ACTIVE MANAGEMENT OF CACHE RESOURCES

A Dissertation
Presented to
The Academic Faculty

By

Subramanian Ramaswamy

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
in
Electrical and Computer Engineering

School of Electrical and Computer Engineering
Georgia Institute of Technology
Aug 2008

ACTIVE MANAGEMENT OF CACHE RESOURCES

Approved by:

Prof. Sudhakar Yalamanchili, Advisor
School of ECE
Georgia Institute of Technology

Prof. Yorai Wardi
School of ECE
Georgia Institute of Technology

Prof. Jeffrey Davis
School of ECE
Georgia Institute of Technology

Prof. Umakishore Ramachandran
College of Computing
Georgia Institute of Technology

Prof. David Schimmel
School of ECE
Georgia Institute of Technology

Date Approved: June 30 2008

Dedicated to Appa, Amma and Divya

ACKNOWLEDGMENT

As a doctoral student, the one decision that matters most is choosing an advisor. I chose the best. Prof. Sudhakar Yalamanchili is not only a great researcher and teacher, but also a wonderful friend. The motivation, support and guidance he provided were the core elements in my graduate student career. His encouragement kept me going in stressful times. I owe a debt of gratitude to him that will never be fulfilled.

My proposal and defense committee members including Professors Jeffrey Davis, Yorai Wardi, David Schimmel, Sean Lee and Umakishore Ramachandran provided me with valuable suggestions and feedback which helped make my contributions stronger.

If I completed my thesis, I owe it to my loving wife Divya. She was always there for me. She put up with my insane working hours and my inability to spend any measurable time with her over the last couple of years as I entered the business end of my doctoral studies. She was an unending source of happiness and joy for me. The few hours I spent in her company were always priceless.

Anything I ever accomplished or will accomplish owes a lot to my dearest parents. They instilled the value of education in me and taught me many life lessons that I value in my personal and professional endeavours. My late father always provided me with the support and encouragement to pursue whatever interests me and my mother taught me never to be satisfied with second prize.

My sister Geetha was another person who was there for me with her support and love during my graduate studies. Visiting her in Chicago during December always filled me with warmth in spite of the winter temperatures.

Last, but definitely not the least, I owe a lot to all my friends and colleagues who helped me at various stages of my career by acting as a sounding board, providing me with unlimited support and helping me with the logistics and paperwork—Jeff Young, Tushar Kumar, Jaswanth Sreeram and Nawaf Al-moosa deserve special mention.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Thesis Statement: Active cache management customized to application memory behavior improves cache efficiency.

The shift from scaling frequency to scaling the number of cores, continues the trend of stressing off-chip memory bandwidth and reliance on larger on-chip caches. Caches typically occupy 40–60% of the chip area and account for 15–30% of energy consumption on chip. For a fixed die size, execution performance will drive the need for more cores while the increased memory bandwidth required to sustain these cores will drive the need for larger caches while keeping energy costs at a minimum. This dissertation is focused on the development of techniques to reconcile these conflicting area and energy demands. The solutions provided in this dissertation are driven in part by the observation of very low performance efficiencies and energy efficiencies for modern data caches. For a range of application domains, data cache utilizations are below 20%, performance efficiencies below 15% and energy efficiencies *below 1%*! These low efficiencies are not sustainable for a critical architectural resource that is also a dominant resource consumer on-chip. Consequently, the ability to scale the number of cores while concurrently pushing back the memory and power walls will require significant improvements in cache efficiency. Concepts and implementations to achieve such improvements are addressed in this dissertation.

Two significant causes of low cache efficiencies are: (i) transistor leakage, and, (ii) the fixed manner in which main memory lines share the cache. At technologies below 70 nm, a L2 cache with size greater than 256 KB has 95% of its energy consumption attributed to leakage. Further, typical L2 cache line is accessed only once every hundreds of thousands of cycles, whereas, the transistors comprising the cache line leak *every cycle* leading to poor energy efficiencies. The low utilization and performance efficiencies can be attributed to the fixed mapping of main memory lines to the cache. This fixed mapping typically

results in the majority of the cache storing data that will not be accessed in future. Indeed, a utilization number of 25% indicates that, on average only 25% of the cache stores data that will be accessed in the future!

The approaches proposed for efficiency improvement is predicated on addressing the preceding two sources of inefficiency. The approach to improving energy efficiency primarily relies on *sizing* the cache to match application memory footprint or working set during a program phase and powering down all remaining cache lines. The approach to improving cache utilization and performance efficiency primarily relies on changing the placement function—the manner in which main memory lines share cache resources motivated in part by compiler-driven approaches where application-data shares registers via compiler based register allocation techniques. This is referred to as *shaping* the cache. The approach to sizing differs from past approaches in that these techniques produce fully functional smaller caches. There can be no references to inactive cache lines. This is feasible since sizing is used in conjunction with shaping. These techniques are predicated on partitioning main memory into *conflict sets*—each set of memory lines is mapped to the same cache set. The optimization problem is the application-driven construction of conflict sets to maximize energy and performance efficiencies. The engineering challenge is the development of hardware-software techniques for static and dynamic sizing and shaping. Collectively, the application of these techniques is referred to as *active management*. This dissertation makes the following specific contributions.

First, a model is developed for quantifying cache utilization, energy efficiency and performance efficiency. Efficiency metrics are introduced followed by an analysis of common single-threaded and multi-threaded benchmarks. The resulting efficiencies are found to be very low ($< 25\%$ for utilization, 15% for performance efficiency, $< 1\%$ for energy efficiency). The analysis exposes some of the key sources of cache inefficiency (e.g.,leakage and how memory lines share the cache) and provides the insights for the development of energy improvement through active management techniques.

Second, profile-driven algorithms for computing placement functions for caches are provided. This was first applied to the design of application specific caches for network processors. In such architectures memory accesses are dominated by accesses to a few major data structures: in the case of network processors, it is the route lookup data structures. This dissertation demonstrates the design of an application specific cache, named the data trace cache for application in networking applications. The customized cache exploited the predictability of accesses to the routing table data structure. The proposed solution has the flavor of stream buffers in general-purpose caches.

This notion of profile driven placement is developed further by adapting it to a more general solution for embedded processors. Greedy algorithms are used on profile data to develop application specific customized cache placement to manage the cache for embedded processors. This resulted in a sharp decrease in conflict misses in embedded processor caches. Another application of these profile-driven solutions was aimed at masking out faulty cache lines with minimal performance degradation which was found to be very effective. The low performance degradation seen using customized placement strategies exposed the existing redundancy in traditional cache storage, which could be intelligently adapted for energy and performance benefits.

Third, the preceding two techniques are consolidated in a solution for application specific cache configurations. The key insight is to recognize the relationship between voltage level and fault free memory cells in the cache. A configuration profile is generated for the cache where each voltage level is mapped to a size and shape (placement function). This shape in turn can be profile driven particularly useful for embedded processors. During program execution voltage levels and consequently cache size and shape can be selected for each program region to concurrently improve both energy and performance efficiencies.

Fourth, active management techniques are developed for static sizing and shaping caches for scientific computing applications which have predictable memory reference behavior

with well defined memory footprints. This domain knowledge was used to derive customized cache placement solutions, which in conjunction with sizing improved performance and energy efficiencies significantly over traditional caches (Section 6.1). The solutions exploited the property of strided accesses to multi-dimensional array data structures being a common feature in scientific applications. The improved sharing achieved using active management, i.e., cache shaping, led to significant drops in miss rates and execution time in addition to improved energy efficiency by sizing the cache to the program phase memory footprint. This work on static sizing and shaping strategies is further extended to include the current environment of multi-threaded applications. In Section 6.2. The application of static sizing and shaping strategies is extended to multi-threaded domains to optimize the efficiencies of caches shared among multiple application threads.

Fifth, active management techniques are developed for runtime sizing and shaping for applications lacking statically characterizable profiles. The concept of miss folding is introduced as a technique for dynamically sizing and shaping the cache. Simple folding schemes relying on various heuristics were found to be effective in decreasing the energy-delay product (EDP) significantly and increasing energy and performance efficiency.

To summarize, state of the practice is focused on applying power down events such as gated-Vdd or drowsy state to selected components of the cache. Consequently references can be made to inactive cache components with accompanying performance penalties, for example the need to power and reload a line or way. In contrast this dissertation sizes a cache in conjunction with customized placement to produce fully functional caches of varying sizes with references accessing only active (i.e., powered) cache components.

Sizing and shaping can be applied at different phase of the design cycle. For example, after burn in test, sizing and shaping can be used to mask faulty cache elements and perform a per application configuration of the on-chip cache hierarchy (Techniques 2 and 3). During compile time, cache configurations (a size and shape) can be computed for various program phases and effected via a software interface (Technique 4). Such an approach is particularly

effective for applications where memory access patterns can be determined via program analysis such as in the scientific computing domain. A more flexible variant is the runtime inference of memory access patterns (Technique 5) and the dynamic determination of a new cache size and shape configuration. This dissertation contributes techniques at all of the preceding points in the life cycle of a program.

Finally, this dissertation opens up a new degree of freedom for compiler optimizations. Compilers have been restricted to a cache with fixed mapping from main memory. Opening up the cache placement function provides another avenue for optimization. The relaxation of the placement coupled with the conflict set construction and notion of liveness provides the means for formulating a range of tractable memory system optimizations.

CHAPTER 1

INTRODUCTION

This dissertation addresses two major sets of challenges facing processor design as the industry enters the deep sub-micron region of semiconductor design. The first set of challenges relates to the well understood memory bottleneck [1] that shows no signs of easing even as the focus shifts from scaling processor frequency to scaling the number of cores. This trend has led to the increasing reliance on larger on-chip caches which occupies 40–60% of area on chip and consuming 15–30% of energy expended on chip.The second set of challenges is posed by transistor leakage and process variation (both inter-die and intra-die) at future technology nodes with leakage power anticipated to increase exponentially and sharply lower defect-free yield with successive technology generations. This dissertation focuses on resolving these two challenges by abstracting them as one problem—developing efficient caches.

The first set of challenges results in efforts to decrease the average memory access latency through the addition of larger and deeper cache hierarchies. However, this has led to an over reliance of caches, with caches typically occupying 40–60% [2] of the chip area and are estimated to account for 15–30% [3, 4] of the overall chip energy consumption. For example, the Intel Itanium 2 processor 9010 has a 6 MB L3 cache [5], with the cache hierarchy occupying almost 60% of the die area. Apart from the increasing cache costs, adding more cache resources results in lower resources for the processing cores limiting Moore's law expansion in raw processing power.

Transistor leakage at sub-micron technologies is at the forefront of the second set of challenges facing the semiconductor industry. Borkar estimates leakage energy to be 60% of the overall processor energy consumption at 65 *nm* and anticipates the leakage current to increase by a factor of 7.5 with successive technology generations [6]. This increase in leakage current is expected to scale total leakage power on the chip by a factor of 5.

Leakage is a function of area on the chip, and caches, because of their significant area budgets are major contributors to leakage [3, 4]. Additionally, since the majority of the cache is idle most of the time (especially L2 and L3 caches), the SRAM cells in caches contribute signficantly more to leakage energy than to switching energy.

The efficiencies of the cache hierarchy is quantified and it was determined that across a range of application domains, data cache utilizations have been found to be below 25%, performance efficiencies below 15% and energy efficiencies *below 1%*! These low efficiencies are not sustainable. Consequently, the ability to scale the number of cores concurrently while pushing back the memory and power walls will require significant improvements in cache efficiency, which is the focus of this dissertation.

Two significant causes of low cache efficiencies are identified: (i) transistor leakage, and, (ii) the fixed manner in which main memory lines share the cache. At technologies below 70 nm, a L2 cache with size greater than 256 KB has 95% of its energy consumption attributed to leakage. A typical L2 cache line is accessed only once every hundreds of thousands of cycles, whereas, the transistors comprising the cache line leak *every cycle* leading to poor energy efficiencies. The low utilization and performance efficiencies can be attributed to the fixed mapping of main memory lines to the cache. In traditional caches the set of memory lines mapping to a cache set (a *conflict set*, Section 3.4), is fixed at design time and this structure is one of the major sources of cache inefficiency. This fixed mapping typically results in the majority of the cache storing data that will not be accessed in future. Indeed, a utilization number of 25% indicates that, on average only 25% of the cache stores data that will be accessed in the future! Thus, the dominance of leakage energy coupled with the static cache architecture leads to inefficiency.

This thesis focuses on actively managing the cache structure, leading to increased utilization and performance efficiency while at the same time limiting any adverse impact on energy consumption. Managing the cache resources in an active manner improves the sharing of cache resources among main memory lines by making this mapping malleable.

This increases cache efficiency—this is the focal point of this dissertation. As technology scales and designs incorporate larger caches for greater performance, cache efficiency is of paramount importance, and this dissertation takes a first step towards that direction.

The key to improving cache energy and performance efficiencies is to construct *conflict sets* customized to the application memory access pattern (i.e., *shaping*) and minimizing the footprint of the application in the cache (i.e., *sizing*). Collectively, *sizing* and *shaping* are referred to as *active management*, as they match the cache size and placement structure to the application memory access behavior. The vehicle that is adopted to achieve active management in this thesis is customizing the cache placement. The result is lower conflict misses (better performance efficiency) and smaller footprints with the ability to turn off unused lines (better energy efficiency). This dissertation focuses on hardware-software approaches embodying this philosophy to improve cache efficiencies.

Sizing and shaping can be applied at different phase of the design cycle. For example, after burn in test, sizing and shaping can be used to mask faulty cache elements and perform a per-application configuration of the on-chip cache hierarchy. During compile time, cache configurations (a size and shape) can be computed for various program phases and effected via a software interface. Such an approach is particularly effective for applications where memory access patterns can be determined via program analysis such as in the scientific computing domain. A more flexible variant is the runtime inference of memory access patterns and the dynamic invocation of a new cache configuration. This dissertation contributes techniques at all three of the preceding points in the life cycle of a program.

The optimization problem is one of forming conflict sets customized to the memory reference pattern of a program phase. The engineering problem is to architect low cost software and hardware solutions to effect sizing and shaping. The design problem is to decide where such techniques should be applied: one-time (per-application) configuration, statically at compile time, or dynamically at run time.

The state of the practice in making caches more energy efficient has been to power

down cache components such as cache lines, sets or ways—turn them off or maintain them in a low voltage state. Strategies focus on *when* to turn off *which* components. Studies improving cache performance include pseudo-associative designs and compiler optimizations along with fixed hardware modifications. Strategies to improve performance efficiency rely on hardware strategies that implement new fixed management structures or through compiler optimizations, noticeable for scheduling and data layout. Existing strategies to improve efficiencies are constrained by one of the primary causes of inefficiency—the fixed manner in which main memory shares the cache. Additionally, improving cache efficiency leads to solutions that might seem counter-intuitive; for example, accesses concentrated on a few cache sets can lead to better efficiency than distributing those accesses across the entire cache but may degrade performance substantially if applied in an agnostic manner. By making the cache *size* and *shape* fluid, efficiencies can be improved by exploiting memory access behavior. Thus, active management can be viewed as an evolution of present day energy saving and performance improvement strategies.

This thesis is organized as follows. The following chapter describes several related techniques used in the literature for improving cache performance and contrasts the strategies proposed in this thesis with them. Chapter 3 describes a model for quantifying cache utilization and efficiency and applies this model to empirically analyse the performance of single threaded and multi-threaded applications in traditional caches. This analysis provides the necessary insights for developing customized active management techniques. Chapter 4 describes the programmable placement system model including the programming model, the operational model and the architecture model employed along with the range of options customization. Chapters 5, 6 and 7 describe various techniques for improving cache performance and efficiencies subdivided into offline profile driven strategies, compile-time strategies driven by program analysis, and dynamic strategies driven by runtime measurements. The dissertation concludes with a brief discussion of the potential extensions to this body of work and a brief summary of this dissertation.

CHAPTER 2

RELATED WORK

A vast body of work applies hardware and software strategies to enhance cache performance and power profiles. However, techniques in the literature are constrained by the passive cache management structure which limits optimization opportunities. Additionally, the optimizations outlined tend to either improve performance or save energy; whereas active management focuses on performance and energy; i.e., cache management to minimize energy and delay simultaneously. The proposed work couples programmable cache placement functions to a domain of applications that can be programmed using analysis (either statically via compilers or at run-time via dynamic optimizers). Active cache management subsumes many of the optimizations outlined in this chapter and, additionally, is complementary to many techniques that can lead to further optimizations. The existing work in literature can be broadly classified into three categories—those which seek to improve execution time, those which seek to improve energy savings and those which provide defect tolerance to caches. A brief discussion of the literature in the three categories follows.

2.1 Execution Time Optimizations

This class of optimizations focus on decreasing cache miss rates and program execution time by means of hardware adaptations or through compiler optimizations. While overall energy consumption may be reduced using these optimizations because of the decrease in execution time, none of these optimizations focus on efficiency, i.e., the maximum performance that can be obtained per transistor. Finally, when performance is being chased, energy becomes second priority. In this dissertation, the focus is on improving cache efficiency—both energy efficiency and performance efficiency. Additionally, many of these performance optimizations are complementary to the approach described in this dissertation and can be applied together. Finally, some of these optimizations may be realized as

instances of the active management approach prescribed in this thesis.

Pseudo Associativity Mechanisms

Multiple optimizations focus on providing the performance of higher associativity caches with access latencies similar to those of lower associativity caches to decrease conflict misses and miss rates. For example, Jouppi [7], proposed small fully-associative buffers called victim caches for reducing misses to heavily accessed entries in a traditional cache, Qureshi *et al*.[8] designed a V-way associativity cache by doubling the number of ways of associativity and using global replacement strategies, Peir *et al*.[9] proposed the adaptive group-associative cache (AGAC), which improves the performance of first level direct-mapped caches by using multiple banks for swapping data with variable access latency. Various other forms of pseudo-associative caches, which trade variable hit latency for increasing associativity, include the hash-rehash cache proposed by Agarwal *et al*.[10], the column associative cache proposed by Agarwal and Pudar [11], and the predictive sequential-associative cache proposed by Calder *et al*.[12]. Hallnor and Reinhardt [13] proposed the Indirect Index Cache (IIC) to achieve full-associativity through software management relying on chain traversal. In the NuRAPID cache proposed by Chishti *et al*.[14], the access latency of different cache lines varies depending on the physical placement of data within the data-store and lowers the cache access latency for increased associativity caches. The proposed active management techniques are complementary to many of these schemes since the mapping of a memory line to a cache set is adapted (as opposed to being mapped to a fixed cache set).

Indexing Schemes

Innovative approaches in cache indexing seek to find better *fixed* design time placement functions than modulo placement functions, and active cache management subsumes many of these schemes. For example, the two-way skewed associative cache proposed by Seznec [15]

attempts to distribute memory accesses uniformly to minimize conflicts by having two indexing functions per cache set. Other techniques aimed at distributing memory accesses across cache sets uniformly by modifying the indexing function include the prime modulo scheme proposed by Kharbutli *et al.*[16], the randomized cache placement scheme proposed by Topham and Gonzalez [17], and the balanced cache proposed by Zhang [18] that uniformly distributes accesses to a direct-mapped cache for reducing misses using programmable content addressable memory decoders for embedded systems. Uniform distribution does not necessarily result in better efficiency, as all cache sets have to be maintained as active or powered on, thereby expending leakage energy.

Cache Replacement Optimizations

An orthogonal body of work for improving cache performance focuses on the replacement policies, i.e., the placement of a memory line within a set. This includes many innovations, including frequency-based replacement proposed by Robinson and Devarakonda [19], recency-based replacement proposed by Jiang and Zhang [20], and adaptive replacement schemes proposed by Subramanian *et al.*[21] and Smaragdakis *et al.*[22]. Puzak [23] proposed the inclusion of extra tags in a shadow directory to provide feedback to a local replacement engine in a set-associative cache.

Compression Optimizations

Cache compression can store larger chunks of data in the cache as shown by Lee *et al.*[24], Zhang *et al.*[25], Alameldeen and Wood [26], Wilson *et al.*[27]. Cache compression is complementary and can be applied concurrently to active management strategies. However, compressing and decompressing data upon entry and exit from the cache involve a substantial overhead.

Partitioned Caches

Many techniques such as sub-banking achieve power reduction by partitioning the data cache into smaller, low-power components. This includes work done by Ghose and Kamble [28] and Su and Despain [29]. Other approaches to partitioned caches categorize locality and reference behavior and cache the different categories separately for energy and performance benefits. Several strategies partition the cache to exploit reference locality across scalar and vector data. Examples include split caches proposed by Dahlgren and Stenstrom [30] and techniques proposed by Petrov and Orailoglu [31]. Lee and Tyson[32] partitioned the cache for memory regions such as heap and stack for lowering energy dissipation.

Stone *et al*.[33] studied optimal static partitioning of the cache ways between two or more applications. Suh *et al*.[34] described a scheme using hit recency of the lines in the cache to estimate the utility of the cache for each application and share the cache ways accordingly. Partitioning mechanisms for shared caches were described by Iyer [35]. Hsu *et al*.[36] studied different policies for partitioning a shared cache among competing applications. Qureshi and Patt [37] proposed a utility-based cache partitioning for sharing the cache across two applications while lowering miss rates. Chang and Sohi [38] proposed dynamic mechanisms for providing latency comparable to that of a private cache while obtaining the benefits of a shared cache. Liu *et al*.[39] proposed a split L2 cache for chip multiprocessors with non-uniform access latencies. Many of these strategies seek to partition ways in a shared cache, whereas the strategy proposed in the research herein partitions a uniform latency shared cache by sets. Additionally, the cache can be made private or shared by customizing the cache placement using active management strategies.

NUCA caches

Non-uniform cache architectures (NUCA) are offered as an alternative to monolithic fixed latency set-associative caches and were proposed by Kim *et al*.[40], Chishti *et al*.[41], Huh *et al*.[42]. NUCA caches are specifically targeted to multiprocessor memory hierarchies.

Dynamic NUCA caches migrate sets of memory lines closer to the processor that might use the data to optimize latency. On the other hand, active management schemes determine the construction of the sets of memory lines and are, therefore, complementary to such optimizations.

Stride Prediction and Pre-fetching

Memory latency can be decreased by employing data pre-fetching. These techniques involve capturing memory access patterns (MAP) and using them to pre-fetch data into the cache. Examples include Luk and Mowry [43], Mowry *et al*.[44], Kim *et al*.[45, 46].

Stride predictors, as proposed by Sazeides and Smith [47], Fu *et al*.[48], Sair *et al*.[49], etc., can be used to predict access strides, which can lead to better pre-fetching decisions. The proposed active management cache design can decrease conflicts among pre-fetched data, leading to better cache efficiencies and bandwidth preservation. Furthermore, memory behavior can be analyzed for better pre-fetching and caching strategies, as shown by Ghosh *et al*.[50], and these techniques can be used to optimize cache management decisions.

Data Layout and Scheduling Optimizations

Data re-layout techniques optimize cache performance for specific access patterns. This includes works proposed by Chilimbi *et al*.[51], Panda *et al*.[52], and, Rabbah and Palem[53]. Carter *et al*.[54] propose using a memory controller to modify the virtual layout to make better use of pre-fetching. Other works proposing active memory controllers include Kim *et al*.[55], and, Heinrich *et al*.[56]. Another theme is reordering memory reference streams via loop transformations such as those proposed by Panda *et al*.[57], and, McKinley *et al*.[57].

The research proposed herein does not alter the memory layout (virtual or physical), or, the instruction schedule in any manner, and is therefore complementary to all such compiler optimizations. However, in certain scenarios, active management can provide similar performance benefits obtained using these compiler optimizations at lower costs. For example, re-mapping data is expensive, and at run-time, data re-mapping becomes infeasible

for large data structures. In such cases, the active management schemes proposed herein provide similar benefits without any of the added disadvantages. The compiler optimizations found in the literature are targeted to fixed cache designs with modulo placement. These optimizations can benefit from actively managed caches since they provide greater opportunity for optimizations. For example, a specific layout may no longer preclude certain instruction schedules for good performance.

Scratch-pad Memories

Several efforts propose compiler controlled on-chip scratch-pad memories as an alternative to hardware caches. Examples include Banakar *et al*.[58], Steinke *et al*.[59], Miller and Agarwal [60], Udayakumaran *et al*. [61], and Panda *et al*. [62]. Scratch-pad memories require explicit control of all data movement between the scratch-pad and the off-chip memory, which leads to an increase in code size and software complexity. In this context, Chiou *et al*.[63] proposed column caching to map specific application data within a specified region in the cache, thereby approximating scratch-pad memory behavior.

2.2 Energy Optimizations

The optimizations in this category are focused on energy savings and simultaneously minimizing performance degradation. While this approach can increase energy efficiency, the degradation in performance can compensate for energy savings thereby causing the adopted strategies to be very conservative. This thesis approaches energy savings by increasing efficiency, i.e., energy savings with performance improvements or adaptations that limit performance degradation even with significant energy savings through aggressive cache line turn off strategies.

Scheduling Cache Turn-offs and Drowsy States

The state of the practice in reducing leakage energy has been to power down cache components such as cache lines, sets or ways. Examples include Albonesi [64], Abella *et al*.[65],

Kaxiras *et al*.[66], Powell *et al*.[67], C. Zhang *et al*.[68], M. Zhang and Asanovic [4], and Zhou *et al*.[69]. Another approach is to keep the cache in a low voltage *drowsy* state as proposed by Flautner *et al*. [70]. W. Zhang *et al*.[71, 72] use special instructions to schedule instruction cache turn offs using loop and branch information or maintain the cache in a drowsy state, activating cache lines prior to access, while Geiger *et al*. [73] combine region-based caching with drowsy caching to reduce cache power dissipation. These strategies focus on *when* to turn off *which* components. Poor decisions lead to expensive misses and power-up events and therefore strategies tend to be conservative. These approaches tune the cache on system level metrics, such as the number of accesses and misses, and do not exploit the concept of conflict set live ranges in data caches to increase leakage savings with minimal performance degradation. Furthermore, many of these approaches can be combined with customized placement to increase energy savings.

Filter Caches

In a two-level inclusive cache, an access to the L2 cache esults in the data being brought into the L1 cache. Therefore, the L1 cache acts as a filter to the L2 cache. For example, there will not be any stride zero access to the L2 cache. Kin *et al*.[74] propose using a small-sized L1 cache to save energy while having an L2 cache sized similar to a typical L1 cache to improve performance. Memik *et al*.[75] use the notion of filtering and prediction to prevent accesses to L2 caches if the access potentially results in a miss, thus saving energy. Bloom filters were used to reduce the energy of virtual cache synonym look-ups by Wyoo *et al*.[76].

2.3 Defect Tolerance

Defect tolerance optimizations have traditionally focussed on next neighbor re-mapping or redirecting faulty cache cell accesses to memory; these approaches are not scalable and result in rapid performance degradatation with increases in the number of faulty cells, as expected with DSM technologies. This dissertation, by focussing on efficiency provides

approaches that can result in caches with a high degree of defect tolerance, as performance loss due to faulty cells can be compensated by applying active management schemes.

Redundancy and ECC Schemes

Adding redundant blocks, as proposed by Turgeon *et al*.[77], Lucente *et al*. [78], Nokolos *et al*. [79], etc., is one approach to designing fault tolerant caches. Using error correcting codes (ECC) as proposed by Kalter *et al*. [80] also provides fault tolerance for on-chip caches. However, these techniques are limited in the number and distribution of faulty cells that can be tolerated, and are improved upon by active cache management approaches that use re-mapping.

Re-mapping or Eliminating Faulty Cache Blocks

The re-mapping approach modifies the cache placement policy to map main memory lines to non-faulty cache lines and is proposed by Shirvani *et al*. [81] and Agarwal *et al*.[82]. Other approaches for fault tolerant caches include avoiding faulty ways in a set-associative cache as proposed by Ooi *et al*. [83], or placing code intelligently in faulty set-associative caches to minimize misses, as proposed by Zarandi *et al*.[84]. The goal of those efforts was fault tolerance, while performance optimization was not addressed. These techniques are additionally limited by their selection of block size (not the same as cache line size). The proposed active management techniques optimize for performance when faults are present and achieve a greater degree of fault tolerance as measured by lower performance degradation for the same distribution of faults.

CHAPTER 3

AN EFFICIENCY MODEL FOR CACHES

This chapter describes a model for quantifying and analysing the efficiency of caches. Several metrics are introduced for the purpose, followed by an analysis of common single-threaded and multi-threaded benchmarks from the efficiency viewpoint. The analysis exposes several sources of inefficiency that can be ameliorated through active management techniques for improving both the cache performance and energy efficiencies.

3.1 Analytical Model

At a clock cycle, a cache line may be *active* (powered) or *inactive* (turned off). Thus, without any energy management, all cache lines are active. A cache line is *live* at a clock cycle if it contains data that will be used prior to eviction, and it is *dead* otherwise [85, 66]. Thus, on any clock cycle, a cache line is live, dead, or inactive. For a cache with L lines over T cycles, the total cache cycles expended is the sum of the *live cycles*, the *dead cycles*, and the *inactive cycles*.

Cache utilization, η_u, is the average percentage of cache lines containing live data at a clock cycle [85, 66]. Utilization is computed as shown in Equation 1.

$$\eta_u = \frac{\sum_{i=0}^{i=L-1} live\_cycles_{line_i}}{\sum_{i=0}^{i=L-1} active\_cycles_{line_i}} \tag{1}$$

The *effectiveness* of the cache, E, is the percentage of cache cycles devoted to live lines and is shown in Equation 2. Effectiveness serves as a metric for comparing programmed cache line shutdown strategies; the higher the effectiveness, the higher the percentage of the active cache that retains live data.

$$E = \frac{\sum_{i=0}^{i=L-1}(live\_cycles_{line_i} + inactive\_cycles_{line_i})}{\sum_{i=0}^{i=L-1}(active\_cycles_{line_i} + inactive\_cycles_{line_i})} \tag{2}$$

Effectiveness can also be represented as shown in Equation 3, where $total\_cycles$ refers

to the program execution time. Here, the number of active cycles for a cache line, i.e., the number of cycles the cache line is powered on, is equal to the sum of the number of dead and live cycles for that cache line. The most effective scheme is one where all cache cycles are either live or inactive. Effectiveness is equivalent to utilization without any energy management.

$$E = 1.0 - \frac{\sum_{i=0}^{i=L-1} dead\_cycles_{line_i}}{total\_cycles * L} \tag{3}$$

Effectiveness can be increased at the expense of a high miss rate. For example, shutting down all but one line in a direct-mapped cache can produce high effectiveness for structured accesses. An efficient cache must be *effective* with high performance. Cache *performance efficiency,* η_p, is defined in Equation 4 as the product of effectiveness and a scaling factor, where t_c is the cache access time, t_p is the miss penalty, and m is the miss rate. A cache has 100% performance efficiency if it does not contribute any dead cycles and has a 100% hit rate.

$$\eta_p = E * \frac{t_c}{t_c + m * t_p} \tag{4}$$

Energy efficiency, η_e, is the ratio of *useful work* to total work. Useful work is the switching energy (i.e., energy consumed during access to the cache) expended in a cache hit. The total work is the sum of the switching energy consumed during all cache accesses (hits and misses) and the leakage energy. A cache has an energy efficiency of unity if all the energy consumed by the cache is equal to the switching energy consumed during cache hits. Energy efficiency is defined in Equation 5, where sw_{energy} represents the switching energy and $leak_{energy}$ represents the leakage energy. The switching energy consumed during a cache hit is assumed to equal the switching energy consumed during a cache miss; this approximation affects the results marginally as explained in the next section.

$$\eta_e = \frac{sw_{energy} * num_{hits}}{sw_{energy} * (num_{hits} + num_{misses}) + leak_{energy}} \quad (5)$$

Although cache sets or lines may be turned off to reduce leakage energy, additional misses that may result from the powering down of parts of the cache can increase program execution times. These increases in execution cycles can lead to higher energy consumption by all active lines, and therefore, the choice of lines or sets to turn off is critical. These additional cache misses and the resultant increase in execution times will in turn drop effectiveness and utilization since the number of dead cycles will also increase. Understanding this relationship between energy efficiency and cache line shutdowns helps identify good strategies for increasing cache efficiency.

3.2 Empirical Analysis of Single-threaded Applications

This section analyses the efficiency of several single-threaded applications. The execution of benchmarks from the SPEC2000 [86], Olden [87] and DIS [88] suites was simulated using the *Simplescalar* [89] simulator, which was modified to obtain cache efficiency.

The initialization phases for the SPEC2000 programs were fast-forwarded during simulation. Energy estimates were derived using *Cacti 4.2* [90] for 70 *nm* technology. The L2 cache access latency was assumed to be fixed at 15 cycles independent of the size and associativity of the cache. Varying the L2 cache latency affected execution times by less than 2%. The definition of energy efficiency assumed that the switching energy for a read was equal to the switching energy for a write. This assumption artificially increased energy efficiency because the real switching energy for a write is lower than that for a read (only one bank is accessed for a write compared to all banks for a read).

Leakage power constituted 95% of the total cache power at 70 *nm* for cache sizes of 256 KB or greater [90], and cache writes constituted a small fraction of the total number of accesses. Therefore, these assumptions affected efficiencies by less than 1%. Finally, the energy was calculated assuming that the cache operated at the highest frequency as given

by *Cacti*.

Cache utilization for a 256 KB 8-way L2 cache with 128-byte lines averaged 24% and utilization for the L1 cache averaged 12%. Cache utilizations for the various benchmarks are shown in Figure 1. Since the L2 cache accommodated some data structures entirely for certain applications (*164.gzip*, *field*), conflict misses were restricted, which resulted in higher utilization values for the L2 cache compared to the L1 cache. The low utilization numbers suggest that both levels of the cache maintain more *dead* lines than *live* lines. Thus, the majority of cache costs are spent in maintaining data that will not be re-used.

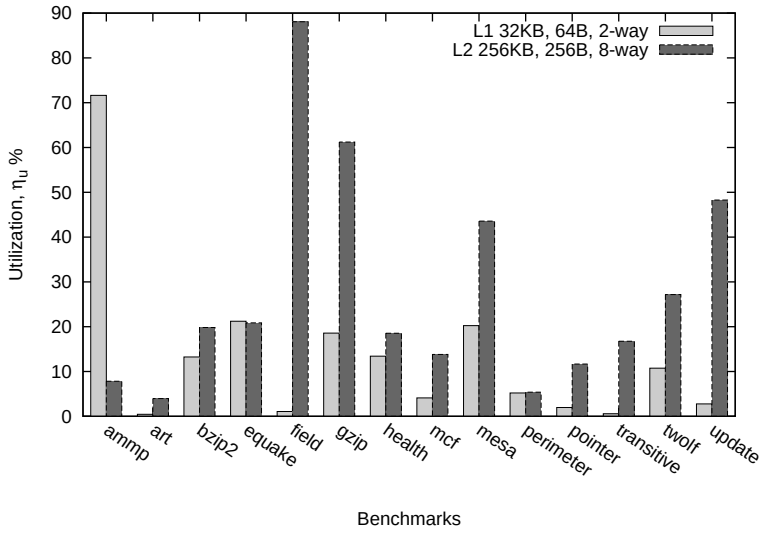


Figure 1. Cache utilizations for traditional cache designs.

Performance efficiency is shown in Figure 2 and averaged 4.7% for the L2 cache. Cache energy efficiency for current designs averaged 0.17% as observed from Figure 2. The major contributor to this low efficiency value was transistor leakage.

Utilization captures the temporal residency of live data in the cache. Performance efficiency captures how well this residency of live data in the cache is exploited. Thus, a live line that is accessed 10 times during its period of residence is more efficient than if it was only accessed twice during its period of residence. Energy efficiency captures the percentage of overall energy that is *useful* (energy expended servicing cache hits).

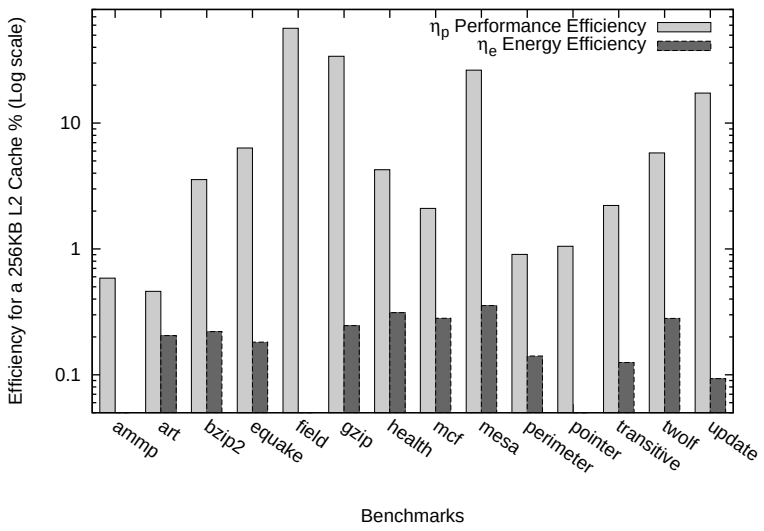


Figure 2. Cache efficiencies for traditional cache designs.

Some benchmarks had good utilization; for example, 164.*gzip* had 60% L2 cache utilization. For this benchmark, two supporting data structures had a 64 KB footprint each. While miss rates for both caches were low for 164.*gzip*, conflicts arose in the L1 cache because of its smaller size. Since the L1 cache size was only 32 KB, it was unable to fit the supporting data structures; this resulted in low utilization. However, the two data structures fit in the larger 256 KB L2 cache, which resolved most conflicts. Therefore, the larger L2 cache had better utilization than the smaller L1 cache for some benchmarks. The high utilization for 164.*gzip* was attributable to the linear manner in which the compression object was accessed. L2 cache misses were primarily accesses to this data structure; misses only occurred once per cache line, which resulted in high utilization.

The sensitivity of energy and performance efficiencies to cache size and associativity is shown in Figures 3 and 4. Efficiencies dropped with increasing cache size and rose slightly with increasing associativity. Larger cache sizes can increase utilization if application footprints fit in the cache. However, as cache sizes were increased further, the percentage of dead cycles also increased, which lowered efficiency. When associativity was increased, miss rates were lowered with a resultant drop in execution time that should have resulted in

improved efficiency. However, each dead cache line stayed longer in the cache as a consequence of the deeper LRU stack; deeper LRU stack means that the stack distance is larger, therefore dead memory lines stay longer in the cache. This compensated for any utilization gains. Therefore, performance efficiency remained flat with changes in associativity. Energy efficiency improved slightly with associativity and is explained by the switching energy increases and reductions in execution times. Cache line sizes also had a limited impact on efficiencies; efficiencies varied within 5% for a range of 128-byte to 512-byte line sizes.

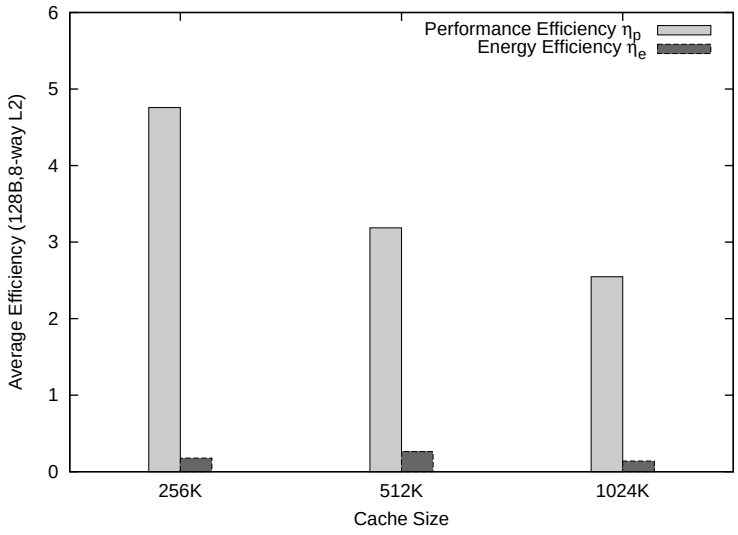


Figure 3. Sensitivity of efficiencies to cache size.

A major cause of cache inefficiency is the fixed manner in which main memory lines share the cache. This observation naturally leads to an approach where memory lines share cache resources in an application-aware manner, much in the same way that application data shares registers via compiler based register allocation techniques. The result of such an active approach to managing caches can be lower conflict misses (better performance efficiency) and smaller footprints with the ability to turn off unused lines (better energy efficiency). The primary source of inefficiency from an energy perspective is the low access frequency for the L2 cache. A detailed analysis of the reference behavior of benchmark

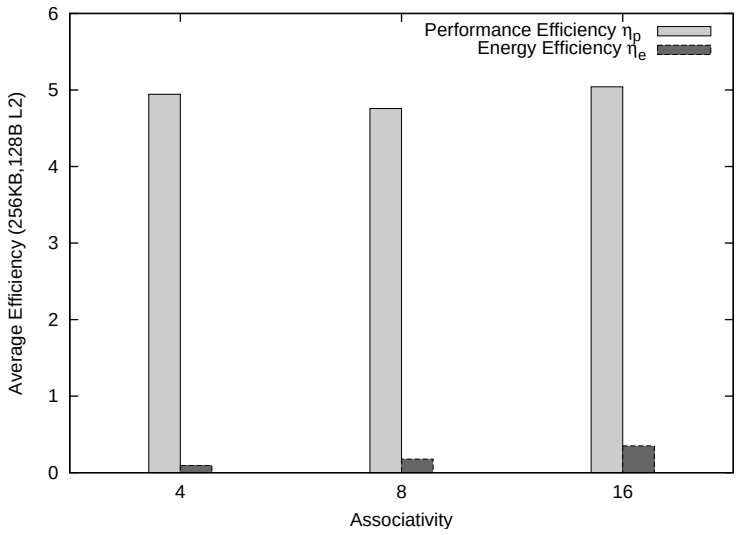


Figure 4. Sensitivity of efficiencies to cache associativity.

kernels [65] identifies inter-reference intervals to be tens of thousands of clock cycles. Millions of cache transistors remain powered up during these long intervals, which adds to leakage energy. The goal now is to dramatically scale back the cache size and re-map memory to the smaller cache. This scaling (up or down) should occur periodically to match the program reference behavior.

For other application domains, such as scientific computing and embedded processing, the utilization and efficiency values for traditional caches remained low. The Lawrence Livermore loops, the LINPACK100 benchmark, and several linear algebra kernels representative of computations used in scientific applications were studied; the utilization values for these benchmarks for a 256KB L2 cache were below 10%, with performance efficiencies below 5%, and energy efficiencies below 0.5%.

Finally, embedded processing benchmark kernels from the *Mibench* suite were also studied and their utilization values were low (< 20% with an 8 KB L1 cache with performance and energy efficiencies in the same low range as those of other application domains).

Customizing the cache design through sizing and shaping are required to maintain cache

efficiencies that are sustainable. Unlike registers or main memory that are software controlled, caches are typically not controlled substantively through software. Traditionally, compilers have been constrained to optimize for the fixed cache design. Active cache management offers compilers opportunities to both control the cache and optimize for the customized cache.

3.3 Efficiency Analysis of Multi-threaded Applications

The preceding discussion of the model and analysis has focused on single-threaded applications. One might expect an increase in cache utilization for multi-threaded applications with traditional caches as multiple threads share the cache. However, a detailed analysis showed that any increases were at best, marginal. The two overwhelming sources of inefficiency—leakage energy and the fixed manner of cache placement more than compensated for any improvements which was further exacerbated by interference among threads. While this dissertation only focusses on uniprocessor single-threaded and multi-threaded applications, the analysis techniques can be extended and applied to multi-core and multiprocessor domains as well.

The execution of various multi-threaded applications from the SPLASH suite [91] was simulated along with other parallel linear algebra routines to quantify cache efficiencies. The applications were simulated in single and multi-threaded modes. The SIMICS [89] full-system simulator configured for x86 processors running the Linux OS was used for the simulations.

Figure 5 shows that the cache utilization for a 512KB L2 cache averages 25% and that for a 32KB L1 cache averages 32%. Utilizations are not seen to vary significantly between single threaded and multi threaded executions. The low utilizations mean that the majority of the cache energy and area costs are spent in maintaining *dead* lines. Only very few benchmarks have good utilization, e.g., *mm* (matrix multiply) has 60% L1 utilization, and *watersp* has 50% L2 utilization. One reason that the L1 cache has a higher utilization

than the L2 cache is because the L2 is rarely probed for the unit-stride accesses. These numbers were not found to be substantially better than those studied in Section 3.2 in the single threaded domain using *simplescalar* with no operating system running. Thus, any advantages of multi-threading was compensated by the sources of inefficiency (poor sharing of cache lines and high leakage). The effect of the operating system on cache efficiencies is considered in Chapter 6 .

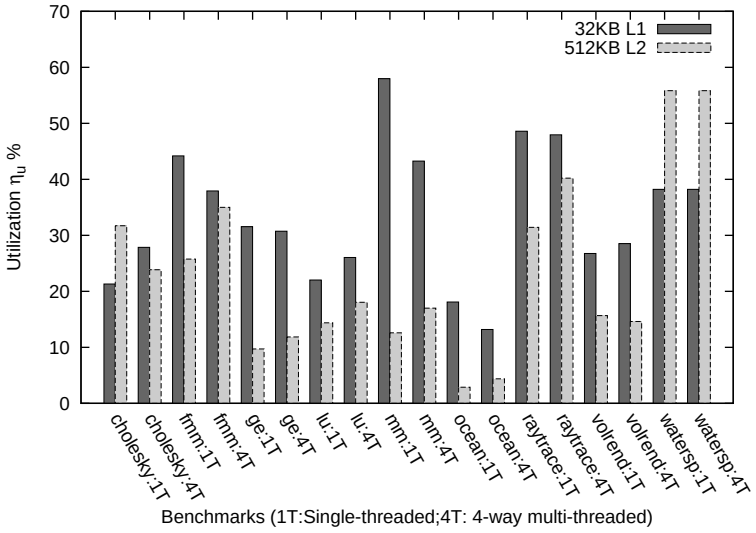


Figure 5. Utilization with modern caches

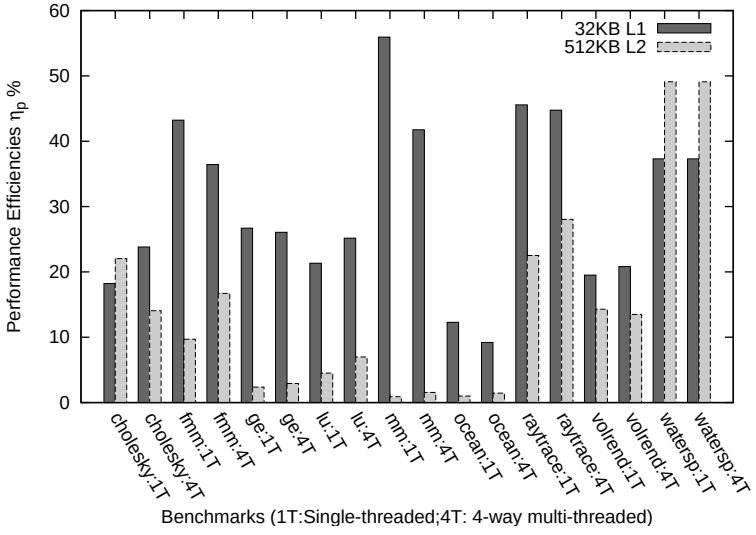


Figure 6. Performance efficiency with modern caches

Performance efficiency, for which the upper bound is utilization is shown in Figure 6

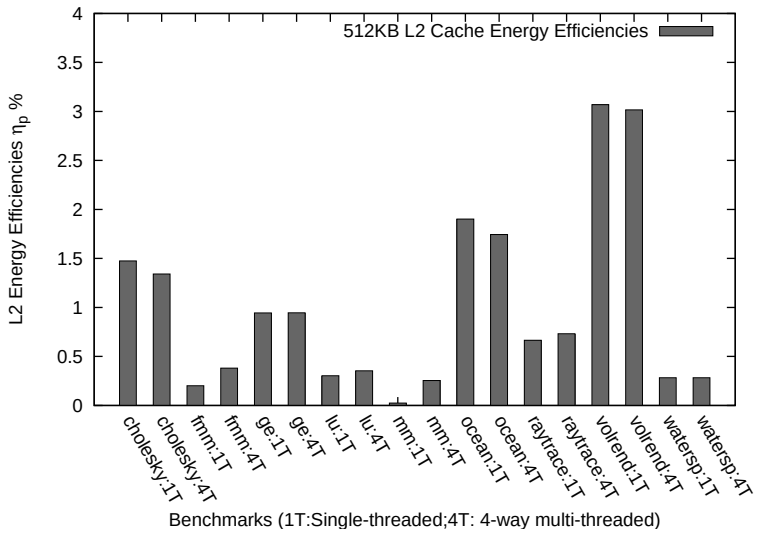


Figure 7. Energy efficiency with modern caches

and averages less than 15% for the L2 cache. Although, most of the applications had a low miss rate, performance efficiencies were still considerably lower than utilizations reflecting that the residency of live lines was not well exploited for performance. Cache energy efficiency with current designs averages under 1% (Figure 7) with leakage being the prime source of inefficiency.

When associativity is increased, miss rates are lowered, but dead cycle counts increase because of the deeper LRU stack. When the cache size is increased, it may decrease miss rates, but a larger number of lines are dead. Thus, utilizations remain flat with increasing cache size and associativity. Energy efficiency improves slightly with associativity due to switching energy increases and reductions in execution time.

Multi-threading can help improve utilization and efficiencies because of more accesses to live data (i.e, thread locality). However, efficiencies and utilization are not affected significantly; interference among threads, the dominance of leakage energy and poor sharing of cache lines across memory is found to override any threading related improvements in traditional caches.

To summarize, efficiencies and utilizations are low for multi-threaded applications with traditional caches as leakage energy and poor sharing of cache resources among memory

lines overwhelm any effects of thread locality. Active management schemes however, due to their application memory behavior centric approach are able to decrease these inefficiencies leading to better cache efficiencies as demonstrated in this dissertation.

3.4 Improving Efficiency

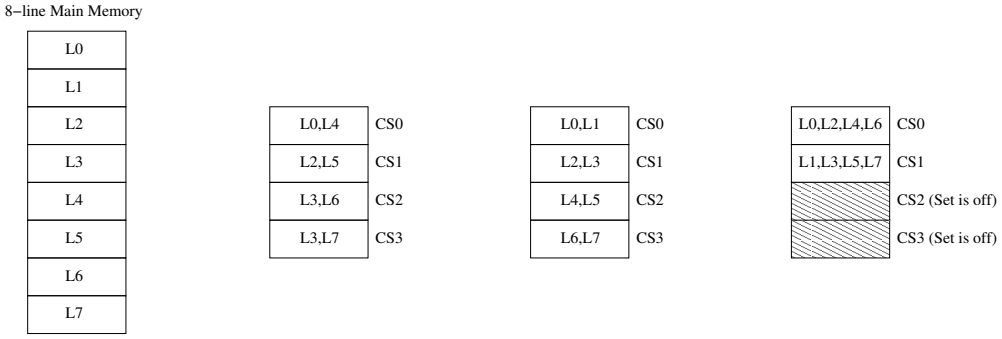
The sharing of cache resources by main memory lines is represented by the construction of *conflict sets*. A *conflict set* is the set of main memory lines that is mapped to a cache set. Conflict sets in modern caches are constructed using *modulo* placement. Using modulo placement, a memory line at line address L is placed in the cache set L mod S, assuming that the cache has S sets. For an eight line memory shown in Figure 8(a), the placement policy used in traditional caches is illustrated in Figure 8(b).

Traditional cache designs assume that a uniform distribution of references across cache sets increases hit rates. Therefore, the modulo placement strategy tries to create a uniform distribution of references and misses across conflict sets. This assumption is not true for many classes of applications. Furthermore, improved hit rates do not necessarily translate to efficient caches. For example, larger cache sizes can lead to higher hit rates but lower efficiency.

For example, assume accesses to a matrix where the data is accessed with a constant stride. Many conflict sets will be sparsely accessed leading to low energy efficiency, and some conflict sets will be densely accessed leading to low performance efficiency (as a result of the increased conflict misses). Additionally, if the application memory footprint is smaller than the cache size, many cache sets will remain unused. Unused cache sets contribute to low energy efficiency.

The key to improving cache efficiencies is constructing conflict sets customized to the application memory access pattern (shaping) while minimizing the footprint of the application in the cache (sizing). For improving efficiency, the notion of liveness of program variables can be extended to main memory lines. If any of the variables resident in the memory

line are live, a memory line is *live*. For example, memory lines with non-overlapping live ranges can be mapped to the same cache set without increasing conflict misses: this results in a merging of cconflict sets, a technique called *folding*. Some examples of using cache placement to improve performance and energy efficiencies are shown in Figures 8(c) and 8(d).



(a) Eight line main memory

(b) Modulo placement

(c) Customized placement

(d) Miss Folding

Figure 8. Conflict set construction

Informally, one can associate a set of live memory lines with a loop nest. If the entire set of live memory lines is resident in a portion of the cache during the execution of a procedure or a function, all the remaining cache lines can be powered off. Ideally, the cache sizing implementation should track the size of the set of live memory lines at any point during program execution. The cache can be appropriately sized to host this set of live memory lines, and the placement function can be defined to distribute the memory lines across the *sized* cache (to minimize conflict misses). The remaining cache lines or sets (beyond the sized active-portion) can be turned off to save energy.

On exiting the loop nest, the cache may be reset to its original configuration; it may also be resized for the next loop nest. The success of this approach is dependent on the application domain. For example, the memory behavior of scientific applications operating on multidimensional arrays has been well studied and characterized; this knowledge can be used to develop cache sizing schemes and implementations to improve cache efficiency.

For applications which lack realistic profile data, heuristics can be used to resize the cache. These heuristics seek to track the live ranges of memory lines as by the generational model of caches [92].

Alternatively, conflict sets can be combined to form larger conflict sets—a technique referred to as *miss folding*. This reduces the total number of conflict sets, allowing cache sets to be powered down. An increase in the number of misses may be balanced against improved energy savings. An example of miss folding, wherein four conflicts sets that originally existed are folded to two (accompanied by two cache sets that are turned off) for energy savings is illustrated in Figure 8(d). For an array access with unit stride or a matrix accessed in row-major order exactly once, all misses are compulsory with a traditional cache. If all live memory lines are mapped to form a single conflict set, it results in the same number of misses; conflict misses are converted to compulsory misses.

For many applications, domain specific knowledge or profile data may be absent. Analysis of these applications also proved that efficiencies and utilizations are exteremely low across the board. For such applications, to increase the efficiencies, runtime heuristics are used in this research which help characterize memory reference behavior across conflict sets and program phases. This information is collected at runtime while the application is executing and drives sizing and shaping algorithms which adaptively customize the cache placement based on memory access behavior. Miss folding and powering cache sets off can also be applied in these domains to increase energy efficiency, while improving the sharing of cache lines among main memory lines to improve performance efficiency. This thesis exploits domain specific knowledge and runtime characterization to improve efficiencies through customized placement for single-threaded and multi-threaded applications across multiple domains at multiple data points using available program characterization inputs.

CHAPTER 4

SYSTEM MODEL

Managing the cache resources in an active manner improves the sharing of cache resources among main memory lines using various strategies relying on concepts including conflict sets, liveness, and footprints to essentially size and shape the cache. Additionally, these concepts can be applied at different points in program execution with the placement being tuned with memory behavior collected using varying methods such as profiles, runtime measurements, static analysis etc. Finally, the complexity of hardware implementation for customized placement varies with the flexibility desired; more flexible the placement, more complex the hardware required.

This chapter first outlines the concepts of sizing and shaping and draws on the notion of program memory footprint to explain the concepts of customized placement. This is followed by presenting the scope of optimizations that are possible using the vehicle of customized placement, followed by describing a traditional cache implementation as the baseline. Then the operational model of customized placement is described with a general architecture model.

4.1 Programmable Placement

Traditional caches can be viewed as being passively managed, i.e., the set of memory lines mapping to a cache set (a *conflict set*, Section 3.4), is fixed at design time as is the replacement policy. This structure is one of the sources of inefficiency as it must accommodate large variations in program memory reference behavior. This research employs active management strategies to improve the sharing of cache resources among main memory lines, thereby leading to improved efficiencies.

The active management vehicles employed are by *sizing* and *shaping* the cache. *Sizing* the cache matches the size of the cache that is *active*, i.e., powered *ON*, to the program

memory footprint. *Shaping* the cache refers to the policy by which the program memory footprint shares a region in the cache, thereby permitting the remainder of the cache to be powered down for energy savings. Cache shaping can also increase cache performance by mapping the program memory footprint onto the cache in an optimized manner. Cache sizing and shaping provides fault tolerance naturally—the faulty cache sets correspond to cache sets that are turned off. Figure 9 illustrates the principles of cache sizing and cache shaping over a program region. The idea is sizing the cache such that it meets the memory reference requirements of the program during each phase of execution. Following the sizing step, the cache is shaped to minimize performance degradation due to conflict misses by constructing conflict sets in an intelligent program and memory behavior aware manner.

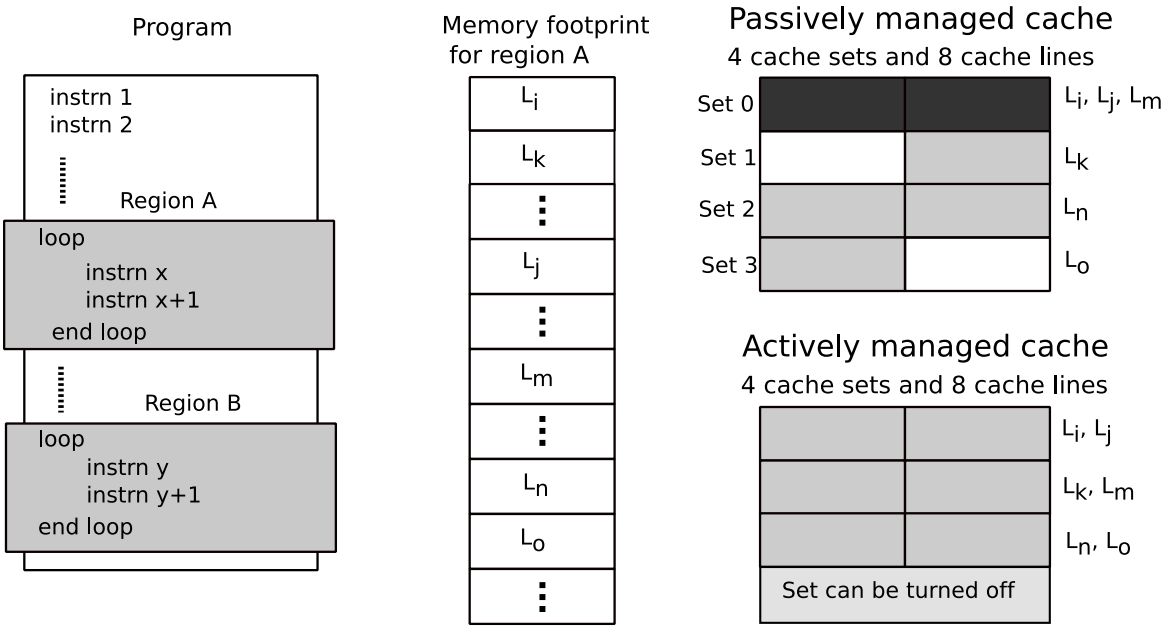


Figure 9. Cache sizing and shaping.

Customized cache placement shapes the cache by reconstructing *conflict sets* (Section 3.4). Customizing the cache placement reconfigures the mapping of main memory lines to cache sets; i.e., a memory line that was originally mapped to cache set X using modulo placement, can now be mapped to any of the cache sets, only limited by the number of cache sets that are available. Programmable placement refers to keeping the cache

placement policy software controlled. Programmable placement places control of the cache placement structure in the hands of the compiler, the programmer, or the runtime. Such flexibility also permits dynamic trade-offs between performance and energy. For example, if it is deemed that the compiler sacrifice performance for energy, the placement can be programmed such that more cache sets are maintained in an off state improving energy performance while potentially sacrificing execution time performance.

4.2 Scope of Optimizations

The optimization problem that this research targets can be formulated as follows: For a cache with S sets ($s_0, s_1, \ldots s_{S-1}$) and M main memory lines ($L_0, L_1 \ldots L_{M-1}$), compute a placement function, $placement(L_i) = S_j$ such that cache utilization/efficiency is maximized for a reference stream ($r_0, r_1, \ldots$).

In the most general case, a memory line can be mapped to any set in the cache. The difficulty with this approach lies in the problem size (mapping 10^6–10^9 lines) and the complexity of the resulting address decoding circuitry. A range of approaches to placement policies from fully customized placement to fixed placement strategies are shown in Figure 10

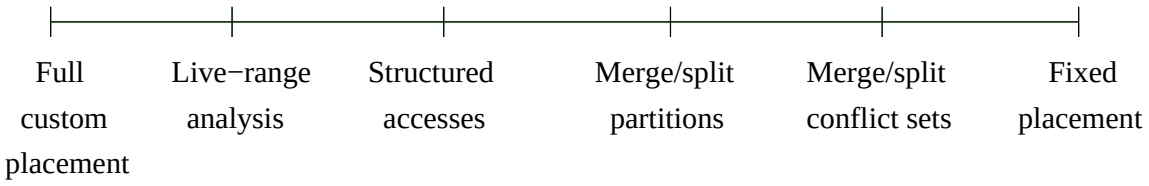


Figure 10. Range of optimization possibilities.

On one end is full customization, i.e., each memory line is mapped individually to a cache set, while at the other end of the spectrum is the fixed cache placement currently used in traditional cache hierarchies. Techniques proposed here fall in various points within the two extremes. The more customizable and flexible a solution is (i.e., moving towards the left end of the spectrum), the greater the cost is in terms of hardware support. Using a

particular solution depends on the application domain.

4.3 A Representative Traditional Cache Implementation

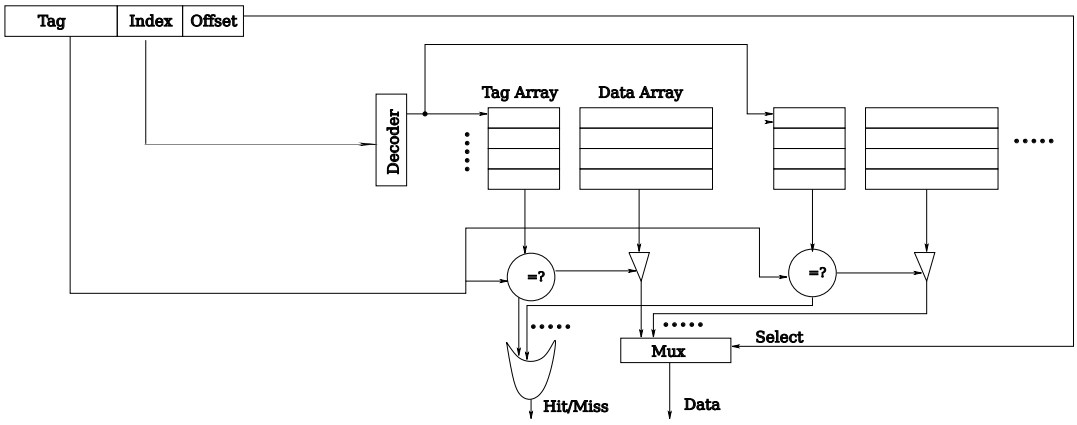


Figure 11. Base Cache

The traditional cache architecture shown in Figure 11 is used as the baseline in this thesis. Any configuration changes to this implementation of a traditional cache will be outlined in the performance analysis sections of various chapters. The memory address is split, as usual, into a tag field, an index field and a line-offset field. The exact configuration of the base cache in terms of size, associativity and block size varies with the particular domain under discussion (for example, smaller cache sizes are selected for embedded systems and larger caches for scientific computing) and are provided in the performance evaluation sections of the varies studies included in this dissertation. During a cache access, the address is decoded into the three fields as shown. The index field is decoded to access the data from the multiple data arrays (the number of data arrays equal the associativity). The tags are compared to the tag from the address field. If a tag matches, the cache access results in a hit and the corresponding data array is read out. It is also assumed that the tags and data arrays are accessed in parallel.

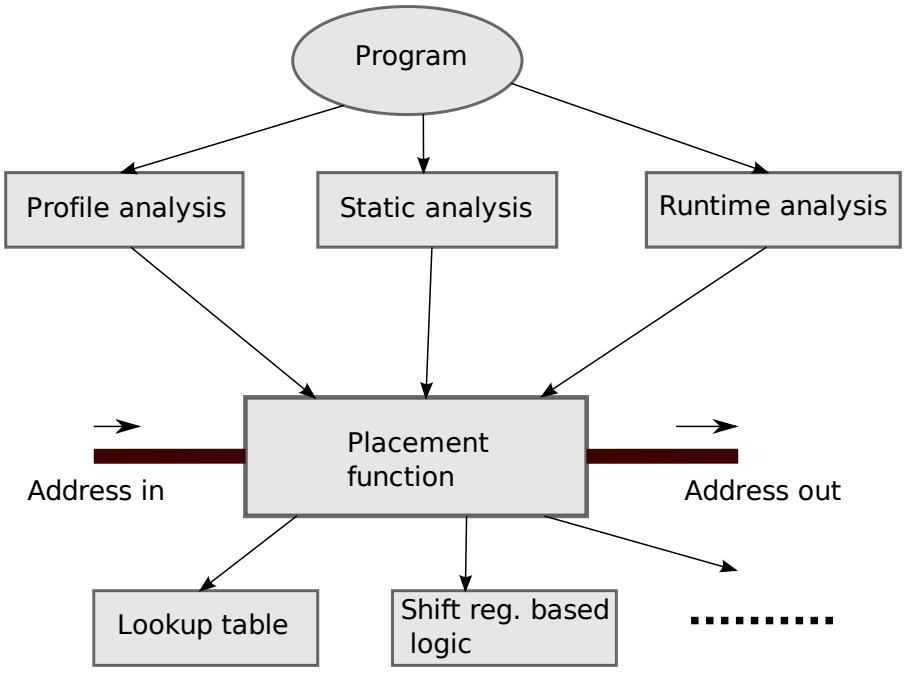


Figure 12. Operational Model

4.4 Operational Model

The operation model for active management is shown in Figure 12.

Several inputs can be used to synthesize optimized placement functions to improve efficiencies. Examples include program profiles, static program analysis, program liveness characteristics, strided memory access patterns, conflict set reference behavior, etc. Similarly, the placement strategy can be applied at multiple program life-cycle points; it can be applied as: i) a one-time (per application) reconfiguration technique which is useful for tolerating manufacturing defects, ii) a compile-time technique applied at specific stages during program execution, or, iii) a dynamic run-time mechanism.

The placement function decodes an incoming cache set address to a new cache set address using the customization function selected. This effectively changes the mapping from main memory to the cache by allowing for more optimal placement functions than the modulo placement function employed in traditional caches.

Hardware implementations for applying customized placement may vary, with the hardware complexity being proportional to the degree of customization desired. General, less flexible implementations of customization can use simpler hardware like logic based on shift-registers, whereas more flexible manifestations of customization would require relatively complex hardware such as lookup tables.

This thesis focuses on solutions in profile-driven, static-analysis based and runtime measurement driven categories and identifies feasible hardware solutions with varying complexity targeted to the domain under consideration.

4.5 Architecture Model

The architecture model shown in Figure 13 is a general model and is limited to uniprocessors, and is modified according to the specific domain or implementation pursued. For example, the customized placement in embedded processors is assumed to be implemented for the L1 cache which frequently have only a single level cache, whereas it is implemented for the L2 cache in architecture with L1 and L2 caches (this is shown in Figure 13).

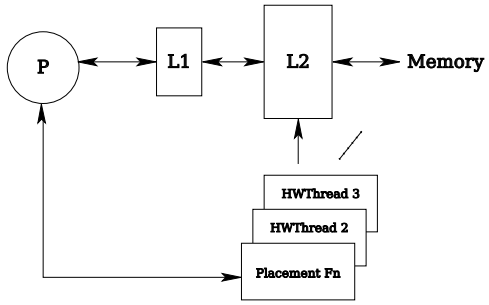


Figure 13. Architecture Model

For extension to the multi-threaded domain, with multiple software threads, the placement function state will have to be saved along with the thread state on context switches. For hardware threads (simultaneous multi-threading), it is assumed that there are separate placement function hardware (for example, multiple lookup tables) for each hardware thread which maintains the cache placement information on a per-thread basis.

4.6 Compilation Model

Program execution evolves through *phases* [93] wherein each phase is characterized by a working set of memory references. Within a program phase, memory reference behavior exhibits spatial and temporal locality around a set of memory locations.

The general compilation model adopted in this dissertation has program regions preceded by operations that modify the cache placement function with the custom placement invoked in hardware only during the execution of these regions. Thus, after the completion of the execution of one such phase where customized placement was invoked, one can modify the placement to a new customized placement for the upcoming phase or revert to traditional modulo placement. For certain applications, such as embedded processing applications, the compilation model is modified to be applied on a per kernel or per application basis. For example, using off-line profile analysis an optimized customized placement solution is identified and invoked through the duration of the kernel or application. For larger applications, customized placement is applied selectively at certain program points, either statically through program analysis or dynamically based on runtime measurements.

4.7 Applying Placement Directives

For applying the placement funtion offline for embedded processors, the off-line profile data is collected and used as input to generate a customized placement function. Thus, post-manufacturing or prior to program execution, the new placement function is enabled for improved performance.

If compile-time data is used to customized the placement function, it is applied by the compiler as a placement directive. The compiler uses static analysis of the program or dynamic optimization data to drive the placement function.

For run-time adaptation, specialized hardware is used to capture metrics used to drive the computing of placement. A separate thread of execution is invoked every few million cycles to probe the hardware and recompute the placement function.

In each of the above cases, the provision is also maintained for the programmer to manually insert the placement function. This is particularly useful if the programmer needs to customize the placement for particular power or performance demands.

The actual address translation (that is, the placement function) is performed by dedicated hardware mechanisms irrespective of how the placement function is computed or invoked. Separate hardware decoding schemes customized to specific domains are presented in this thesis.

The remainder of this thesis describes the various solutions that were proposed for different domains to improve the efficiency of caches. The chapters are subdivided into solutions that deal with off-line profile driven strategies followed by those applying compile-time strategies and strategies that are applied at runtime using runtime measurements.

CHAPTER 5

OFF-LINE STRATEGIES

This chapter identifies several active management solutions in the context of improving cache performance using off-line profile driven strategies, that are applied once per application or kernel. These strategies which reconfigure the cache on a per application basis are well suited for domains such as embedded processing, since embedded processors typically execute a fixed set of applications, that are well understood through program profiles. Section 5.1 adapts the concepts of cache shaping for network processors, arguing for including a small single level cache hierarchy in network processors; a program specific adaptation exploiting the knowledge of applications that execute in network processors. This is followed by a general application of cache shaping through customized placement for embedded processors using profile data in Section 5.2. Section 5.3 applies active management techniques for promoting fault tolerance in embedded processors, again using profile data aimed at limiting performance degradation as the percentage of usable cache size decreases because of defects.

5.1 Data Trace Cache

This section outlines techniques to address the memory bottleneck problem for network processors with the goal of minimizing off-chip memory demand and average memory access latency by the use of a small application specific compiler-visible data trace cache. The key insight is to provide small caches that service accesses to key data structures that account for the bulk or a significant portion of the data accesses. This section this technique is applied to network processors where a significant portion of the memory accesses are performed to tree data structures that implement the routing table. Accesses to tree data structures contribute significantly (greater than 75% of memory references) to the memory traffic in packet processing applications. A tree access creates a simple to characterize

trace of memory references and the data trace cache proposed herein exploits the locality of reference in these data traces to improve performance.

These application specific data trace caches outperform conventional caches in off-chip bandwidth demand and miss rates by up to 50% for accesses to rooted tree data structures at small (256–1024 bytes) cache sizes. This is found to translate to a 30% reduction in average memory access time (AMAT) for the entire application kernel, i.e., route lookup. Such caches are philosophically in the same category as victim caches, stream buffers, and pre-fetch buffers in that relatively small investments in silicon realizes substantive reduction in off-chip memory bandwidth demand resulting in improved performance.

5.1.1 Caching for Network Processors

In network processors, a cache hierarchy is normally absent[94]. This is due to the belief that packet data has little re-use, and therefore, there is no reference locality across packets. However, it is found that caching the application data structure alone (in this case, the routing table tree or packet classification tree) using the data trace cache can yield significant performance benefits. The approach described in this section takes a close look at the application data structures for an important group of kernels: those with large rooted tree data structures.

If a tree data structure access requires a traversal from the root to a *leaf node* (i.e., a node which has no children) and is partitioned into tree node sets with each set consisting of nodes at the same height, it is observed that a tree access is characterized by the following properties: i) a tree node is accessed from each set, and, ii) no more that one tree node in each set is accessed. Thus, a tree access generates a trace of accesses to relatively few tree nodes, or few memory addresses, compared to the size of the entire tree data structure. Even if traversal to the leaves are not necessary (i.e., traversal can stop at an intermediate node with children), the tree access still generates some references. The nodes closer to the root are always accessed more often than those closer to the leaves. Thus, with respect to the memory footprint of the tree as whole, memory access patterns are skewed in that some

regions of memory consistently accessed more often than others.

Many applications often query large data structures where the processing of the query leads to highly skewed access patterns. Characterizing these access patterns can lead to performance improvements by combining compiler optimizations (data re-layout for example) and cache design. This philosophy is applied to large rooted trees. The resulting cache design is called a *data trace cache* (DTC). The DTC design utilizes a flexible placement policy, distinguished from the fixed application independent placement policies of traditional cache architectures. Only memory accesses to the application tree data structures are processed by the DTC. Non-tree data structure references are handled by a conventional memory hierarchy that operates in parallel with the DTC.

This work can be easily extended to packet classification algorithms [95, 96] which employ search trees. Most of the literature on network processor caches rely on caching recently accessed routing table entries [97, 98, 99]. For example, Baer *et al.*[100] suggests a scheme for compressing the routing table data structure such that it can be more effectively cached. Unlike the approach described here, these schemes do not exploit access patterns in the routing table data structure to improve locality.

5.1.2 Characterizing Memory Reference Behavior

This section presents a characterization of the reference behavior of accesses to tree data structures which drives the design of the DTC. There are numerous applications of search trees, ranging from routing table lookup and packet classification in network processors to indexing databases, dictionaries and file systems.

5.1.2.1 Tree Data Structure MAPs

Figure 14 illustrates a binary search tree data structure. A tree access involves traversing the tree from the root to a leaf based on matching a key or other values in each intermediate node. If the tree is complete, and assuming any leaf node is equally likely to be accessed during a search operation, the probability with which a node at level i is accessed during

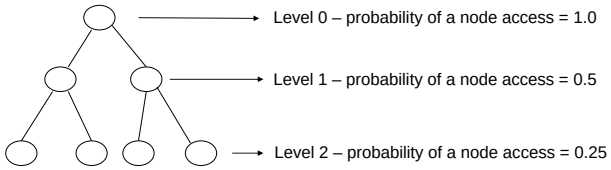


Figure 14. A Binary Search Tree

a tree access is $1/2^i$. There is significant reuse of nodes close to the root, whereas nodes close to the leaves are not accessed as often. Only one node at each level is accessed during a tree traversal. For example, if one considers a tree with 24 levels there can be more than 1 million nodes at a level of the tree with each traversal accessing one of these nodes. This suggests that a relatively small number of cache lines devoted to these 1 million nodes will suffice with respect to capturing any reference locality in accesses to this level of the tree. *The cache placement policy is the vehicle for determining relationship between memory locations and cache locations. A flexible, configurable placement policy can take advantage of such lack of locality in references to specific (large) regions of data structures.* While trees may not be balanced or be necessarily binary, the preceding observations concerning locality generally apply.

5.1.2.2 Exploiting Reference Locality in Tree Data Structures

Each tree traversal produces a number of accesses: a balanced binary tree with 4 billion nodes will have at most 31 nodes. Thus, a trace of tree access references will access a maximum of 31 levels. If two memory accesses per node is assumed, this represents a trace of 62 memory addresses. This sequence of memory addresses is the *memory footprint*. Application tree data structures may be very large (>100000 nodes) but footprints are small (50–60 memory references). Distinct footprints tend to have common references at levels close to the tree root and footprints are the same for accesses to the same leaf in a tree—these observations are exploited for performance benefits.

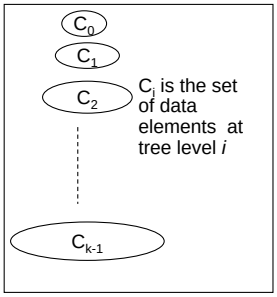


Figure 15. A n-ary Tree Data Structure

Figure 15 captures another way of visualizing tree data structure memory access patterns. The application data structure in this example is a complete n-ary search tree. The set C_i represents the data nodes at tree level i. C_0 will have one node, whereas C_1 will have n nodes, C_2 will have n^2 nodes and C_i will have n^i nodes. Each tree access will reference exactly one node in each C_i. Also, $C_i s$ are accessed in ascending order and the cardinality of the sets increase with level.

Memory is partitioned into conflict sets. In traditional caches, each conflict set has the same number of entries. The key idea is to have a placement policy where all nodes of C_i are grouped into a conflict set, i.e., the cardinality of each conflict set is different.

5.1.3 Architecture and Design

5.1.3.1 Data Trace Cache: Principle

The placement policy used in traditional cache architectures assigns memory line L to cache set *L mod S* where there are S sets in the cache. Thus S contiguous lines in memory are mapped to S distinct sets in the cache exploiting spatial reference locality. The placement policy determines membership of a memory line in a conflict set. Existing placement policies create conflict sets of equal size, i.e., the same number of memory lines map to each set. Successive references to lines in a conflict set can create conflict misses in the cache.

For a tree data structure, a traditional cache allows the higher level nodes in the tree which are accessed more frequently to conflict with lower level tree nodes that are accessed rarely which leads to many misses. The DTC is designed to exploit locality in footprints.

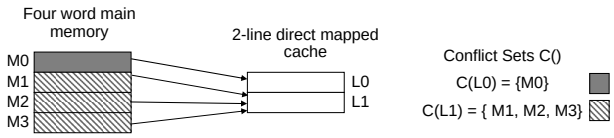


Figure 16. Data Trace Cache Principle

This is achieved by redefining the placement policy such that (ideally) all nodes at a level in the tree reside in the same conflict set, and conflict sets are allocated across sets in the cache. Thus, successive memory references in a tree access will reside in distinct sets in the cache. If the cache is large enough to hold a few complete traces, accesses to memory can be lowered.

As illustrated in Figure 16, fixed conflict sets are replaced by an allocation of conflict sets via a flexible placement policy. In Figure 16, the conflict set sizes increase with the level of the tree data structure. Additionally, one conflict set may be allocated to multiple sets in the cache. The practical difficulty with arbitrary conflict set resolution is in the run-time indexing of the DTC directory, thereby constraining the actual placement policies employed.

5.1.3.2 System Architecture

The DTC is primarily targeted towards embedded systems which have a relatively small sized L1 cache (1KB-4KB) with no L2 cache. A small sized DTC (512–1K bytes) in addition to a primary L1 cache significantly outperforms an L1 cache of the same magnitude.

The architecture employed for using a DTC is shown in Figure 17. The DTC is employed parallel to the L1 cache, and every memory address passes through both caches. Since the data contained in L1 and the DTC are mutually exclusive, there will be less pollution of non tree data in L1 and can lead to better miss rates for L1. The performance evaluation is limited to that between a traditional L1 and a DTC making the reported improvements conservative; if the improvements in L1 miss rate due to lower pollution were also factored in, the results will have higher improvements.

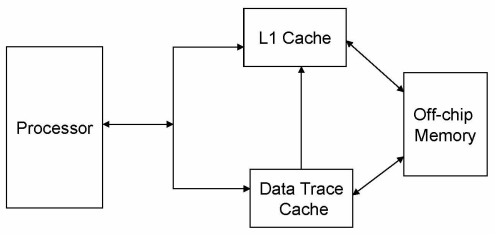


Figure 17. Data Trace Cache: System Architecture

The architecture model consists of a multi-threaded processor with a cache hierarchy. Each thread is assumed to execute the same code fragment (thus replication of the placement hardware is not required). Additionally, the code fragment is assumed to access a large application data structure (e.g., indexing a binary search tree) repetitively. Multi-threading follows a switch on miss strategy, i.e., thread switching occurs on a cache miss. This strategy is commonly implemented in network processors [94].

Since the L1 and the DTC operate in parallel, the additional overhead involved in indexing the DTC would not affect cache performance on a hit on L1. A miss in L1 indicates that the reference may be a tree reference that may or may not be present in the DTC, or is a non tree data reference that missed in L1. The L1 cache proceeds on a memory fetch only if it is a non tree data reference. This is indicated using a flag that is set by the DTC (set in a couple of cycles within accessing the DTC) indicating whether the incoming reference was a tree data reference. If it is a hit in the DTC, no memory fetch occurs. If, on the other hand it is a miss in the DTC, a memory fetch is initiated by the trace cache. Thus, the memory reference is brought into the L1 cache or the DTC, depending on which cache initiated the memory fetch. This architectural setup ensures in most applications, that, even if additional cycles are consumed while indexing the DTC, the average memory latency is relatively unaffected, since the L1 is not delayed more than a few cycles on an L1 miss. Whereas, on a DTC hit, execution cycles can be saved by not going to off-chip memory.

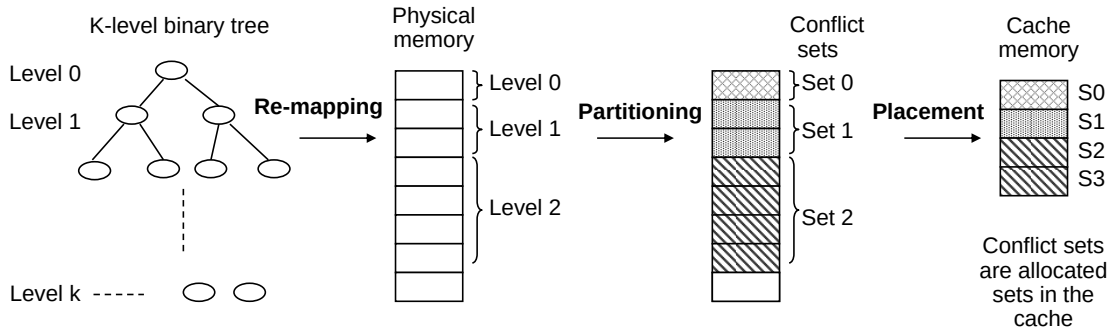


Figure 18. Methodology

5.1.3.3 Design

To exploit the reference locality in tree data structures, the rooted tree is statically remapped and augmented with a cache that employs a placement policy that is tailored to the remapped tree and, to the tree access patterns. Figure 18 outlines the strategy steps:

1. Remap the tree data structure so that all nodes occupy a contiguous region of memory
2. Partition this memory region into conflict sets
3. Assign or place conflict sets to sets in the cache

The goal is to find combinations of remapping, partitioning and placement schemes that collectively realize lower miss rates. Traditional caches create conflict sets guided by expectations of spatial locality in the reference stream, e.g., successive lines in memory are mapped to distinct cache sets. In principle, the static remapping of the tree data structures can be such that nodes at a level can be relocated to memory locations that are in such a conflict set. This would simplify indexing into the cache but would leave "holes" in memory. Such an approach which leads to memory wastage is eschewed in favor of greater control over the relationship between remapping and partitioning steps.

5.1.3.4 Remapping

The prescribed approach exploits tree data structure access characteristics for reductions in cache miss rate via flexible placement policies. To gain control over the placement policy for tree nodes in the cache, the tree is remapped into a contiguous region of memory. The remapping can also improve the spatial locality of nodes closer to the root that have higher probabilities of access when traversing the tree. Any existing remapping algorithm can be employed [53, 51]. Even a simple remapping of the tree into a contiguous segment of memory using a breadth-first traversal of the tree was found to be sufficient for performance gains.

5.1.3.5 Partitioning and Placement

After the tree is remapped to occupy a contiguous segment of memory as illustrated in Figure 18, the next step defines conflict sets (partitioning) and allocates (placement) sets in the cache to each conflict set. Run-time indexing of the cache (i.e., which is determined by the placement policy) is kept simple and fast by restricting placement choices. Additionally, sufficient cache resources is provisioned to store a few footprints.

For partitioning, a simple power of two assignment is chosen as shown in Figure 19. The first two conflict sets will have two nodes each, followed by a conflict set with four nodes, then eight nodes and so on; this approach is tailored for a balanced binary tree data structure. The partitioning can be performed independent of the placement function, but for optimizing performance, the placement function has to be tailored to the partitioning algorithm.

The inputs to the placement algorithm are the number of memory lines that were remapped, the total number of cache sets, and, the cache line size. The partitioning and placement algorithm used is shown in Figure 19. The algorithm outputs a set of masks which are the offsets of the first cache set allocated for each conflict set. These values are used for runtime decoding of incoming memory addresses. Additionally, the algorithm also returns a DTC code that is used to identify whether a given memory address is cached in

Input: $Num\_memory\_lines, Num_cache\_sets, Cache\_LS(linesize)$
Output: $Mask[], DTC_code$

```
1: intCS, Elements[], Mask[], pn, i
2: Num_CS = 1; CSG = 0
3: for i = 2 to i < Num_memory_lines do
4:    NUM_CSG+ = 1; Elements[CSG] = i {Power of two partitioning of conflict sets}
5: end for
6: {Allocation of Cache sets to conflict sets}
7: pn = Num_Cache_Sets/Num_CS {Initialize packing number}
8: for i = 0 to i < Num_CS do
9:    if Elements[CS] > pn then
10:      Cache_Sets[CS] = Elements[CS]/Cache_LS
11:   else
12:      Cache_Sets[CS] = pn
13:   end if
14:   Remaining_Cache_Sets− = Cache_Sets[CS]
15: end for
16: while Remaining_Cache_Sets > 0 do
17:   pn∗ = 2
18:   for i = 0 to i <= Num_CS do
19:      if Cache_Sets[CS] == Elements[CSG]/Cache_LS then
20:         break {Do nothing}
21:      else
22:         if Remaining_Cache_Sets > pn/2 then
23:            Cache_Sets[CS]+ = pn/2
24:            Remaining_Cache_Sets[CS]− = pn/2
25:         else
26:            Cache_Sets[CS]+ = Remaining_Cache_Sets
27:            Remaining_Cache_Sets[CS] = 0
28:         end if
29:      end if
30:   end for
31: end while
32: for i = 0 to i <= Num_CS do
33:   if i == 0 then
34:      Mask[i] = Cache_Sets[i] − 1
35:   else
36:      Mask[i]+ = Mask[i − 1] + Cache_Sets[i]
37:   end if
38: end for
39: DTC_code = binary_complement(Num_CSG)
40: return DTC_code, Mask[]
```

Figure 19. Algorithm for Allocating Cache Sets to Conflict Sets

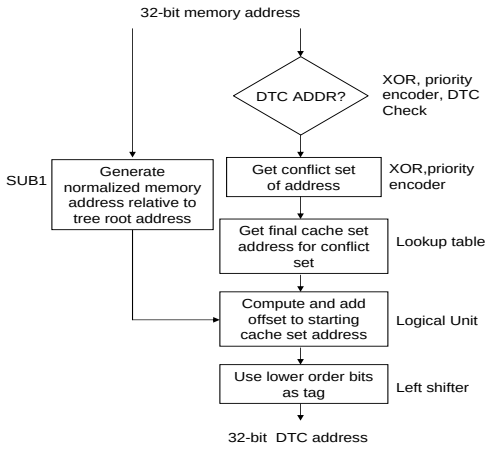


Figure 20. Flowchart for Indexing the DTC

the DTC or not. The placement algorithm initially allocates an equal number of cache sets to each conflict set ;the algorithm assumes that there are at least as many cache sets as there are conflict sets. If this assumption is not true, one can double the number of levels (new conflict sets) per cache set. Following which, a greedy distribution of cache sets to conflict sets is performed. The mask values returned by the placement algorithm are stored in a lookup table. There will be as many entries in the lookup table as there are conflict sets. Since for a 32 bit address, the maximum number of conflict sets possible is 32, a maximum of 32 entries can be stored in the lookup table.

Cache indexing in the case of a traditional cache is simple as all conflict sets are of the same size and a modulo approach is sufficient. For the DTC, however, due to cache sets having varying conflict set sizes, a different approach is required. An overview of the runtime decoding scheme is shown in Figure 20. Figure 21 depicts the implementation of the scheme for a given 32-bit byte addressed memory address.

First, the runtime indexing identifies a given memory address as belonging to the DTC or not. The DTC check block performs this function along with the priority encoder and the XOR gate. Next, the conflict set to which the incoming address belongs is identified using a priority encoder. The priority encoder indexes the lookup table to obtain the cache set and the cache line to which the incoming address is mapped, i.e., the placement information for

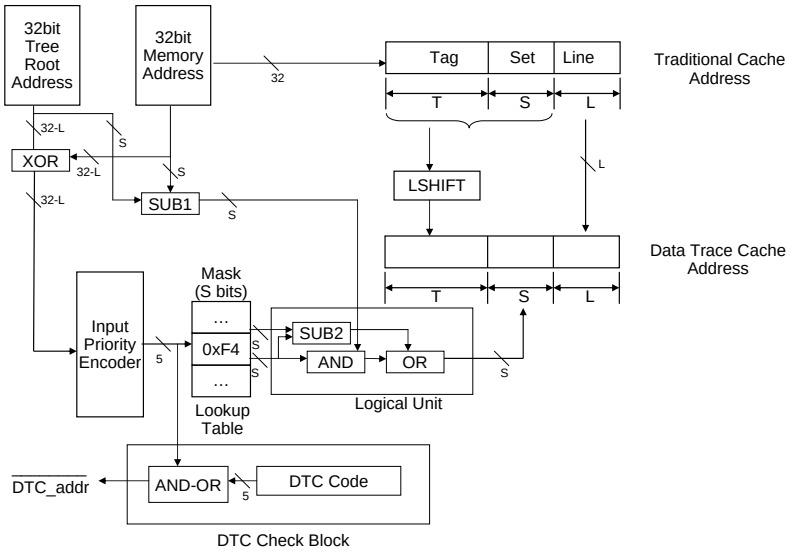


Figure 21. Implementation of Runtime decoding for the DTC

the particular address. This is simple enough if there are only as many cache sets as there are conflict sets. Since one conflict set could point to multiple cache sets, the placement information has to be determined. The cache set in which a particular memory address may be located is a function of the number of cache sets allocated for the particular conflict set (computed using a SUB block), and, the cache set offset of the address (an XOR block provides this). A logical unit as shown in Figure 21 is employed to perform this function.

Since application specific placement in actual hardware is realized by manipulating the lower order address bits (so that each conflict set has a different size), using the higher order bits as a tag in a conventional cache is no longer sufficient to uniquely identify a memory address. Hence, the lower order bits starting from the set address are used as the tag for the DTC which is achieved using a left shift operation as shown in Figure 21.

For incomplete, unbalanced or *n*-ary trees, the tree is remapped using a breadth first traversal as earlier. The partitioning of memory into sets is performed, again assuming a

Table 1. Description of Benchmarks

Benchmark	Description	Tree Data Structure Size
bin	Synthetic application implementing a parametric balanced binary tree	1MB
rtr	Radix tree based routing table lookup benchmark from CommBench [101]	1MB
avl	AVLtree implementation. AVLtrees are commonly used in databases and filesystems	64KB
aa	AAtree implementation. AAtrees are commonly used in databases and filesystems	128KB
treap	Tree heap implementation. Tree heaps are commonly used in databases and filesystems	128KB

complete binary tree (i.e., the same power of two partitioning described earlier is followed). This assumption allows the runtime indexing to remain fast and practical, and handles any general case. However, the proposed design is best suited for a complete binary tree.

5.1.4 Performance Evaluation

A brief explanation of the benchmarks used to evaluate the DTC is given in Table 1. The evaluation infrastructure built for the study is depicted in Figure 22. For simulating the DTC, the tree data structure is statically remapped using a breadth first traversal placing tree nodes in contiguous locations in memory. The application kernels are then compiled using *gcc* (version 3.2.2) and the data memory address traces generated using the *valgrind*[102] profiling tool. This address trace is filtered to remove non tree data structure accesses (stack memory accesses for example). The address trace is processed to apply the cache decoding scheme and produce a trace of directory entries and tags to drive dineroIV[103] cache simulator, which was modified to support multi-threading. The cache parameters are also fed into an area estimator, *Cacti 3.2* [104].

A conservative miss penalty of 100 cycles was assumed in the simulations (200–300

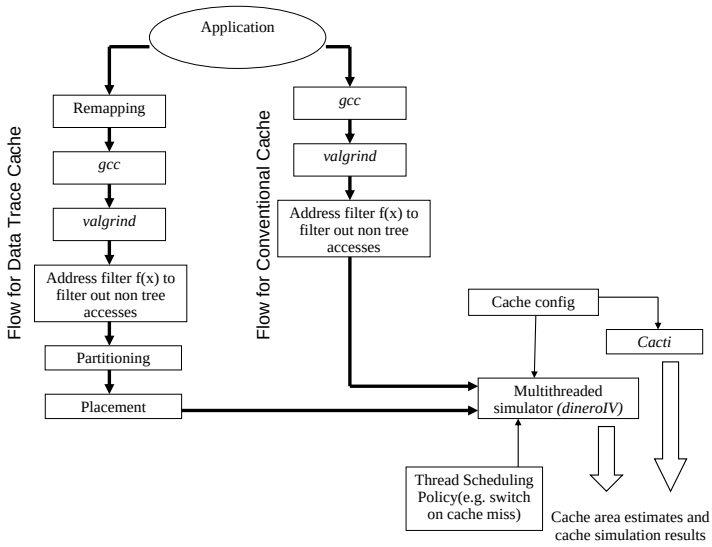


Figure 22. Evaluation Infrastructure

cycles are the norm). As mentioned earlier, network processors (for example, the IXP) lack caches, and this section proposes the introduction of a small cache in the philosophy of stream buffers and victim caches to aid in reducing the memory demand. Hence, the performance evaluation focusses on small sized caches.

Table 2 tabulates the miss rates of the DTC versus a traditional cache for a single threaded processor and a multi-threaded processor. The comparison between a 1KB DTC and a 1KB traditional cache for a single threaded and multi-threaded processor are shown in 23 and 24. It can be seen that there is a significant reduction in miss rates (by 20% to 50%) by using a DTC for a single threaded processor. For a 4-way multi-threaded processor, the skew in miss rates are even higher, since the DTC design considers the non-uniform access probability of tree nodes and the total number of misses do not significantly change with multi-threading, while misses in the traditional cache increase with multi-threading.

Given the large miss rates with small caches, there is a compromise to be made between cache sizes and lower miss rates. This, in addition to the expectation that packet processing applications do not reuse packet data are some of the reasons for network processors lacking caches. From the analysis presented, it is seen that while miss rates are high, given the

Table 2. Performance comparison of Data Trace Cache Vs. a Traditional Cache

		Miss rates for single threaded processor		Miss rates for multi-threaded processor	
Appl.	Cache Size	DTC	L1 cache	DTC	L1 cache
bin	512	57.3%	70.3%	58.6%	75.3%
	1024	54.3%	65.6%	55.7%	69.9%
rtr	512	27.6%	32.2%	28.7%	34.8%
	1024	19.5%	22.6%	20.1%	25.6%
avl	512	33.8%	56.3 %	38.9%	73.4%
	1024	27.7%	56.7%	30.7%	67.8%
aa	512	22.2%	27.3%	46.1%	56.3%
	1024	14.8%	20.7%	29.2%	44.6%
treap	512	24.0%	26.0%	22.5%	19.4%
	1024	12.1%	15.7%	13.2%	16.4 %

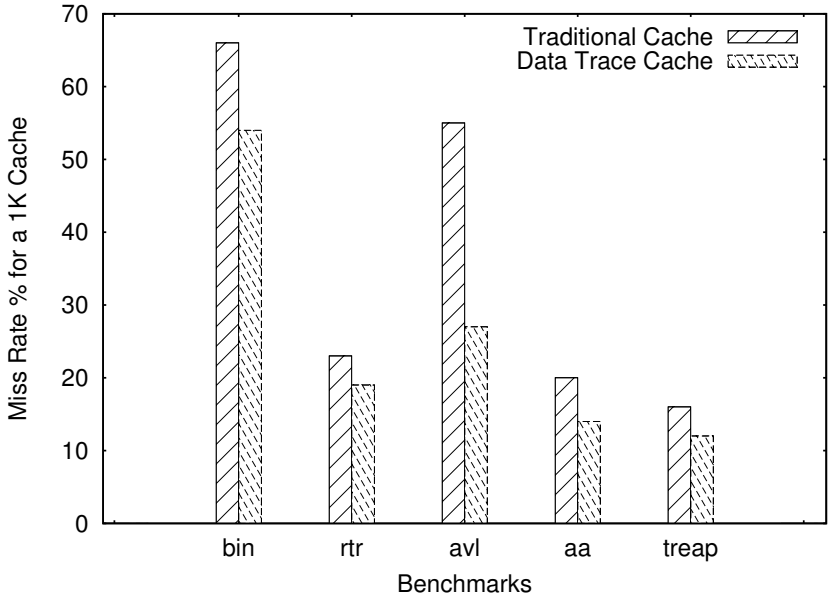


Figure 23. Single threaded DTC Results

significant miss penalties, the addition of a small cache remains beneficial, especially so, if it is a customized cache solution as the area and energy footprints for such caches remain lower while achieving better performance compared to traditional caches.

For the *rtr* benchmark, at lower cache sizes the performance difference between the data trace and the traditional cache was found to be starker (33% at a 256 byte cache size).

In addition, if flow pinning [105] was employed, the DTC was observed to perform better (miss rate for a 256 byte cache drops to 39% as opposed to 60% in a conventional cache). The DTC performs well for all the benchmarks considered though none of these required traversals to a leaf node; i.e., a tree access can terminate at any node where there is a match.

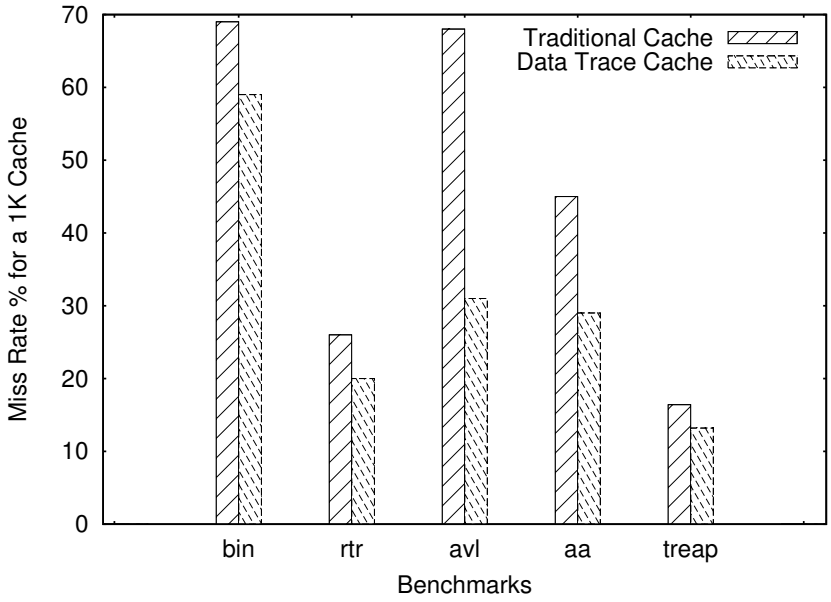


Figure 24. Multi-threaded DTC Results

Additionally, increasing the cache size from 512 bytes to 1024 does not significantly affect the DTC performance. This is because most of the performance gains due to the DTC principle are achieved at smaller DTC sizes. There is a significant reduction in miss rates to the rooted tree data structure in these applications. Since the results tabulate only references to the tree data structure, it is necessary to investigate whether the reduction in miss rates to the tree data structure results in a reduction of average memory latency. This is especially important as the DTC has a higher access latency than a comparable L1 due to the additional logic present. The additional logic as shown in Figure 21 is estimated to increase access latency by 7–8 cycles over a comparable L1 cache.

Figure 25 shows the DTC performance for the *avl* benchmark. It is seen that increasing associativity and size of the cache have lesser impact on miss rates than using a DTC. As the

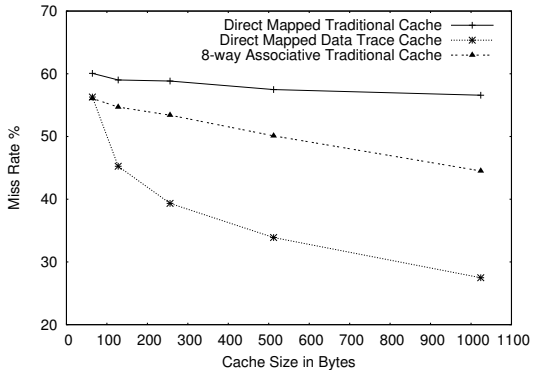


Figure 25. DTC performance for *avl*

size increases, the divergence between DTC and traditional caches narrows, but the DTC still performs better corroborating further that tree benchmarks benefit with the addition of a DTC.

If a same sized L1 was used instead of the DTC (with both caches having a parallel L1—this is referred to as L1base, and the L1 to be used instead of the DTC is called L1addl), the average memory latency can be calculated as:

$$L1addl = (L1_{at} + L1addl_{mr} * m_p)\ cycles$$

where $L1addl_{mr}$ is the miss rate, $L1_{at}$ is the L1 access time assumed, and the miss penalty is m_p cycles.

For the DTC, the average memory latency would be:

$$t_{DTC} = (DTC_{at} + DTC_{mr} * m_p)\ cycles$$

Where DTC_{at} is the DTC access time and DTC_{mr} is the DTC miss rate.

Assuming that the L1 access time is 4 cycles, and very conservatively assuming the DTC access time to be 12 cycles (due to additional logic as shown in Figure 21), miss penalty of 100 cycles, for the *avl* benchmark with a 1024 byte cache, the following is obtained:

$$t_{L1addl} = 4 + 0.7 * 100 = 74\ cycles$$

$$t_{DTC} = 10 + 0.3 * 100 = 40\ cycles$$

It is seen that, DTC performs by a factor of more than 1.5 over a same sized L1 in average memory access latencies for the tree data structure accesses. Assuming that L1base has the same hit rate (5% overall) with and without the trace cache, if 25% of the application memory accesses are to the tree data structure, overall latencies are computed as:

$t_{DTC} = 0.25 * 40 + 0.75(4 + 0.05 * 100) = 16\ cycles$

$t_{noDTC} = 0.25 * 74 + 0.75(4 + 0.05 * 100) = 24\ cycles$

For the entire system, the DTC is observed to decrease the average memory latency by approximately 30%. If the higher hit rate for the L1base in parallel with the DTC (due to lower data pollution), the performance gains will be higher. There is some area costs due to the decoding logic and the lookup table (32 * 5 bits per mask = 20 bytes for the lookup table), but the performance gains offset the slight increase in area.

Though the design in this section is applicable to any type of tree data structure, this idea can be extended to other data structures such as directed graphs. Using profile information to construct conflict sets, the idea can be extended across application data structures. For example, if data structures A and B each have a group of elements accessed more often than the other elements, and if accesses to A and B occur in an interleaved manner, these elements from A and B can be grouped into one conflict set. This would prevent an element from data structure A that is accessed very rarely from phasing out an element accessed more frequently in data structure B. For example the bit vector packet classification algorithm used for packet processing has five tree data structures [96].

Large data structures often have highly skewed non-uniform access patterns to the individual elements (for example, trees and directed graphs). This skewed access pattern can be exploited by using caches customized to the application memory access behavior. Application specific data trace caches with static remapping coupled with partitioning and placement functions can realize significant reductions in miss rates. The DTC demonstrates this phenomenon by achieving significant reductions in miss rates for several network processing kernels based on tree data structures. This philosophy can be extended to other data

structures such as directed graphs.

5.2 Customized Placement Cache

In this section the lessons of the data trace cache are generalized to techniques to customize the placement functions in traditional general-purpose caches to implement sizing and shaping. These techniques are profile driven and primarily targeted at embedded systems.

Embedded processors typically have single level caches with a miss accessing main memory. The power and energy constraints on embedded processors make cache energy and performance profiles especially critical. This section summarizes the work done in the context of embedded processor caches using *partitions*, which are finer grained than conflict sets, for realizing the mapping from main memory lines to cache sets. The concept of interference among partitions and conflict sets are used to drive the programmable placement policy resulting in enhanced performance for embedded processor applications. The analysis in this section is focussed on performance alone in exploring the potential of using customized placement solutions to increase cache performance as opposed to traditional caches with larger sizes and associativity. Thus, the strategy proposed here does not explicitly seek energy savings, though there is energy saved as a result of using lesser energy hungry caches with customized placement in comparison to traditional caches.

5.2.1 Partitions and Conflict Sets

If the total number of lines in the cache is represented by C_l, a *partition* P_i is defined as the set of memory lines, where memory line $L_k \in P_i$ if $L_k \ mod \ C_l = i$. A traditional k-way set-associative cache with S sets will define S conflict sets. The set of lines in each conflict set will be the union of k partitions. In a direct-mapped cache, partitions and conflict sets are equivalent. The optimization problem that is tackled in this section can be formulated thus: Given a memory reference profile, synthesize an assignment of partitions to conflict sets to minimize the number of conflict misses. Figure 26 illustrates the concept of partitions and

alternate customized placement functions.

Partitions correspond to the conflict sets of direct mapped caches. Rather than the general problem of partitioning the set of main memory lines into conflict sets (O(M^S) design space with M lines and S sets), the problem is re-formulated as the problem of grouping partitions into conflict sets (O(C^S) design space)

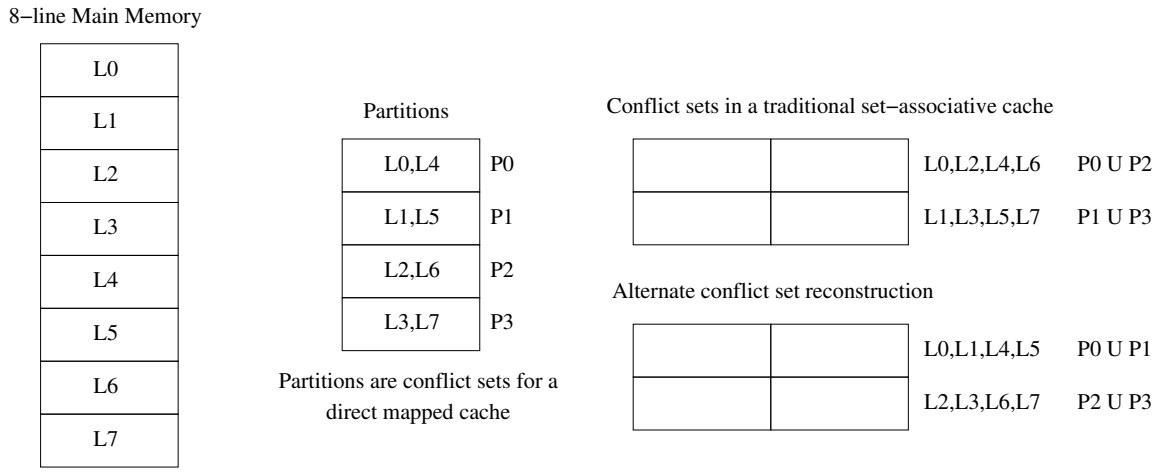


Figure 26. Partitions and conflict sets in a traditional cache.

Within a program phase, memory reference behavior exhibits spatial and temporal locality around a set of memory locations. The utilization of each partition in a program phase is measured and used to capture the potential conflicts between references to two partitions using the concept of *interference potential*, which is quantified as follows.

5.2.2 Greedy Algorithm Based Placement

A memory reference trace is partitioned into contiguous segments of references called *windows* where each window represents a program execution phase. For window w the number of references to partition i is defined as $r[w][i]$. In this study, the interference potential between partitions i and j in window w is $min(r[w][i], r[w][j])$ - representative of the average increase in the number of conflict misses in window w if partitions i and j are mapped to the same cache set (the maximum increase in the number of conflict misses is twice the

interference potential). The interference potential between two partitions is the sum of the interference potential between the partitions across all windows. As associativity increases, the interference potential is an increasingly pessimistic measure of conflict misses as the merged partitions will share an increasing number of lines in the target set. Customized placement will create new conflict sets by composing partitions based on their interference potential and assign these conflict sets to cache sets.

The algorithm shown in Figure 27 is the pseudo-code for the computation of customized placement for a set-associative cache. For a k-way set-associative cache with C_s sets, there are $P = k * C_s$ partitions. These P partitions are merged to form C_s sets of partitions, which form the conflict sets for the cache. The primary inputs to the algorithm are the reference counts for each partition in each profile window ($r[P][W]$, where W represents the total number of windows), interference potential between partitions ($ip[P][P]$), and, the number of sets (C_s).

A greedy algorithm iteratively traverses the interference potential matrix to select the partition pair with the minimum interference potential (line 5) and merges them to form a new conflict set (line 6) and updates the reference counts (line 7) and the interference potential matrix (line 8) to reflect the removal of one partition. This process is iterated $P - C_s$ times, to ensure that the resulting number of sets of partitions (the new conflict sets) is equal to the number of cache sets. A partition can be allocated a maximum of one cache set, and multiple partitions may share a single cache set. Finally, the mapping is updated so that the partitions map to actual cache set indexes (line 10). The output is the conflict set membership for all partitions ($map[P]$).

Direct-mapped caches are a special case where the number of partitions is equal to the number of cache sets (associativity = 1) and each set is one cache line. Thus partitions cannot be merged without leaving some cache lines unassigned and therefore unused. Thus direct-mapped caches are treated as a separate case and the algorithm is shown in Figure 28. The algorithm for direct-mapped caches consist of an allocation stage where partitions are

allocated cache lines (with a minimum of one cache line) based on their access demand, followed by merging partition pairs based on interference potential and updating the reference counts and the interference potential matrix. The algorithm returns an assignment of partitions to conflict sets (cache lines), and an allocation count equal to the number of cache lines allocated to each partition.

This flexibility in placement is accompanied by a relatively complex translation of 32-bit physical addresses to access the cache set, tag, and word. In a conventional modulo placement, $log_2\, C_s$ bits of the address, the *set index*, are used to determine the set containing the referenced memory line. With custom placement a partition may be mapped to any of the sets in the cache, and therefore requires the re-mapping of the set index bits. This re-mapping is implemented using a look-up table as shown in Figure 29. There is no constraint on the composition of conflict sets and therefore two lines in memory with the same tag may share the same cache set. Therefore the conventional tag is extended with the original set index to ensure that all lines can be correctly differentiated in a cache set with a unique tag.

Address decoding in a direct mapped cache is a bit more complex because each conflict set corresponds to a single partition and is traditionally allocated to a single cache line. If

Input: $r[P][W],\ ip[P][P],\ C_s$
Output: $map[P]$

1: **for** $i = 0$ to $P - 1$ **do**
2: $map[i] = i$ {Initialize}
3: **end for**
4: **for** $i = 1$ to $P - C_s$ **do**
5: find i_{min}, j_{min} s.t. $ip(i_{min}, j_{min}) = min(ip[P][P])$
6: $map[j_{min}] = map[i_{min}]$ {Merge i_{min}, j_{min}}
7: $update(r)$ {Update reference counts after merging}
8: $update(ip)$ {Update ip[P][P] after merging}
9: **end for**
10: update(map[P]) {Update mappings to point to actual cache sets}
11: **return** $map[P]$

Figure 27. Algorithm for determining set-associative cache placement.

partitions are coalesced based on interference potential to form new conflict sets, and if all cache lines are used, then at least one partition will have to be allocated multiple cache lines. When this happens, the memory lines in a partition are mapped to the allocated cache lines using modulo placement. In this implementation, the number of cache lines that can be allocated a to a single partition is limited to be 1, 2, or 4. Address translation now operates as illustrated in Figure 30 and implemented as shown in Figure 31.

5.2.3 Performance Evaluation

The techniques described in this section were evaluated by modifying the *Cachegrind* [102] simulator to support alternative placement functions. *Cachegrind* has a write-allocate policy on write misses with LRU replacement in set-associative caches. The benchmarks

Input: $r[P][W]$, $ip[P][P]$, C_s
Output: $map[P], alloc[P]$

1: $avg = (\sum_{i=0}^{P-1} \sum_{j=0}^{W-1} r[i][j])\ /\ P$
2: $partitions\_to\_merge\ =\ 0$
3: **for** $i = 0$ to $P - 1$ **do**
4: $map[i] = i$ {Initialize}
5: $alloc[i] = 1$ {Initialize one cache line per partition}
6: $p\_ref = \sum_{j=0}^{W-1} r[i][j]$ {Partition reference count}
7: **if** $p\_ref > avg$ **then**
8: $alloc[iter]=2^{\lfloor log_2\ val \rfloor}$ {Allocate lines to partition in powers of two}
9: $partitions\_to\_merge$ += $alloc[i]$
10: **end if**
11: **end for**
12: **for** $iter = 1$ to $partitions\_to\_merge$ **do**
13: find i_{min}, j_{min} s.t. $ip(i_{min}, j_{min}) = min(ip[P][P])$
14: $map[j_{min}] = map[i_{min}]$ {Merge i_{min}, j_{min}}
15: $update(r)$ {Update reference counts after merging}
16: $update(ip)$ {Update ip[P][P] after merging}
17: **end for**
18: update(map[P]) {Update mappings to point to actual cache sets}
19: **return** $map[P]$, $alloc[P]$

Figure 28. Direct-mapped cache placement

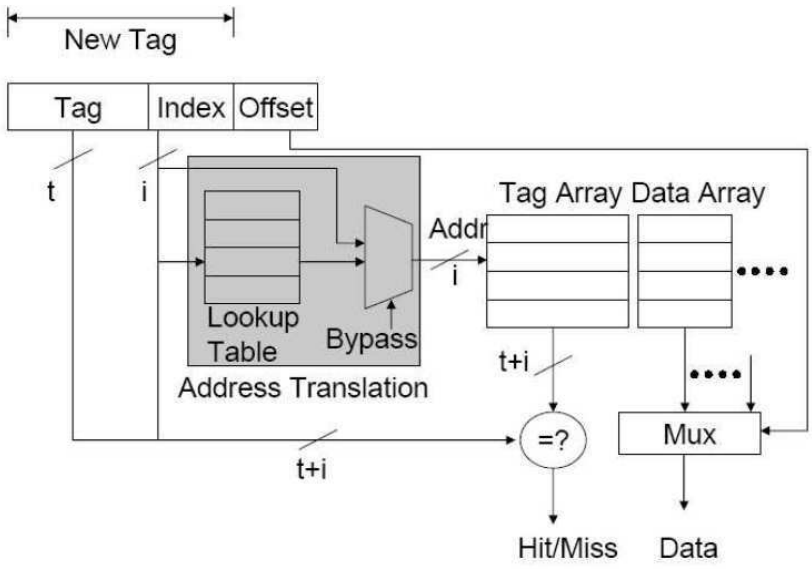


Figure 29. Address decoding for set-associative caches.

studied included a subset of kernels from the *Mibench* [106] benchmark suite including *basicmath*, *cjpeg*, *djpeg*, *fft*, *inverse fft*, *susan*, *tiff2bw*, *tiff2rgba*, *tiffmedian*, *tiffdither*, *patricia*, *ispell*, and, *ghostscript*. The area, power and cache latency estimates were generated using *Cacti 3.2* [104] for 90 *nm* technology.

This analysis does not implement phase detection, but rather, each phase (window) was chosen to be one million references empirically. The window chosen has to be sufficiently large such that interference potential is not tied too closely to the profile (since any slight change in profile would affect performance), whereas it has to be sufficiently small to ensure the interference information captured is useful enough to drive miss rates down. The kernels had reference counts ranging from 40 million to 100 million references, and a value of one million references was chosen as the window size as a compromise between the conflicting arguments.

Figure 32 illustrates the average memory access time (AMAT) averaged over the *Mibench* kernels obtained for various cache configurations, with a miss penalty of 200 cycles. In this case, no access time penalty was assessed for the look-up table access. The AMAT using the customized placement cache (CPC) is consistently better, and can be seen to offer the

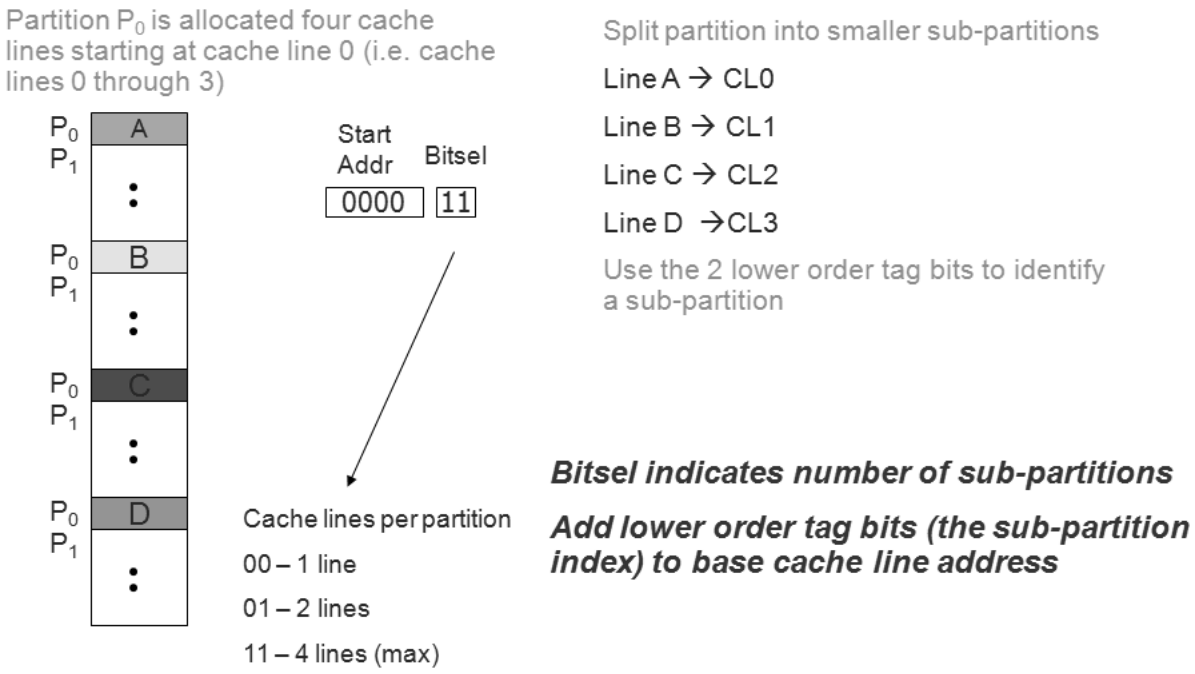


Figure 30. Address translation of direct-mapped caches - concept

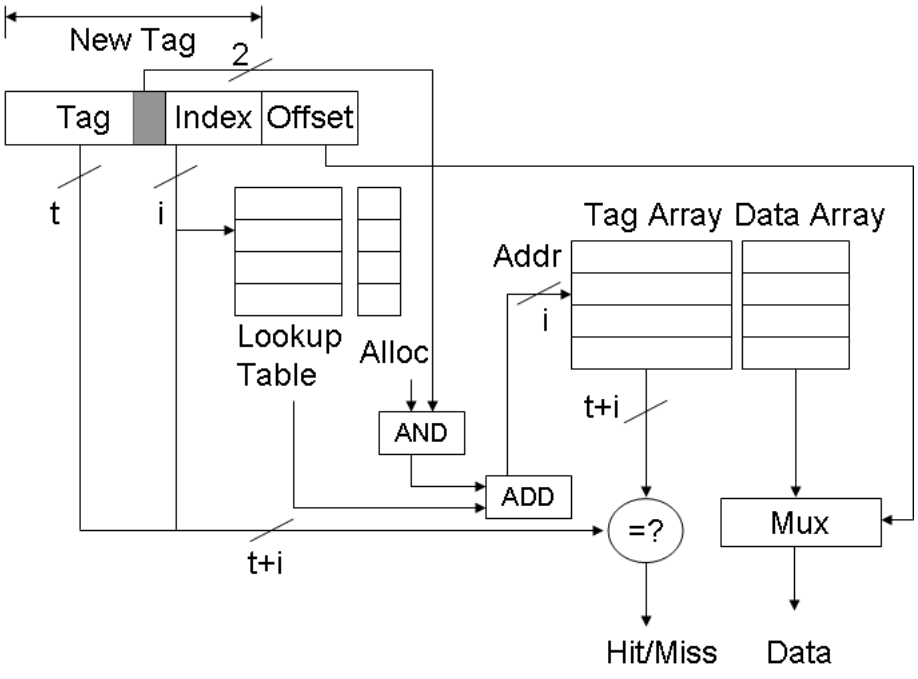


Figure 31. Address decoding for direct-mapped caches (bypass path not shown)

same effect as increasing associativity in traditional caches. This is caused by improved sharing which comes at the expense of the additional re-mapping but garners the effect of increased associativity.

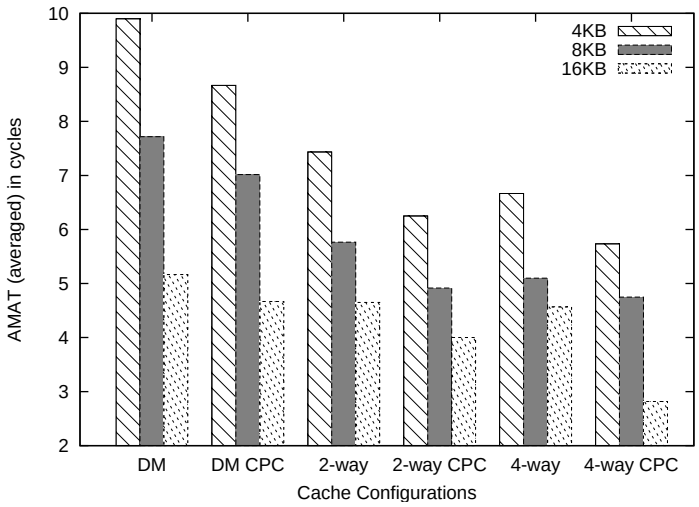


Figure 32. AMAT comparison for various cache configurations.

The latency penalty for customized placement is approximately 0.3 *ns*, over the base latency of 0.6 *ns* for a direct-mapped 4 KB cache and 0.8 *ns* for a 4-way 4 KB cache. The latency cost consists primarily of the decoding necessary for indexing the SRAM based look-up table. The effect of the added latency decreases as the cache size increases or the associativity increases. For the caches considered, the access latency including the look-up stage is well within 1.2 *ns* corresponding to a clock of less than 750 MHz for single cycle hit time - well within the scope of modern embedded processors. Therefore the analysis assumed single cycle hit times. Even if a penalty of half a cycle was applied to the customized placement cache, it would still outperform traditional caches, as the difference in AMAT between traditional caches and customized placement cache is greater than one cycle for most configurations.

Figures 33 and 34 illustrate the area and energy costs for various cache configurations. Figure 33 plots the area in mm^2 for various cache configurations. The additional area cost of the CPC is very low(2–5%) with the lookup table contributing to most of the increase

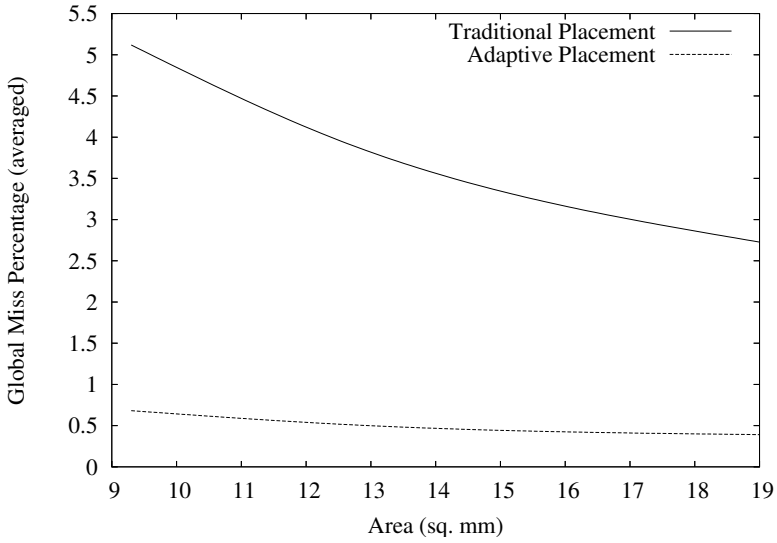


Figure 33. Area costs of various cache configurations

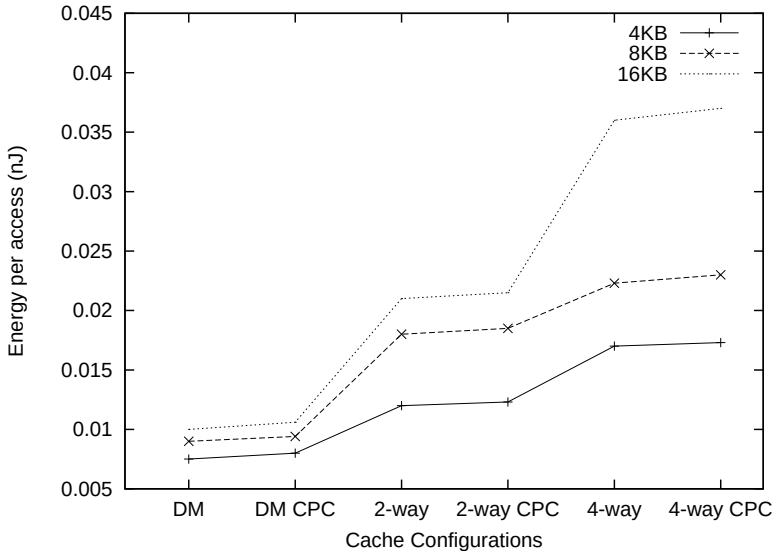


Figure 34. Energy costs of various cache configurations

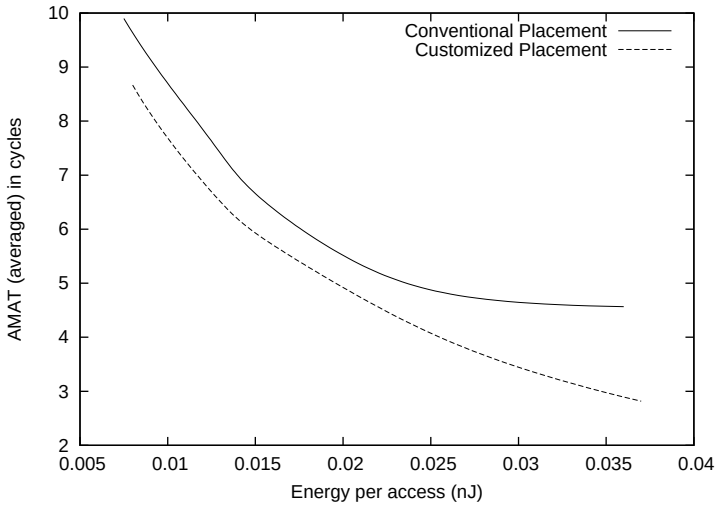


Figure 35. Energy-AMAT curves compared for traditional and customized placement caches.

in area. Figure 34 plots the per access energy consumed (in nJ) by various cache configurations. It is seen that associativity increases energy costs significantly, whereas the increase due to the addition of customized placement (keeping associativity fixed) is again low (2–5%). Looking at the these results in conjunction with Figure 32, the desirable design point is to use smaller caches with customized placement, or lower associativity with customized placement to have miss rates comparable to larger caches or caches with higher associativity resulting in energy savings.

Figure 35 illustrates the (interpolated) relationship between the AMAT provided and the energy consumed by traditional and customized placement caches. For caches with the same configuration (consuming approximately the same energy - CPC consumes slightly higher energy than the modulo cache for the same configuration corresponding to two nearby points in the x-axis), customized placement caches can provide significantly lower AMAT (10–40%) compared to traditional placement caches.

If caches for a specific AMAT were designed, customized placement offers energy savings (25% or more) over the traditional placement caches since caches of smaller sizes and lower associativity may be chosen. The energy savings increase as lower AMATs are obtained—this is because with customized placement lower AMATs are realized with

lower associativity and cache sizes compared to traditional caches. This discussion did not consider the energy saved in off-chip memory owing to a lower number of misses resulting in fewer requests to off-chip memory. Considering the decreased number of off-chip requests would lead to higher energy savings than reported in this section.

5.3 Fault Tolerant Cache (FTC) Architecture

The preceding designs led to the observation that shaping via customized placement could be used to mask errors in the cache. Conventional fault tolerant memory includes the use of redundant rows and columns of memory cells. Customized placement permits a more flexible masking of faulty rows by remapping memory lines to cache lines, building on the basic observation from Agarwal *et al*. [82].

The operation of the fault tolerant cache is based on a simple principle—main memory lines which are mapped to faulty cache lines are re-mapped to non-faulty cache lines. The determination of this mapping or placement is driven by application reference profiles and is under software control reflecting the active management principles. The small hardware footprint and fast hit times of these customized caches make them attractive for embedded processors. However, the techniques have natural extensions to larger set associative caches found in general-purpose processors.

In comparison to Section 5.2, the level of abstraction dealt with in this section is conflict sets, as opposed to partitions, and the techniques presented herein are focussed on providing graceful degradation in performance to caches with faulty lines.

5.3.1 Placement Model

The target of this approach are embedded processors that typically have a single level cache and where servicing a miss from off-chip memory is typically two orders of magnitude slower, e.g., 300 cycles in the IXP 2800 [94].

In a direct mapped cache a specific operating supply voltage is applied and consequently some memory cells fail as defined in Agarwal *et al*. [82]. Assuming a RAM Tag

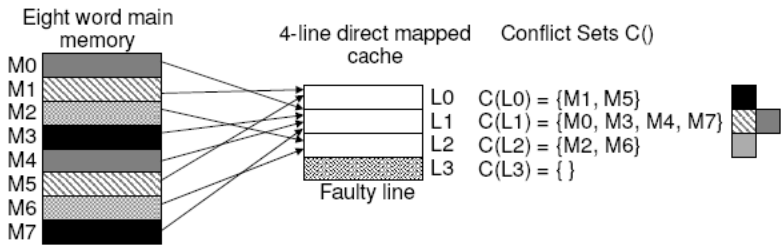


Figure 36. Cache with Customized Placement

implementation, tags and lines with failed cells are marked as faulty lines. Now consider a direct mapped cache where there are f such faulty lines. If a tag is faulty, the corresponding line is treated as faulty. The optimization problem is to find a new assignment of the S conflict sets to the $S - f$ non-faulty cache lines. The assignment is computed with the goal of minimizing conflict misses. Conflict sets from the fault free cache are *merged* in the faulty cache leading a lower number of conflict sets (and cache sets, since there are less fault free sets available). Thus, there are now $S - f$ conflict sets in the fault tolerant cache. An example of profile-driven customized placement is illustrated in Figure 36. The challenges are i) the development of algorithms to determine the most effective placement policy, and ii) the efficient implementation of address translation mechanisms for the customized placement policy. The former is driven by the locality properties of the reference profile measured as described in the following section.

5.3.2 Capturing Reference Locality

During a program phase, program reference behavior exhibits spatial and temporal reference locality around a set of memory locations. *Group temporal* locality has been defined as behavior wherein memory references in a phase that are not spatially local, are temporally local, i.e., they are clustered in time [107]. For example, when access to a data element allocated on the heap is strongly correlated (i.e., invariably followed by) an access to a local variable allocated on the stack. Studies have proposed various metrics [53, 108, 107] for capturing group temporal locality.

In the presence of faulty cache lines, multiple conflict sets are mapped to a non-faulty cache line. This results in the number of conflict misses to the particular cache line to increase. The optimization problem is to find an assignment of conflict sets to cache lines that will minimize the increase in conflict misses in the FTC. Conflict sets that exhibit group temporal locality are poor candidates for sharing a cache line. To identify good candidates for sharing a cache line, interference potential is used.

A memory reference trace is partitioned into contiguous segments of references called windows as defined earlier. For window w the number of references to cache line i is defined as $r(w, i)$. In this study, the interference potential between conflict sets i and j is $min(r(w, i), r(w, j))$ - representative of the potential increase in the number of conflict misses in window w if conflict sets i and j are mapped to the same cache line in that window. The interference potential between two conflict sets for the application is the sum of the interference potential between the conflict sets across all windows.

5.3.3 Fault Tolerant Placement Policies

Two placement policies for fault tolerant direct mapped caches are evaluated. The first is a profile-agnostic policy that is representative of existing approaches to address faulty cache sets [81, 82]. The second is a profile-driven placement policy customized from a reference stream.

5.3.3.1 Modulo Placement

Modulo placement is employed in existing fault-free caches and is representative of fixed alternatives realized by existing proposals for by-passing faulty cache sets or lines [81, 82]. Consider a cache with f faulty lines and $S - f$ non faulty lines that are addressed contiguously 0 through $(S - f)$. Main memory line L is mapped to $L \; mod \; (S - f)$. The practical difficulty is performing modulo $(S - f)$ arithmetic on every cache access. Since the fault pattern is not known a priori, address translation must be programmable. Thus, the decoder

has to be programmed such that faulty sets in the cache are bypassed. Similar implementations are also described in literature [81, 82]. For comparison purposes, the implementation of address translation for custom placement is also used to implement modulo placement. This is shown in Figure 37, where a lookup table performs the modulo function and bypass faulty lines.

Depending on the fault pattern, two lines with the same tag can now map to the same non-faulty cache line. Therefore to ensure unique tags across all memory lines that map to cache line, the new tags are comprised of the old tags concatenated with the index bits. For example, in Figure 36, if the line size was 16 bytes, memory addresses 0x00000010 and 0x00000070 map to the same line in the faulty cache (line 1), but the original higher order tag bits (0x000000 in both cases) are not sufficient to identify the memory location uniquely. If the index bits are concatenated with the old tag, the tags are now 0x0000001 and 0x0000007, which can be used to differentiate the memory lines.

5.3.3.2 Customized Placement

The goal of customized placement is to map conflict sets to non-faulty lines so as to minimize conflict misses. The algorithm for determining such a placement is outlined in Figure 38. The number of conflict sets in the original cache is S, which is equal to the number of cache lines in a direct-mapped cache and the total number of windows is W. The input to the algorithm consists of the reference count per cache line per window for all cache lines, $r[S][W]$, the interference potential matrix, $ip[S][S]$ and the number of faulty lines f. The algorithm first initializes the placement, by mapping each conflict set in the original fault-free cache to the corresponding cache line. This if followed by an iterative traversal of the interference potential matrix to i) select the conflict set pair with the minimum interference potential, ii) merge them to form a new conflict set (lines 5–6) and iii) update the interference potential matrix and reference counts (lines 7–8). The algorithm terminates after f merges, i.e., the number of new conflict sets have been made equal to the number of

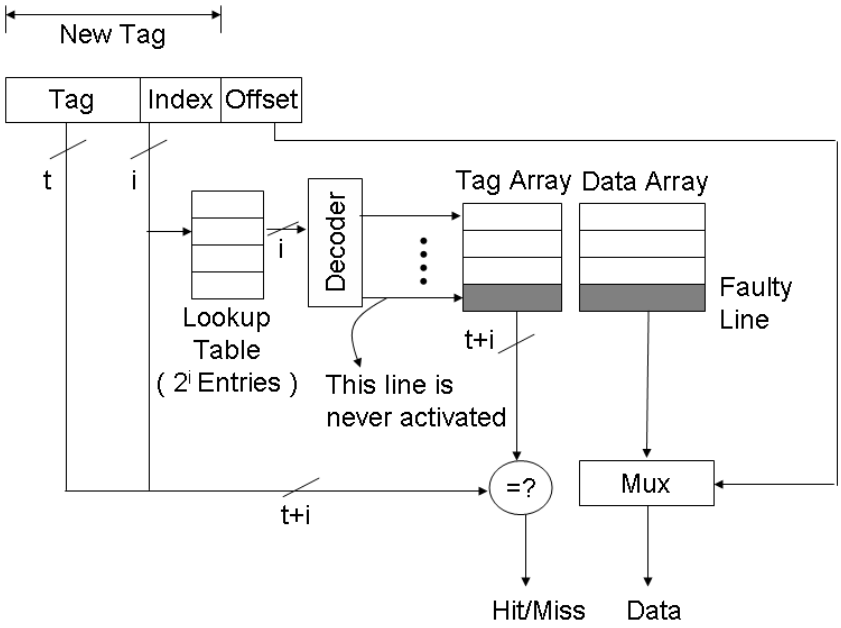


Figure 37. FTC Placement Implementation

fault-free cache lines. At this point the S conflict sets in the fault-free cache have been assigned to the $S - f$ cache lines. Multiple conflict sets in the fault-free cache can be mapped to a single cache line. The algorithm returns a placement $map[S]$, which maps conflict sets in the original cache to fault-free cache line. *All conflict sets are re-mapped—not just those sets that are mapped to faulty-cache lines!* The only requirement is that there are f merge operations. This enables a tighter control of AMAT as the number of faulty lines increases. Hence, a final update of the $map[S]$ array is required (line 9), where the mappings of the conflict sets are updated such that they point to fault free lines in the cache.

The technique and algorithms are similar in concept to that applied in Section 5.2. However, in this section, customized placement is performed at the granularity of conflict sets, as opposed to partitions that was used in Section 5.2. Additionally, techniques provided in Section 5.2 were not tailored to providing fault tolerance to defects in the cache.

Address translation is achieved using a lookup table as shown in Figure 37. Main memory addresses are translated via a lookup table, which is indexed by the original cache index (i.e., the index for the fault-free case), the output of which contains the new cache

Input: $r[S][W], f, ip[S][S]$
Output: $map[S]$
1: **for** $iter = 0$ to $S - 1$ **do**
2: $map[iter] = iter$ {Initialize}
3: **end for**
4: **for** $iter = 1$ to f **do**
5: find i, j s.t. $ip[i][j] = min(ip[S][S])$ {Find conflict sets having minimum ip}
6: $map[j] = map[i]$ {Merge the two conflict sets}
7: $update(r[S][W])$ {Update reference counts}
8: $update(ip[S][S])$ {Update the ip matrix}
9: **end for**
10: $update(map[S])$ {Update to remove mapping to faulty lines}
11: **return** $map[S]$

Figure 38. Placement Algorithm for FTC

line to which a memory address is mapped. The width of the tag array is increased by the size of the index to differentiate distinct memory lines as in the case of the modulo fault tolerance scheme. Faulty cache lines are never activated.

5.3.4 Results and Analysis

5.3.4.1 Evaluation Methodology

In the simulations, the following assumptions were made. The probability of a cache line being faulty is assumed to be normally distributed with mean μ and standard deviation σ. The fault detection model in Agarwal *et al.*[82] is assumed, where the cache has BIST circuitry which identifies defects and errors post fabrication and marks faulty SRAM cells. Access time failures or read/write failures due to process variation are considered faulty behavior. If a single bit in a cache line or tag is faulty the entire line is marked faulty. An S-bit register records the result of the BIST operation. The contents of this register can be read by compiler/configuration software. The customized placement is loaded by the software into the lookup table.

The fault tolerant cache placement schemes were simulated using *valgrind* [102]. The area, latency and power estimates were derived using *Cacti 3.2* [104]. The kernels that were analyzed belong to the *Mibench* embedded benchmark suite [106] which covers a broad domain of embedded system kernels. With a target of embedded processors, the focus

is on smaller caches where the effect of parameter variations is expected to be relatively higher and the benefits of the proposed approach the greatest. The miss penalty used in the analysis was as 100 cycles for a 32-byte cache line predicated on off-chip DRAM accesses of 100–300 cycles [94]. The miss penalties for 64-byte and 128-byte cache lines were 108 and 124 cycles , with the typical memory bank model assumed [109]. A fixed window size was used in the analysis, 100000 references, which was about 1% of the trace length for most of the *Mibench* kernels.

5.3.4.2 Fault Tolerance

Figure 39 compares the performance of a fault tolerant cache using a modulo scheme and one using the customized placement scheme. It is observed that the performance degradation for the customized placement cache in AMAT is less than 5% with 12.5% of the cache faulty, and the degradation is only 20% when 50% of the cache is faulty, compared to degradations of 10% and 60% for the modulo scheme. As the percentage of cache area that is faulty nears 100%, the difference between the two schemes will be less noticeable, as both techniques will yield very high miss rates. The better performance of the customized placement cache is due to the more efficient sharing of cache resources among memory lines. Since, the conflict sets that are grouped (equivalently *merged*) have a relatively lower interference potential, the effect on miss rates and hence AMAT and cache performance are reduced.

From Figure 41, it is observed that for larger caches using customized placement, the AMAT remains flat for a higher percentage of faulty cache area. This is because, the AMAT will not be affected as long as the number of fault-free lines in the cache is larger than the application footprint. Thus, the slope of the curves decrease as the cache size increases. The AMAT shown in Figures 39 and 41 represent AMAT averaged over the *Mibench* kernels. Figure 40 captures the performance variation of the individual *Mibench* kernels with faults. From Figure 40, it is seen that for certain benchmarks, the performance degradation is less than 5% even with 50% of the cache area being faulty using the customized placement FTC.

Thus customized placement shares cache resources effectively, that even with the effective cache being halved, there is very little (< 5%) performance degradation. The trends were found to be similar with varying cache line sizes.

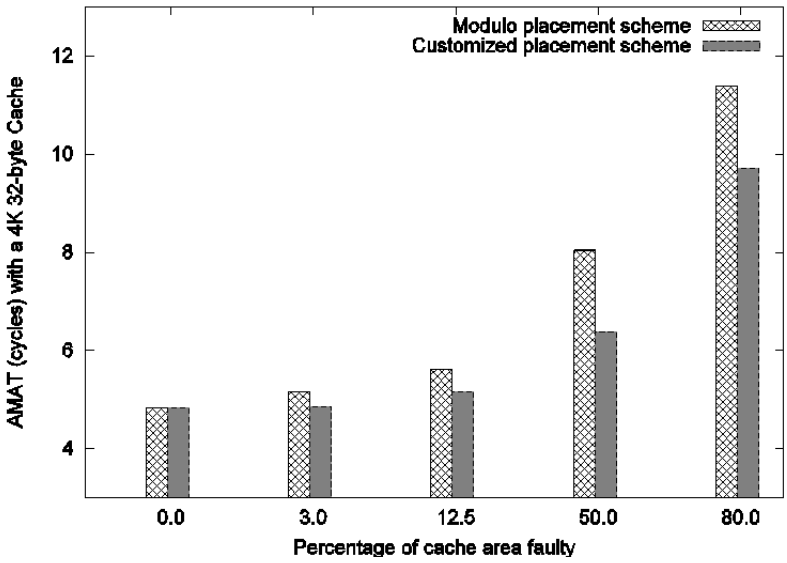


Figure 39. Modulo Vs Custom Placement

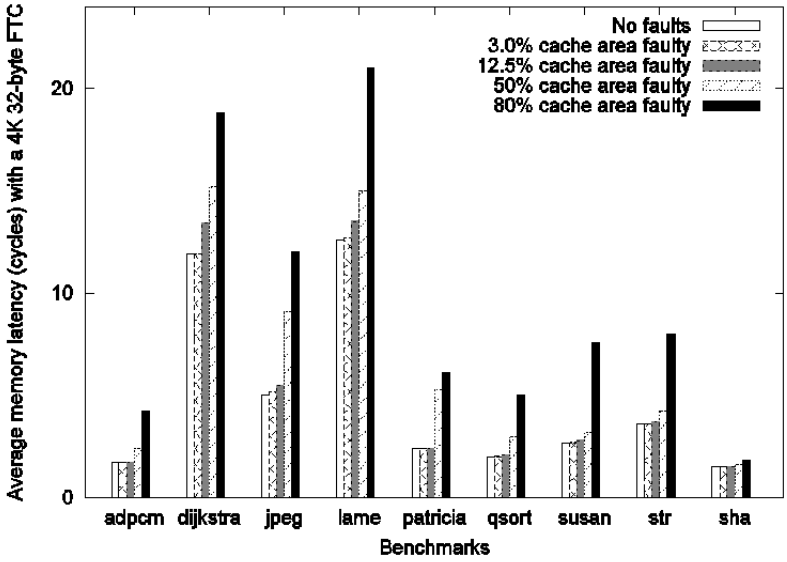


Figure 40. AMAT Variation with Faults

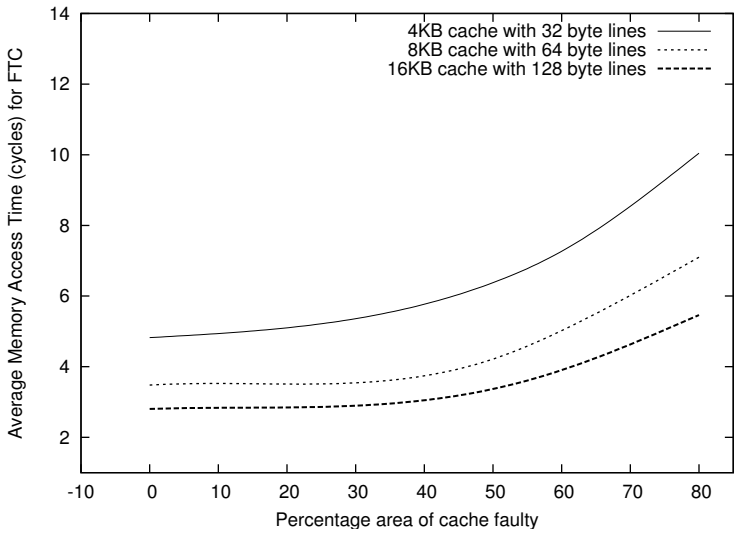


Figure 41. AMAT Variation with Faults for Various Cache Configurations

5.3.4.3 Area, Latency and Power

The area and latency costs of the fault tolerant cache were measured using *Cacti 3.2* [104]. The area cost for a 4K direct mapped cache with 32 byte lines is a lookup table consisting of 128, 7-bit entries plus an additional $128 * 7$ bits of increased tag store and a 128-bit register where the fault status of the cache lines are stored which together constitute approximately 5% of the overall cache area. For a 16K cache with 128 byte lines, the increased storage is again 310 bytes, which is less than 2% of the overall cache area.

The access time for an 4K direct mapped cache with 32 byte lines is 0.48 ns (0.58 ns for a 16KB, 128 byte line cache) using 90 nm technology, whereas the added latency for the lookup is 0.23 ns (comprising mainly of the decoding latency necessary for indexing the lookup table). Thus, the combined access time of the variable placement cache is around 0.7 ns. For an embedded processor running at 1.2 GHz, the access can therefore be performed within one cycle. However, modern embedded processors typically operate at 250–500 MHz and therefore the separate lookup stage will not adversely affect the AMAT. Larger the cache size or higher the associativity, the effect of the additional latency will be reduced considerably (for a 16KB cache, the lookup table increases latency by 30%, as opposed to nearly 45% for a 4KB cache). For the modulo fault tolerant scheme, the

hardware implementation of the address translation is also on the critical path, although its effect is not significant.

Since the lookup table size is very small compared to the cache size, and additionally, there are no tags or multiplexors required, the energy consumed by the lookup table is again concentrated in the decoding circuitry. This adds about 12% power increase to a 4K cache, and the increase diminishes with larger caches and caches with higher associativity (for example, for a 4K 4-way cache, the increase is approximately 2.5%, and for a 16 KB cache the increase is 8%).

5.3.4.4 Performance Yield

Performance yield recognizes that at the micro-architecture level yield is measure of the ability to meet performance goals and it is not a measure of identification of defect-free substructures. Thus we measure yield as the percentage of implementations (for convenience we use the term die) that produce an AMAT that deviate no more than 5% from the non-faulty die. The simulations were carried out for 1000 dies and for various μ, σ values of the number of faulty cache lines. The measured AMAT was averaged across all of the kernels. Those die whose AMAT was within 5% of the averaged AMAT for non-faulty designs were classified as usable die. The results are summarized in Table 3.

From Table 3, it is observed that the number of usable dies is substantially higher for the customized placement scheme over the modulo placement scheme. For a distribution with $\sigma = 8$ and $\mu = 8$, it is seen that die yield is over 85% for the customized placement scheme compared to 27% for the modulo placement scheme. The difference in the number of usable dies is noticed to be significant irrespective of the fault distribution. This effect on performance yield is graphically captured in Figure 42, where customized placement increased the total number of usable dies by as much as 600 for a set of 1000 dies for a given fault distribution.

If the application footprint fit into the set of fault free lines, the difference in yield is not expected to be significant. However, in practice this is rarely the case. Further, the results

Table 3. Performance Yield Comparison

μ, σ	Dies usable with Mod. Placement	Dies usable with Cust. Placement
1,2	921	998
1,8	380	958
2,2	820	990
4,4	604	984
4,8	355	925
8,8	279	859
16,16	130	612

suggest a strategy of using smaller caches to save area and power with minimal sacrifice is performance. This performance loss may well be recovered by putting the recovered silicon area to good use, e.g., larger register file.

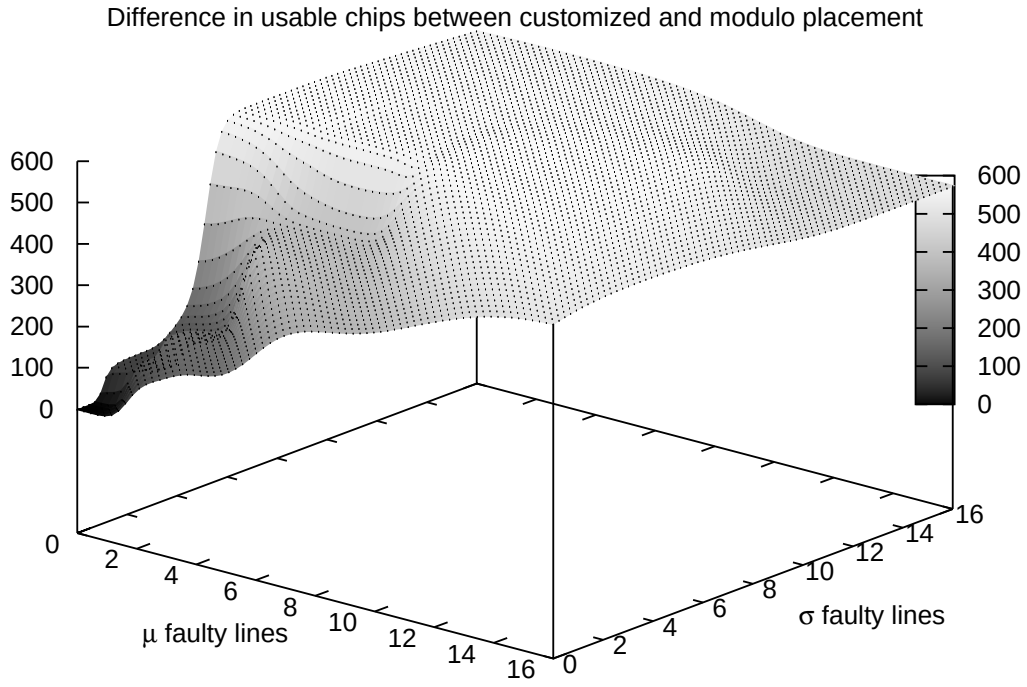


Figure 42. Performance Yield

The definition of yield includes a measure of performance resulting in a more stringent requirement than only requiring that the design tolerate faults and enable the die to be functional. Therefore, if a weaker definition of yield sufficed, the number of usable die

would be higher. Finally, a better definition of yield would include binning by frequency and power (leakage). That is the subject of current efforts.

5.4 Concluding Remarks

Approaches to improve the cache performance and providing tolerance to faults using off-line profile driven strategies, which configure the cache one time per application were described in this chapter. Several problems were identified and solutions were prescribed with a view on increasing cache energy and execution time performance. The solutions were matched with characteristic of the embedded processing domain, where they are applied. These techniques were found to improve cache execution times and energy performance significantly compared to traditional cache designs in addition to providing graceful performance degradation in the presence of defects in the cache.

CHAPTER 6

STATIC STRATEGIES FOR IMPROVING EFFICIENCY

This chapter identifies several active management solutions in the context of improving cache energy and performance efficiencies using static program analysis. Static strategies are well suited for domains such as scientific computing and embedded processing, since application workloads in these domains are well defined. In this dissertation, the analysis is limited to identifying program working set sizes and access strides. Section 6.1 applies these techniques to the scientific domain exploiting the domain specific knowledge for improving efficiencies in the scientific domain, called strided placement, and Section 6.2 extends these principles to the multi-threaded scientific domain with studies accounting for the effect of the operating system as well.

6.1 Strided Placement Cache

Strided array accesses are common in many scientific computing kernels. These strides are either statically computable at compile-time or are dynamically computed at run-time. For example, grid processing systems partition multi-dimensional array data among nodes using a run-time determination of available processors (this leads to each node having a different access stride to the array data structure). Sequences of accesses across orthogonal dimensions of multi-dimensional arrays typically generate many conflict misses. Such structured access patterns can benefit from customized placement with dramatic reductions in conflict misses and energy consumption with low overhead. The strided placement caches proposed here to optimize efficiencies for scientific computing results in a large reduction in execution times, average memory access times (AMAT), overall energy consumption and the energy delay product (EDP) with performance and energy efficiency improvements being greater than 4X and 10X.

The proposed technique can extend existing compiler memory optimizations to improve

performance and lower energy. This work was motivated in part by the data skewing techniques for concurrent access to interleaved memory banks including Lawrie and Vora [110], Raghavan and Hayes [111], Sohi [112], and, Rau [113]. The work specifically targets large array data structures with memory accesses to multidimensional arrays realized as accesses to addresses in a linear memory address space. The techniques are first developed for one-dimensional arrays with a single stride and extended to multi-dimensional arrays with multiple strides. The techniques are applied to single-threaded and multi-threaded applications and are found to very effective in improving cache efficiency.

6.1.1 Strided Placement for One-dimensional Arrays

Consider a one dimensional array, $A[N]$, with $N = 2^n$ elements where a memory block or *line* consists of $x = 2^b$ array elements. Thus, the array A is stored in $L = N/x$ contiguous memory lines, addressed $l_0, l_1 \cdots l_i \cdots l_{L-1}$, with line l_i containing the elements $A[ix], A[ix+1] \cdots A[ix+x-1]$. Traditional caches use the fixed modulo placement policy where memory line l_i is stored in cache set $l_i \; mod \; S$, with S sets in the cache.

An access pattern is a sequence of memory accesses to the array. Now consider an access pattern that begins with an access to element $A[0]$ and has a stride $k = 2^p$, i.e., consecutive accesses are to elements $A[k], A[2k], A[3k]$, and so on. This corresponds to a memory *line access stride* of $K = k/x$, i.e., memory lines are accessed in the order $(l_0, l_K, l_{2K} \cdots l_{(L-1)-(L-1) \; mod \; K)})$. When $k < 2x$, all lines in the array are touched in a single array traversal. When $k \geq 2x$, all lines are not touched in a traversal. For example, with $k = 2x$, every even line is touched when traversal is started at any of the elements $A[0] \cdots A[x-1]$, and every odd numbered line is touched when traversal starts at elements $A[x] \cdots A[2x-1]$. The *stride set* is the set of lines accessed during one traversal of the array with a stride of k. Thus, there will be K stride sets, each with cardinality L/K. The cardinality of a stride set corresponding to an access of stride k is the *stride number*, sn_k. These concepts are illustrated in Figure 43.

Now consider a traditional direct-mapped cache with modulo placement and S lines

(equivalently S sets). For accesses with stride k, if $S = K$, all of the lines in a stride set belong to the same conflict set producing a conflict miss on *every access*! To eliminate conflict misses, lines $l_0, l_K, l_{2K} \cdots$ can be mapped to separate cache sets via the customized placement called *strided* placement.

Using *strided placement*, memory line l_i is mapped to cache set $\lfloor l_i/K \rfloor \ mod \ S$. Now the first S strided accesses to lines in this set, with stride k, will not produce any conflict misses! When $S \geq sn_k$, all lines in a stride set can be resident in the cache and thus *all* conflict misses can be eliminated leaving only compulsory misses, whereas with modulo placement there are sn_k misses for every array traversal. This is significant for column major access to matrices stored in row major order where the number of conflict misses that are eliminated can be as high as $sn_k * (x - 1)$. Figure 43 also illustrates this property of strided placement. Increasing the associativity of the traditional cache will not help because all the lines being accessed still belong to a few conflict sets (all the lines will be grouped into k_a sets, where k_a is the associativity). An associativity of sn_k is required to eliminate conflict misses in a single array traversal with stride k. This is undesirable and infeasible for larger values of sn_k. If $sn_k = S$ the number of conflict misses for strided placement equals that of a fully-associative cache with S lines as illustrated in Figure 43. This is achieved without the area and power penalties of a fully-associative cache. However, if there is pollution in array accesses, a fully-associative cache will have all accesses miss since it has only one set, whereas the strided placement cache by virtue of having multiple cache sets is more resilient. For strided placement, if the cache associativity is k_a with S sets in the cache, the strided placement model changes to mapping memory line l_i to cache set $\lfloor l_i/K * k_a \rfloor \ mod \ S$, because k_a lines are present in each cache set.

For multidimensional arrays with strided access patterns, the goal, is to similarly sequentially refer elements in distinct conflict sets. Thus the lines corresponding to the accessed elements can be concurrently resident in the cache. Conflict set formation is the key. For example, to have conflict-free access to all elements along the third dimension

of a 3-D array stored in row-major order, a placement function that places each xy-plane (corresponding to the first and second dimension) in a distinct conflict set is required as shown in Figure 45.

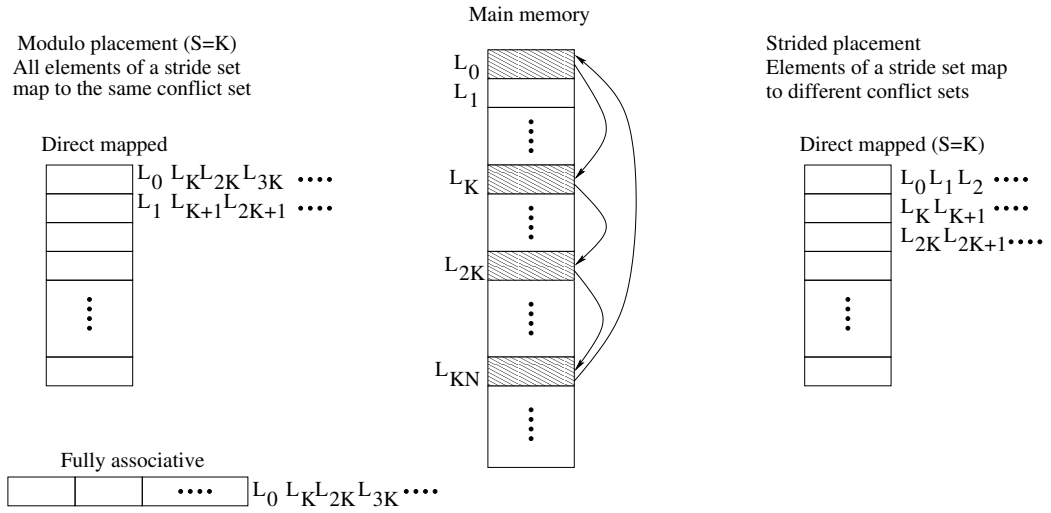


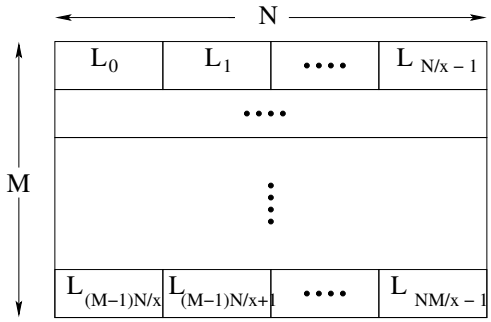
Figure 43. Strided placement: Basic concepts.

The criterion for conflict-free accesses using modulo placement has the following conditions: $S * k_a \geq sn_k$, and, $S * k_a \geq sn_k.gcd(S, K)$, where $gcd(S, K)$ represents the greatest common divisor between the number of cache sets and the stride K. If S and K are co-prime, the requirement reduces to $S * k_a \geq sn_k$ for conflict-free access. If S is even, and the stride is even, modulo placement cannot provide conflict-free accesses. With customized placement, the criterion for conflict-free access is just $S * k_a \geq sn_k$ for conflict-free accesses.

6.1.1.1 Multi-dimensional arrays and Multiple Strides

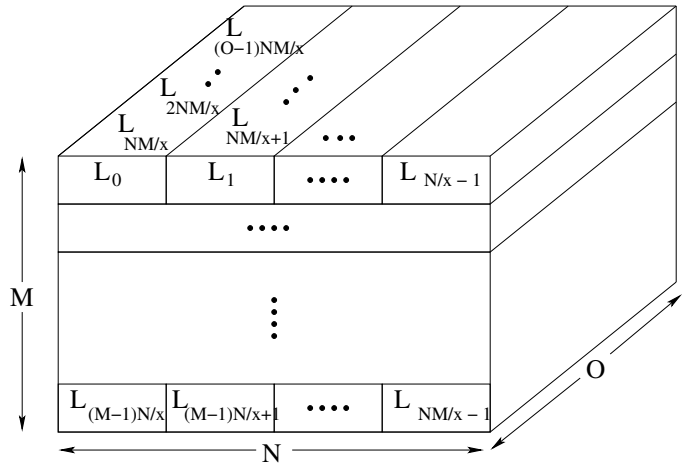
For a 2-D array with dimensions M, N (stored in row-major order with M rows and N columns), row-major accesses translate to an elemental stride of unity, and, column-major access translate to an elemental stride of N. Therefore, the placement function that minimizes conflict misses for column order accesses will map line l_i to set $\lfloor l_i / \lfloor N / x * k_a \rfloor \rfloor \ mod \ S$ in the cache as illustrated in Figure 44. Similarly, the conflict set construction for three dimensional arrays is shown in Figure 45. To traverse a matrix in both row and column-major orders, storage can be row-major and the placement be optimized for column-major order

to minimize conflict misses The same placement policy can be used for forward and back diagonal accesses, because the conflict set construction remains the same.



Row−order access: Each column is a conflict set (L_i mod S)
Col−order access: Each row is a conflict set (L_i/(N/x) mod S)
Diagonal access: Each row is a conflict set (L_i/(N/x) mod S)

Figure 44. Strided placement for matrices.



x−order access: Each column is a conflict set (L mod S)
y−order access: Each row is a conflict set (L /(N/x) mod S)
z−order Diagonal access: Each row is a conflict set (L /(NM/x) mod S)

Figure 45. Strided placement for 3D arrays.

If a one dimensional array, A is accessed with line strides P and Q in distinct program regions $seg1$ and $seg2$, distinct placement functions can be employed in each region. If A is accessed with line strides P and Q in the same program region, placement is optimized

for the stride $K' = gcd(P, Q)$, which yields conflict-free accesses if $S * k_a \geq sn_{K'}$, because the stride set for each stride will be either placed in distinct cache sets (if direct-mapped) or mapped to cache sets such that each cache set accommodates references from both stride sets without thrashing. For multiple strides, therefore, the general solution is a placement function optimized to the stride $K' = gcd(P, Q, \cdots .)$.

If two arrays with the same dimension are accessed within the same loop nest with the same stride, K, the placement is optimized for the stride K, if the cache is two-way associative. The general solution is to optimize for the stride $K * k_a/X$, when X arrays are present. If two arrays have different strides P, and Q, the placement policy must be optimized for the stride $K' = gcd(P, Q)$. Accesses will be free of conflicts as long as the condition, $S * k_a \geq 2 * sn_{K'}$ is satisfied. As only one placement function is applied to both arrays, the stride numbers are calculated based on the larger sized array. All these relations can be extended to multiple arrays with multiple strides.

One can also apply different placements at different levels of the cache hierarchy. For example, customized placement optimized to stride P is used in one cache, say the L1 cache, and placement optimized to stride Q is used in the L2 cache. Alternately, one can apply multiple placement functions (based on addresses) for different arrays, i.e., array A has a placement function P_a applied to it, and array B has a placement function P_b applied to it, and so on.

6.1.2 Miss Folding

Miss folding is the technique through which multiple conflict sets are merged to form a single conflict set (mapping to a single cache set) with no increase in the number of misses. Miss folding results in conflict sets with higher cardinality, but reduces the total number of conflict sets, allowing cache sets to be switched off, increasing energy efficiency with no drop in performance. An increase in the number of misses may be tolerated to benefit from higher energy savings and is a design compromise. The applied placement policies ensure that a memory line will have an active cache set to map to, limiting performance

degradation. An example of miss folding was illustrated in Figure 8(d). The technique focuses on folding conflict sets on live range information without increasing the total number of misses. In this case, miss folding folds conflict sets as well as conflict misses. The challenge has been development of low cost mechanisms that determine the lines to be turned off, and, when they should be turned off with minimal performance degradation (for example see Kaxiras *et al.*[66] and Abella *et al.*[65]). This segment of research exploits domain knowledge about structured array accesses and application footprints to improve energy and performance efficiency through customized placement.

A sizing algorithm restricts the number of active sets in the cache based on the stride number. The computation of the placement function follows directly from the cache sizing step. Informally, the goal is to have the active number of cache lines equal to the stride number. In practice, issues including the number of available sets, the stride number, matrix dimensions, performance versus energy optimizations will direct the specific sizing and placement functions. Algorithms shown in Figure 46 and Figure47 outline the implementation.

The cache lines required for eliminating conflicts for strided accesses to 1-D arrays is $S = sn_k/k_a$, so the remaining sets may be turned off without affecting performance. Both the stride and the stride number are programmable, so the programmer can override the stride number to increase energy savings at the expense of performance. This is useful in many scenarios. For example, if the footprint of the application reference pattern (represented by the stride number) is contained in the cache, the cache can be downsized. The benchmark 164.*gzip* requires a 256 KB cache to saturate hit rates. The stride can be identified as unity and the stride number can be programmed to keep the downsized cache at 256 KB even if the physical cache is larger for energy savings.

Traditional caches map successive memory lines to successive cache sets. If the access stride is unity, and, the data is touched only once, there will only be compulsory misses. This pattern is common—an array access with unit stride or a matrix that is accessed in

rowmajor order exactly once. All the conflict sets in this case can be merged to form a single conflict set resulting in the number of misses remaining the same as misses migrates from compulsory to conflict.

Input: l_i, K, sn_k, S, k_a
Output: $cache\_set(l_i)$
1: **if** $bypass = 1$ **then**
2: $return(l_i \ mod \ S)$ {Modulo placement}
3: **else**
4: $S_a = active\_sets\ (sn_k, S, k_a)$
5: $return\ (\frac{l_i}{K * k_a} \ mod\ S_a)$ {Customized placement}
6: **end if**

Figure 46. Algorithm computing the cache placement function.

Input: sn_k, S, k_a
Output: S_a, the number of active cache sets
1: **if** $S n_K < S * k_a$ **then**
2: $return\ (S)$ {Keep all sets active}
3: **else**
4: $return\ (S - \frac{sn_k}{k_a})$ {Number of active cache sets after turnoffs}
5: **end if**

Figure 47. Algorithm computing the number of active cache sets.

If an application data structure includes pointer intensive or random accesses with high miss rates that are changed with larger cache sizes, multiple conflict misses can be folded keeping the number of misses the same, but saving energy. Several strategies for customization by the programmer or the compiler may be applied based on the metric that is being optimized, e.g., reducing power via decreased performance.

6.1.2.1 Hardware Implementation

Customized placement is manually invoked by the programmer by calling the placement function with the stride and the stride number as arguments. If a programmer wishes to turn off more cache sets than computed by the placement function, a smaller stride number can be provided deliberately. Although customized placement can be implemented at all cache levels, for the scientific domain the L2 caches are larger and consume higher energy (leakage) and area budgets compared with the L1 cache. L2 cache misses result in expensive long latency memory accesses, therefore miss rate reductions have a more potent effect on execution time. Hence L2 caches are targeted for the scientific domain. Additionally, any

address translation time overhead can be overlapped with L1 cache access and thus hidden.

When the customized placement block is activated, the cache address decode path has another step as shown by the shaded region in Figure 48. In strided placement, memory line l_i is placed in set $\lfloor l_i/K \rfloor \ mod \ S$, where K equals $\lfloor k/x \rfloor$. The number of array elements per cache line, x is always a power of two enabling $K = \lfloor k/x \rfloor$ to be easily obtained with a shift operation. Furthermore, if the resulting value of K is also a power of 2, the computation of the placement reduces to a shift operation on the line address. Alternatively, compilers have been known to pad arrays to simplify addressing and maximize pre-fetch mechanisms. Those techniques are feasible here as well.

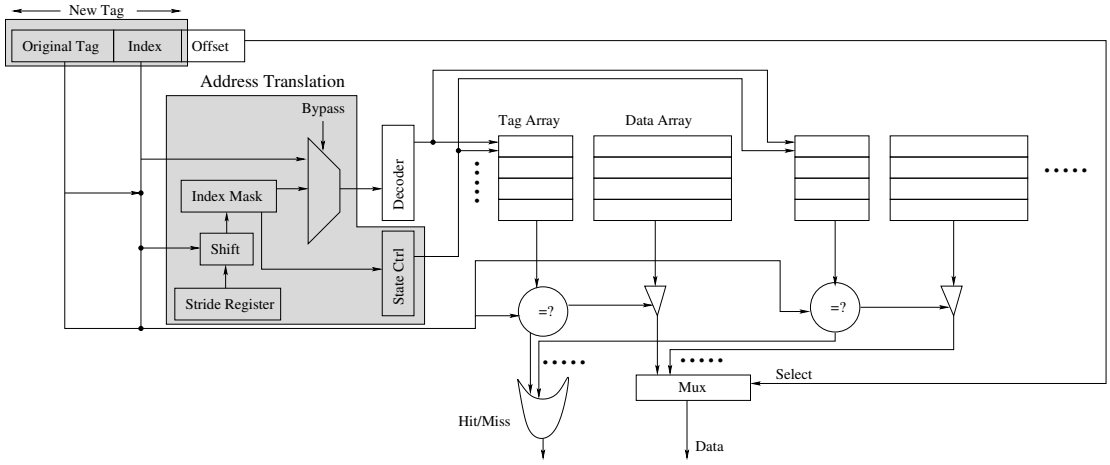


Figure 48. Run-time address translation for strided placement.

If the stride is not a power of two, one can adapt the placement for a stride K' which is a power of two, where $K' = gcd(K, K')$. This makes the resulting stride odd, and therefore co-prime with the number of cache sets, which distributes accesses evenly reducing conflicts.

The hardware implementation of the run-time address decoding for customizing placement depicted in Figure 48 consists of a stride register that contains the number of bits by which the line address (tag plus index) must be shifted to compute the set address, i.e., $\lfloor l_i/K \rfloor \ mod \ S$ prior to traditional tag look-up (i.e., the value $log_2(K)$). For even strides,

that are not a power of two, this register will contain the number of bits for the *gcd* of the stride and the closest power of two. The new tag is the old tag augmented with the original cache set index because the cache tag array may contain tags from customized as well as modulo placement. Traditional modulo placement can be invoked by writing a zero into the stride register or by invoking the bypass circuit. Writing to the stride register can be implemented as a customized instruction, or, by making the stride register an addressable region in memory. With multiple software threads, the stride register state will have to be saved along with the thread state on context switches. For hardware threads (simultaneous multi-threading), there must be separate stride registers for each hardware thread which maintains the cache placement information on a per-thread basis.

Zhang *et al.*[114], and, Mamidipaka and Dutt *et al.*[115] explain the leakage mechanisms in a 6-T SRAM cell and provide leakage models which are used in *Cacti*. The mechanism for turnoff of cache sets used is called *gated-Vdd* and was proposed by Powell *et al.*[67]. The technique enables turning off the supply voltages to caches lines to bring the leakage energy dissipation to negligible levels. The state of cache sets, i.e., whether a cache set is on or off is maintained in a register and is programmable. Customized placement ensures that a turned off cache set is never accessed to provide energy savings with low performance loss.

For implementing the customized turn-offs, the mask corresponding to the number of sets required to be active is maintained in the index mask register, which is decoded (the state control decoder) to turn cache sets on or off, by controlling the gated-Vdd transistor. This mask is the value $sn_k/k_a - 1$ computed from the algorithms presented in the previous section. This ensures that whenever cache sets are turned off, no inactive cache sets are accessed. The current design downsizes or up-sizes the cache in powers of two. A more general implementation is writing to the gated-Vdd transistors directly, but this simple design was sufficient for scientific computing as evident by the efficiency results. Thus, an access to a conflict set that originally mapped to an inactive cache set does not have to

result in a miss as happens with techniques reported in the literature.

The additional area costs for the customized placement cache consists of the extra tag bits required for identifying memory addresses uniquely, i.e., the tag matching logic requires the original tag and the original index to be stored. The stride register and index mask do not add any appreciable area costs. Thus, for a 256 KB 8-way L2 cache with 128 byte lines, an additional eight bits per set has to be stored, a total of 256 bytes, which is an addition of 0.1% area to the cache.

6.1.3 Performance Evaluation

The utilization and efficiency comparisons are shown in Figures 49, 50, and, 51. Traditional placement caches are represented in the figures as TC, and CC represents customized placement caches. Customized placement share cache resources better among memory lines than traditional caches as a result of better conflict set construction, even when cache sets are turned off adverse effects on AMAT and execution time were marginal as indicated by the higher efficiency numbers. Many of the chosen benchmarks had accesses to large multi-dimension array data structures with dimensions exceeding 1024 in many applications.

Figure 52 shows the average efficiencies across benchmarks studied among various configurations of traditional and customized placement caches.

The performance efficiency numbers with customized placement caches average 22% versus 5.6% for the traditional placement caches. Increasing associativities had a very limited effect on efficiencies for the strided benchmarks, because associativities that were impractical were required to reduce conflict misses substantially. Increasing cache sizes had only marginal improvements in performance efficiency compared to their energy costs as seen from Figure 52.

The numbers indicate that merging of conflict sets based on live ranges through customized placement created few extra misses attributable to cache set turn offs overall. The numbers show that customized placement was tolerant to pollution since each polluting

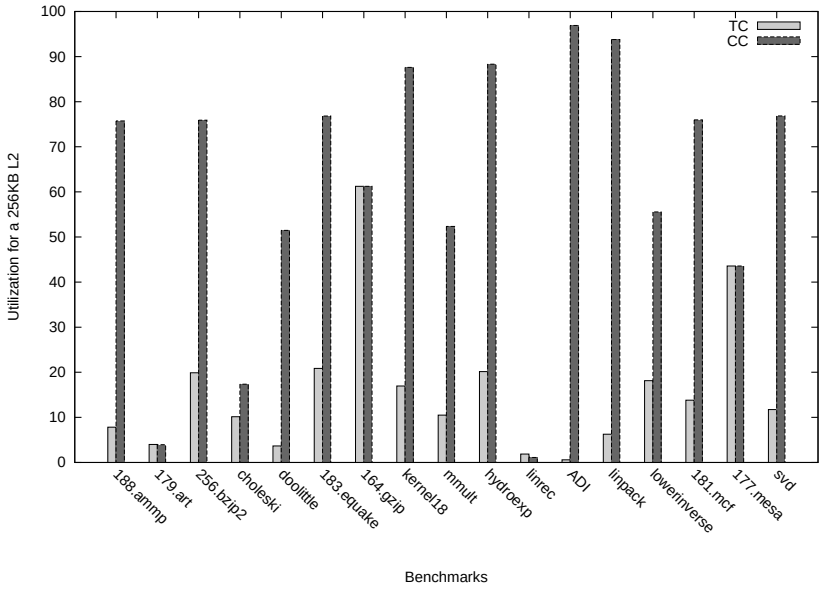


Figure 49. Cache utilization comparison for scientific computation.

access affects only one cache set, compared to higher associative caches which are more susceptible because the number of cache sets is smaller.

The utilization values for customized placement was 60% compared to 15% for a 256 KB 8-way traditional L2 cache. The average energy efficiency of customized placement caches were 5.5% from 0.38% for traditional placement caches. Section 3 has more explanations on the causes of inefficiency in traditional caches. The utilization improvement causes were two fold—first, by decreasing the percentage of dead cycles by turning off cache sets, and second, by customizing the cache placement, live data percentage increased as data likely to be reused was maintained in the cache as opposed to traditional cache where the same data was being evicted. The latter phenomenon was especially noticeable in the benchmarks with strided accesses.

AMATs and execution time for customized placement cache generally dropped compared to traditional caches because of the better sharing of active cache sets among memory lines, and because of the smarter conflict set construction, even when cache sets are turned off, adverse effects on AMAT and execution time were marginal. Figure 53 shows the average EDP across benchmarks and customized placement caches the EDP by more than an

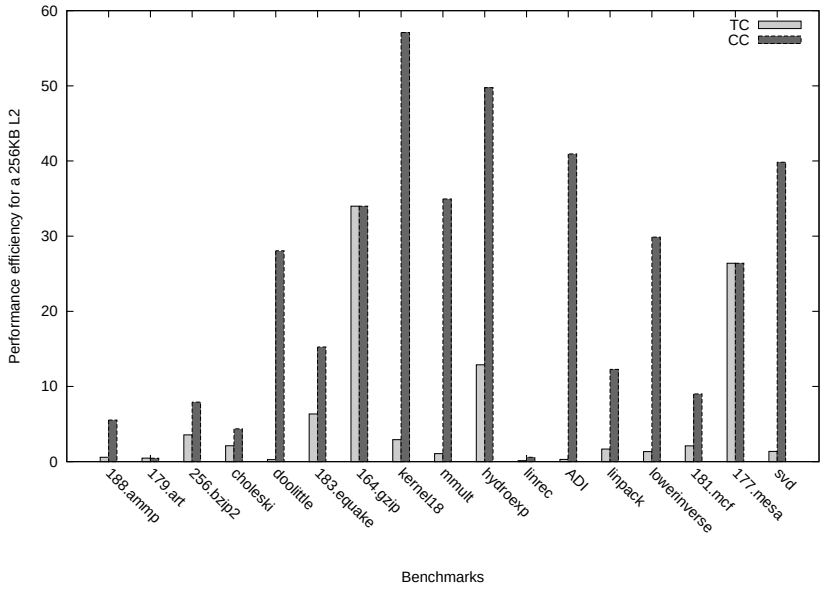


Figure 50. Performance efficiency comparison for scientific computation.

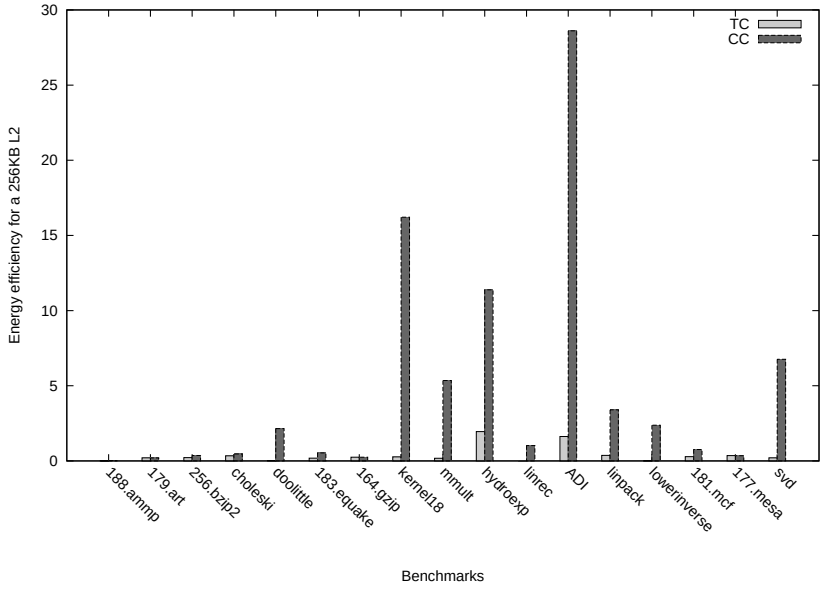


Figure 51. Energy efficiency comparison for scientific computation.

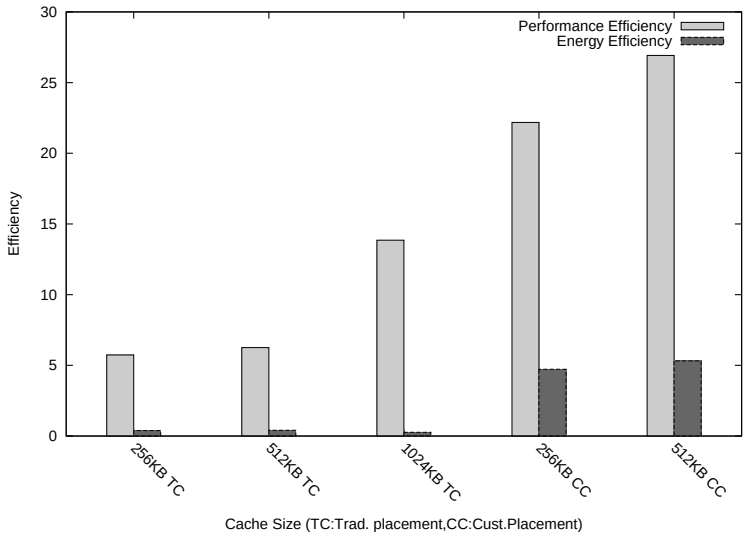


Figure 52. Efficiency variation with cache size.

order of magnitude as both delay and energy gets reduced.

Figure 54 shows the instructions per cycle (IPC) comparison of various cache configurations. While the IPC of the customized cache with downsizing rarely drops by more than 5% for the individual benchmarks, the overall IPC shows an improvement of 20% to the baseline traditional cache. This IPC improvement, coupled with the significant savings in energy indicate the efficacy of customized placement as a technique.

A drawback with customizing placement is that whenever the placement function is changed within an application, write-backs are necessary to ensure consistency. This is expected to cause slowdowns because of replacements, but, eager write-backs have been shown to improve performance [66, 116]. However, placement is changed relatively few times (mostly twice of thrice) within a program and the number of cache writes are a fraction of the total number of accesses. The number of invalidations were generally lower than 1% of total accesses and the results suggesting conflict misses were almost eliminated for many benchmarks indicate that any adverse effects were low.

The benchmark 164.*gzip* required 128 KB for supporting data structures and a cache

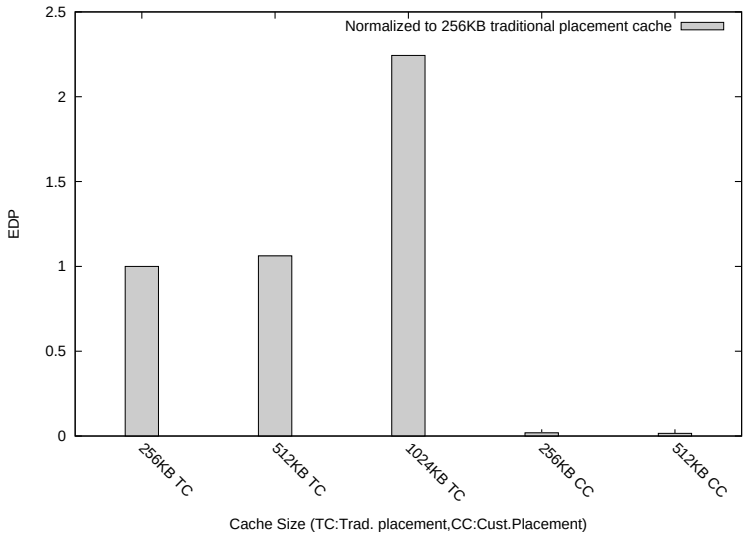


Figure 53. EDP variation with cache size.

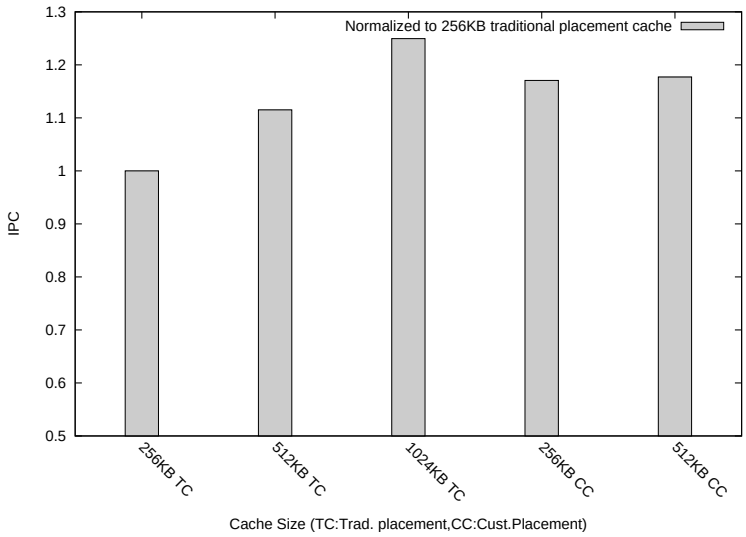


Figure 54. IPC variation with cache size.

size of 256 KB is expected to saturate performance, hence customized placement was optimized to the 256 KB footprint, and this resulted in energy savings compared to larger traditional caches with almost no increase in execution time.

For 177.*mesa*, the customized placement chosen was the same as traditional placement, and thus led to no improvements–this is one option to pursue. For the benchmark

256.*bzip*2, the cache was downsized by two, and that resulted in a 20% increase in execution time although significant savings in energy and decrease in dead cycles followed. Thus, the programmer can determine the time criticality of the application and decide accordingly on downsizing. These results show the potential of using active cache management techniques to improve cache efficiency.

Scientific computing applications sometimes adopt customized algorithms that are finely tuned to the hardware. For example, algorithms may be customized for a particular cache size, line size and associativity. In the performance evaluation, the comparison is made on generalized algorithms that are not tuned to the particular cache parameters. Making this comparison might yield a smaller divergence in energy and performance efficiency between strided placement and modulo placement. However, strided placement is an orthogonal architectural optimization to the algorithms used. Therefore, it may be possible to come up with more powerful algorithms to optimize energy and performance efficiency of caches using strided placement. This evaluation is outside the scope of this thesis and offers a potential extension to this work.

Additionally, such customization by the developer is not scalable as it has been pursued with a specific cache configuration in mind. Thus, for every machine configuration change, the algorithms have to be re-customized. With customized placement, the solution is scalable as the new placement function can be computed with the new cache as it primarily depends on the memory access pattern which is not subject to change as long as the algorithm remains the same, as opposed to re-writing the algorithms.

6.2 Extensions to Multi-threaded Applications

This section extends the discussion of the strided placement cache in the previous section to multi-threaded applications. The efficacy of active management techniques in improving cache efficiencies in the multi-threaded domain is seen by the significant improvements in

energy and performance efficiencies with relatively low overheads in hardware and software costs. The simulations in this section are carried out with the Linux OS running to make the results realistic and the impact of the OS on cache efficiencies is also discussed.

6.2.1 Hardware Extensions for Multi-threading

With multiple software threads, the stride register state will have to be saved along with the thread state on context switches. For hardware threads (simultaneous multi-threading), there must be separate stride registers for each hardware thread which maintains the cache placement information on a per-thread basis. When processes/threads are swapped out by the operating system, the dirty cache lines have to be written back to maintain consistency.

6.2.2 Impact of the Operating System

Additional misses mean additional accesses to memory which increases execution time, which forms a feedback loop as more OS accesses interfere, further increasing execution time (similar to a damped feedback loop).

The simulator was configured to track the application alone ignoring the OS. This allowed us to measure OS involvement, and found that it is generally a small percentage (less than 5% of the total accesses) and can be ignored. Most of the OS related memory accesses were hits in the L1 and L2 caches. Thus, if adopting customized placement causes the execution time to be dropped sharply, along with the total number of references (because of shortened execution time, OS interference related accesses also decrease) it is possible for miss rates to vary interestingly. For example, the L1 miss rates will be different after applying customized placement to the L2 because the number of accesses to the L1 is decreased. For example, if the L1 cache had 5 million accesses with 1 million accesses caused by the OS (with a 99% hit rate), and 4 million references to the application having 0.5 million misses, and after applying customized placement to the L2, the execution time was shortened, and OS only contributed 0.5 million accesses, the L1 miss rate will be seen to increase from 0.5/5.0 to 0.5/4.5! Thus, one has to factor in all metrics including

efficiency, execution times and miss rates to account for such anomalies.

6.2.3 Performance Evaluation

6.2.3.1 Simulation Methodology

The execution of the benchmarks were simulated on the SIMICS full system simulator which was configured for x86 processors running the Linux OS. The cache simulator was modified to obtain cache efficiencies. Strided placement was implemented by modifying the cache simulation module of SIMICS.

Energy estimates were obtained using *Cacti* 5.0 [117] for 70 *nm* technology. It is assumed that the L2 cache access latency to be 15 cycles independent of the size and associativity of the cache (this affected execution time by less than 2%). The efficiency definitions assume that the read and write energies are the same—which increases energy efficiency compared to a more precise definition. Leakage energy is predominant and cache writes constitute a small fraction of the total number of accesses, (for example, Tarjan *et al*.[90] estimate that at 70 *nm*, greater than 95% of the total cache power is leakage) therefore this assumption affected efficiency by less than 1%, as given by *Cacti* estimates. Finally, the energy was calculated with the cache operating at the highest expected frequency as given by *Cacti* estimates.

6.2.3.2 Results and Discussion

The performance of the strided placement cache was evaluated against the traditional modulo placement cache for a variety of single threaded and multi-threaded benchmarks. Many of the chosen benchmarks had accesses to large multi-dimension array data structures with dimensions exceeding 1024 in many applications.

Figure 55 shows that effectiveness has increased significantly across all applications. This means that strided placement caches were able to eliminate a large portion of dead cycles, when the cache lines were storing dead data. This increase in effectiveness can be also seen in performance efficiency as shown in Figure 56. The increase in performance efficiency is also broad across all benchmarks, suggesting that effectiveness was increased

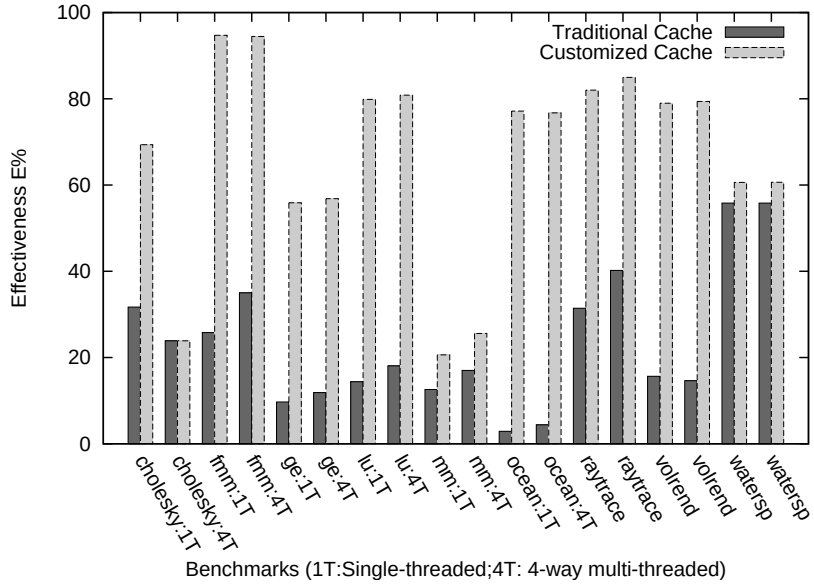


Figure 55. Effectiveness comparison

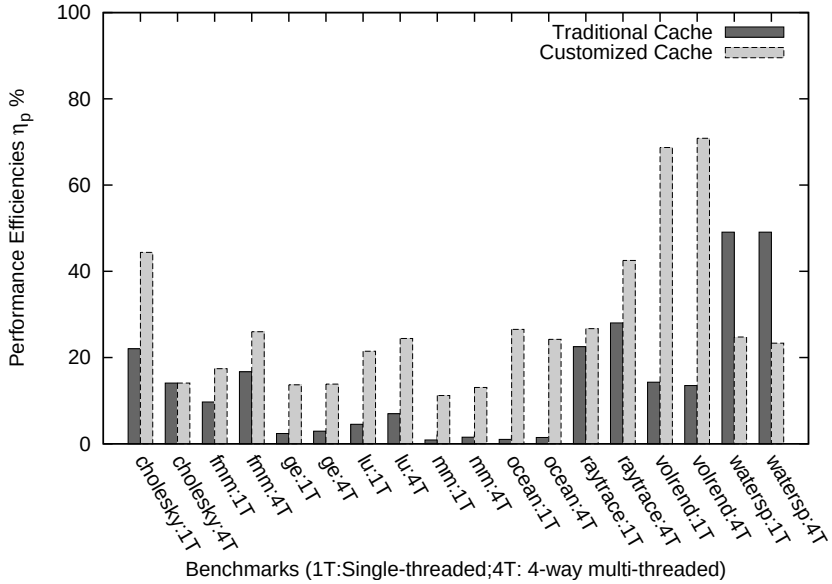


Figure 56. Performance efficiency comparison

without significantly affecting miss rates or execution times. The benchmark *watersp* is an outlier where the performance efficiency decreases, suggesting that aggressively shutting of cache lines hurt its performance.

The comparison of energy efficiencies is shown in Figure 57. Again, it is seen that energy efficiency increases across all applications and energy efficiencies double and triple for the strided placement cache over the traditional cache for many benchmarks. Though

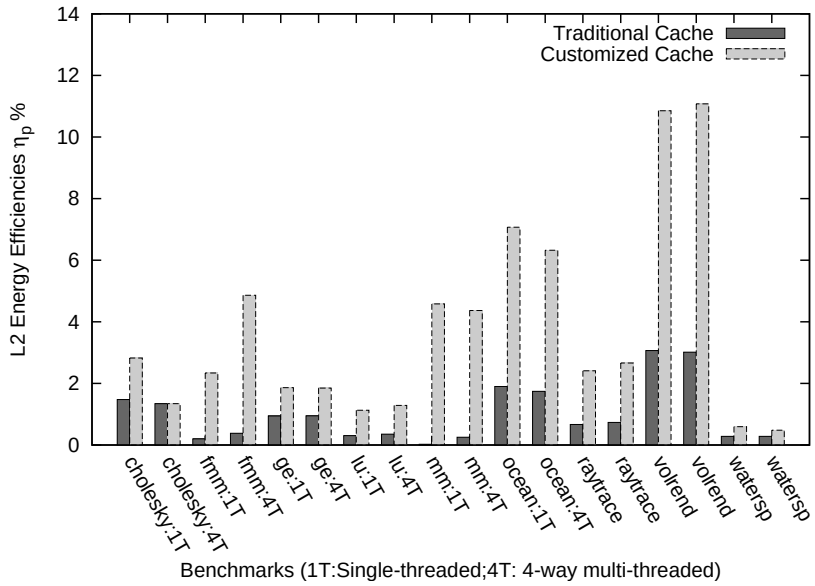


Figure 57. Energy efficiency comparison

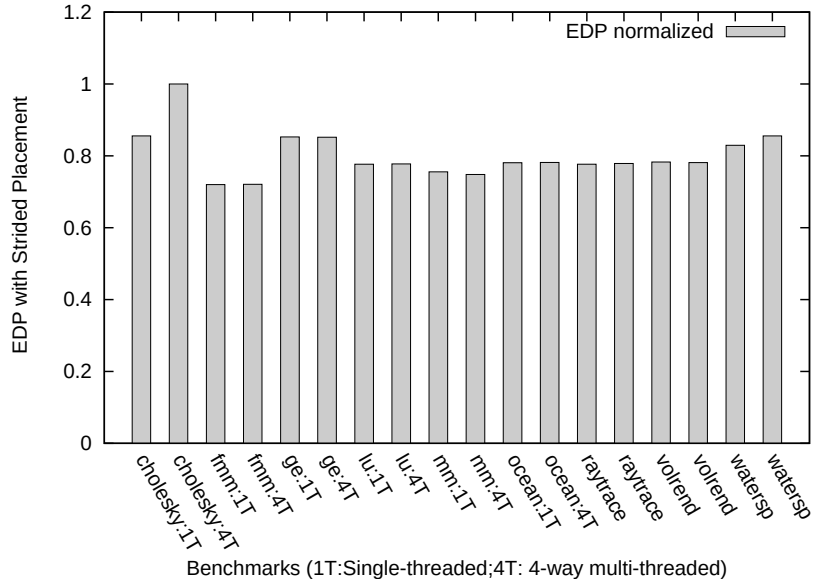


Figure 58. EDP comparison

the energy efficiency improvements as an absolute percentage seem modest, it has to be factored in that there are some fundamental constraints while attempting to improve energy efficiency. Given, that an L2 cache line is accessed only once every few million cycles, leakage energy dominates. Thus, in spite of the improvements brought forth by strided

placement caches, absolute energy efficiency improvements were limited although relatively large.

For multi-threaded benchmarks, the performance gains are as strong as that for single threaded benchmarks, signifying that the conflicting data demands across threads were satisfied by strided placement with the benefit of energy savings. All the improvements are seen with an OS under realistic conditions.

The Energy Delay Product shown in Figure 58 is calculated for the entire processor (not for the cache alone), assuming that the L2 cache accounts for 30% of the processor energy consumption. Even under these assumptions, EDP for most benchmarks are reduced by more than 15% (if the EDP were captured only for the cache, this would mean an increase of over 66%).

By providing the software control over the cache placement, reductions in execution time and energy consumption are achieved for single and multi-threaded applications executing on a Linux OS. This approach is able to increase energy efficiency by a factor of four and decrease EDP by 33% for a range of benchmarks. Although the improvements are significant the results still indicate much headroom left to explore.

Similar arguments made in Section 6.1.3 regarding the adoption of customized algorithms tuned to the cache parameters hold true for multi-threaded applications as well, and these techniques can be applied together for performance and energy benefits.

6.3 Concluding Remarks

This chapter describes techniques to optimize the memory performance of multidimensional arrays by relaxing the fixed placement constraint currently used by cache hierarchies by using information available at compile-time to drive sizing and shaping strategies. By building on the work of data skewing techniques, multidimensional array accesses are analyzed and cache placements developed that significantly improve cache performance and energy efficiencies. By giving software greater control over the cache structure, some of

the performance headroom that is obscured by fixed, design time partitioning of cache resources across memory lines is recovered.

CHAPTER 7

RUNTIME STRATEGIES FOR IMPROVING EFFICIENCY

This chapter provides solutions for improving cache efficiencies at runtime using heuristics to reconfigure the cache based on runtime measurements of memory reference patterns. These strategies are well suited for applications lacking statically characterizable profiles. Miss folding is used for dynamically sizing and shaping the cache in Section 7.1. Simple folding schemes relying on various heuristics were found to be effective in decreasing the energy-delay product (EDP) significantly and increasing energy and performance efficiency. Section 7.2, in addition to providing an efficient mechanism for tracking the runtime utilization of applications using a low-overhead approach, provides a framework to exploit utilization and efficiency concepts to identify program memory behavior phases and uses that information to re-configure the cache to improve cache efficiencies.

7.1 Improving Efficiency via Resizing+Remapping

This section proposes an evolution to the dynamic scaling of cache resources for improving both cache performance and energy efficiency. Resource scaling is a natural extension to the prior techniques for intelligently turning off portions of the cache for short periods of time, and which requires solving a basic problem of mapping all of memory into the scaled down cache or a scaled up cache i.e., computing a new placement function.

7.1.1 Cache Downsizing

The concept of conflict sets is combined with run-time reference counts to realize simple (hardware) techniques to dynamically resize the cache to a subset of active components and recompute the placement function . The L2 cache is targeted because of its larger size and consequently greater impact on energy consumption. The concept of *folding* is proposed by which memory regions that normally map to disjoint cache resources are combined to share cache sets producing a new placement function. Folding enables powering down

cache sets at the expense of possibly increasing conflict misses.

Effective folding heuristics can substantially increase energy efficiency at the expense of an acceptable increase in execution time. Section 3 describes the model and metrics for computing cache efficiency and the application to a set of benchmarks. The resulting insights lead into several folding heuristics.

If two conflict sets have non-overlapping live ranges, the two may be merged and a cache set corresponding to one of those sets can be turned off without any increase in the number of misses. This merging of conflict sets is referred to as *folding*. Folding requires re-mapping a set of main memory lines to a new set. Conversely, *splitting* is the reverse process and is accompanied by a corresponding up-sizing of the cache.

Re-constructing conflict sets alters the mapping from memory lines to cache sets, i.e., the cache placement is altered. Conflict set folding and splitting, along with cache sizing customize the placement for energy and performance efficiency. All conflict sets do not necessarily have the same cardinality as opposed to conflict sets in traditional caches which have the same cardinality. That is, the number of memory lines contained in a conflict set may vary over time using customized placement. A newly created conflict set by merging original conflict sets has a higher cardinality than other conflict sets which were not altered.

Associated with each set is a reference counter that approximates the "liveness" of the lines in a cache set. This reference count is used in the heuristics described later in this section. The existence of the ability to turn off cache lines using the *Gated-Vdd* approach proposed by Powell *et al.*[67] is assumed.

7.1.2 Runtime Heuristics

Three heuristics were evaluated—cache decay resizing, power of two resizing and segment resizing. The cache decay resizing heuristic is an extension to the cache decay strategy proposed by Kaxiras *et al.*[92]. Based on a 4-bit counter value, cache sets are turned off. If a cache set is accessed while it is turned off, a cache miss results and the cache set is powered back on during the servicing of the miss. This is extended this by folding conflict

sets that have long periods of inactivity or little activity. Splits from the merged conflict set occur if multiple accesses (four) to the original conflict set occur within the sampling period. Eager write-backs of dirty lines are required whenever the cache is resized. Two lines with the same tag can now map to the same cache set. Therefore to ensure unique tags across all memory lines that map to a cache set, the new tags are comprised of the old tags concatenated with the index bits. The address decoding is shown in Figure 59.

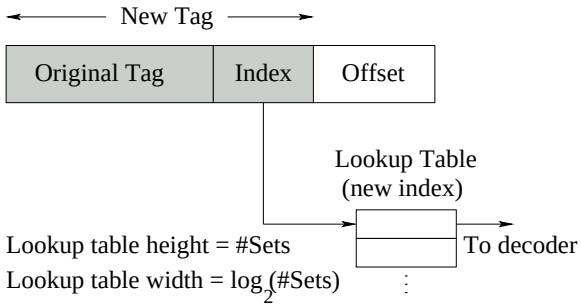


Figure 59. Cache decay resizing.

A simple version of customized placement is downsizing or sizing the cache upwards in powers of two. Powell *et al.*[67] adopt such a mechanism for instruction caches where the active cache size is increased or decreased based on miss rates being within a certain bound determined by profiling. The advantage of this strategy is the simple address translation mechanism which uses simple bit masking as shown in Figure 60. This scheme is adopted for data caches. The number of references for a time interval is used as our basis for sizing the cache. This scheme creates a smaller number of uniformly sized denser conflict sets upon downsizing.

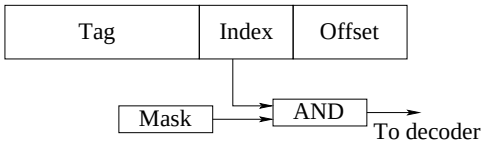


Figure 60. Power of two resizing.

In the segment resizing approach, the cache is split into segments that are a power of

two, where a segment is a group of contiguous cache sets. For example, a cache with 256 sets can be split into 8 segments of 32 sets each, or 16 segments of 16 sets, and so on. Conflict sets from one segment are folded with those in another segment based on reference count within a time interval. For example, with 8 segments and 256 conflict sets originally, conflict sets belonging to two segments can be folded which results in 224 conflict sets allowing 32 cache sets to be turned off.

This strategy attempts to size the cache on utilization. For example, if the utilization was 12.5%, and if the cache was divided into eight segments, ideally only one of the eight segments has to be turned on if the application footprint in the cache is contiguous. On the other hand, if the application footprint was small but non-contiguous, finer grained approaches are better suited.

The incoming address is split into a segment offset and the segment index. The look-up table is indexed using the original segment index to identify the new segment. The new segment index concatenated with the segment offset gives the new cache set index to be sent to the cache decoder. The new tag is the original tag appended with the segment index to ensure unique tags. The address decoding is shown in Figure 61.

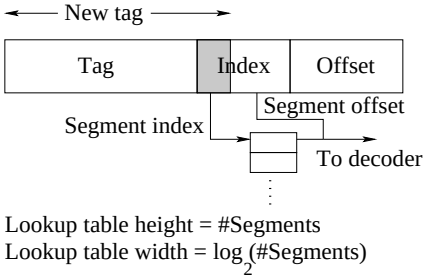


Figure 61. Cache segment resizing.

7.1.3 Performance Evaluation

The heuristics are named as follows: i) *decay* representing the cache decay technique with a sample interval of 512K cycles, ii) *dec-res:4b:2s*, representing the decay resizing heuristic

with a 4-bit counter with all folded conflict sets merged to two sets with the other *dec-res* heuristics representing changed parameters, iii) *pow-res* represents the power of two resizing technique with a threshold of 1000 accesses for downsizing, iv) *seg-res:125:8S* represents segment resizing with eight segments and a threshold of 125 accesses per segment per 512K cycles. The thresholds were chosen on the basis that on average access to an L2 cache line occurs once every 80000 cycles.

Average energy efficiency is shown in Figure 62. The normalized average execution time increases less than 4% for the heuristics, compared to a 5.5% increase for the cache decay technique. The folding heuristics have each memory line mapped to an active cache set. This provides resilience against occasional accesses to a conflict set that was folded, because the active cache set can satisfy the request. This feature allows more conflict sets to be folded and more aggressive turn offs to be scheduled.

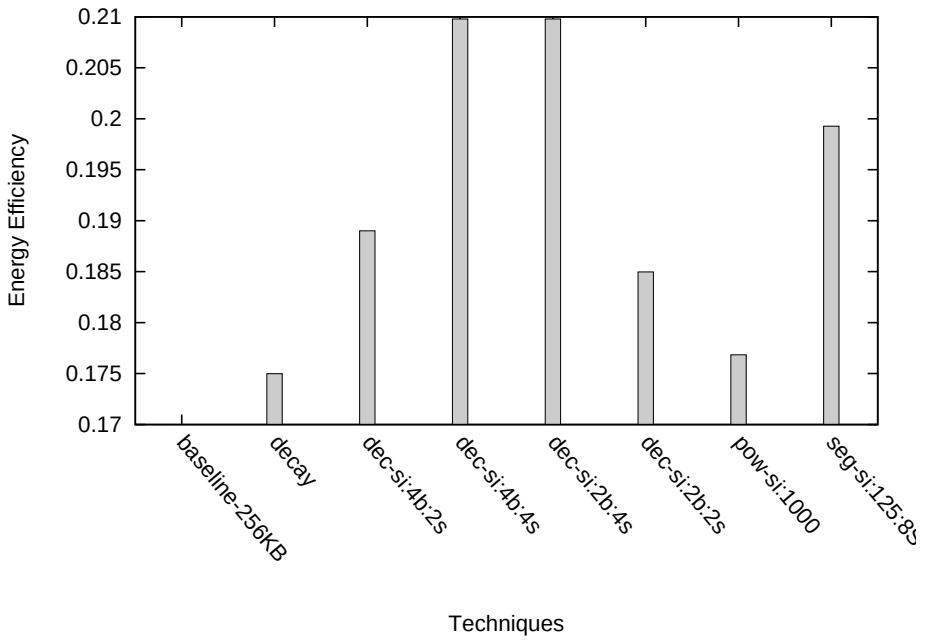


Figure 62. Energy efficiency comparison of folding heuristics.

Performance efficiency numbers are shown in Figure 63 also improves indicating that the heuristics folded conflict sets with large inactivity periods and therefore cache misses were kept low. Energy efficiency increases by about 20% relative to the base line. If half

the cache was turned off to reduce the leakage energy by half, the number of hits must stay constant for energy efficiency to double. Every added miss will add to execution time, and to leakage energy increasing the denominator and lowering the numerator in the energy efficiency equation. These effects provide the context for evaluating the efficiency improvement of 20%. This improvement is captured more effectively in the drop in EDP shown in Figure 64. The EDP improves by up to 45% compared with the base line cache.

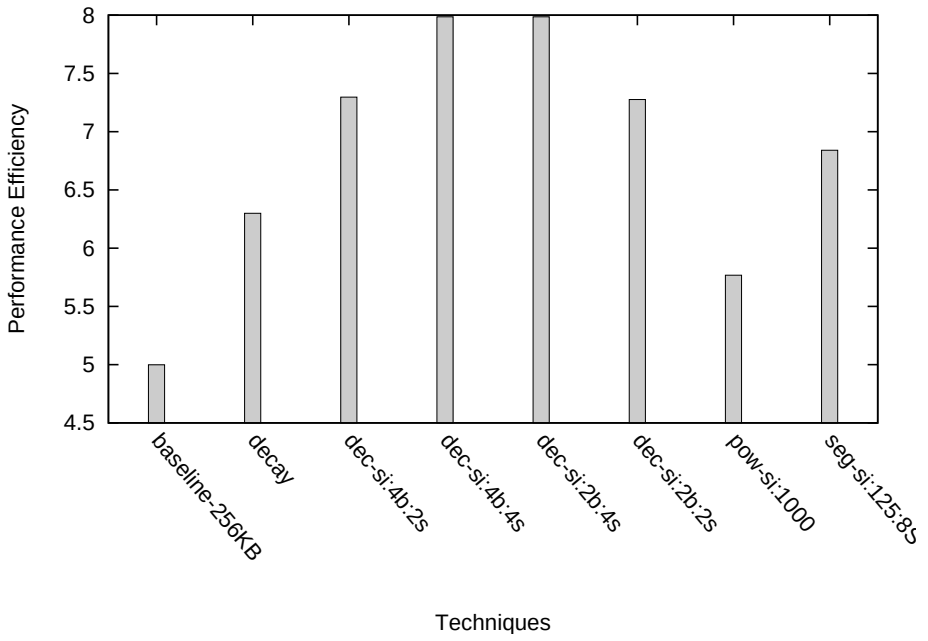


Figure 63. Performance efficiency of folding heuristics.

Among the heuristics studied, the 2-bit hierarchical counters for the decay resizing heuristic provide the largest improvements in EDP, because of the aggressive turn offs folding all unused conflict sets into just two sets. The segment resizing technique, reduces the EDP by 30%, but affects the execution time by less than 1%.

The power of two sizing heuristic lowered EDP by 20%, and performed worse than the original cache decay approach and the other heuristics. The reason for the under-performance is that it tries to create different "conventional caches," by resizing conflict sets uniformly, and therefore the sources of inefficiency remained.

The folding heuristics were found to be stable independent of the exact thresholds,

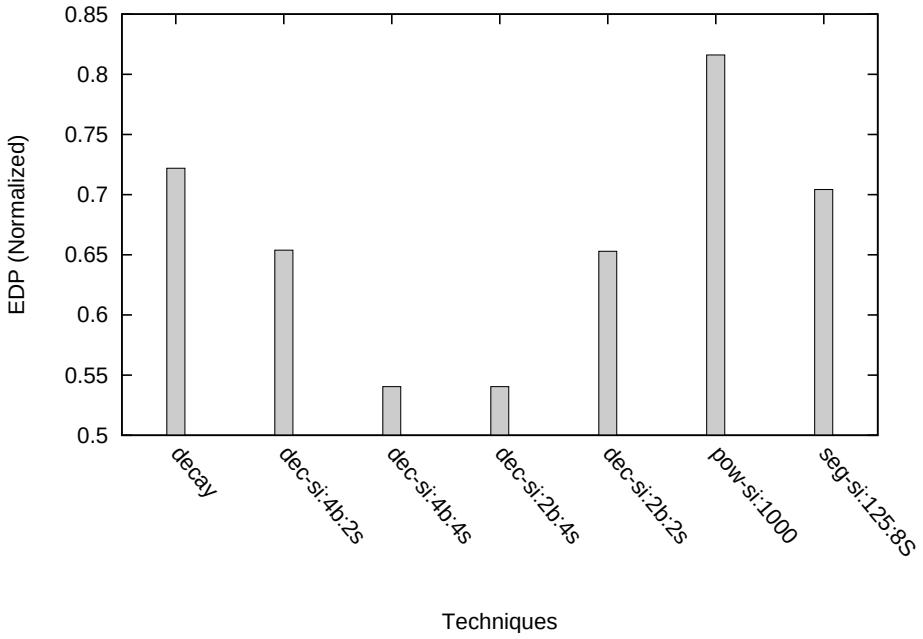


Figure 64. EDP of folding heuristics.

counters or banks chosen, indicating that the techniques are robust to the variance in memory reference behaviors, as long as the parameters chosen were in the expected reference range. The IPC degradation followed the execution time pattern, and was limited to less than 4% for all the heuristics.

Active management strategies by shaping and sizing the cache were found to increase cache efficiency and the performance yield of caches. The research completed thus far exhibits the potential of this approach in dealing with the costs associated with designs in the deep sub-micron region. Furthermore, it lays the foundation for improved approaches to be studied in increasing cache efficiency and reliability.

7.2 An Utilization Driven Framework for Improving Efficiency

With the shift of scaling from frequency to the number of cores, efficiency has become a primary design constraint. It is determined that cache performance efficiencies are less than 10% and energy efficiencies are in the range 1–5% for L2 data caches! This is especially significant because caches are dominant consumers of energy and area on-chip.

Consequently, the ability to scale the number of cores concurrently with pushing back the memory and power walls with small overall increases in die size will require significant improvements in cache performance efficiency and cache energy efficiency. This section provides strategies to improve L2/L3 cache efficiencies by coupling voltage scaling with flexible cache management policies to enable significant efficiency improvements in data caches. Specifically, a framework with a low overhead mechanism is proposed for the runtime computation of line/set utilization to i) voltage scale and size the cache to match program-phase footprint, and ii) shape the cache to the application memory behavior, i.e., change the placement function. The proposed techniques were applied to several benchmarks resulting in performance efficiency more than doubling, energy efficiency improving by 10% with a 10% improvement in an EDP.

Sizing and shaping are consolidated into a solution for improving L2/L3 cache efficiency. The key insight is to combine sizing and shaping with voltage scaling in the cache. As voltages are scaled down, the defect-free failure rate for cache memory cells increases, for example due to timing failures [118]. Additionally, some program phases have smaller cache footprints than others. These two insights are combined into an off-line generated voltage-sizing profile of the cache and this profile is traversed dynamically. First, we propose a low overhead heuristic for the on-line computation of cache utilization. An empirical analysis has demonstrated that this heuristic tracks the actual utilization very closely. The utilization measurements are used to *up-size/down-size* the cache. Each change in the cache size is accompanied by a refinement of the placement function to map memory to the active cache lines. Companion measurements of miss rates are used to modulate sizing decisions to prevent efficiency gains that carry excessive performance penalties e.g., significantly higher miss rates. The result is significant improvements in performance and energy efficiencies with an EDP improvement of over 10%.

7.2.1 Empirical Analysis

Detailed analyses of the utilization and efficiency of traditional caches can be summarized as follows. The simulations were performed using *Simplescalar* [89] and the energy modeling was performed using *Cacti* 5.1 [117] using the ITRS-HP model. More details about the simulation methodology can be found in Section 7.2.6.1. The utilization and performance efficiency numbers are shown in Figure 65, and the energy efficiency numbers is Figure 66.

- The low utilizations indicate that the majority of the cache energy and area costs are spent in maintaining *dead* lines.
- Performance efficiency, for which the upper bound is utilization, is shown in Figure 66 and averages less than 10% for the L2 cache. Although, most of the applications had a low miss rate, performance efficiencies were still considerably lower than utilizations reflecting that the residency of live lines was not well exploited for performance.
- Cache energy efficiency with current designs averages under 5% (Figure 66) with leakage being the prime source of inefficiency.
- When associativity is increased, miss rates are lowered, but dead cycle counts increase because of the deeper LRU stack. When the cache size is increased, it may decrease miss rates, but more lines are dead. Thus, utilizations remain flat with cache sizes and associativity.
- L1 cache utilizations were comparable to that of the L2 cache.

To make significant improvements in efficiency it is imperative to: i) reduce the number of dead lines in the cache, and, ii) make better use of active lines. The first step improves energy efficiency and the second improves performance efficiency. The approach reported here differs from prior efforts in: i) coupling voltage scaling with resizing and remapping,

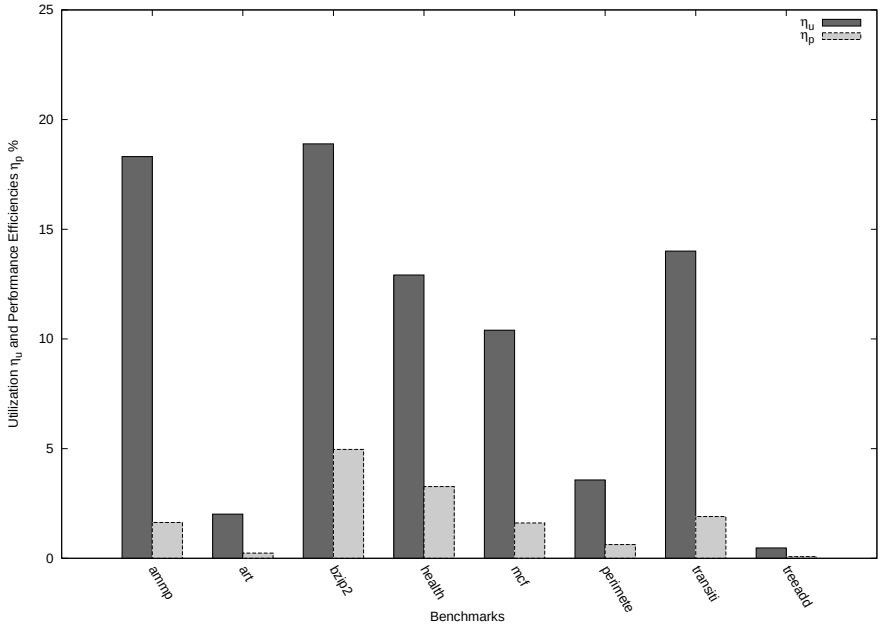


Figure 65. Utilization and Performance Efficiencies for a traditional 256KB L2 Cache

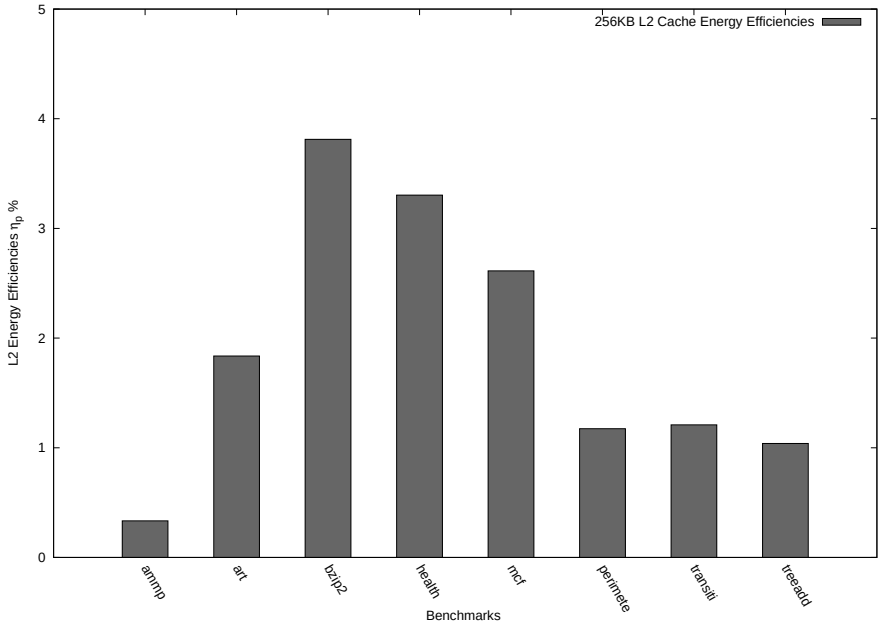
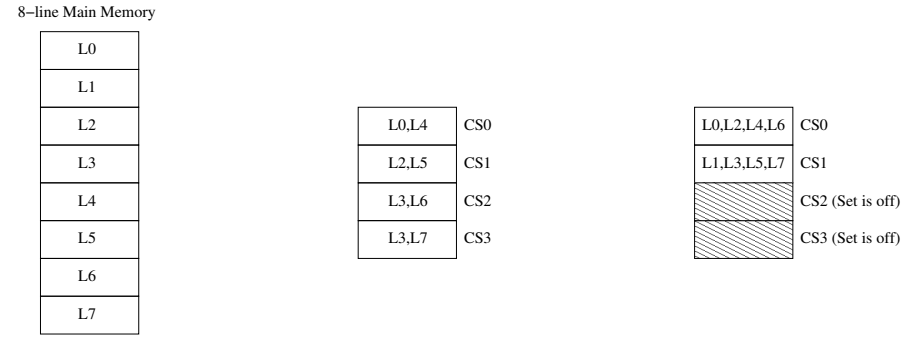


Figure 66. Energy efficiencies for a 256KB L2 cache

and ii) creation of a static voltage-sizing profile that is dynamically traversed, with iii) cache shaping performed dynamically using a model employing utilization and miss-rate. Thus, caches are one-time reconfigured to produce this profile (post manufacturing) and



(a) Eight line main memory

(b) Modulo placement

(c) Energy Efficient Cache

Figure 67. Conflict set construction

this profile is traversed in an application-specific manner. Specifically we need to identify triggers to move to a new voltage-sizing point concurrently with the computation of a new placement function.

7.2.2 Operational Model

Customized placement is implemented for the L2 cache as described in Section 4.5. The L2 access is remapped through a lookup table (to implement customized placement) and this additional lookup is overlapped with the L1 access hiding the address translation. Operationally, when a down-sizing or up-sizing operation is performed, the remapping table is reloaded with address translations, which is performed in software. The existence of the ability to turn off cache lines using the *Gated-Vdd* approach proposed by Powell *et al*.[67] is assumed, which enables turning off the supply voltages to caches lines, and has an additional area cost of 3%. The area and energy costs of the lookup table is addressed in Section 7.2.6 and is negligible.

7.2.3 On-Line Cache Management

Improving performance and energy efficiencies relies on three main steps: i) off-line characterization, ii) on-line computation of cache utilization, iii) cache sizing, and, iv) cache shaping.

7.2.3.1 *Off-line Characterization*

Several studies have documented the challenges of the manufacturing process to fabricate devices with design tolerances because of the significant variations in transistor device characteristics within a die (WID), across dies (D2D), and between wafers (W2W) [119, 6]. Parameter variation can cause SRAM cell failures due to a variety of factors for example, threshold voltage fluctuation in transistors can occur due to random dopant effects, as well as parametric variations due to the manufacturing process including sub-wavelength lithography, imprecision in chemical polishing, and uneven exposures. Thus the number of fault free cells in the cache will vary with the cache voltage level and parametric effects are generally spatially correlated.

Assumptions include i) the existence of built-in-self-test (BIST) capability for the cache as described in Agarwal *et al*.[82], ii) operation of the L2 cache as a separate voltage island (for example, the Barcelona die has separate voltage island for the L3 cache), and iii) the availability of four voltage levels. The BIST is operated off-line at each voltage level to identify fault free sets. A cache set is marked as failed if one cache line within it is marked failed—this is a line with at least one failed bit cell. Failures are assumed to be monotonic, i.e., a line marked faulty at a voltage level cannot be operational at a lower voltage level. This information is captured in a voltage-sizing map that reflects available fault free cache sets at each voltage level. This map can be made more aggressive by further down-sizing the cache at each voltage. However, the converse is not true. The following sections deal with *when* to resize (equivalently change voltage levels and power down unused lines) and *how* to remap memory to the active cache sets.

7.2.3.2 *Online Computation of Cache Line Utilization*

Effectively sampling activity in the data cache is found to provide a reasonable estimate of utilization. The approach is as follows. Within each set, the last hit is recorded via a hit status bit associated with each line. The status bit will be equal to 1 for the last line hit in the set, and 0 for all other lines in the set. The hit status bits of all lines are sampled

every S cycles. The utilization is computed every $W > k \times S$ cycles for some integer k. The utilization of a line is determined by the number of times its hit status bit is set in this interval W. If it is k, the utilization for that line is 100%. This cache line utilization averaged across all cache lines is the measured global cache utilization. The simulations used a value of one million cycles for one sampling period, and used 5 million L2 cache references as the value for W.

Figure 68 illustrates the accuracy of the measured utilization by comparing it with simulations that accurately keep track of real utilization. The sampled values are within 10% of the actual utilization values in absolute terms for all benchmarks (for the majority of the benchmarks the measured utilization is within 5% of actual). As shown in Section 7.2.6, this degree of accuracy is sufficient to drive sizing decisions to produce substantive improvements in efficiency.

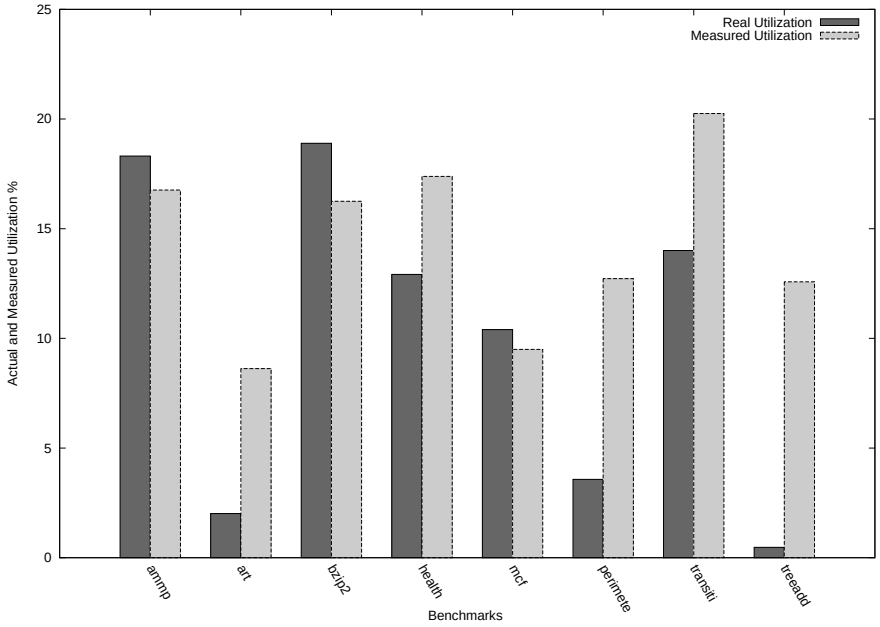


Figure 68. Utilization Measured vs. Actual

7.2.4 Cache Sizing

The memory behavior of programs can be viewed as progressing through a sequence of phases where the memory referencing behavior of each phase is characterized by its working set of memory lines (often referred to as its *footprint*). At phase transitions, online utilization measurements drive sizing decisions. Phase detection is a challenging problem that has inspired several efforts [93]. While this approach can naturally accommodate such phase detectors, here it is shown that even with empirically determined fixed duration phases, efficiencies can be substantially improved. Specifically the results are presented for phases that are 5M references long. As described in the preceding section, the measured utilization at the end of each phase maps is used to select a cache size/voltage.

Since real failure rate data necessary to construct the maps in Section 7.2.3.1 is lacking, two synthetic voltage-sizing maps are utilized. The first provides caches sized at 25%, 50%, 75%, and 100% of the original cache with voltage levels assigned accordingly (i.e., the highest voltage level is associated with a fully sized cache). The second map utilizes cache sizes of 60%, 70%, 80%, and 100% with increasing voltage levels. This map is informed in part by the work in Chapter 5 that noted significant performance penalties when the unusable cache size exceeded 40%. Utilization values uniformly index both maps, i.e., a measured average line utilization of 50% would index into the either a cache size of 50% or 70% respectively. Thus operationally, at the end of a phase, the average measured cache utilization is used to index a map to obtain a new cache size, and a new placement function is computed and loaded into the address remapping table in the L2.

7.2.5 Shaping the Cache

Shaping the cache is comprised of two steps—i) computing the placement function for a specific cache size, and ii) configuring the cache to implement this new mapping. Two heuristics are defined for the former.

7.2.5.1 Computing the Placement Function

At the end of a phase, a software implementation updates or improves the cache efficiency as follows. The global cache utilization across all sets is computed from the cache line utilizations. This utilization is used to index the voltage-sizing map which identifies the sets that will be active in the next phase. The group of sets that will be powered down are known through the model described in Section 7.2.3.1. The corresponding conflict sets (memory lines that map to these sets) must be merged with conflict sets that map to cache sets that are active in the next phase. Merging takes place with active cache sets that had the lowest utilization in the last phase. In this implementation cache down-sizing is performed in steps as described here. However, only one cache up-sizing operation is performed—full cache operation.

For comparison purposes the nearest neighbor merging described in Agarwal *et al*. [82] where only the pairs of conflict sets that map to adjacent sets in the cache are merged is also evaluated. This approach is simpler in that no status information is maintained nor is any online decision made and is described as Option 2 in Figure 70. However, both techniques are invoked based on phase changes as determined by the online computation of cache utilization using the heuristic and cache miss rates and the existing voltage level (corresponding to a certain number of cache sets that are powered down).

7.2.5.2 Configuring the Cache

The remapping is implemented by updating the address remapping table which is illustrated in Figure 69. The configuration update is performed in software. The hardware logic requires BIST data, which contains information on which sets fail at which voltage level. A set of four mask registers contain the status of each cache set at each of four voltage levels. For a given cache set, the voltage level indicator indexes the mask bits for the set. The selected mask bit is used to power down the corresponding set or leave it powered on. These mask registers will use $256 * 4$ bits. Additionally, hit status bits have to be maintained per cache line (2048 bits for a 256KB 8-way 128 byte L2 cache). The major

investment in area comes from the lookup table used for remapping. For the 256 KB L2 used in our experiments, 256, byte-wide entries are required for this look up table. The new tag is the old tag augmented with the original cache set index because the cache tag array may contain tags from customized as well as modulo placement. This adds another 256*8 bits, resulting in a total additional storage of 896 bytes for customized placement, which is insignificant compared to the L2 cache size. The energy consumption of this additional logic is less than 1% of the total cache energy consumption. Finally, when the placement is changed, cache lines that had multiple conflict sets mapping to it are written back if dirty and invalidated to avoid any synonym or aliasing problems.

In terms of latency, an additional cycle is required for the address translation logic. While this can be easily masked by performing the lookup in parallel with an access to the L1 cache, even if the lookup was not performed in parallel with the L1 cache access, the additional cycle does not alter execution time by more than 2% in our simulations.

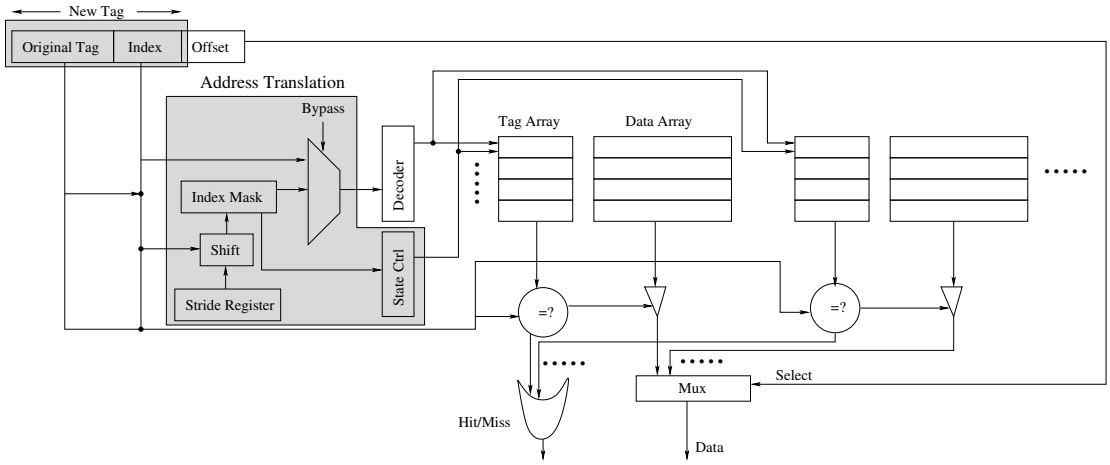


Figure 69. Hardware Implementation

7.2.6 Performance Evaluation

7.2.6.1 Simulation Methodology and Assumption

The execution of a subset of applications from the SPEC2000 suite [86] and DIS [88] and Olden [87] was simulated. The applications were simulated using the *Simplescalar*

Input: $util[\ ], utilization, missrate$
Output: $remap[\ ]$

1: $initialize(trigger\_old, trigger\_new, new\_voltageindex, old\_voltageindex)$
2: $trigger\_new = utilization * t_c/(t_c + t_p * missrate)$ { This is the metric used to detect phases in memory behavior;this combines utilization and miss rate}
3: **if** $trigger\_new > trigger\_old$ **then**
4: $new\_voltageindex = int(utilization)/25$ {Defining four voltage indices based on utilization, with index 3 corresponding to the highest voltage level}
5: **else**
6: $turn\_on\_all\_sets(); new\_voltageindex = 3$ {if performance is suffering more than the savings in utilization, turn all sets back on}
7: **end if**
8: **if** $new\_voltageindex > old\_voltageindex$ **then**
9: $new\_voltageindex = old\_voltageindex$ {up-sizing the cache cannot occur in steps other than turning all sets back on, whereas down-sizing can occur in steps}
10: **end if**
11: $turn\_off(new\_voltageindex)$ {schedule cache sets to turn off based on corresponding voltage level which were identified during BIST}
12: **for all** faulty cache sets, fc, at $new\_voltageindex$ **do**
13: OPTION 1: $remap(fc) = faultfree\_nextneighbor\_set$
14: OPTION 2: $remap(fc) = faultfree\_lowestutil\_set$
15: OPTION 2: $update(util[\ ])$ { the utilization array is updated with the utilization of the set being remapped being added to the set to which it is remapped}
16: **end for**{ Option 1 signifies the next neighbor strategy, whereas Option 2 is the utilization driven remapping strategy}
17: $trigger\_old = trigger\_new$
18: $old\_voltageindex = new\_voltageindex$

Figure 70. Shaping Algorithm using Utilization

3.0 [89] simulator.

Energy estimates were obtained using *Cacti* 5.1 [117] for 70 *nm* technology, which is the latest version of *Cacti* (therefore, the efficiency numbers without any energy management were revised upwards from the numbers in earlier chapters; this is because the latest version use the ITRS-HP technology models). The L2 cache access latency was assumed to be 15 cycles. The efficiency definition assumes that the read and write energies are the same—which increases energy efficiency compared to a more precise definition. Leakage energy is predominant and cache writes constitute a small fraction of the total number of accesses, these assumptions affected efficiency by less than 1%, as given by *Cacti* estimates.

The energy was calculated with the cache operating at the highest expected frequency as given by *Cacti* estimates. The L1 cache configuration was 16KB 2-way with 64-byte lines. Results focus on the efficacy of the shaping algorithm in saving energy without regressing performance by turning off cache sets. As a result, only the effect of the cache size on energy savings (though this results from voltage scaling) is estimated; the energy saved in the cache and processor through voltage scaling as well as the increases in the cycle period is not accounted in the results.

7.2.6.2 Results and Discussion

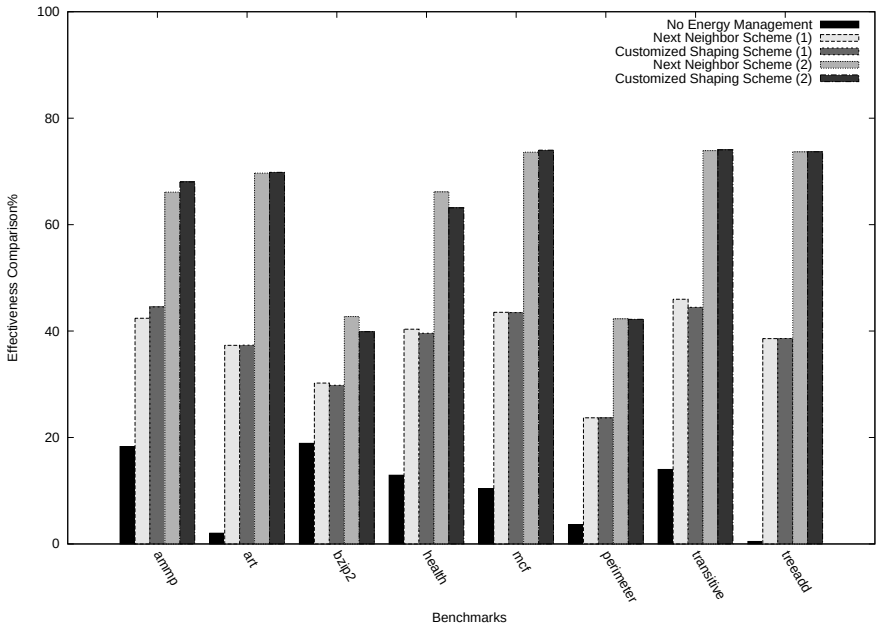


Figure 71. Effectiveness comparison

The five schemes represented in the figure are as follows:

1. No energy management representing traditional caches
2. Next neighbor scheme (1) representing the next neighbor scheme with cache sizing maps of 60, 80, 90 and 100%
3. Customized Shaping scheme (1) representing the utilization based remapping scheme with cache sizing maps of 60, 80, 90 and 100%

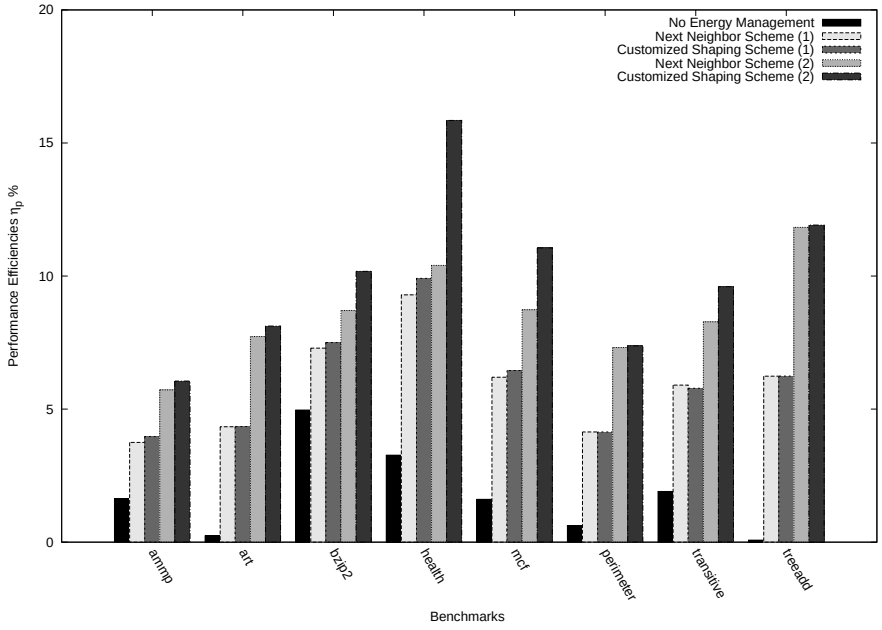


Figure 72. Performance Efficiency comparison

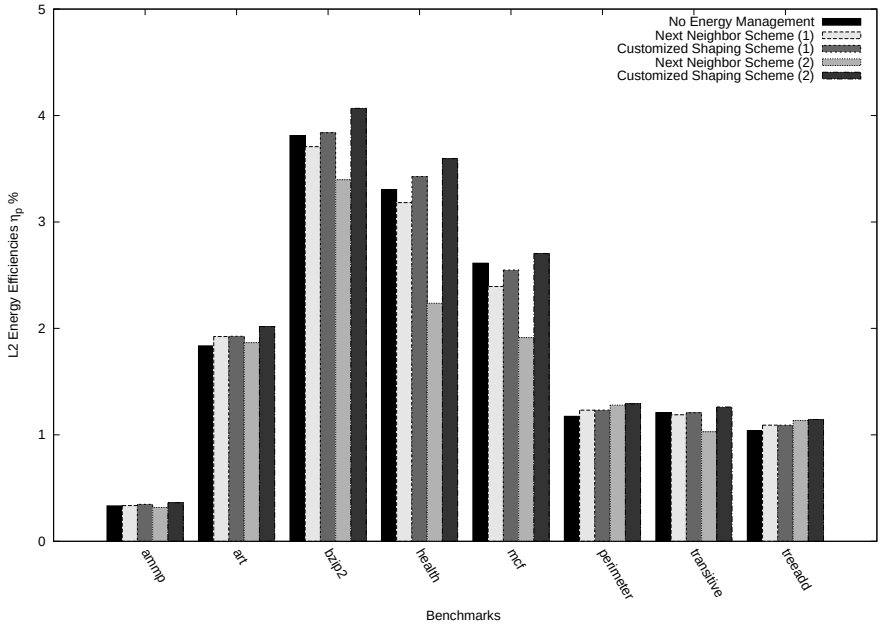


Figure 73. Energy Efficiency comparison

4. Next neighbor scheme (2) representing the next neighbor scheme with cache sizing maps of 25, 50, 75 and 100%

5. Customized Shaping scheme (2) representing the utilization based remapping scheme

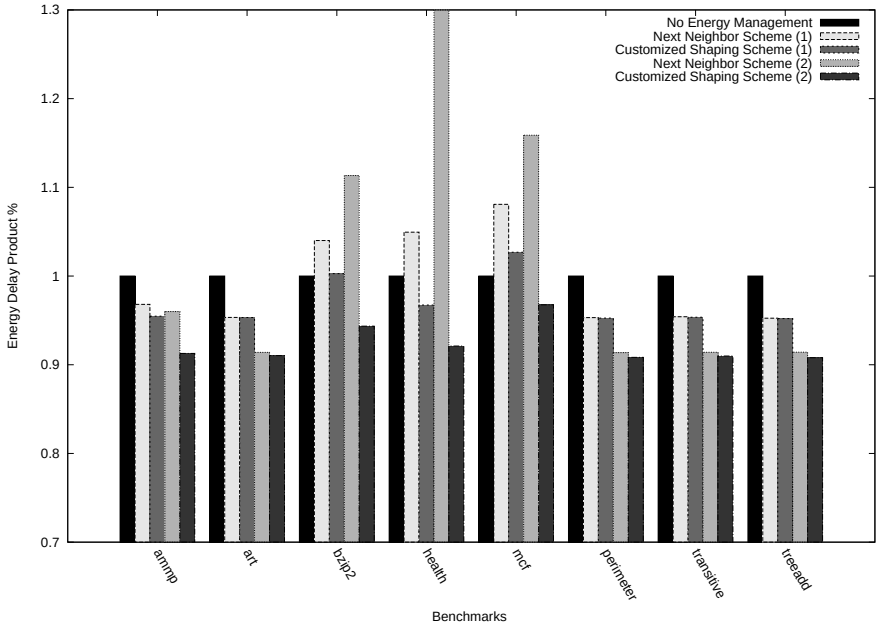


Figure 74. Energy Delay Product comparison

with cache sizing maps of 25, 50, 75 and 100%

From Figure 71, it can be observed that effectiveness has increased significantly across all applications with all the energy management schemes. The next neighbor schemes perform as well as the customized shaping schemes in terms of effectiveness—this can be explained by the utilization tracking mechanism which predicts shut down phases well.

However, the performance of the schemes diverge when performance efficiencies are compared, as shown in Figure 72. When performance efficiencies are compared, the customized shaping schemes relatively out perform the next neighbor schemes on average by 10% (a 1% difference in performance efficiency represents a 10% improvement). Compared to the traditional caches, performance efficiencies are more than doubled in many benchmarks.

The comparison of energy efficiencies are shown in Figure 73. Again, it is seen that energy efficiency increases for all applications over the traditional cache for many benchmarks. Though the energy efficiency improvements as an absolute percentage seem modest, it has to be factored in that there are some fundamental constraints while attempting to

improve energy efficiency. Given, that an L2 cache line is accessed only once every few million cycles, leakage energy dominates and this constraint keeps tabs on any improvements achieved. A 1% improvement in energy efficiency is a 25% increase—this has to be factored in the evaluation. The customized shaping scheme consistently outperforms the next neighbor schemes by 0.05–1.0%, representing an average improvement of 0.5% (again, this is a 10–15% improvement relative to the values of energy efficiency).

The increase in energy efficiency can be captured by looking at the Energy Delay Product as shown in Figure 74. EDP is calculated for the cache alone, since it is assumed to be in a separate voltage domain. EDP for most benchmarks are reduced by more than 15% using the customized shaping scheme with *no EDP regressions* for any benchmark. The next neighbor approach, however suffers significant regression for many benchmarks; for example, the EDP for benchmark *health* doubled using next neighbor (this is not represented in the figure). In summary, both energy management schemes perform well using the online utilization tracking mechanism, whereas the customized shaping scheme is able to provide an additional 10–15% relative improvements in energy efficiencies and EDP.

Relative to the prior work demonstrated in this thesis, the approach here is distinct in the following: i) integration of voltage scaling, ii) computation of a static profile of voltage-sizing behavior, and iii) dynamically traversing this profile using run-time measured utilization. In addition this chapter proposes a low cost, and effective approach for the run-time computation of cache set/line utilization that could readily be used to drive other cache management policies.

7.3 Concluding Remarks

The shift to scaling via increasing the number of cores now places a premium on more efficient cache design. Large L2/L3 caches are dominant and very inefficient consumers of energy and therefore buying performance with larger caches is no longer feasible. Efficiency will be a dominant design driver for future processors. Many applications are hard to

characterize statically, and therefore runtime strategies are necessary. This chapter presents frameworks and implementation strategies for substantive improvements in the efficiency of on-chip caches. The first part of the chapter focusses on using runtime heuristics to improve efficiencies, whereas the second part provides a means to track runtime utilization and efficiency and couples this with voltage scaling to realize gains in cache efficiency.

CHAPTER 8

CONCLUSION

This chapter explores some of the potential research directions that can emerge from this thesis and briefly summarizes how these techniques presented in this thesis can be adopted and improved upon. Following this, this chapter concludes this dissertation by summarizing the work presented in this dissertation.

8.1 Future Extensions

Design trends indicate a shift towards multiprocessing as envisioned by the increase in the number of multi-core processors. Many of these processors have independent L1 caches with a shared last level cache. In traditional shared caches, the cache allocated to any core is a function of the memory behavior of applications executing on both cores. Although this approach is simple, this may lead to inefficiency as the competing programs from the multiple cores may thrash in the cache leading to poor efficiencies.

Sizing and shaping the cache provides for a single cache having shared partitions as well as private partitions for competing applications to improve efficiencies further. Private partitions can eliminate thrashing among competing applications. Sharing cache sets allows certain cache sets to be unused for periods of time permitting power down strategies to be scheduled for energy savings. Similarly L1 and L2 caches need not be hardwired for inclusiveness or exclusiveness; they may be customized for inclusiveness or exclusiveness using placement.

The research can also be extended to allocate the cache across multiple threads and multiple cores. For example, a program for which performance improves with a larger cache size is allocated a larger portion of the cache at the expense of a program that does not benefit from increased cache size. The cache partitions allocated to the competing threads or applications can be shaped to minimize misses and improve efficiencies further.

This research also opens up new opportunities for compiler-driven placement optimizations. Compiler techniques for optimizing strided array accesses include data pre-fetching, data re-layout and loop transformations (e.g., Mowry *et al*.[44], Chilimbi *et al*.[51], Panda *et al*.[52], Rabbah and Palem[53]). These optimizations are targeted to fixed modulo placement caches. By making the cache placement customizable, these techniques have greater freedom, e.g., a specific layout may no longer preclude certain instruction schedules for good performance. This can lead to powerful optimizations combining compiler optimizations with cache placement optimizations. Another related area is using customized placement for scientific computing applications in conjunction with algorithms that are tuned to cache parameters such as associativity and line size.

Another promising branch of this work is adaptive power management built on accurate models of devices that incorporate failure modes, leakage power, short circuit power etc. The goal is to adaptively trade-off bit error rate, power, and performance inspired by the work with microprocessor datapaths [120]. A convergence of these techniques with the datapath techniques can produce robust single chip embedded processor power management strategies for deep sub-micron implementations.

8.2 Summary

This dissertation addresses two major sets of challenges facing processor design as the industry enters the deep sub-micron region of semiconductor design. The first set of challenges relates to the well understood memory bottleneck [1] that shows no signs of easing even as the focus shifts from scaling processor frequency to scaling the number of cores. This trend has led to the increasing reliance on larger on-chip caches which occupies 50–60% of area on chip and consuming 15–30% of energy expended on chip.The second set of challenges is posed by transistor leakage and process variation (both inter-die and intra-die) at future technology nodes with leakage power anticipated to increase exponentially and sharply lower defect-free yield with successive technology generations. This dissertation

focuses on resolving these two challenges by abstracting them as one problem—developing efficient caches.

This dissertation defines efficiency as it applies to caches and analyses the efficiency of modern caches for various workloads and finds them to be extremely low (with performance efficiencies less than 15% and energy efficiencies in the order of 1%), with a majority of the cache storing data that will not be re-used (with utilization less than 25%).

This is followed by strategies to improve efficiency, predicated on addressing the two sources if inefficiency—transistor leakage and the manner in which main memory lines share the cache in traditional caches. The approach to improving energy efficiency primarily relies on sizing the cache to match application memory footprint or working set during a program phase and powering down all remaining cache lines. The approach to improving cache utilization and performance efficiency primarily relies on cache shaping, i.e., changing the placement function.

Sizing and shaping are applied at different phase of the design cycle. The thesis described the various solutions that were proposed for different domains to improve the efficiency of caches. These included solutions that deal with off-line profile driven strategies followed by those applying compile-time strategies and strategies that are applied at runtime using runtime measurements. This dissertation contributes techniques at all of the preceding points in the life cycle of a program.

By giving software greater control over the cache structure, some of the performance headroom that is obscured by fixed, design time partitioning of cache resources across memory lines is recovered leading to significant improvements in cache execution time performance and energy performance leading to improvements in cache energy efficiency, cache performance efficiency, average memory access times and Energy-Delay products.

In conclusion, the active management techniques presented were found to improve cache execution times and energy performance significantly compared to traditional cache

designs leading to improved energy and performance efficiencies. In addition active management of cache resources provides graceful performance degradation in the presence of defects in the cache leading to improved performance yield. The presented techniques offer a more flexible and cost-effective approach to solve current-day technology challenges in terms of energy consumption and performance scaling compared to traditional approaches including increasing associativity, sizes and agnostic powering down of cache segments. The flexibility of the approach lends it to be applied in various domains at various program execution points; for example, active management can be applied one-time, at compile-time or at run-time using available data including off-line profile analysis, compile-time analysis or run-time measurement. Additionally, the improvements are achieved with relatively moderate hardware and software overheads.

REFERENCES

[1] S. A. McKee, "Reflections on the memory wall," in *Proc. of the Conference on Computing Frontiers*, 2004.

[2] P. Ranganathan, S. Adve, and N. P. Jouppi, "Reconfigurable caches and their application to media processing," in *Proc. of the International Symposium on Computer Architecture*, pp. 214–224, 2000.

[3] Y.-F. Tsai, A. Hegde, N. Vijaykrishnan, and M. J. Irwin, "Chippower: An architecture-level leakage simulator," in *Proc. of IEEE System on Chip Conference*, pp. 395–398, September 2004.

[4] M. Zhang and K. Asanovi, "Fine-grain CAM-tag cache resizing using miss tags," in *Proc. of the International Symposium on Low power Electronics and Design*, pp. 130–135, 2002.

[5] Intel Corporation, *Dual-Core Intel Itanium 2 Processor 9000 Series Datasheet*, July 2006.

[6] S. Borkar, "Design challenges of technology scaling," *IEEE Micro*, vol. 19, no. 4, pp. 23–29, 1999.

[7] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers.," in *Proc. of the International Symposium on Computer Architecture*, pp. 364–373, 1990.

[8] M. K. Qureshi, D. Thompson, and Y. N. Patt, "The v-way cache: Demand based associativity via global replacement.," in *Proc. of the International Symposium on Computer Architecture*, pp. 544–555, 2005.

[9] J.-K. Peir, Y. Lee, and W. W. Hsu, "Capturing dynamic memory reference behavior with adaptive cache topology," in *Proc. of the International conference on Architectural support for programmin g languages and operating systems*, pp. 240–250, 1998.

[10] A. Agarwal, J. Hennessy, and M. Horowitz, "Cache performance of operating system and multiprogramming workloads," *ACM Trans. Comput. Syst.*, vol. 6, no. 4, pp. 393–431, 1988.

[11] A. Agarwal and S. D. Pudar, "Column-associative caches: A technique for reducing the miss rate of direct-mapped caches," in *Proc. of the International Symposium on Computer Architecture*, 1993.

[12] B. Calder, D. G, and J. Emer, "Predictive sequential associative cache," in *Proc. of the International Symposium on High Performance Computer Architecture*, 1996.

[13] E. G. Hallnor and S. K. Reinhardt, "A fully associative software-managed cache design.," in *Proc. of the International Symposium on Computer Architecture*, pp. 107–116, 2000.

[14] Z. Chishti, M. D. Powell, and T. N. Vijaykumar, "Distance associativity for high-performance energy-efficient non-uniform cache architectures," in *Proc. of the International Symposium on Microarchitecture*, 2003.

[15] A. Seznec, "A case for two-way skewed-associative caches.," in *Proc. of the International Symposium on Computer Architecture*, pp. 169–178, 1993.

[16] M. Kharbutli, K. Irwin, Y. Solihin, and J. Lee, "Using prime numbers for cache indexing to eliminate conflict misses.," in *Proc. of the International Symposium on High Performance Computer Architecture*, pp. 288–299, 2004.

[17] N. Topham and A. Gonzalez, "Randomized cache placement for eliminating conflicts," *IEEE Trans. Comput.*, vol. 48, no. 2, pp. 185–192, 1999.

[18] C. Zhang, "Balanced cache: Reducing conflict misses of direct-mapped caches," in *Proc. of the International Symposium on Computer Architecture*, (Washington, DC, USA), pp. 155–166, IEEE Computer Society, 2006.

[19] J. T. Robinson and M. V. Devarakonda, "Data cache management using frequency-based replacement," in *SIGMETRICS '90: Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, (New York, NY, USA), pp. 134–142, ACM Press, 1990.

[20] S. Jiang and X. Zhuang, "Lirs: An efficient low inter-reference recency set replacement policy to improve buffer cache performance," in *Proc. of SIGMETRICS*, 2002.

[21] R. Subramanian, Y. Smaragdakis, and G. H. Loh, "Adaptive caches: Effective shaping of cache behavior to workloads," in *Proc. of the International Symposium on Microarchitecture*, (Washington, DC, USA), pp. 385–396, IEEE Computer Society, 2006.

[22] Y. Smaragdakis, S. Kaplan, and P. R. Wilson, "EELRU: Simple and effective adaptive page replacement," in *Measurement and Modeling of Computer Systems*, pp. 122–133, 1999.

[23] T. R. Puzak, *Analysis of cache replacement-algorithms*. PhD thesis, 1985.

[24] J.-S. Lee, W.-K. Hong, and S.-D. Kim, "Design and evaluation of a selective compressed memory system," in *Proc. of the IEEE international Conference on Computer Design*, (Washington, DC, USA), p. 184, IEEE Computer Society, 1999.

[25] Y. Zhang, J. Yang, and R. Gupta, "Frequent value locality and value-centric data cache design," *SIGPLAN Not.*, vol. 35, no. 11, pp. 150–159, 2000.

[26] A. R. Alameldeen and D. A. Wood, "Adaptive cache compression for high-performance processors," in *Proc. of the International Symposium on Computer architecture*, (Washington, DC, USA), p. 212, IEEE Computer Society, 2004.

[27] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis, "The case for compressed caching in virtual memory systems.," in *USENIX Annual Technical Conference, General Track*, pp. 101–116, 1999.

[28] K. Ghose and M. B. Kamble, "Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line segmentation," in *Proc. of the International Symposium on Low power Electronics and Design*, (New York, NY, USA), pp. 70–75, ACM Press, 1999.

[29] C.-L. Su and A. M. Despain, "Cache designs for energy efficiency," in *Proceedings of the Hawaii International Conference on System Sciences*, (Washington, DC, USA), p. 306, IEEE Computer Society, 1995.

[30] F. Dahlgren and P. Stenstrom, "On reconfigurable on-chip data caches," in *Proc. of the International Symposium on Computer Architecture*, pp. 189–198, 1991.

[31] P. Petrov and A. Orailoglu, "Towards effective embedded processors in codesigns: customizable partitioned caches," in *Proc. of the International Symposium on Hardware/Software codesign*, (New York, NY, USA), pp. 79–84, ACM Press, 2001.

[32] H.-H. S. Lee and G. S. Tyson, "Region-based caching: an energy-delay efficient memory architecture for embedded processors," in *Proc. of the 2000 international conference on Compilers, architecture, and synthesis for embedded systems*, (New York, NY, USA), pp. 120–127, ACM Press, 2000.

[33] H. S. Stone, J. Turek, and J. L. Wolf, "Optimal partitioning of cache memory," *IEEE Trans. Comput.*, vol. 41, no. 9, pp. 1054–1068, 1992.

[34] G. E. Suh, S. Devadas, and L. Rudolph, "A new memory monitoring scheme for memory-aware scheduling and partitioning," in *Proc. of the International Symposium on High Performance Computer Architecture*, (Washington, DC, USA), p. 117, IEEE Computer Society, 2002.

[35] R. Iyer, "Cqos: a framework for enabling qos in shared caches of cmp platforms," in *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, (New York, NY, USA), pp. 257–266, ACM Press, 2004.

[36] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni, "Communist, utilitarian, and capitalist cache policies on cmps: caches as a shared resource," in *Proc. of the IEEE international Conference on Parallel Architectures and Compilation Techniques*, (New York, NY, USA), pp. 13–22, ACM Press, 2006.

[37] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proc. of the International Symposium on Microarchitecture*, (Washington, DC, USA), pp. 423–432, IEEE Computer Society, 2006.

[38] J. Chang and G. S. Sohi, "Cooperative caching for chip multiprocessors," in *Proc. of the International Symposium on Computer Architecture*, (Washington, DC, USA), pp. 264–276, IEEE Computer Society, 2006.

[39] C. Liu, A. Sivasubramaniam, and M. Kandemir, "Organizing the last line of defense before hitting the memory wall for cmps," in *Proc. of the International Symposium on High Performance Computer Architecture*, (Washington, DC, USA), p. 176, IEEE Computer Society, 2004.

[40] C. Kim, D. Burger, and S. W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in *Proc. of the international conference on Architectural support for programming languages and operating systems*, (New York, NY, USA), pp. 211–222, ACM Press, 2002.

[41] Z. Chishti, M. D. Powell, and T. N. Vijaykumar, "Distance associativity for high-performance energy-efficient non-uniform cache architectures," in *Proc. of the International Symposium on Microarchitecture*, (Washington, DC, USA), p. 55, IEEE Computer Society, 2003.

[42] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler, "A nuca substrate for flexible cmp cache sharing," in *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, (New York, NY, USA), pp. 31–40, ACM Press, 2005.

[43] C.-K. Luk and T. C. Mowry, "Automatic compiler-inserted prefetching for pointer-based applications.," *IEEE Trans. Computers*, vol. 48, no. 2, pp. 134–141, 1999.

[44] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching.," in *Proc. of the International conference on Architectural support for programmin g languages and operating systems*, pp. 62–73, 1992.

[45] J. Kim, R. M. Rabbah, K. V. Palem, and W.-F. Wong, "Adaptive compiler directed prefetching for epic processors.," in *PDPTA*, pp. 495–501, 2004.

[46] J. Kim, K. V. Palem, and W.-F. Wong, "A framework for data prefetching using off-line training of markovian predictors.," in *Proc. of the IEEE international Conference on Computer Design*, pp. 340–347, September 2002.

[47] Y. Sazeides and J. E. Smith, "Modeling program predictability," in *Proc. of the International Symposium on Computer Architecture*, pp. 73–84, 1998.

[48] J. W. C. Fu, J. H. Patel, and B. L. Janssens, "Stride directed prefetching in scalar processors," in *Proc. of the International Symposium on Microarchitecture*, pp. 102–110, 1992.

[49] S. Sair, T. Sherwood, and B. Calder, "A Decoupled Predictor-Directed Stream Prefetching Architecture," *IEEE Trans. Computers*, vol. 52, no. 3, pp. 260–276, 2003.

[50] S. Ghosh, M. Martonosi, and S. Malik, "Cache miss equations: a compiler framework for analyzing and tuning memory behavior," *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 4, pp. 703–746, 1999.

[51] T. M. Chilimbi, M. D. Hill, and J. R. Larus, "Cache-conscious structure layout.," in *Proc. of the ACM Conference on Programming Language Design and Implementation*, pp. 1–12, 1999.

[52] P. R. Panda, H. Nakamura, N. D. Dutt, and A. Nicolau, "Augmenting loop tiling with data alignment for improved cache performance.," *IEEE Trans. Computers*, vol. 48, no. 2, pp. 142–149, 1999.

[53] R. M. Rabbah and K. V. Palem, "Data remapping for design space optimization of embedded memory systems.," *ACM Trans. in Embedded Computing Systems*, vol. 2, no. 2, pp. 186–218, 2003.

[54] J. B. Carter, W. C. Hsieh, L. Stoller, M. R. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama, "Impulse: Building a smarter memory controller," in *Proc. of the International Symposium on High Performance Computer Architecture*, pp. 70–79, 1999.

[55] D. Kim, M. Chaudhuri, M. Heinrich, and E. Speight, "Architectural support for uniprocessor and multiprocessor active memory systems," *IEEE Trans. Comput.*, vol. 53, no. 3, pp. 288–307, 2004.

[56] M. Heinrich, E. Speight, and M. Chaudhuri, "Active memory clusters: Efficient multiprocessing on commodity clusters," in *Proceedings of the International Symposium on High Performance Computing*, (London, UK), pp. 78–92, Springer-Verlag, 2002.

[57] P. R. Panda, N. D. Dutt, A. Nicolau, F. Catthoor, A. Vandecappelle, E. Brockmeyer, C. Kulkarni, and E. D. Greef, "Data memory organization and optimizations in application-specific systems," *IEEE Design and Test of Computers*, vol. 18, no. 3, pp. 56–68, 2001.

[58] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: design alternative for cache on-chip memory in embedded systems," in *Proc. of the International Symposium on Hardware/Software codesign*, 2002.

[59] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel, "Assigning Program and Data Objects to Scratchpad for Energy Reduction," in *Proc. of Design Automation and Test Europe*, 2002.

[60] J. E. Miller and A. Agarwal, "Software-based instruction caching for embedded processors," in *Proc. of the International conference on Architectural support for programming languages and operating systems*.

[61] S. Udayakumaran, A. Dominguez, and R. Barua, "Dynamic allocation for scratch-pad memory using compile-time decisions," *ACM Trans. on Embedded Computing Sys.*, vol. 5, no. 2, 2006.

[62] P. R. Panda, N. D. Dutt, and A. Nicolau, "Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications," in *Proc. of the European conference on Design and Test*, (Washington, DC, USA), p. 7, IEEE Computer Society, 1997.

[63] D. Chiou, P. Jain, L. Rudolph, and S. Devadas, "Application-specific memory management for embedded systems using software-controlled caches," in *Proc. of the Design Automation Conference*, pp. 416–419, 2000.

[64] D. H. Albonesi, "Selective cache ways: On-demand cache resource allocation.," in *Proc. of the International Symposium on Microarchitecture*, 1999.

[65] J. Abella, A. Gonzalez, X. Vera, and M. F. P. O'Boyle, "IATAC: a smart predictor to turn-off l2 cache lines," *ACM Trans. Archit. Code Optim.*, vol. 2, no. 1, pp. 55–77, 2005.

[66] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache decay: exploiting generational behavior to reduce cache leakage power," in *Proc. of the International Symposium on Computer Architecture*, pp. 240–251, 2001.

[67] M. Powell, S.-H. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar, "Gated Vdd: A circuit technique to reduce leakage in deep-submicron cache memories," in *Proc. of the International Symposium on Low power Electronics and Design*, pp. 90–95, 2000.

[68] C. Zhang, F. Vahid, and W. Najjar, "A highly configurable cache architecture for embedded systems," in *Proc. of the International Symposium on Computer Architecture*, pp. 136–146, 2003.

[69] H. Zhou, M. C. Toburen, E. Rotenberg, and T. M. Conte, "Adaptive mode control: A static-power-efficient cache design," *ACM ACM Trans. on Embedded Computing Sys.*, vol. 2, no. 3, pp. 347–372, 2003.

[70] K. Flautner, N. Kim, S. Martin, D. Blaauw, and T. Mudge, "Drowsy caches: Simple techniques for reducing leakage power," in *Proc. of the International Symposium on Computer Architecture*, 2002.

[71] W. Zhang, J. S. Hu, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, "Compiler-directed instruction cache leakage optimization," in *Proc. of the International Symposium on Microarchitecture*, pp. 208–218, 2002.

[72] W. Zhang, M. Kandemir, M. Karakoy, and G. Chen, "Reducing data cache leakage energy using a compiler-based approach," *Trans. on Embedded Computing Sys.*, vol. 4, no. 3, pp. 652–678, 2005.

[73] M. J. Geiger, S. A. McKee, and G. S. Tyson, "Drowsy region-based caches: minimizing both dynamic and static power dissipation," in *Proc. of the conference on Computing frontiers*, (New York, NY, USA), pp. 378–384, ACM Press, 2005.

[74] J. Kin, M. Gupta, and W. H. Mangione-Smith, "The filter cache: an energy efficient memory structure," in *Proc. of the International Symposium on Microarchitecture*, (Washington, DC, USA), pp. 184–193, IEEE Computer Society, 1997.

[75] G. Memik, G. Reinman, and W. H. Mangione-Smith, "Just say no: Benefits of early cache miss determination," in *Proc. of the International Symposium on High Performance Computer Architecture*, (Washington, DC, USA), p. 307, IEEE Computer Society, 2003.

[76] D. H. Woo, M. Ghosh, E. Ozer, S. Biles, and H.-H. S. Lee, "Reducing energy of virtual cache synonym lookup using bloom filters," in *Proc. of the international conference on Compilers, architecture and synthesis for embedded systems*, (New York, NY, USA), pp. 179–189, ACM Press, 2006.

[77] P. R. Turgeon, A. R. Steel, and M. R. Charlebois, "Two approaches to array fault tolerance in the IBM enterprise system/9000 type 9121 processor," *IBM J. Res. Dev.*, vol. 35, no. 3, pp. 382–389, 1991.

[78] M. A. Lucente, C. H. Harris, and R. M. Muir, "Memory system reliability improvement through associative cache redundancy," *IEEE Journal of Solid-State Circuits*, vol. 26, pp. 404–409, 1991.

[79] D. Nokolos, "Performance recovery in direct-mapped faulty caches via the use of a very small fully associative spare cache," in *Proc. of the International Computer Performance and Dependability Symposium*, p. 326, IEEE Computer Society, 1995.

[80] H. L. Kalter, C. H. Stapper, J. E. B. Jr., J. DiLorenzo, C. E. Drake, J. A. Fifield, G. A. K. Jr., S. C. Lewis, W. B. van der Hoeven, and J. A. Yankosky, "A 50-ns 16-mb DRAM with a 10-ns data rate and on-chip ECC," *IEEE Journal on Solid-State Circuits*, vol. 25, pp. 1118–1128, October 1990.

[81] P. P. Shirvani and E. J. McCluskey, "PADded cache: A new fault-tolerance technique for cache memories," vol. 00, p. 440, IEEE Computer Society, 1999.

[82] A. Agarwal, B. C. Paul, and K. Roy, "A novel fault tolerant cache to improve yield in nanometer technologies.," in *Proc. of the IEEE International Online Testing Symposium*, pp. 149–154, 2004.

[83] Y. Ooi, M. Kashimura, H. Takeuchi, and E. Kawamura, "Fault-tolerant architecture in a cache memory control LSI," *IEEE Journal on Solid-State Circuits*, vol. 27, pp. 507–514, April 1992.

[84] H. R. Zarandi, S. G. Miremadi, and H. Sarbazi-Azad, "Fault detection enhancement in cache memories using a high performance placement algorithm," *IOLTS*, vol. 00, p. 101, 2004.

[85] D. C. Burger, J. R. Goodman, and A. Kagi, "The Declining Effectiveness of Dynamic Caching for General-Purpose Microprocessors," Tech. Rep. UWMADISONCS CS-TR-95-1261, University of Wisconsin, Madison, January 1995.

[86] "The SPEC CPU2000 Benchmarks." http://www.spec.org/cpu/.

[87] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren, "Supporting dynamic data structures on distributed-memory machines," *ACM Trans. on Programming Languages and Systems*, vol. 17, pp. 233–263, March 1995.

[88] "DIS Stressmark Suite: Specifications for the Stressmarks of the DIS Benchmark Project v 1.0.," tech. rep., 2000.

[89] "The Simplescalar Simulator." Available at http://www.simplescalar.com/.

[90] N. P. J. David Tarjan, Shyamkumar Thoziyoor, "CACTI 4.0," tech. rep., HP Laboratories.

[91] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *ISCA*, 1995.

[92] Z. Hu, M. Martonosi, and S. Kaxiras, "Improving cache power efficiency with an asymmetric set-associative cache," in *Workshop on Memory Performance Issues*, 2001.

[93] T. Sherwood, G. Varghese, and B. Calder, "A pipelined memory architecture for high throughput network processors.," in *Proc. of the International Symposium on Computer Architecture*, pp. 288–299, June 2003.

[94] Intel Corporation, *Intel IXP2800 Network Processor Hardware Reference Manual*, November 2002.

[95] P. Gupta and N. McKeown, "Packet classification on multiple fields.," in *Proc. of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pp. 147–160, 1999.

[96] V. Srinivasan, S. Suri, and G. Varghese, "Packet classification using tuple space search.," in *Proc. of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pp. 135–146, 1999.

[97] P. C., "Locality and route caches," in *Proc. of the NSF Workshop on Internet Statistics Measurement and Analysis*, February 1996.

[98] T. cker Chiueh and P. Pradhan, "High performance routing table lookup using CPU caching," in *Proc. of the IEEE Conference on Computer Communications*, vol. 3, pp. 1421–1428, March 1999.

[99] K. Gopalan and T. cker Chiueh, "Improving route lookup performance using network processor cache.," in *Proceedings of the ACM/IEEE conference on Supercomputing*, pp. 1–10, November 2002.

[100] J.-L. Baer, D. Low, P. Crowley, and N. Sidhwaney, "Memory hierarchy design for a multiprocessor look-up engine.," in *Proc. of the International Conference on Parallel architectures and compilation techniques*.

[101] T. Wolf and M. Franklin, "Commbench — a telecommunications benchmark for network processors.," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 154–162, April 2000.

[102] "Valgrind tool suite version - 2.1.2." Available at http://www.valgrind.org.

[103] "Dinero IV Trace-Driven Uniprocessor Cache Simulator."

[104] P. Shivakumar and N. P. Jouppi, "CACTI 3.0: An integrated cache timing, power and area model," tech. rep., HP Laboratories.

[105] I. Stoica, "Stateless core: A scalable approach for quality of service in the internet," 2001. Doctoral Dissertation.

[106] M. Guthaus, J. Ringenberg, D. Ernst, T. M. T. Austin, , and R. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proc. of the IEEE International Workshop on Workload Characteristics*, pp. 3–14, 2001.

[107] P. Petrov and A. Orailoglu, "Towards effective embedded processors in codesigns: customizable partitioned caches.," in *Proc. of the International Symposium on Hardware/Software codesign*, pp. 79–84, 2001.

[108] K. Hazelwood, M. C. Toburen, and T. M. Conte, "A case for exploiting memory-access persistence," in *Workshop on Memory Performance Issues*, June 2001.

[109] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kauffman, 3 ed., 2003.

[110] D. H. Lawrie and C. R. Vora, "The prime memory system for array access.," *IEEE Trans. Computers*, vol. 31, no. 5, pp. 435–442, 1982.

[111] R. Raghavan and J. P. Hayes, "Reducing Interference Among Vector Accesses in Interleaved Memories," *IEEE Trans. Computers*, vol. 42, no. 4, pp. 471–483, 1993.

[112] G. S. Sohi, "High-bandwidth interleaved memories for vector processors - a simulation study," *IEEE Trans. Comput.*, vol. 42, no. 1, pp. 34–44, 1993.

[113] B. R. Rau, "Program behavior and the performance of interleaved memories.," *IEEE Trans. Computers*, vol. 28, no. 3, pp. 191–199, 1979.

[114] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. Stan, "Hotleakage: A temperature-aware model of subthreshold and gate leakage for architects," Tech. Rep. TR-CS-2003-05, Univ. of Virginia, Dept. of Computer Science, March 2003.

[115] M. Mamidipaka and N. Dutt, "ecacti: An enhanced power estimation model for on-chip caches," Tech. Rep. 04-28, CECS, University of California, Irvine, September 2004.

[116] H.-H. S. Lee, G. S. Tyson, and M. K. Farrens, "Eager writeback – a technique for improving bandwidth utilization," in *International Symposium on Microarchitecture*, pp. 11–21, 2000.

[117] S. Thoziyoor, N. Muralimanohar, and N. P. Jouppi, "Cacti 5.0," tech. rep.

[118] S. Mukhopadhyay, H. Mahmoodi-Meimand, and K. Roy, "Modeling of failure probability and statistical design of SRAM array for yield enhancement in nanoscaled CMOS.," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 24, no. 12, pp. 1859–1880, 2005.

[119] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De, "Parameter variations and impact on circuits and microarchitecture.," in *Proc. of the Design Automation Conference*, pp. 338–342, 2003.

[120] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge, "Razor: A low-power pipeline based on circuit-level timing speculation," in *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, (Washington, DC, USA), p. 7, IEEE Computer Society, 2003.