terms of access time and power consumption. These figures reveal that the percentage improvement resulting from my approach when compared to other organizations. In each figure, I also include the average improvement for all benchmarks. From Table 6.3 I can see that my split cache does not results in huge reduction in cache miss rate. However Figures 6.6 and 6.7 show that for each of the benchmark my split cache provides huge reduction in cache access time and power consumptions over all of the traditional methods. For example for benchmark **ar,** my split cache achieves a more than 60% improvement in access time when compared with a direct-mapped cache with a victim cache. When compared for power consumption, my split data caches show almost 60% improvement for benchmark bc over both 2-way, set-associative caches and MRU [17, 75] caches. From this example again I can see that access time and power consumption are more useful than miss rate in evaluating the performance of a cache scheme.

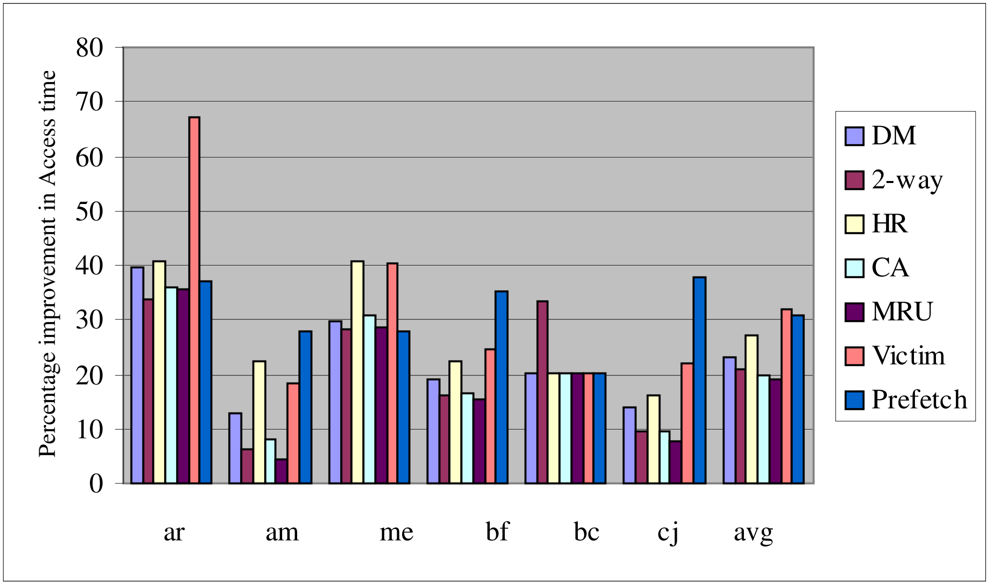


Figure 6.6: Percentage Improvement in Access Time by Using the Split Data Caches

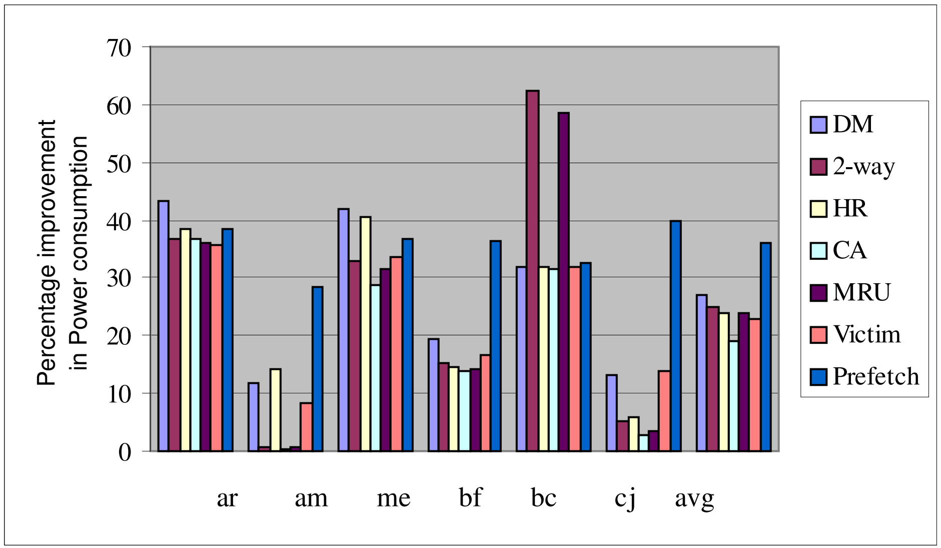


Figure 6.7: Percentage Improvement in Power Consumption by Using the Split Data Caches

After evaluating my split data cache, I perform the comprehensive evaluation of my integrated approach. Because miss rate does not depict the true picture, (As seen from previous example.) for results of my integrated approach I did not show the reductions in miss rates. Rather I include analysis of reduction of area consumption, because for embedded systems, it can be beneficial. Here it should be mentioned that in evaluating of split cache, I did not perceive a reduction in cache area consumption because each of my array and scalar cache was 4 kilobytes and their combined area was proportional to the 8 kilobytes unified caches. However, my integrated approach reduced my scalar cache's size by 88% of its previous size. In Figure 6.8, I compare the percentage improvement in silicon area needed by my proposed integrated cache when compared to the size of areas needed by other cache architectures. For cache parameters see Table 6.1. Figure 6.8's Y-axis presents the percentage improvements for my cache. For example, my cache organization showed a 43% reduction in area when compared with a unified, 8 kilobytes, 2-way, set-associative cache. I was unable to compare the area savings of my integrated cache with probe caches (Hash rehash [1], Column associative [2] and MRU [17, 75].) because published literature provided too little information on their hardware implementations. However, I posit that the silicon area required for these three alternatives will be at least as much as a 2-way set-associative cache. As we can see from Figure 6.8 my proposed cache architecture consistently showed a reduction in area over all other organizations used in my study.

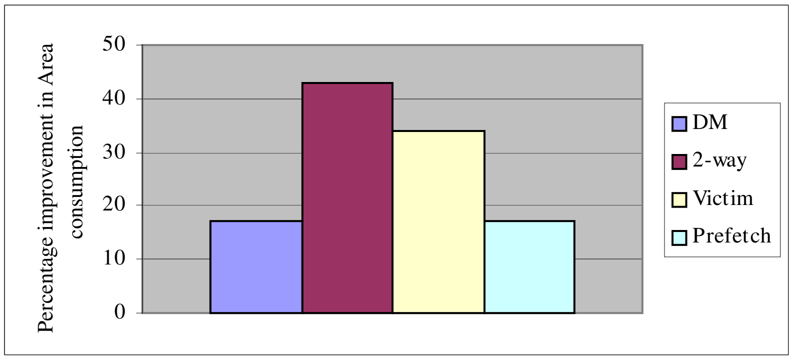


Figure 6.8: Percentage Improvement in Area Consumption by Using the Integrated Approach

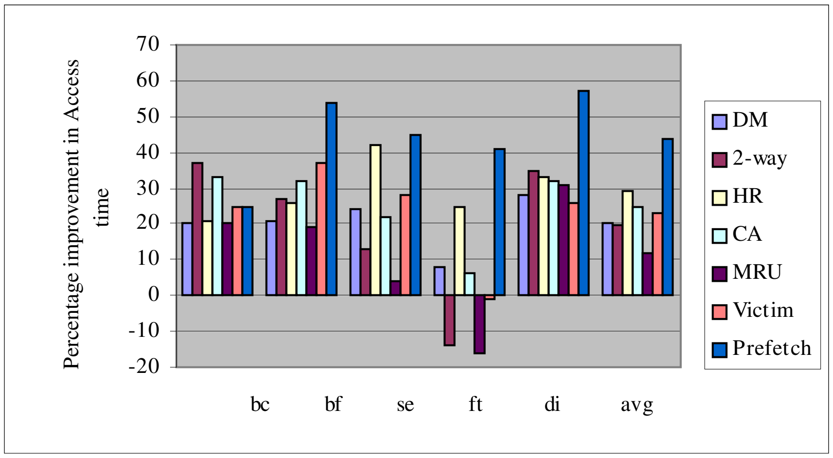


Figure 6.9: Percentage Improvement in Access Time by Using the Integrated Approach

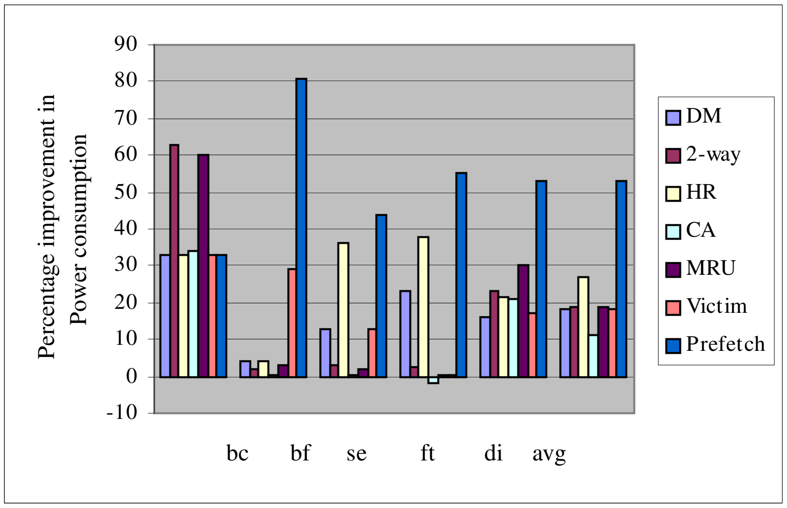


Figure 6.10: Percentage Improvement in Power Consumption by Using Integrated Approach

The percentage improvements in access time achieved by my integrated approach when compared with access times achieved by other cache organizations are plotted in Figure 6.9. My results show that split data caches allowed a significant reduction in access time for all benchmarks except ff. This benchmark is a fast fourier transform loop. For ff because both 8 kilobytes 2-way set-associative and 8 kilobytes MRU are providing faster access to data, I believe the benchmark has more conflicting scalar data. As a result, my 512 bytes scalar cache was too small to accommodate the benchmark's needs. This led to more time being required as main memory had to be accessed. Still, as we can see in Figure 6.9, for the streaming benchmark bc my approach achieves a 37% improvement in access time over conventional 8 kilobytes, 2-way, set-associative cache configurations.

In Figure 6.10, I compare my split data caches with other cache organizations in term of power consumption. My cache organization shows significant improvements for streaming benchmarks. For example, for benchmark bc, my split cache achieves a 63% reduction in power consumption when compared with an 8 kilobytes, 2-way, set-associative cache. The average savings for all benchmarks are also included in Figures 6.9 and 6.10.

6.7 Conclusions

In this chapter I have shown that, when carefully designed, embedded systems can benefit significantly from small split caches. My integrated method permits a systematic trade-off between memory size, power and performance, which has up to now, been unexplored for embedded systems. My approach can significantly reduce the power consumed as well as the memory size while providing better performance (By reducing execution time.) than the same applications executing with conventional direct-mapped or set-associative caches. By separating data accesses into scalar and array (Or stream) references, we learned, one can eliminate conflicts between competing locality types. This conflict elimination, in turn, allows to reduce total cache size. A smaller (combined) cache leads to smaller footprints and reduced power requirements.

The most significant achievement of my work is the ability to include prefetching and victim caches for embedded systems. As mentioned earlier, because of their high-energy requirements, victim caches and prefetching are seldom used in embedded systems. My experimental data using a unified, direct-mapped data cache with a victim cache and a unified, direct-mapped data cache with prefetching both support this assertion. However, I have found that a split data organization with very small scalar and array caches can benefit from the use of victim caches and stream buffers. While traditional prefetching techniques have been explored [8], (premature) prefetching resulted in poor performance due to cache pollution which caused displacing needed data in an untimely manner. This formed the primary reason for not using prefetching in embedded systems. However, my findings in this study persuade me that a carefully designed cache system not only solves the deficiencies of general prefetching; it also solves the problem of stream buffers. Jouppi's analysis [39] included a stream buffer for a unified data cache, and the buffer required flushing each time a scalar data was accessed (Because stream buffers assume contiguous data items.). Because I remove contaminating scalar data from array caches, stream buffers associated with the array cache are flushed less frequently. This results in a huge reduction (As high as 80%.) in power consumption. Likewise, victim caches are unpopular in embedded system design because fully associative victim caches consumes significant amounts of energy. Again, I show that carefully designed cache systems can benefit from victim caches while maintaining low energy requirements. In a split cache organization, as the array references are removed from the scalar cache, the victim cache encounters fewer scalar references. Reduced costs of using non-swapping victim cache that augments my 512 bytes scalar cache allow to generate a 33% reduction in power consumption when compared with the reduction obtained with a traditional unified cache with victim caches. My integrated cache also performs better than larger unified caches using additional levels of cache hierarchy (Such a large L-2 cache.). Ideally, a split cache organization should be dynamically reconfigurable to meet application requirements. For example, for applications possessing very few array or stream references, the array cache supplements the scalar data cache. However, such dynamic reconfigurations require additional hardware. I will investigate trade-offs and reconfiguration options in next chapter.

CHAPTER 7

RECONFIGURABLE SPLIT DATA CACHE: AN APPROACH FOR EMBEDDED APPLICATIONS

In this chapter I evaluate my reconfigurable cache architecture with split data caches (Separate array and scalar data caches.) and instruction cache complemented by a very small prefetch buffer. My goal is to reduce (silicon) area, access time, and dynamic power consumed by embedded systems' cache memories while retaining performance.

7.1 Introduction

The performance of a given cache architecture is largely determined by the behavior of the applications running on the system. Unfortunately the manufacturer sets the cache architecture as a compromise across several applications. This leads to conflicts in deciding on total cache size, line size and associativity. For embedded systems, where every parameter needs to be customized to be cost effective, this "one-size-fits-all" design philosophy is not adequate. This will lead to suboptimal performance and/or power consumption profiles.

Reconfigurability is a design alternative that provides flexibility in the system so that resource constraints like area, power and performance can be balanced based on applications' needs. In the previous chapter I explored the trade-offs in embedded system's area, power and performance within the context of my split data caches. Since for embedded applications, it is necessary to provide the required performance within specified size and power budgets, reconfigurability will be an appropriate option. For that reason I feel that it is worthwhile investigating new reconfigurable cache organizations to address area, performance and power requirements of embedded applications.

The key contributions of this work are as follows. In my design, I first address the problem of improving Level one (L-1) data cache performance for embedded systems through the use of reconfigurable separate array and scalar data caches. I extend my architecture by augmenting the scalar cache with a victim cache [39]. For my reconfigurable L-1 instruction caches, I augment a direct mapped instruction cache with a small prefetch buffer. The sizes of all cache componens can be configured for each application. For L-1 caches my configurations are based on exhaustive searches of cache sizes to find the optimum configuration in terms of three metrics: area, access time and power. Finally, inspired by the reduction in silicon areas and power consumptions resulting from my L-1 caches I implement reconfigurable Level two (L-2) caches. By using a simpler tuning method, at L-2 I emphasized more on area reduction and find an optimum cache size for each application needed to achieve the desired performance. When using my proposed caches for embedded applications, my results show excellent reductions in both memory size and memory access time, translating into reduced power consumption. If we consider tradeoffs in performance of data caches we can achieve as much as 83% reduction in area consumption (Without any increase in execution time.) and as much as 61% in cycles (Without any increase in silicon area.). My cache architecture reduces the cache area by as much as 85% for L-1 instruction and 78% 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 8 kilobytes L-1 instruction and 8 kilobytes L-1 unified data caches. These reductions can be profound when working with small L-1 caches often found in embedded systems. For L-2 instruction cache I achieved on average 50% improvement for power and more than 80% reduction in access time. Whereas for L-2 data cache the average improvement was 50% for power and more than 60% in access time.

My design enables the cache to be divided into multiple partitions so that unused portion can be used for other processor's activities to improve performance (Such as hardware prefetching, instruction reuse, branch predictions.) or the unused portions can shutdown to save power. These performance improving techniques have not been recommended for embedded systems as they require additional hardware for implementing look-up tables, which lead to increased size and power requirements. Since reductions in cache sizes are possible in my designs, the prefetch buffer or look-up tables for these optimizations could be implemented in a partition of the reconfigurable cache instead of using other valuable chip area. Since my reconfigurable approach leverages the sub array partitioning that is already present in modern caches, only minor changes to conventional caches are required. In this chapter I first evaluated how the unused cache sub arrays can be used for instruction or data prefetching. I show that such a use will lead to as much as 67% reduction in execution times when compared with the base case (8 kilobytes direct-mapped unified data cache with a 32 kilobytes L-2 cache.). Even accounting for additional power consumed by prefetching, my structures show an average power reduction for embedded applications of 64% over traditional unified data caches.

Then I explored how unused cache portions can be used for branch predictions. Finally I explored the static and dynamic power reduction by partial/complete shutdown of L-2 instruction and data caches. By setting a few bits in a configuration register, the cache can be configured in software for optimum sizes for each of my caches (In L-1 and L-2 levels.) to utilize the unused area for other processor activities (Such as larger branch prediction tables or prefetch buffers.) or to shut down. For both cases, the reconfiguration leads to only a small overhead in terms of time, power, and silicon area and hardware complexity. Some of these results were published in [62, 64].

7.2 Architectural Design of Reconfigurable Split Caches

Figure 7.1 shows my 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. My cache organization with separated component caches can be reconfigured to meet the specific needs of an application.

In section 7.2.1 I describe the sub array cache partitioning that is usually present in modern caches. The basic design of cache architecture and the modifications needed to design a reconfigurable cache are described in section 7.2.2. In section 7.2.3 I describe my reconfigurable split cache in details.

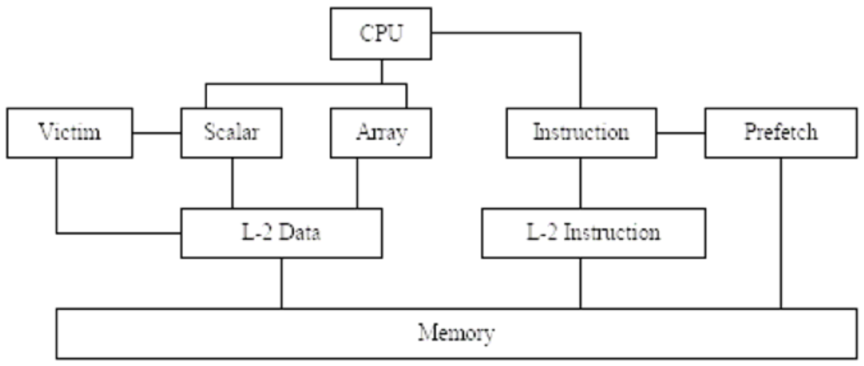


Figure 7.1. Reconfigurable Split Cache Organization

7.2.1 Segmentation of Tag and Data Arrays

Current cache implementations are partitioned into multiple sub arrays. For example, the SA-110 embedded microprocessor [34, 55] uses 32-way associative 16 kilobytes L-1 instruction and data caches, each of which is divided into 16 fully associative sub arrays. The Hewlett Packard PA-8500 microprocessor uses two 512 kilobytes data banks of 1 megabytes 4-way set associative L-1 data cache that are partitioned into four 128 kilobytes sub arrays [51].

In a basic cache organization the data and tag arrays consist of S rows and 8*B*A bit columns, where B is the line (Or block) size in bytes, A is the associativity of the cache, and S is the number of sets. Such an organization is oblong in either along the columns or rows, making accesses slower. To alleviate this problem, cache arrays are broken horizontally and vertically into multiple sub arrays, each approximating a square. Two parameters N_{dwl} and N_{dbl} indicate to what extent the array has been divided -- N_{dwl} indicates how many times the array has been split with vertical cut lines (Creating more, but shorter, word-lines.), while N_{dbl} indicates how many times the array has been split with horizontal cut lines (Causing, shorter bit-lines.). The total number of sub arrays is N_{dwl} * $N_{dbl.}$ With this partitioning in place, the partitioning required for my reconfigurable caches can easily be implemented if there are at least as many sub arrays as the maximum number of partitions (Because for reconfigurable cache different partitions must be implemented in physically different sub arrays of the cache to be indexed by different addresses.).

7.2.2 Hardware Organization

Figure 7.2 [67] shows the structure of a cache using Static Random Access Memory (SRAM) technology. This figure also includes the areas where additional multiplexors are added to implement reconfigurability (Referred by numbered blocks.).

Decoder: The decoder decodes the address and selects an appropriate row by driving one word-line in the data array and one word-line in the tag array. Each sense amplifier monitors a pair of bit-lines and detects the value of the bit.

Comparators: The information read from the tag array is compared with the tag bits of the address. In an A-way set-associative cache, comparators are required to select a line from the set.

Multiplexors Drivers and Output Drivers: The results of the A comparisons are used to drive a valid (hit/miss) output as well as to drive the output multiplexors. These output multiplexors select the proper data from the data array (In set-associative cache or a

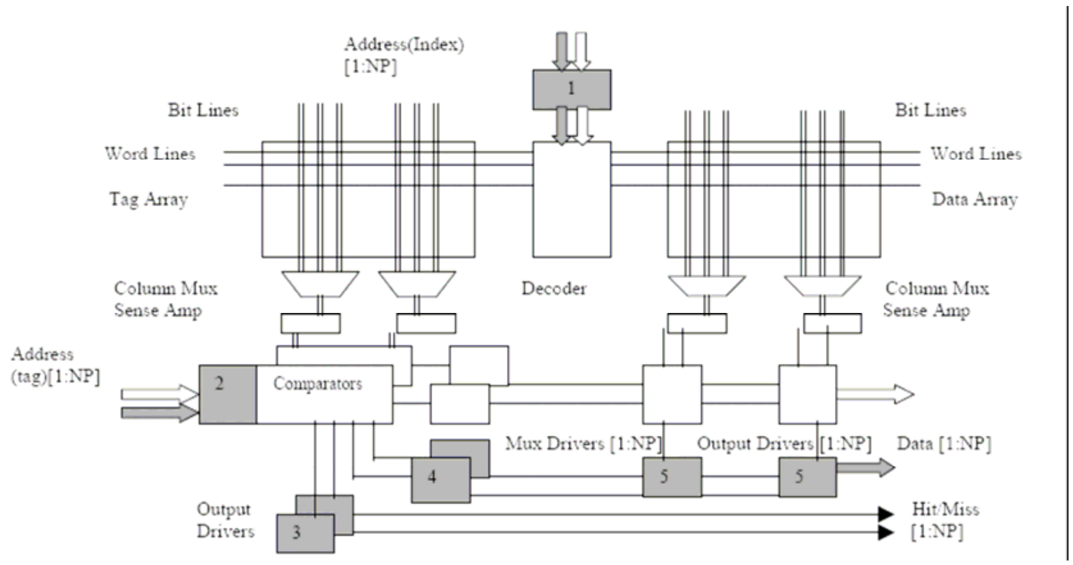


Figure 7.2. Additional Logic for Reconfigurable Cache [67]

cache in which the data array width is larger than the output width.), and export the selected data out of the cache.

In order to implement reconfigurability only a small amount of additional logic is required. The main difference between conventional and reconfigurable cache is that for the later case, the different partitions must be indexed by different addresses. As a result extra multiplexors are added for each of the above-mentioned structures to make sure that the correct address is forwarded to the selected partition. Additional wiring is also necessary from the cache to the processor for directing data to/from the various partitions. In Table 7.1 I identify the additional multiplexors with their functionality.

The additional logic will add to silicon area, access time and power consumed. Ranganathan *et al* [67] have studied the impact of reconfigurable cache organizations on cache access times and showed that for a small number of partitions, reconfigurable caches increase the cache access time by less than 5%. In a different study, Zhang *et al*

Table 7.1 Additional Multiplexors to Implement Reconfigurability

Multiplexors	Functionality
For address decoder	Correct address selection
For tag comparators	Route tag bits from correct address
For output drivers	Select correct hit/miss signals

have shown that the reconfigurable cache does not consume significantly additional power over traditional cache structures [82, 83]. In this chapter I have used the CACTI timing model to obtain values for these overheads of reconfigurability.

7.2.3 Reconfigurable Split Cache Organization

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 design an addressing scheme that can address any partition. Ranganathan *et al.* [67] have already proposed two partitioning and addressing schemes: “associativity based partitioning” and “overlapped wide-tag partitioning.” The former approach requires only minor changes to well-understood set associative cache organizations. However the key drawback with this scheme is that the number and granularity of the partitions are limited by the associativity of the cache. In my design I use “overlapped wide-tag partitioning” scheme. In this scheme, the key challenge is to devise a mechanism so the size of the array tag can be dynamically changed with the size of partitions (Since the number of bits in tag and index fields of the address will vary based on the size of the partition.). I 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 is needed. This register will be a part of processor state and be designed as any other control register.

A reconfigurable cache can be used in different ways. The best configuration for an application can be determined by extensive simulations (Or actual executions). Software profiling tools, used to identify portions of code that exhibit different cache behaviors, can also be used. Reconfiguration can also be implemented dynamically with appropriate hardware profiling and an automatic cache tuner. I advocate the use off-line analyses to find an optimum cache organization for each application (Or domain), and select that configuration when the application is scheduled. This reduces the hardware complexity resulting from dynamic adaptations. Other major issues in designing reconfigurable split caches include determining how to find the best configuration and maintaining data consistency. For detailed information about maintaining data consistency the reader is referred to [67].

7.3 Related Work

Ranganathan *et al.* [67] proposed reconfigurable unified data cache architecture for general-purpose processors. They proposed dividing cache into different partitions that could be used for different processor activities. Ranganathan *et al.* did not provide an analysis of silicon area involved in the reconfigurable cache, but explored different design alternatives, focusing on one option that of using the saved silicon area for "instruction reuse." I provided a detailed analysis of silicon area savings, reduction in execution cycles and power consumed when my reconfigurable cache structures are used. I also perform detailed analyses of achieving additional performance improvement by using saved silicon area for prefetching and branch prediction. I concentrate on embedded benchmarks for my evaluation. Albonesi *et al.* [5] proposed “selective cache ways” to selectively disable portions of unified data cache, trading off performance with power. In my analyses, in addition to trading off performance with power, I also explore how unused cache portions can be used for other purposes (Such as prefetch buffers.), providing further options in design trade-offs. Unlike mine neither of these analyzed the impact of reconfigurability on instruction cache or on L-2 caches.

Work by Zhang *et al.* [82, 83] is closely related to my research, as they evaluate reconfigurable unified data caches for embedded applications. Unlike the work by this research team, I do not see associativity as an important reconfigurable design parameter. This is because, both my array and scalar caches are designed as direct mapped caches, and I use victim caches to solve associativity for scalar data. Also inclusion of a very small instruction buffer allows to remove the cold misses in my direct mapped instruction cache. In addition to showing performance gains and power reductions, I also analyze silicon area savings obtained from my caches. Instead of shutting down cache area to save power, I also explore how the unused portion of cache area can be used for other architectural features (Other than as a conventional cache system.) that can improve applications’ performance. In a later work [29] they also analyzed the possibilities of reconfigurability in L-2 cache.

The most significant aspect of my work is using split data cache with reconfigurability. Previous research did not consider reconfigurable caches within the context of split data caches [56, 57, 69, 71, 78]. Although there are several works on prefetching [8, 52] this study is not about prefetching per se. In this study I am just implementing prefetching as one use of unused cache portions to improve performance.

7.4 Experimental Methods

The experimental environment is built on the Simplescalar (Version 3.0d) simulation tool set [13] modeling an out-of-order speculative processor with a two-level cache hierarchy. I rely on default parameters defined by Simplescalar as shown in Table 7.2. The base cache system, which is the cache with which I compare my designs, uses an 8 kilobytes L-1 instruction cache, an 8 kilobytes L-1 data cache, a 32 kilobytes L-2 instruction cache and a 32 kilobytes L-2 data cache. My performance evaluation includes silicon area needed for cache structures, because I am interested not only in the performance but also in better use of silicon area. I use CACTI [80] for computing silicon areas needed by caches. I use a modified CACTI timing model to obtain access time and power overheads incurred by reconfigurable caches.

In exploring optimal configuration, I varied only cache-size (Not line-size)—I start from smaller to larger sizes in order to avoid cache flushing (256 bytes to 8 kilobytes range.). Assuming array and scalar (Also instruction and data.) cache-size as different parameters I explored all possible combinations. At L-1, for each benchmark I

Table 7.2: Simulation Configurations of Simplescalar

Pipeline Parameters		Memory Parameters	
Issue Width	4	L-1Instruction Cache	8k, Direct-mapped
Functional Units	4 I-alu, 1 I-mul/div, 1 fp alu, 1 fp-mul/div	L-1 Data Cache	8k, Direct-mapped
RUU	8	L-2 Unified Cache	4 - Way, 32 K
LSQ	4	Line Size	32 bytes
Integer ALU	1 cycle	L-1 Hit Time	1 cycle
Integer Multiply	4 cycles	L-1 Miss Penalty	10 cycles
Integer Divide	20 cycles	Mem Latency/Delay	40/20 cycles
FP Multiply	4 cycles		
FP Divide	12 cycles		

Table 7.3: 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	2 kilobytes	2 kilobytes
qsort	256 bytes	1 kilobytes	4 kilobytes	2 kilobytes	32 kilobytes
dijkstra	1 kilobytes	512 bytes	4 kilobytes	4 kilobytes	8 kilobytes
blowfish	1 kilobytes	512 bytes	4 kilobytes	2 kilobytes	8 kilobytes
sha	256 bytes	512 bytes	1 kilobytes	1 kilobytes	8 kilobytes
rijndael	512 bytes	1 kilobytes	4 kilobytes	4 kilobytes	32 kilobytes
stringsearch	256 bytes	512 bytes	1 kilobytes	1 kilobytes	16 kilobytes
adpcm	256 bytes	1 kilobytes	512 bytes	1 kilobytes	4 kilobytes
CRC32	256 bytes	512 bytes	512 bytes	4 kilobytes	2 kilobytes
FFT	1 kilobytes	1 kilobytes	4 kilobytes	4 kilobytes	16 kilobytes

exhaustively explored all cache configurations to find the best configuration for optimizing power, area and access times. My L-1 augmented split data caches achieve excellent reductions in the number of cache misses, translating to fewer visits to L-2 caches. Since the number of accesses (Which is the number of misses from L-1 caches.) to L-2 caches is small, it is unnecessary to perform an exhaustive search of all possible L-2 cache configurations (As done for L-1 cache, see Figure 7.5 and Table 7.3.). I only explored configurations that reduce the silicon area needed for L-2 caches. I start with a very small L-2 cache, and continuously increase the sizes of the caches until no further reductions in misses are achieved (Compared to the base configuration of 32 kilobytes L-2 instruction and data caches.). Since both cache access times and the number of misses determine power consumption, this method allows me to find the smallest cache sufficient to meet performance requirements without increasing power consumption. In Table 7.3 I provide the optimum configurations for each benchmark.

7.5 Evaluation of Reconfigurable Split Data Caches

In following two sections I describe my empirical results with two different strategies. First I describe the improvement achieved with L-1 data caches while considering trade offs. In this case my goal is to optimize only a single parameter, while not decreasing others. Next I try to find the optimum configuration for each of my parameters, area, time and power consumptions. The results of the second strategy with both L-1 and L-2 data and instruction caches are described in section 7.5.2.

7.5.1 Results with Holding One Parameter Fixed

The standard way to evaluate the impact of several parameters is to vary one of the parameters while keeping the others fixed, which I follow in following two subsections.

7.5.1.1 Area

For some embedded applications size reduction is far more important than being faster or less power consuming. As a side-benefit, in many of these applications, reducing the footprint of processing resources can also lead to reduced power consumption. In this section I show how my cache design leads to substantial reduction in size (In terms of silicon area needed.). For this purpose I compare the areas consumed by my cache while achieving equal or fewer execution cycles as compared to an 8 kilobytes base cache. In other words, I fix the number of cycles to show how my design requires a smaller footprint. The first series in Figure 7.3 shows the percentage reduction in area needed by using my system instead of the base cache, while requiring no more execution cycles than the base case of 8 kilobytes unified data cache. From Figure 7.3 I can see that for half of the benchmarks, my system offers more than 80% reduction in silicon area.

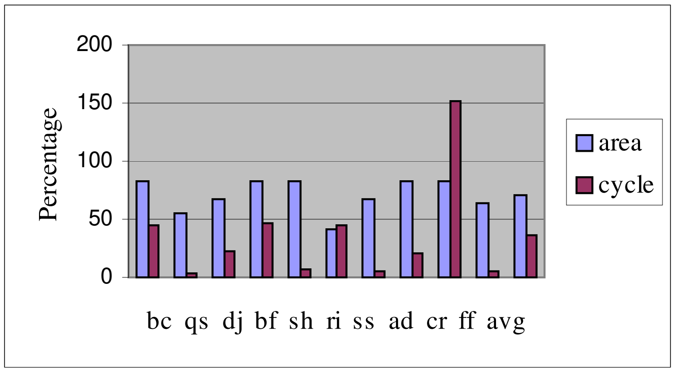


Figure 7.3: Percentage of Area and Cycle Reduction

In Figure 7.3 I also compare the number of cycles (Total execution time.) needed when using my cache system with that using unified data cache of equal size. By this I mean, if there is 75% reduction in area for my cache leading to a total L-1 size for scalar, victim and array portions, I allocate the same cache capacity for the unified data cache – thus keeping cache sizes equal in both designs. For some benchmarks (cr, bf, ri) we can see that there is a large increase in execution time for a unified data cache with smaller overall cache capacity (As compared to my split data caches.). For these benchmarks separation of data into array and scalar data significantly reduces the number of conflict cache miss. For other benchmarks (ss, ff) we can see that my cache shows only negligible reduction in execution time. This is easy to see (Also check Table 4.5.) since these applications have very small number of load/store instructions and thus any optimization to cache substructures has very minimal impact on the program execution. For these cases my cache structure can be reconfigured to gain other benefits including shutting off portions of caches to save energy, or utilizing unused portions of caches for purposes other than caching. I will explore these options in a later section.

7.5.1.2 Performance

Most modern embedded applications are demanding higher performance and added features. In such applications, faster execution of programs may be more important than reducing the footprint of the computing system. Such systems may afford larger caches, say 8 kilobytes or larger L-1 data caches. Here I will show how my design reduces the execution times while using equal (or smaller) amounts of area for caches. Note that my system uses 3 cache structures -- (Scalar cache, victim cache and array cache.). This presents more design choices in terms of selecting a size for each of these structures.

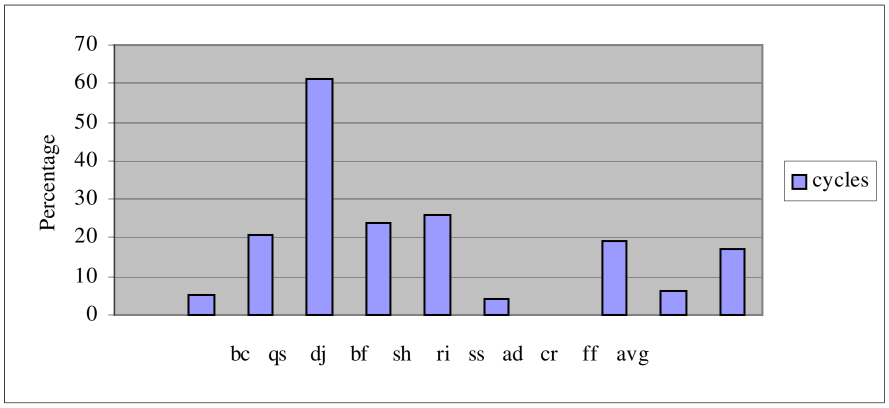


Figure 7.4: Percentage of Cycle Reduction without Increased Area

Optimal sizes for each of the cache structures are selected in order to obtain overall reduction in execution cycles while maintaining the same overall silicon area needed (As that of the base case using an 8 kilobytes unified data cache.). Sometimes the optimal selections of sizes for the different structures may lead to overall cache areas that are somewhat different from my target sizes. For example if I use 4 kilobytes scalar, 512 bytes victim cache and 2 kilobytes array cache, the total size is less than 8 kilobytes, but rounded to the nearest power of 2 (Which is 8 kilobytes.). For most cases the total area needed by my cache is smaller than the size of the base cache. In a few cases, I needed less than 512 additional bytes when compared to the 8 kilobytes base cache. Figure 7.4 shows the percentage improvement in execution times of applications assuming (approximately) equal numbers of bytes of cache for my designs and the base case. Those benchmarks that showed significant improvement in terms of silicon areas (Figure 7.3) also show reductions in execution cycles (Figure 7.4). The benchmarks, for which my design did not show reductions in area, do not show significant performance gains with my designs. Once again this should be expected since these benchmarks do not involve many memory accesses. For two benchmarks sh and ff, although the percentages of memory references are small (19 and 23%) large improvements are achieved by my cache. For these applications, the separation of data types into scalar and stream (Or array) is the main source of the performance gains. In Figure 7.4 I am also showing the average execution time gains across all the benchmarks used in my experiments.

7.5.2 Results with L-1 and L-2 Instruction and Data Caches

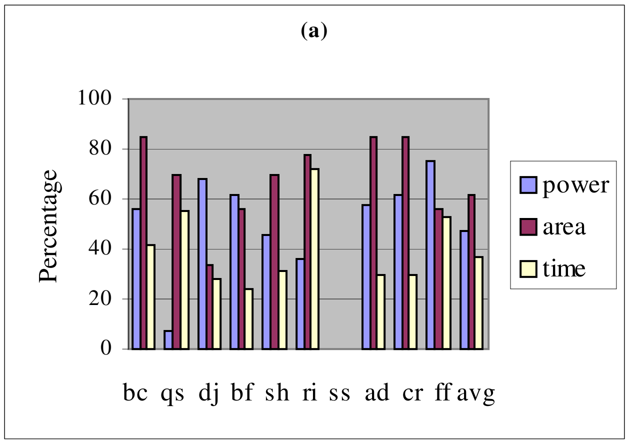
The most important concern for the designers of any embedded system is power consumption. My overall goal is the reduction of energy consumption by capitalizing on both the split cache organization and the reconfigurability of my cache structures.

Moreover the concomitant reductions in execution cycles and silicon areas, can further contribute to energy savings. In this section I will show the overall improvement in energy consumption by combining the efforts described in previous sections in both L-1 and L-2 data and instruction caches.

7.5.2.1 Results with L-1 Instruction and Data Caches

I believe the main problem with L-1 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, I use separate scalar and array caches, and augment direct mapped scalar caches with a small victim cache. In addition, with reconfigurability I permit varying the sizes of scalar and array caches for each application. I augment my L-1 instruction cache with a small buffer to permit for effective prefetching of instructions. Even with the additional power needed by the prefetching, I show significant reductions in total power consumed by all my caches (By 47% on average.).

The three series in Figure 7.5 represent percentage reductions in power, area and access times for L-1 instruction and data caches respectively. In order to obtain these results, I exhaustively searched for optimal cache sizes for each cache structure (Array, scalar, victim data caches and instruction cache.). In this figure I also show the average power, area and cache access time across all the benchmarks used in my experiments. As can be seen, for instruction cache, on average I achieve 47% reduction in power, 62% in area and 37% in cache access time. Here it should be mentioned that for benchmark **ss** the best configuration of instruction cache is 8 kilobytes.



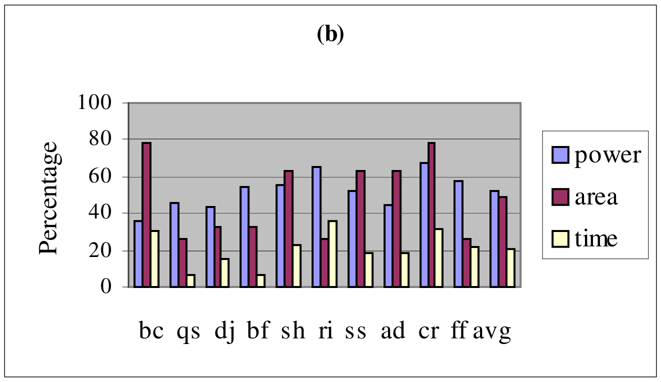


Figure 7.5: Percentage Reduction of Power, Area and Cache Access Time for L-1 (a) Instruction and (b) Data Caches

Hence for this benchmark I did not achieve any reduction in power or area. For data caches, on average I show more than 50% reduction in both power and area. For each of the benchmarks I also achieve reduction in cache access times.

7.5.2.2 Results with L-2 Instruction and Data 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. My L-1 reconfigurable split data caches achieve excellent reductions in the number of cache misses, translating to fewer visits to L-2 caches. For benchmark ff I was able to achieve as much as 96%

reductions in the number of misses. In Figure 7.6 I show the percentage reduction in the total number of access (Which is number of misses in L-1 caches.) in my L-2 caches

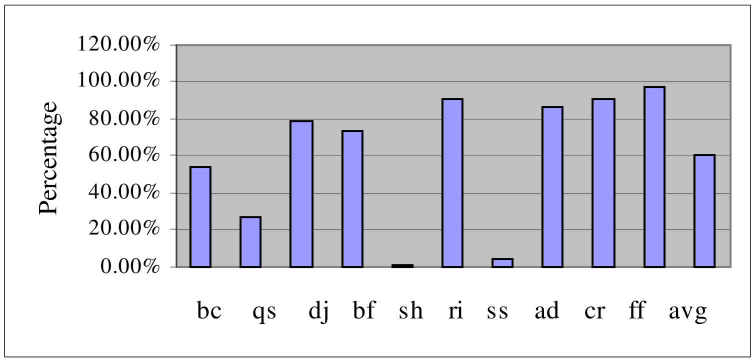


Figure 7.6: Percentage Reduction of Number of Access in L-2 Caches

(Both instruction and data caches.) compared to that of the L-2 caches of the base cache system. This implies that we can reduce the size of L-2 caches for applications to 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 (Which is number of misses from L-1 caches.) to L-2 caches is small I did not see a need for split L-2 data caches. Also

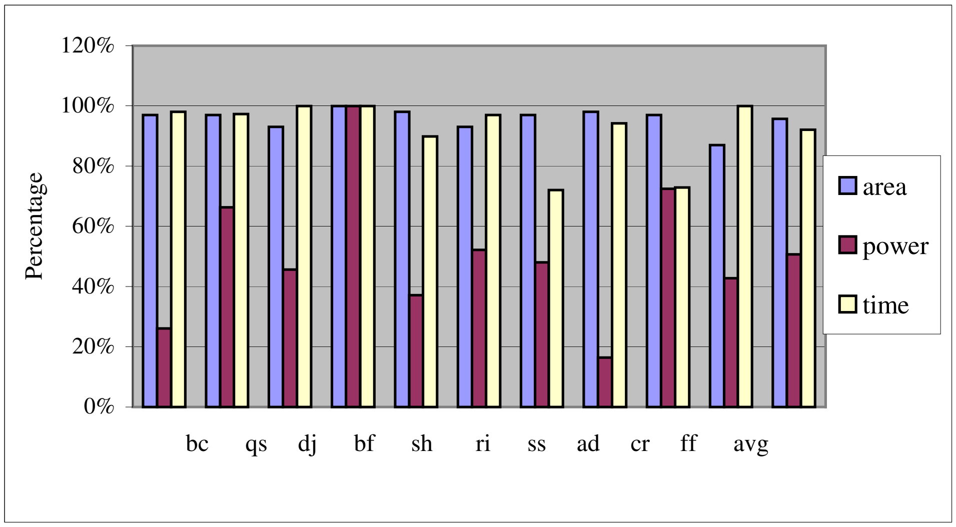


Figure 7.7(a): Percentage Reduction of Area, Power and Cache Access Time for L- 2 Instruction Caches

I felt that it was unnecessary to perform an exhaustive search of all possible L-2 cache configurations (As done for L-1 cache, see Figure 7.5 and Table 7.3.).

The three series in Figure 7.7 represent the percentage reductions in area, power and

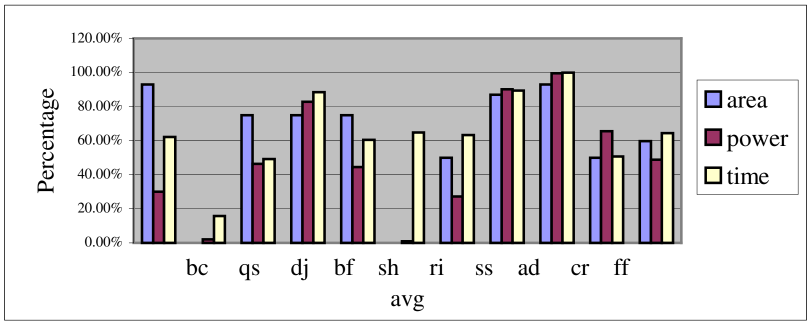


Figure 7.7(b): Percentage Reduction of Area, Power and Cache Access Time for L-2 Data Caches

access time for L-2 instruction (7.7a) and L-2 data (7.7b) caches respectively. In these figures I also show the average area, power and cache access time reductions across all

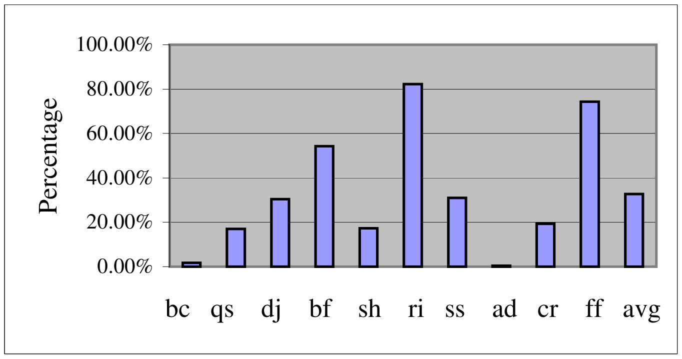


Figure 7.8: Percentage Reduction of Execution Cycles

the benchmarks used in my experiments. My 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 7.8 I compare the execution cycles of the selected benchmarks of my proposed cache systems (With optimal configurations for various L-1 and L-2 structures as outlined previously.) with that of base cache systems. It should be mentioned that here I am comparing total execution cycles for the application – not just memory access cycles.

7.6 Achieving Further Improvement with Reconfigurable Split Data Caches

From the results shown in section 7.5 we can make two observations. First my cache design will result in huge silicon area savings**.** Second, my designs also consume less power than conventional unified data caches. 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. The space savings resulting from my cache structures may be used for many architectural features to further improve the performance of embedded systems. On the other hand, if savings of static and dynamic power consumption is our main concern then we can simply shut down the additional saved area. In the following two sections I evaluated both options.

7.6.1 Utilization of the Unused Areas

In this section I propose my reconfigurable cache to enable its dynamic partitions to be assigned to processor activities other than conventional caching. 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 trace caches, branch prediction buffers.) and the achievable performance gains increase with the size of these tables [72]. Because of additional tables, these techniques are often viewed as inappropriate for embedded systems [84]. Since I show reductions in cache sizes needed for my designs (While not sacrificing performance or increasing power consumptions.), these savings may be used to implement look-up tables or buffers to implement elaborate branch prediction or instruction reuse ideas. To provide evidence of the benefits of reconfigurable caches, I first study one such technique, hardware prefetching, which is followed by explorations of other hardware optimization techniques.

7.6.1.1 Hardware and Software Prefetching

Prefetching or exploiting the overlap of processor computations with data access has proven to be effective in tolerating long memory latencies [8, 52]. 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 before 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. For these reasons prefetching is not used in embedded systems. However in this section I show that a reconfigurable split data organization with very small scalar and array caches can benefit significantly from prefetching. I used prefetching for both array data items and instruction cache at L-1 level. In my reconfigurable cache I can use separate partitions for prefetched data and avoid cache pollution. The prefetching areas can be implemented in cache arrays with minor hardware and software changes. Several software-profiling tools have been developed to identify portions of code that exhibit different cache behavior, to provide insights to programmers. Such tools can be used to implement smarter prefetching, especially for my scalar cache. For my array cache, a simple prefetching method one in which bringing blocks to the cache will also prefetch next two lines to the prefetch buffer, can be very useful. One can explore other prefetching and lookup techniques. For example checking for a line in the cache can also be accompanied by checking the tag for the line two blocks before. If it is a miss for the current line but a hit for the previous line, only then the buffer will be checked (The current line must be available in the buffer.). Other smarter look up variations can also be implemented to reduce cache access energy.

7.6.1.2 Hardware Optimization Techniques with Branch Prediction Tables

Modern processors utilize speculative execution of instructions based on branch prediction; instruction reuse and function reuse technique to improve performance [18, 76]. It has been found that many instructions and functions are repeatedly executed with the same inputs, producing same outputs [76]. Similarly for branch instructions, branch decisions are correlated and can be predicted. This observation can be exploited to reduce the number of instructions/functions executed dynamically as follows: by buffering the previous result of the instruction/function, future dynamic instances of the same static instruction (Or function) can use the result by establishing that the input operands in both cases are the same [76]. For all of these optimization techniques as the microprocessor tries to make the prediction based on a record of what this instruction/function has done previously, having a larger look up table is very helpful [72]. Unfortunately none of these optimization techniques have been studied in detail for embedded applications. I anticipate that since I can save the space needed for cache memories using my cache structures (On average 62% for instruction cache and 49% for data cache.), the saved space can be used to build needed look-up tables to implement instruction and function reuse in embedded systems.

In my study I 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 I 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.

7.6.2 Shut Down Portions of L-2 Instruction and Data Caches

Even if we do not use a portion of a cache it will still consume static power as leakage current. Hence in order to save both dynamic and static power, we should shut down the cache portion, which is not used for the running application.

7.7 Results after Achieving Further Improvement with Reconfigurable Split Data Caches

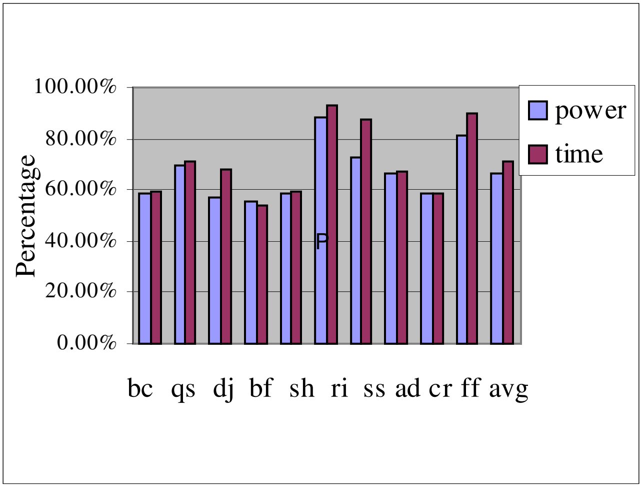
In following sections I show the improvements achieved by using my saved cache area for other processor activities including instruction and data prefetching, branch prediction buffers. I evaluate the potential benefits of such techniques for embedded applications. I also explore the energy savings if the unused cache partitions are shut down.

7.7.1 Results with Utilization of the Unused Areas

Results are provided in the same order as section 7.6.1.

7.7.1.1 Results with Hardware and Software Prefetching

Figure 7.9 shows the percentage improvement in power consumed and cache access time for applications using prefetching in for both L-1 instruction cache and L-1 data cache (Along with my scalar, victim and array caches.) when compared to the base cache



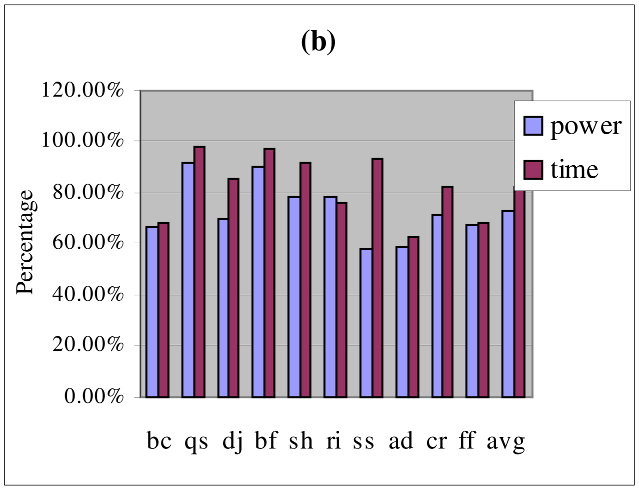


Figure 7.9: Percentage of Power and Access Time Reduction with Prefetching for (a) Instruction and (b) Data Caches

system. As can be seen, for all the benchmarks there are a significant reduction in cache access times and power consumption. While access to memory is hidden by prefetching, additional energy is consumed by prefetching. The data in Figure 7.9 accounts for the added power for prefetching. In Figure 7.10 I present the percentage improvement in terms of execution cycles of an application using prefetching (Along with my scalar, victim and array caches.) when compared to the base cache system. As we can see for benchmark ri I obtain as much as 85% reductions in the number of execution cycles. For two benchmarks, ad and bc, as the percentage of memory references are very low (7 and 11 % respectively.) prefetching did not show further improvements in execution cycles. The average reduction in execution cycles is 47%. Thus my data shows that my split data cache augmented by prefetching and victim cache can improve performance and reduce power consumption of embedded benchmarks when compared to a unified data cache.

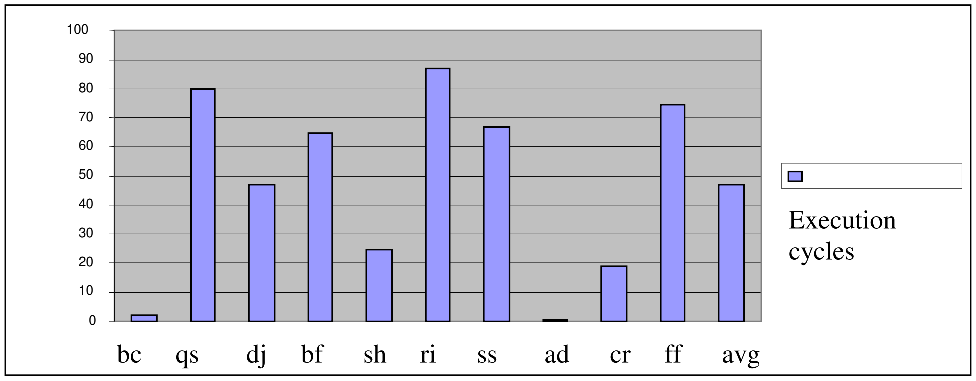


Figure 7.10: Percentage Reduction of Execution Cycle after Implementing Prefetching

7.7.1.2 Results with Hardware Optimization Techniques with Branch Prediction Tables

Figure 7.11 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** I achieved 75 % reduction in execution cycle; the average reduction across all applications is 47%.

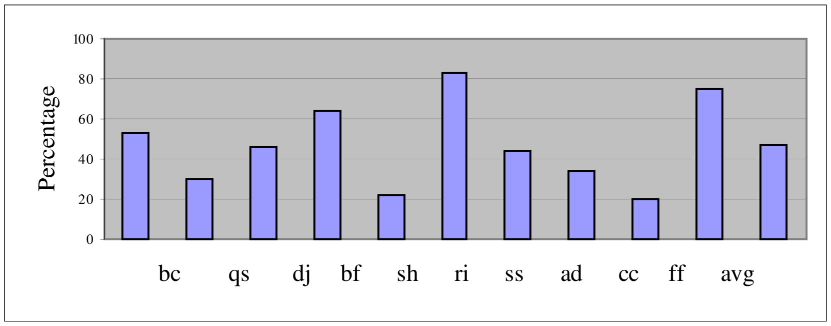


Figure 7.11: Percentage Reduction of Execution Cycle after Implementing Branch Prediction

7.7.2 Results with Shut Down of L-2 Instruction and Data Caches

The most important concern for the designers of any embedded system is power consumed by applications. As my proposed design for L-1 instruction and data caches

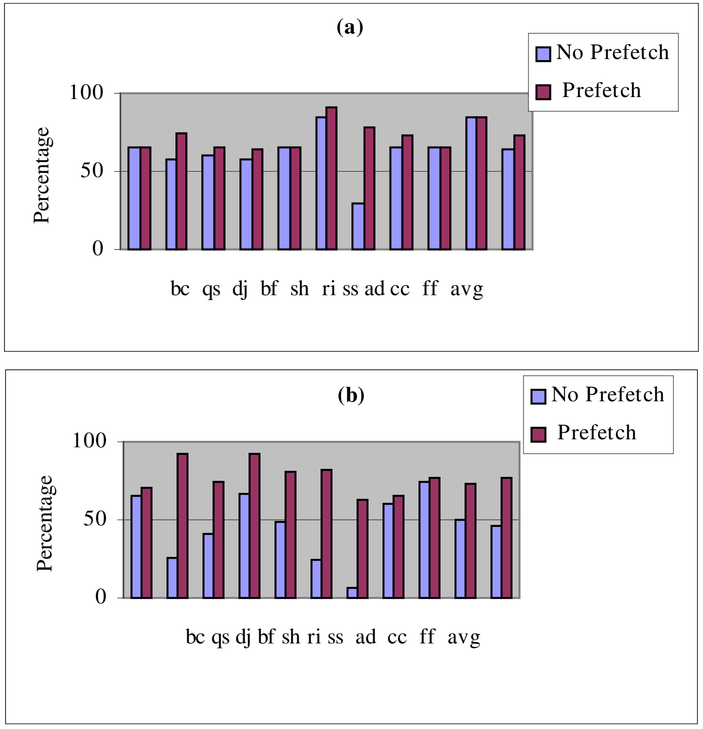


Figure 7.12: Percentage of Dynamic and Static Power Reduction without and with Prefetching for (a) Instruction and (b) Data Caches

result in reductions in the number of misses, translating into fewer accesses in L-2 caches, we may want to shut down unused portions of L-2 cache. In this section I explore the power savings from shutting down the unused portions. This requires to model both static and dynamic power consumed by cache memories. In previous sections I only accounted for dynamic power since all cache portions were still active (Not shut down.). In Figure 7.12 I show the percentage reduction in total power consumption (Both dynamic and static.) for (a) instruction and (b) data caches. In each figure I show the power reductions without and with prefetching. Here for prefetch buffer I am using area saved from L-I instruction and data caches. Here 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.

7.8 Conclusions

In this chapter I introduced novel cache architecture for embedded microprocessor platforms that explores reconfigurable L-1 and L-2 cache memories. At L-1, my cache configurations consist of an instruction cache with prefetching and split data cache with a scalar data cache augmented by a victim cache, and a separate array data cache. The size of these different units can be configured for each application to achieve optimum performance. In addition, at L-2, I can configure the sizes of instruction and (unified) data caches. My results show that I achieve significant reductions in the number of cache misses translating into reduced cache access times, reductions in silicon areas, power consumptions and finally reduction in the number of execution cycles.

I believe there are three reasons behind the success of my cache architecture. First, I used reconfigurability to tune the cache sizes and miss rates for a particular applicatio to reduce overall power consumption. Reconfigurability allows to implement most appropriate size for each of these different caches (For example split array and scalar data caches at L-1; L-1 instruction caches augmented by prefetch buffers, and shutting down L-2 instruction and/or data caches partially or completely.). Second, the separation of array and scalar data items eliminates mutual interference caused by these two types of data and reduces conflict misses. The third reason behind my success is the greater reconfigurable design space afforded by my cache structures allowing for more choices for improvement.

I also show that the saving in cache sizes resulting from my designs can be used for other processor activities including instruction and data prefetching, branch prediction buffers. I evaluate the potential benefits of such techniques for embedded applications. I also explore the energy savings if the unused cache partitions are shut down. Since my reconfigurable approach leverages the sub-array partitioning that is already present in modern caches, only minor changes to cache implementations are required. The reconfiguration only requires a small overhead in terms of silicon area, power and execution times.

CHAPTER 8

DATA FLATTENING: AN APPROACH FOR POINTER INTENSIVE APPLICATIONS

8.1 Introduction

So far, in my proposed cache design, I have been concerned with two data types: array and scalar data. In initial evaluation of my proposed split data cache design, I emphasized scientific applications that store data in array structures and do not support pointers. Many applications (Especially those written in languages such as C and C++ —for example, databases and operating systems.) use of pointer structures to store data extensively. I call such applications pointer intensive programs. Because of their dynamic nature and their reliance on heap allocated storage, pointer structures tend to have less regular access patterns than arrays. As a result, techniques developed to reduce and tolerate latency (That were created primarily for applications that manipulated data stored in arrays, including my split cache design.) do not perform as effectively for pointer-intensive programs. For this reason, I propose data flattening, a data placement method for pointer-intensive applications. I augment data flattening with my split cache design. My proposed split data cache organization allows me to retain a simple design to eliminate conflicts between data types (Scalar and array data, so far.). At the same time, the array cache of my split cache design can take advantage of the uniform regularity of array data through larger lines, stream buffers, and other types of prefetching. I want to utilize array cache even for heterogeneous structures like linked list. In order to achieve this, I convert linked list to linear array. By using data flattening, I hope to improve data cache performance by locating heap variables in contiguous locations in the virtual memory space (Such as an array.). This approach employs data profiling to characterize heap variable usage. Profile information then guides the variable placement solution that increases predicted cache line use and block prefetch. Finally, using array cache for these flattened data decreases inter-variable conflicts. This data flattening solution can be implemented by using a modified linker and customized dynamic allocation routines.

8.2 Proposed Design

Before I describe my data flattening techniques in detail, let me first identify the differences between array and pointer data, and different options for data placement.

8.2.1 Array vs. Pointers

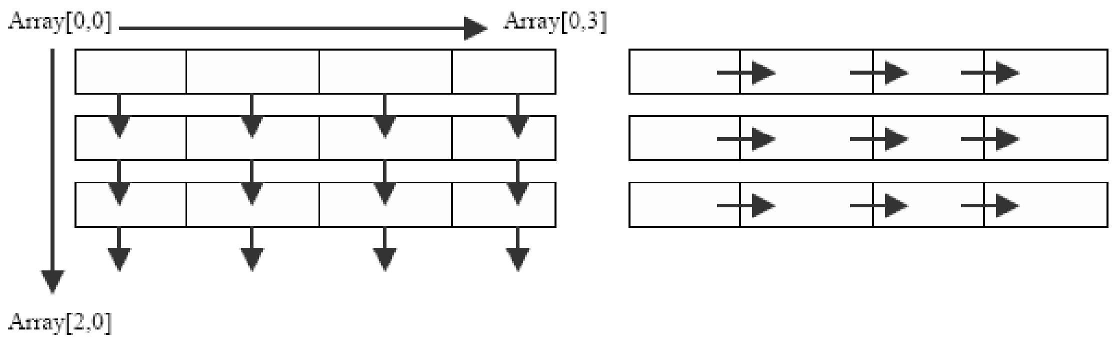
Pointer data structures such as lists and trees are widely used in large applications, including compilers, databases, and graphics applications (Systems that use linked object graphs and function tables.). Typically, all pointer data structures are allocated via dynamic memory management (e.g., malloc) and labeled as heap objects (I interchangeably call them heap or pointer data.). Pointer data are constructed by explicitly connecting data elements, where each data element contains fields that name adjacent elements by address. This connectivity permits easier construction and manipulation of data structures with arbitrary shape, such as trees and graphs. Unfortunately, this flexible, dynamic construction allows pointer data structures to grow to large, irregular foot-prints, which is not only difficult to cache but also noncondusive to prefetching. On the other hand, compact, uniform, contiguous data patterns makes arrays tailor-made candidates for today's prefetching methods, which rely on address stream regularities to extract arithmetic patterns that can be used to predict.

Conventional prefetching techniques analyze the address history associated with an instruction or datum and exploit regularity in the stream to produce prefetching addresses. Because address sequences corresponding to sequential array traversal exhibit arithmetic regularity, we can easily compress address sequences to a pair of numbers: a base value and a stride. These two numbers can be used as a formula to generate previously unseen addresses that closely match actual program accesses. Using a mathematical formula makes the address computation so fast that, finding enough work to overlap the latency is no longer an issue. Historically, however, prefetch mechanisms have had trouble with pointer data structures [20]. Pointer intensive applications generally lack the address stream regularities which allow extracting arithmetic patterns that can be used to make predictions. At the same time, accesses to successive pointer data elements cannot be overlapped, as the process of address generation itself requires an inherently serial evaluation through memory. This condition, which effectively exposes the full latency of each pointer datum access, is known as the pointer chasing problem.

Another major difference between array and pointer data structures is their creation time. An object is assigned an address when it is created. For global variables such as array data, addresses are assigned at compile time, typically when the program is linked. For pointer data variables, however, addresses are assigned at run time when dynamic storage is allocated. The address assigned to a data object affects its location in the data cache, as it is determined by this object's address modulo the number of cache blocks. Consequently, data placement offers a mechanism to control both the cache block's contents and location within the cache. With data placement to control the contents and location, we can influence the data cache's performance [15].

8.2.2 Different Data Placement Techniques

Programs locality of reference can be improved either by changing a program's data access pattern or its data organization and layout [53]. Hence, cache performance depends on two factors: the time data items are accessed and their location in the address space. Usually, cache performance optimization techniques rely on one of two options: either they restructure the computation or they restructure the data layout. Implementing restructuring computation assumes that, given a fixed data layout, we wish to manipulate the ordering of accesses in such a manner that multiple accesses to the same data item (Or cache line.) occur closely in time, thereby enhancing locality. A common example of a restructuring-computation approach is manipulation of dense matrices in scientific programs. Figure 8.1 illustrates this restructuring computation with an example. If we can interchange the loops (As shown in the code snippet on the right.), the resultant data reference pattern will step through all array elements in a cache block before accessing the next block, reducing the number of potential cache misses by a factor of four. Other examples of "restructuring computation" include loop tiling, loop unrolling [16, 81].



```
for i = 0 to n do
   for j = 0 to m do
    ...
    ... = Array(j, i)
    ...
   done
done
```

```
for i = 0 to m do
    for j = 0 to n do
    ...
        ... = Array(j, i)
    ...
    done
done
```

Figure 8.1 An Example of “Restructuring Computation” Approach [21]

In contrast, data placement assumes that, given a set of data items which are accessed closely in time in the original computation, we wish to actively arrange them in the address space such that we create spatial locality by allocating them contiguous addresses. This allocation enhances the effectiveness of long cache lines and simplifies prefetch address generation. We also avoid cache conflicts because data placement ensures that conflicting data do not reside in separate lines which map into the same cache sets [53]. An example of a data placement technique is “list linearization,” an old technique for compacting lists in Lisp programs [19, 58]. List linearization relocates a linked list’s nodes so that they reside in contiguous memory locations. Other examples of “data placement” include coloring, copying, and clustering [12, 20, 21, 23].

While both “restructuring computation” and “data placement” approaches have received considerable attention in the past, my focus in this study is on facilitating data placement optimizations. There are two approaches to implement data placement method. Data relocation (Or simply relocation.), the first approach, moves the heap data (Perhaps more than once.) after it has been allocated. Although relocation, can adapt to dynamic program behavior, it will need complicated hardware modifications. List Linearization is an example of relocation-based optimizations. In the second approach, Static placement, we could assign the optimized address during allocation (When the heap data is created.) [15]. The advantage of static placement is its simplicity. Data flattening, the placement technique proposed by me is a static allocation method.

8.2.3 Data Flattening Method

Data flattening is a profile-based memory allocation technique that compresses sparsely scattered pointer data into regular contiguous memory locations. To apply the approach, I first profile a program to characterize how the program uses data. The profile information then guides the placement of heap variables at run time using customized allocation routines to increase cache line utilization and block prefetch.

As pointer data are scattered sparsely throughout the address space, a straightforward method of actively improving spatial locality would be to take data items that are accessed closely in time and pack them into adjacent memory locations. This form of data placement makes cache lines more effective and potentially reduces the number of capacity (Through compaction.) and compulsory (Through prefetching.) misses. This method also reduces space consumption, alleviates the pointer-chasing problem, and makes a case for array cache for pointer data. Figure 8.2 shows an example of my data flattening method. Without flattening the four nodes, RED, BLUE, PINK and GREEN are scattered throughout memory (See Figure 8.2(a).) As a result, they reside in four separate cache lines (Assuming each cache block contains two nodes.).

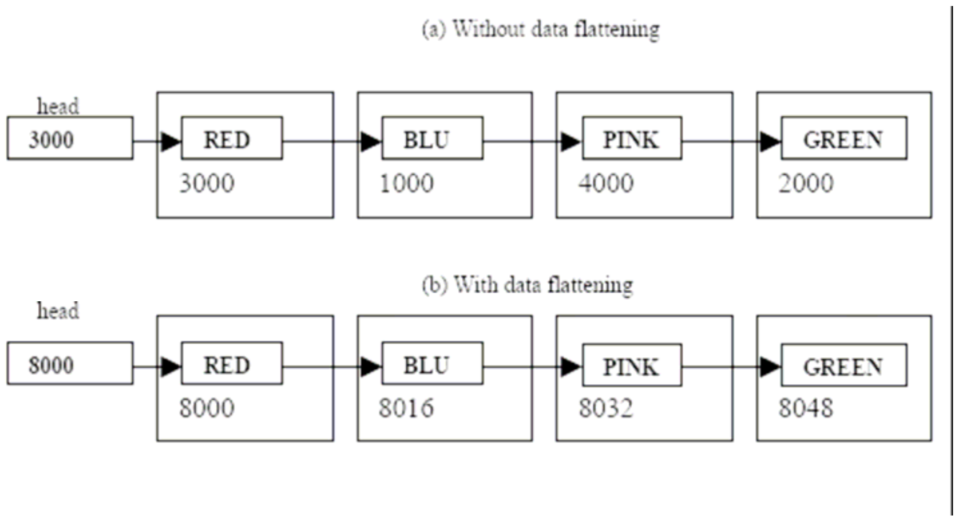


Figure 8.2: Locations of Different Nodes in Memory of a List without Data Flattening (a) and with Data Flattening (b)

When flattening is used (Figure 8.2 (b)), however, the four nodes will be allocated to a contiguous memory region. As a result, rather than occupying four separate lines, the nodes occupy only two lines. Hence data flattening not only eliminates as much as half of the space consumption but also potentially eliminates half of the cache misses this list causes as it is continuously revisited. At the same time, I reduce pointer chasing problem, the major concern with Pointer data. Consider the same figure (8.2) with regard to pointer chasing. Assume that we wish to hide the entire miss latency and, to do so, we need to prefetch three nodes ahead. If so, we want to prefetch node GREEN as soon as we arrive at node RED. However, with prefetching, we do not know the address of node GREEN until we have dereferenced nodes RED, BLUE, and PINK. In contrast, with data flattening, we have linearized the list, and we can prefetch node GREEN at node RED by prefetching the next cache line. Finally, my biggest achievement is the effect of my proposed split data caches for pointer-intensive applications. Because these data are now processed through data flattening into more uniform patterns (No longer scattered.) the data are now tailor made for my array cache.

However, compressing a pointer data access stream can be a difficult task. Linear layout of pointer data is usually the result of allocator strategy, compacting garbage collection, or careful hand optimization and this linear layout is often compromised as the data structure evolves [15]. In my simulation framework, I have implemented a simple approach. I assume that all pointer data elements of equal size belong to a single data structure. I first profile a program to be optimized to gather information characterizing its data usage, including total number of equally sized nodes. For example, let me assume that through profiling I found that 64 equally (16 bytes) sized nodes have been allocated. In the original run, when the first request for a node of 16 bytes occurs, instead of allocating 16 bytes, I allocate 1 kilobytes as a free list and mark the byte for this particular data structure. The first 16 bytes determine the data segment's new starting location. During this custom allocation, any later request for a 16 bytes node will be fulfilled from this corresponding free list (Chosen by tag.) so that the data flattening is performed.

In real environments, this procedure becomes more complicated. Once the profiling data are generated, they are fed back into the compiler/linker for data-placement optimization. Heap-allocation placement is implemented at run-time using a customized malloc routine. The modified malloc first computes the heap-allocation name. The data placement framework requires that profile information collected in one program run be used to direct variable placement in subsequent runs [15]. To implement this binding, profile and placement tools must assign names to all variables. Generating names for heap variables presents a more challenging task, as heap variable addresses can change with different inputs to the program. In reality, we can model previously proposed heap allocators that map objects of similar sizes to the same pages of memory during allocation [73]. The difference is, we use data placement to guide heap objects into particular address spaces, along with the elements of same data structure (list or tree). In my data flattening custom allocator, there will be several free lists. Each list will have an associated tag corresponding to a particular data structure. Heap objects belonging to the same data structure (allocation locality) will be assigned the same allocation tag. Objects with the same tag will use the same free list for allocation and benefit from potentially being allocated close to one another.

8.3 Related Work

This work totally differs from the work of Carr *et al.* [16] and Wolf *et al.* [81]. First of all, instead of reconstructing computation, I focus on facilitating data layout optimization. Also both of these work focused on regular array accesses, whereas my work considers an entirely different class of data structures. Pointer based structures do not support random access, and hence changing a program’s access pattern is impossible in general.

My work relates with the work of Seidl and Zorn [73] and Calder *et al.* [15] as we all implement an allocation strategy, based on a history of the previously allocated object obtained by profiling. However Seidl's [73] studies focused on a program's paging behavior, not its cache behavior. My work differs, not only because of the vast difference in cost between a cache miss and a page fault, but also because cache blocks are far smaller than memory pages. On the other hand, although Calder *et al.* [15] focused on cache, they did not emphasize on only heap objects. Their technique, shows little improvement for heap objects but significant gains for stack objects and globals. By contrast, I provide tools for cache-conscious heap layout that produce significant improvement.

Unlike my work, Chilimbi's first placement optimization technique which implements structure splitting is designed only for tree data structures [21, 22]. In addition, they used an entirely different allocation strategy, based on programmer-supplied hint to co-locate objects, rather than using profile data gathered by running a training program. Whereas like this work, their second approach [20] relies on profile information to guide heuristic algorithms in placing instructions to minimize instruction cache conflicts, and maximize cache line utilization and block prefetch. However their design is for instruction cache. Also their work relies on properties of object-oriented programs and requires copying garbage collection, whereas my study focuses on C programs.

The major difference between my work and that of Luk and Mowry [52], is the timing as I implement data flattening during alloction. Unlike my work, their schemes incur serious overheads in the form of runtime storage and additional code needed to maintain history pointers and linear data layout, respectively.

The major way my proposed cache organization differs from all of these above mentioned work that they used placement optimizations to improve the performance of a unified data cache. These works complement this one, as they are also concerned with improving the cache performance of a data structure by reorganizing its internal layout, while the orthogonal techniques in my study improve performance by arranging collections of flattened data in separate array and scalar data caches.

8.4 Experimental Methods

In my experimental evaluations of the data flattening I use select three benchmarks from the Olden pointer-intensive benchmark suite [19], three from Standard Performance Evaluation Corporation (SPEC) integer benchmark suit [32], one from memory pointer benchmark suit [70] and one from Mi- bench suit [30]. A summary of the benchmarks is shown in section 4.3.3.

For the purpose of studying the performance implications of data flattening I extend Simplescalar (version 3.0d) simulation tool set [13] modeling an out-of-order speculative processor. I rely on default parameters defined by Simplescalar as shown in table 8.1. The base cache system, which is the cache with which I compare my designs, uses an 8 kilobytes Level one (L-1) instruction cache, an 8 kilobytes L-1 data cache, a 32 kilobytes Level two (L-2) instruction cache and a 32 kilobytes L-2 data cache. My performance evaluation for Data Flattening includes performance in execution cycles, cache access time and cache power consumption. I use CACTI [80] for computing cache access time

Table 8.1: Simulation Configurations of Simplescalar

Pipeline Parameters		Memory Parameters	
Issue Width	4	L-1 Data Cache with DF	8k
Functional Units	4 I-alu, 1 I-mul/div, 1 fp alu, 1 fp-mul/div	L-1 Data Cache block size	64k
RUU	8	L-1 Data Cache block size	Direct maped
LSQ	4	Array/Scalar Cache with DF	4k
Integer ALU	1 cycle	Array/Scalar Cache block size	32k
Integer Multiply	4 cycles	Array/Scalar Cache block size	Direct maped
Integer Divide	20 cycles	L-1 Hit Time	1 cycle
FP Multiply	4 cycles	L-1 Miss Penalty	10 cycles
FP Divide	12 cycles	Mem Latency/Delay	40/20 cycles

and cache power consumption. Since I did not evaluate the area consumption, I set total cache area (Including array and scalar caches.) equal to that of base cache. Table 8.1 also lists the various architectural parameters for each cache configuration used in my studies.

8.5 Empirical Results with Data Flattening

In this chapter I present results of my experiments with "data flattening" and "cache splitting." As the base cache I use conventional 8 kilobytes unified data cache. When collecting data, to really show the impact of array caches for data flattening, I do the comparison between the followings,

1. Performance of original code
2. After data flattening, performance using single data cache
3. After data flattening performance using split data caches

So I want to show how much the performance improves because of better localities (Compare 1 and 2.), and by how much split data caches improve performance (Compare 1 and 3 and 2 and 3.).

My evaluation compares the cache designs for three metrics: the cache access time, the power consumption and the performance (execution cycles). I demonstrate that my proposed "split" cache with flattened data gains improvements for all three metrics when compared to other approaches. The results support my view that a complete separation of array and scalar data caches allow array cache to play a key role in boosting cache performance for flattened data for the pointer intensive applications.

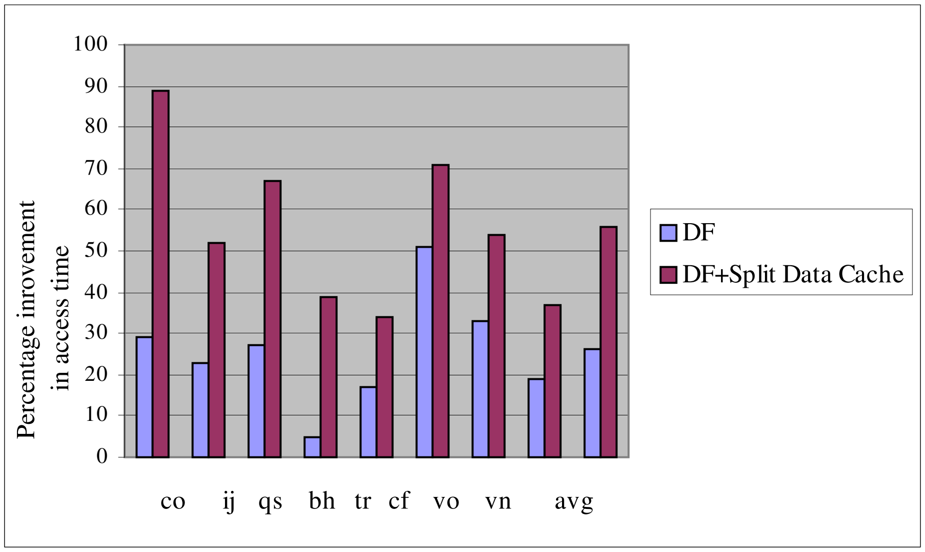


Figure 8.3: Percentage Improvement in Cache Access Time

The percentage improvements in terms of cache access time, power consumption and execution time achieved by my "data flattening" approach are plotted in Figure 8.3, 8.4 and 8.5 respectively. In these figures I show the relative advantage of using separate array and scalar caches with data flattening method. In the first series I compare the results

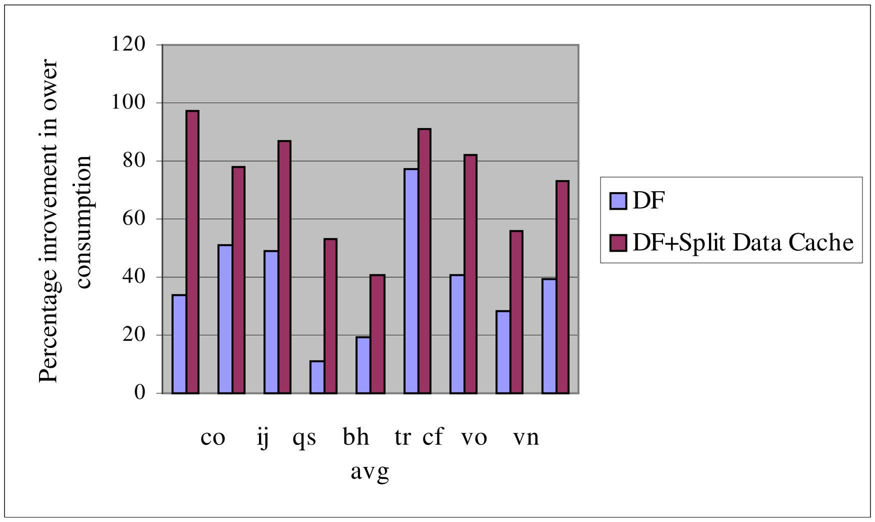


Figure 8.4: Percentage Improvement in Cache Power Consumption

obtained with original code (without data flattening) using unified data cache, with that of data flattened code using unified data cache. Whereas in the second series I compare the results obtained original code (without data flattening) using unified data cache, with that of data flattened code using split data caches. In each figure, I also include the average for all the selected benchmarks. From these figures I can see that the split data cache organization with flattened data has led to a significant reduction in access time, power consumption and overall performance.

From these figures, we can make another observation. Although data flattening itself produce reduction in access time, power consumption and number of execution cycles; by sending these flattened data to my split data caches we an achieve additional significant improvement. For benchmarks co, ij, bh and vn, the reason behind not achieving significant improvement by using data flattening itself (Not using split data caches.) is total percentage of pointer load is very low. For bh and vn, the second reason is the data allocation pattern itself. As, both of these benchmarks create their pointer-based structures at program start up and do not subsequently modify them, there is not

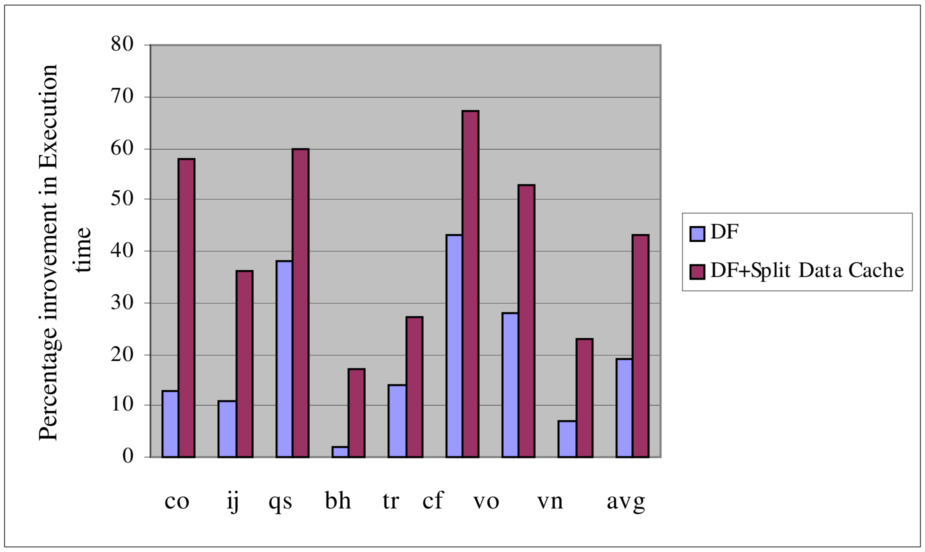


Figure 8.5: Percentage Reduction of Execution Cycles

enough opportunity left for my “data flattening” method. Although it improves performance (Figure 8.5) the gain is only 10-15% because pointer structure elements are created in dominant traversal order, which produces a natural cache-conscious layout (like array). For the benchmark tr and vo, although the percentage of pointer load is moderate, there is another reason for “data flattening” to hold back from its optimum effect. For both of these benchmarks, the sizes of pointer data nodes are very big. Benchmarks cf and qs with large number of small sized pointer data performed very good with data flattening alone. However my split data caches with flattened data provide excellent improvement for all of the benchmarks; on average 56%, 73% and 43%

reduction in access time, power consumption and performance respectively, when compared with original code (without data flattening) using unified data cache.

8.6 Conclusions

In this chapter, a cache-conscious data placement optimization, called "data flattening," is introduced to improve data cache performance of pointer intensive applications. The idea behind "data flattening" is to allocate the nodes of a pointer data linked structure (like list or tree) to reside in compressed contiguous memory locations. In this chapter I showed that this compression brings a threefold impact. First of all, converting pointer data to array through data flattening will not only make pointer data a suitable candidate for my array cache, it will not create conflicts (As they are not scattered sparsely throughout the address space anymore.). Large, irregular foot-print of pointer data stream makes them not only very difficult to cache, but also difficult to prefetch. As compression will definitely bring a huge reduction in the footprint of pointer data stream and will make them more cachable. Finally compression will also bring eased in the prefetching. Ordinarily, a prefetch address for a pointer data element cannot be generated until the addresses of all *previous* elements in the structure are known. Compression is attractive because it allows for generation of prefetch addresses for arbitrary pointer data elements without the need for a serial evaluation.

CHAPTER 9

CONCLUSIONS

In today's microprocessors, cache has become a vital element in improving performance for a wide range of applications. The central theme of this dissertation is to address the deficiencies in existing cache memory systems. I have proposed a cache organization that can significantly reduce the power consumed as well as the cache size while providing even better performance (By reducing execution time.) when compared applications executing with conventional direct-mapped or set associative caches. I have demonstrated that even very small data caches, when split to serve data streams exhibiting temporal or spatial localities, and augmented with smarter techniques like prefetching, victim caching, data flattening and reconfigurability, can improve performance of wide-ranging applications, without consuming excessive silicon real estate or power.

Existing cache organization suffers from the inability to distinguish different types of localities, and non-selectively cache all data rather than making any attempt to take special advantage of the locality type. This causes unnecessary movement of data among the levels of the memory hierarchy, cache pollution and unnecessary increases in miss ratio, memory access time and memory bandwidth. I have proposed a split cache architecture that groups memory accesses as scalar or array references according to their inherent locality and subsequently maps each group to a dedicated cache partition. In this system, because scalar references and streamed references no longer negatively affect each other, cache interference, thrashing and pollution problems have been diminished, delivering better performance. Further improvements have been achieved by the introduction of victim cache, prefetching, data flattening and reconfigurability to tune the array and scalar caches for specific application.

Unfortunately many traditional approaches (Like prefetching and victim caching.), which have been proved very efficient for desktop systems, do not work well for embedded systems. One of the achievements of my work is the transformation of prefetching and victim caching into promising techniques for embedded systems. I have demonstrated that a split data organization with very small scalar and array caches can solve the deficiencies of victim caches and prefetching in embedded systems, and benefit significantly.

One of the most significant contribution of this work is the introduction of a novel cache architecture for embedded microprocessor platforms. My proposed cache architecture uses reconfigurability coupled with split data caches to reduce (silicon) area and dynamic power consumed by cache memories while retaining performance gains. When using my augmented split caches for embedded applications, my results show excellent reductions in both memory size and memory access times, translating into reduced power consumption. Studies have shown that on-chip caches are responsible for 50% of an embedded processor's total power dissipation and, thus, savings in cache memory power can be significant in overall power savings. Since there was a huge reduction in miss rates at Level one (L-1) (Which is the number of times one needs to access Level two (L-2) cache.) further power reduction is achieved by partially or completely shutting down L-2 data or L-2 instruction caches. I also showed that the saving in cache sizes resulting from my designs can be used for other processor activities including instruction and data prefetching, branch prediction buffers and evaluated the potential benefits of such techniques for embedded applications. The energy savings from the unused L-1 cache partitions being shut down is also addressed in my research.

I also have explored data flattening for *pointer-intensive applications* and augmented this method with my split cache design. Data flattening which is a profile based memory allocation technique to compress sparsely scattered pointer data into regular contiguous memory locations (Like array data.). I have explored the potential of Spit cache organization for data placement with data flattening method.

The biggest advantage of my proposed cache system is its efficiency over a wide ranged of applications. Because this system is equipped with a special array cache, it is obviously beneficial for scientific applications and showed significant improvement with Standard Performance Evaluation Corporation (SPEC) floating point benchmarks. My data flattening work with pointer intensive Olden benchmarks and SPEC integer benchmarks have shown significant performance improvement. However, the most benefited applications are those in the embedded arena. Challenges to design of processing elements for embedded systems prove more stringent (Than those for desktop processors.), as they often must meet strict timing constraints and be designed to function within limited resources such as memory size, available power, and allowable weight. My proposed integrated method has met this challenge by permitting a systematic trade-off between memory size, power and performance, which has up to now not been feasible for embedded systems.

As performance metrics to evaluate cache performance improvement I include all possible metrics: miss rate, cache access time, silicon real-estate area, and power consumption. In order to demonstrate overall, whole system performance improvement, I use execution cycles.

Since my research displays a new path in increasing cache performance, which is expected to be an important limitation in future, I believe that my split cache architecture will find its way into the future microprocessors and the multiprocessor/multicomputer systems.

In future work, I plan to explore my proposed cache organization for multithreaded environment. I also want to work with intelligent memory management. I will explore integrating my data flattening with memory management such that an intelligent management can dynamically profile applications and flatten pointer data types to improve localities and utilize reconfigurable caches effectively. To fully benefit from split and reconfigurable caches, architecture must be modified to differentiate between scalar and array caches, and compilers must generate appropriate instructions to access either the scalar or the array cache. Compilers must also generate needed cache configuration parameters. I will explore developing such compiler augmentations.

REFERENCES

[1] A. Agarwal, J. Hennessy and M. Horowitz, Cache performance of operating systems and multiprogramming, ACM Transactions on Computer Systems, Volume 6 Issue 4 (Nov. 1988), 393-431.

[2] A. Agarwal and S. D. Pudar, Column–associative caches: a technique for reducing the miss rate of direct–mapped caches, in Proceedings 20th Annual International Symposiumon Computer, Jun. 1993, pp. 179-190

[3] A. Agarwal, H. Li, and K. Roy, DRG-Cache: A Data Retention Gated-Ground Cache for Low Power, Design Automation Conference, Jun. 2002, pp. 473-478.

[4] G. Albera, R. I. Bahar, Power/Performance Advantages of a Victim Buffer in High-Performance Processors, IEEE Volta International Workshop on Low Power Design, Mar. 1999, pp. 43-51.

[5] H. Albonesi, Selective Cache Ways: On-Demand Cache Resource Allocation, Journal of Instruction Level Parallelism, Volume 2 (May 2000) 141-152.

[6] ARM Microprocessor Brief Datasheet, April 2000.

[7] J. Arul, K. M. Kavi and S. Hanief, Cache Performance of Scheduled Dataflow Architecture, Proc. of the 4th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP2000), Dec. 2000, pp. 834-846.

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

[9] R. I. Bahar, D. Grunwald, B. Calder, A Comparison of software code reordering and victim buffers, ACM SIGARCH Computer Architecture News, Volume 27 Issue 1 (Mar. 1999) 51-54.

[10] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu and S. Dwarkadas, Memory hierarchy reconfiguration for energy and performance in general-purpose processor architecture, 33rd International Symposium on Microarchitecture, Dec. 2000, pp. 245-257.

[11] J. Banerjee, W. Kim, and J. F. Garza, Clustering a DAG for CAD databases, IEEE Transactions on Software Engineering, Volume 14 Issue 11(Nov 1988), 1684–1699.

[12] V. Benzaken and C. Delobel, Enhancing performance in a persistent object store: Clustering strategies in O2, in Technical Report 50-90, 1990.

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

[14] B. Calder, D. Grunwald and J. Emer, Predictive sequential associative cache, in Proceedings of the Second IEEE Symposium on High-Performance Computer Architecture, Feb. 1996, pp. 244-253.

[15] B. Calder, C. Krintz, S. John, and T. Austin, Cache-conscious data placement, ASPLOS-VIII, Oct. 1998, pp. 139-149.

[16] S. Carr, K. S. McKinley, and C.W. Tseng, Compiler optimizations for improving data locality, in Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI), Oct. 1994, pp. 252–262.

[17] J. H. Change, H. Chao, and K. So, Cache design of a sub-micron CMOS Systerd370, in Proceedings of the 14th Annual International Symposium on Computer Architecture, Jun. 1987, pp. 208-213.

[18] P. Chen, K. Kavi and R. Akl, Performance enhancement by eliminating redundant function execution, in Proceedings of the IEEE 39th Annual Simulation Conference, Apr. 2006, pp 143-150.

[19] C. J. Cheney, A nonrecursive list compacting algorithm, Communications of the ACM, Volume 13 Issue 11 (Nov. 1970), 677-678.

[20] T. M. Chilimbi, and J. R. Larus, Using generational garbage collection to implement cache-conscious data placement, in Proceedings of the 1998 International Symposium on Memory Management, Oct. 1998, pp. 37-48.

[21] T. M. Chilimbi, B. Davidson, and J. R. Larus, Cache-conscious structure definition, in Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, May 1999, pp. 13-24.

[22] T. M. Chilimbi, J. R. Larus, and M. D. Hill, Tools for cache-conscious data structures, PLDI, May 1999, pp. 51-68.

[23] D. Comer, The ubiquitous B-tree, ACM Computing Surveys, Volume 11 Issue 2 (Jun 1979), 121–137.

[24] S. Cotterell and F. Vahid, Synthesis of customized loop caches for core-based embedded systems, International Conference on Computer Aided Design (ICCAD), Nov. 2002, pp. 655-662.

[25] R. Espasa and M. Valero, A Victim Cache for Vector Registers, ICS-11. ACM International Conference on Supercomputing, Jul. 1997, pp. 293-300.

[26] A. Eustance and A. Srivastava, ATOM: A flexible interface for building high performance program analysis tools, Western Research Laboratory, TN-44, 1994.

[27] T. Givargis, J. Henkel and F. Vahid, Interface and cache power exploration for core-based embedded system design, in International Conference on Computer-Aided Design (ICCAD), Nov. 1999, pp. 270-273.

[28] C. Gonzalez, A. Aliagas and M. Valero, Data cache with multiple caching strategies tuned to different types of locality, in Proceedings of International Conference on Supercomputing, Jul. 1995, pp. 338-347.

[29] 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), Feb. 2004, pp. 208-213.

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

[31] J. L. Hennessy and D. A. Patterson, Computer architecture a quantitative approach, Morgan Kaufmann Publishers, Third Edition 2003, pp 423-430.

[32] L. Henning, SPEC CPU2000: Measuring CPU performance in the new millennium, IEEE Computer, Volume 33 Issue 7 (Jul. 2000), 28-35.

[33] D. Hormdee, J. D. Garside, S. B. Furber, An Asynchronous Victim Cache, in Proceedings of DSD, Sep. 2002, pp. 4-14.

[34] Intel StrongARM SA-1110 Microprocessor Brief Datasheet, April 2000.

[35] B, Jacob, Cache design for embedded real-time systems, Embedded Systems Conference, Danvers MA, June 30, 1999.

[36] P. Jain, S. Devadas, D. W. Engels and L. Rudolph, Software-assisted cache replacement mechanisms for embedded systems, ICCAD, Nov. 2001, pp 119-126.

[37] T. L. Johnson and W. W. Hwu, Run-time adaptive cache hierarchy management via reference analysis, in Proceedings the 24th International Symposium on Computer Architecture, Jun. 1997, pp 315-326.

[38] T. L. Johnson, M. C. Merten, and W. W. Hwu, Run-time spatial locality detection and optimization, in Proceedings the 30th International Symposium on Microarchitecture, Dec. 1997, pp. 57-64.

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

[40] P. Jung-Wook, K. Cheong-Ghil, L. Jung-Hoon and K. Shin-Dug, An energy efficient cache memory architecture for embedded systems, in Proceedings of the ACM symposium on Applied computing, Mar. 2004, pp. 884-890.

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

[42] M. B. Kamble and K. Ghosse, Analytical energy dissipation models for low power caches, in Proceedings of International Symposium on Low Power Electronics and Design, Aug. 1997, pp.143 -148.

[43] K. M. Kavi, A. R. Hurson, P. Patadia, E. Abraham and P. Shanmugam, Design of cache memories for multi-threaded dataflow architecture, in Proceedings of the 22nd International. Symposium on Computer Architecture (ISCA-22), Jun 1995, pp. 253-264.

[44] K. M. Kavi and A. R. Hurson, Performance of cache memories in dataflow architectures, Euro-micro Journal on Systems Architecture, Volume 44 Issue 9-10 (Jun. 1998) 657-674.

[45] C. Kulkarni, C. Ghez, M. Miranda, F. Catthoor and H. D. Man, Cache conscious data layout organization for conflict miss reduction in embedded multimedia applications, IEEE Transactions on computers, Volume 54 Issue 1 (Jan. 2005), 76-81.

[46] G. Kurpanek, K. Chan, J. Zheng, E. DeLano and W. Bryg, PA7200: a PA-RISC processor with integrated high performance MP bus interface, COMPCON Digest of Papers, Volume 94 Issue 28 (Feb. 1994), 375-382.

[47] M. S. Lam, P. R. Wilson, and T. G. Moher, Object type directed garbage collection to improve locality, in Proceedings of the International Workshop on Memory Management, Sep. 1992, pp. 16–18.

[48] J. H. Lee, G. H. Park, K. W. Lee, T. D. Han, and S. D. Kim, A Power Efficient Cache Structure for Embedded Processors Based on the Dual Cache Structure, in Proceedings of the ACM LCTES, Jun. 2000.

[49] J. H. Lee, J. S. Lee and S. D. Kim, A new cache architecture based on temporal and spatial locality, Journal of Systems Architecture, Volume 46 Issue 15 (Sep. 2000), 1451-1467.

[50] J. H. Lee, S. D. Kim and C. Weems, Application adaptive intelligent cache memory system, ACM Transactions on Embedded Computing Systems, Volume 1 Issue 1 (Dec. 2002), 56-78.

[51] G. Lesartre and D. Hunt. PA-8500: The continuing evolution of the PA-8000 family, in Proceedings of Compcon, 1997.

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

[53] C. Luk and T. C. Mowry, Memory forwarding: Enabling aggressive layout optimizations by guaranteeing the safety of data relocation, in Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA), Jun. 1999, pp. 88-99.

[54] S. A. McKee, R. H. Klenke, K. L. Wright, W. A. Wulf, M. H. Salinas, J. H. Aylor, A. P. Barson, Smarter Memory: Improving Bandwidth for Streamed References, IEEE Computer (Jul. 1998) 54-63.

[55] E. McLellan, The Alpha AXP architecture and 21064 processor, IEEE Micro, Volume 13 Issue 4 (Jun 1993), 36–47.

[56] V. Milutinovic, M. Tomasevic, B. Markovic and M. Tremblay, The split temporal/spatial cache: initial performance analysis, SCIzzL-5, Mar. 1996.

[57] V. Milutinovic, M. Prvulovic, D. Marinov and Z. Dimitrijevic, The splits spatial/non-spatial cache: a performance and complexity evaluation, IEEE Technical Committee on Computer Architecture Newsletter, (Jul. 1999), 11-18.

[58] D. A. Moon, Garbage collection in a large LISP system, in Conference Record of the 1984 Symposium on LISP and Functional Programming, Aug. 1984, pp. 235–246.

[59] A. Naz, K.M. Kavi, P.H. Sweany and M. Rezaei, A study of separate array and scalar caches, in Proceedings of the 18th International Symposium on High Performance Computing Systems and Applications (HPCS 2004), Winnipeg, Manitoba, Canada, May, 2004, pp 157-164.

[60] A. Naz, M. Rezaei, K. Kavi and P. Sweany, Improving data cache performance with integrated use of split caches, victim cache and stream buffers, in Proceedings of the Workshop on Memory performance dealing with applications, systems and architecture (MEDEA-2004), also published in SIGARCH's ACM Computer Architecture News (CAN), Volume 33 Issue 3 (Jun. 2005) 41-48.

[61] A Naz, K. Kavi, M. Rezaei and W. Li, Making A Case For Split Data Caches For Embedded Applications, in Proceedings of the Workshop on Memory performance

dealing with applications, systems and architecture (MEDEA-2005), also published in SIGARCH's ACM Computer Architecture News (CAN), Volume 34 Issue 1 (Mar. 2006) 19-26.

[62] A. Naz, K. Kavi, P. Sweany and W. Li, A Study of Reconfigurable Split Data Caches and Instruction Caches, in Proceedings of the ISCA 19th International Conference on Parallel and Distributed Computing (PDCS-2006), Sep. 2006, pp. 154-160.

[63] A. Naz, K.M. Kavi, P.H. Sweany and W. Li, Tiny split data caches make big performance impact for embedded applications, Special Issue on Embedded Single-Chip Multicore Architectures and related research - from System Design to Application Support of ACM Journal of Embedded Computing, Volume 2 Issue 2, (Nov 2006) 207-219.

[64] A. Naz, K. Kavi, P. Sweany and W. Li, Reconfigurable Partitioned Data cache: a Novel Approach for Embedded Systems, in Proceedings of SAC 2007 22nd ACM Symposium on Applied Computing SPECIAL TRACK on Embedded Systems: Applications, Solutions, and Techniques, Mar. 2007, pp. 707-712.

[65] S. Palacharla and R. E Kessler, Evaluating Stream Buffers as a Secondary Cache Replacement, in Proceedings of the 21th International Symposium on Computer Architecture, Apr. 1994, pp. 24-33.

[66] P. Petrov, A. Orailoglu, Towards Effective Embedded Processors in Codesigns: Customizable Partitioned Caches, in International Symposium on Hardware/Software Codesign (CODES), Apr. 2001, pp. 79-84.

[67] P. Ranganathan, S. V. Adve and N. P. Jouppi, Reconfigurable caches and their application to media processing, in Proceedings of the 27th International symposium on Computer Architecture, Jun 2000, pp. 214-224.

[68] E. Rashid, A CMOS RISC CPU with On-Chip Parallel Cache, ISSCC Digest of Papers, Feb. 1994, pp. 210-211.

[69] J. A. Rivers and E. S. Davidson, Reducing conflicts in direct-mapped caches with a temporality based design, in Proceedings of the International Conference on Parallel Processing, Aug. 1996, pp. 154-163.

[70] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren, Supporting Dynamic Data Structures on Distributed Memory Machines, ACM Transactions on Programming Languages and Systems, Volume 17 Issue 2 (Mar. 1995), 233-263.

[71] F. J. Sanchez, A. Gonzalez, and M. Valero, Software management of selective and dual data caches, IEEE Technical Committee on Computer Architecture Newsletter, (Mar. 1997) 3-10.

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

[73] M. L. Seidl, and B. G. Zorn, Segregating heap objects by reference behavior and lifetime, in Proceedings of the Eight International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII), Oct. 1998, pp. 12-23.

[74] A. J. Smith, Cache memories, ACM Computing Surveys, Volume14 Issue 3 (Sep. 1982), 473-530.

[75] K. So and R. N. Rechtschaffen, Cache operations by MRU change, IEEE Transactions on Computers, Volume 37 Issue 6, (Jun. 1988) 700-709.

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

[77] D. Stiliadis, Selective victim caching: A method to improve the performance of direct-mapped caches, IEEE Transactions on Computers, Volume 46 Issue 5 (May 1997) 603-610.

[78] M. Tomasko, S. Hadjiyiannis and W. A. Najjar, Experimental evaluation of array and scalar caches, IEEE Technical Committee on Computer Architecture Newsletter, (Mar. 1997) 11-17.

[79] O. S. Unsal, I. Koren, C. M. Krishna and C. A. Moritz, The minimax cache: an energy-efficient framework for media processors, 8th International Symposium on High-Performance Computer Architecture, HPCA8, Feb. 2002, pp. 131-140.

[80] 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 (May 1996) 677 -688.

[81] M. E. Wolf and M. S. Lam, A data locality optimizing algorithm, in PLDI, Jun 1991, pp 30–44.

[82] C. Zhang, F. Vahid and W. Najjar, Energy benefits of a configurable line size cache for embedded systems, IEEE International Symposium on VLSI Design, Feb. 2003, pp. 87-91.

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

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