IMPROVING PERFORMANCE AND ENERGY CONSUMPTION IN REGION-BASED CACHING ARCHITECTURES

by

Michael J. Geiger

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2006

ACKNOWLEDGEMENTS

I wish I could say that this dissertation was a labor of love, but on many days, it was just labor. Thankfully, I had the guidance and support of many people to help me through the last several years. Some of them helped guide my research in the right direction; others helped me deal with the times when I had no direction. Thanks are in order for every single one of them, whether I actually remember to thank them in this space or not.

First of all, I want to thank my advisors, Gary Tyson and Trevor Mudge. Gary's ideas were the driving force behind all of this research, but he allowed me the freedom to solve each problem in my own way. Trevor's questions helped me to consider all angles of a particular topic, leading me to be as thorough as possible in my work. I've appreciated their insights, their patience, and their support throughout this entire process. I'd also like to thank the other members of my dissertation committee—Marios Papaefthymiou, Steve Reinhardt, and Dennis Sylvester—for their time and their input.

A special thank you goes to Sally McKee, a wonderful friend and mentor. Sally helped me extensively with this research—she edited ridiculously wordy drafts of papers she, Gary, and I wrote together, served as a sounding board for ideas, and called me frequently to motivate me to keep progressing (which may have been more difficult than actually doing the research). More than anyone else, she believed in the quality of this work, and that belief provided a spark that helped me enjoy the research a little more.

If I had written every bit of code I needed from scratch, this work would have taken twice as long as it did. I'd like to thank Hsien-Hsin Lee and Chinnakrishnan Ballapuram at Georgia Tech for providing me with their semantic-aware memory simulator and for answering a lot of questions at the very beginning. Thanks also to Major Bhadauria, Karan Singh, and Chris Dolen at Cornell University, who used the code I wrote in their own research and, in doing so, uncovered bugs I might not have found otherwise.

I was blessed with a wonderful group of friends to lean on throughout this process. Whether they realize it or not, they helped me fight off complete insanity for much longer than I believed possible. The list begins with Sean Holleran, Tami Ursem, Kat Penzkover, Dmitri Dolgov, Anya Osepayshvili, Brooke Haueisen, Ben Gould, Chris Brinkerhoff, Dylan Heldsinger, Bill Rand, Colleen Van Lent, and Allen Cheng, most of whom suffered through—I mean, thoroughly enjoyed—their own doctoral studies at the University of Michigan. I'd also like to thank my friends from my undergraduate days at Cornell University who stayed in touch with me despite my repeated failures to promptly return their phone calls: Jeremy Smith, Julia Rotman-Smith, Matt Greene, John Houghton, and Aaron Allen.

My family supported and encouraged me in many ways throughout my graduate career. My sister, Stacey, and I rarely discussed my work over these last few years, and I didn't mind that at all. Talking to her always took my mind away from the stress of research and provided me with plenty of much-needed laughter. My father frequently reminded me to think positively and to make whatever I'm doing as fun as possible. I don't know how well I followed that advice, but I appreciated his good thoughts and his constant faith in my ability to get past any obstacle. And my mother offered the most frequent and most fervent encouragement, because I spoke with her almost every week. She has always urged me to work my hardest and do my best, and her confidence in me has helped me get to where I am today. My family's pride in what I can accomplish has always been a driving factor in my life, and I know that the completion of this dissertation will make each of them proud.

Last—but certainly not least—I want to thank my wife, Brenda. Anything she could do to help me over the past six years, she did. She listened to me ramble about my research and somehow managed to look interested and ask useful questions, despite having almost no idea what I was talking about. She helped me keep my head on the many occasions where I thought the stress might make it explode. She baked countless batches of her awesome chocolate chip cookies, because every researcher working late nights needs comfort food to keep going, and I am no exception. She even helped me with my data analysis, entering results and making most of the graphs in this dissertation! (I think she deserves a co-author credit on my next paper, but she says it would screw up her CV.) In short, she is the rock I lean on and my best friend in the whole world, and without her love and support, I never could have gotten this far.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Embedded systems must simultaneously deliver high performance and low energy consumption. Meeting these goals requires customized designs that fit the requirements of the targeted applications. This philosophy of tailoring the implementation to the domain applies to all subsystems in the embedded architecture. For the memory system, which is a key performance bottleneck and a significant source of energy consumption, generic caching strategies are insufficient. The system requires specialized cache structures that match the manner in which programmers use data. Since different data subsets exhibit varying degrees of locality, a partitioned cache offers the best opportunity to optimize performance and energy consumption for all memory accesses.

In this dissertation, I explore several different methods that utilize partitioning for an energy-efficient, high performance data cache. Region-based caching, which replaces a unified data cache with multiple caches optimized for stack, global, and heap references, serves as the starting point for this research.

I begin by addressing energy consumption in the level one data cache. Drowsy region-based caches combine static and dynamic energy reduction techniques to simultaneously lower both sources of energy consumption. This combination performs better than the sum of its parts because the separate region caches allow us to use different degrees of drowsy caching. I then show how additional cache partitioning can further reduce energy consumption, presenting a scheme to identify highly local data in the heap region and route their accesses to a smaller cache.

I also study methods for improving memory system performance. The effectiveness of data prefetching can be increased by partitioning the cache in a manner that isolates data that prefetch well. Finally, I discuss how to reallocate data within region-based caches to eliminate unnecessary cache misses.

CHAPTER 1

INTRODUCTION

Embedded designers face the challenge of delivering high performance within a restrictive energy budget. These constraints force designers to reconsider their approach to system design. General purpose systems optimize common cases to provide acceptable performance and energy consumption for all applications—in essence, these systems treat all applications as equal. Because embedded systems typically execute programs from a single domain, a better embedded design approach is to optimize the system to the behavior of its target applications. This methodology sacrifices adaptability for significant improvements in performance and energy consumption.

Architects focus a significant amount of effort on memory system design. As on-chip caches occupy increasingly greater die area, power consumption within the memory hierarchy grows in importance—caches may consume over 40% of a chip's overall power [77]. The memory system is also a significant performance bottleneck due to the growing gap between processor and DRAM speeds [76][108]. Caches provide one solution, offering fast, low power accesses for data with high locality. However, a general cache architecture, which works well for a variety of data, is not the best approach for all accesses. A unified cache implicitly treats all data the same, but programmers use different types of variables in different ways, leading to distinct locality characteristics for each data subset. However, within each class of data, the usage typically remains consistent across applications.

We can therefore apply the embedded design concept of an application-specific system to the design of the memory subsystem. Tailoring an entire system to a particular application domain sacrifices programmability because other applications will suffer. However, the consistency of memory behavior across applications allows us to optimize the memory hierarchy to the behavior of given subsets. Current systems use very general caches, splitting memory references only according to instructions versus data [41]. As an alternative, further specialization of memory structures to better match usage characteristics of the data they hold can both improve performance and significantly reduce total energy expended.

Partitioning the cache according to reference behavior can have another benefit: we can selectively apply optimizations to those subsets on which they work well. Hardware optimizations typically involve tradeoffs between energy and performance, with most techniques attempting to improve one area while minimizing the impact on the other. In caches, these tradeoffs are often due to a data subset on which a technique is not as effective; the penalty for those data is severe enough to significantly impact the overall performance or energy consumption of an application. In some cases, all data benefit from an optimization, but the improvements are greater for some data. For example, **drowsy caching** [31][57][58], a technique for reducing static energy consumption, works best on low locality data that remains idle for long periods of time. In other cases, the optimization is completely useless for a data subset and should only be applied where effective. **Prefetching** techniques [21][26][45][47][54][55][81][85][94][96][97][113], which improve performance by anticipating and eliminating cache misses, are effective for data that exhibit predictable access patterns but not for random accesses. In both cases, partitioning the cache so that differing degrees of optimization can be applied to the appropriate data sets can improve the effectiveness of these techniques.

One form of heterogeneous memory, **region-based caching** [32][33][67][70], replaces a single unified data cache with multiple caches optimized for global, stack, and heap references. This approach works well precisely because these types of references exhibit different locality characteristics. Furthermore, many applications are dominated by data from a particular region, and thus greater specialization of region structures should allow both quantitative (in terms of performance and energy) and qualitative (in terms of security and robustness) improvements in system operation. This approach slightly increases required chip area, but using multiple, smaller, specialized caches that together constitute a given “level” of a traditional cache hierarchy and only routing data to a cache that matches those data’s usage characteristics provides many potential benefits: faster access times, lower energy consumption per access, and the ability to turn off structures that are not required for (parts of) a given application.

Our work focuses on the promise of this general approach to data cache design. We perform detailed analysis of memory reference behavior to identify data subsets for which given techniques work well. In some cases, we find that a region-based partition allows for substantial benefit. We also find other instances in which stretching or redefining the boundaries of our partitions achieves even greater benefits.

The remainder of this dissertation is organized as follows. Chapters 2 and 3 provide additional introductory material relevant to this research. In Chapter 2, we discuss related work in the areas of energy and performance improvement for cache designs. We focus primarily on techniques affecting the level one data cache but reference other relevant methods as well. Chapter 3 outlines the experimental framework used throughout this work. We provide an overview of the simulator infrastructure used to model our proposed improvements as well as a discussion of the benchmarks we used to evaluate those techniques.

Chapters 4 and 5 present methods for reducing energy consumption in cache partitioning. In Chapter 4, we explore the application of **drowsy caching** [31][57][58] to region-based caching. We demonstrate that the combination of the two can be more effective than either alone, as partitioning the cache allows us to apply different degrees of drowsy caching to different regions of data [32][33]. Chapter 5 addresses the caching of the heap region, the most difficult region of data to manage in a cache structure. This chapter shows that the heap region often possesses greater locality than previously thought. We propose a method for tailoring the heap cache to each application, allowing us to accommodate cases in which the entire heap caches well as well as instances in which only a fraction of the heap possesses good locality [33][34][35].

Chapter 6 shifts the focus of the dissertation from energy to performance, showing that partitioning the cache can also improve methods for reducing application run time. We demonstrate that **prefetching** [21][26][45][47][54][55][81][85][94][96][97][113] is most effective when applied only to a subset of the data. We analyze data reference patterns to determine which data prefetch well and show how region-based caching affects the impact of different hardware prefetch algorithms.

Region-based caching implicitly assumes that the data is placed in the appropriate region at compile time. Chapter 7 challenges that assumption, examining data for which the reference characteristics do not match the expectations for its region. We discuss the relocation process for incorrectly placed data, examining the issues involved in identifying and remapping these structures in the compiler.

Chapter 8 concludes the dissertation. We summarize the contributions of this work and offer some potential directions for future research.

CHAPTER 2

RELATED WORK

Caches provide fast memory accesses by temporarily storing data believed to currently be in use [96]. These memories operate on two principles: spatial locality, which states that data near the currently accessed address are most likely to be accessed in the near future, and temporal locality, which states that recently accessed addresses are likely to be accessed again in the near future. In this chapter, we focus on techniques for improving the energy and performance of the data cache, typically the level one (L1) data cache. In several cases, similar techniques have been proposed for the instruction cache; we only briefly mention those methods where applicable. Approaches that address lower levels of the memory hierarchy receive similar treatment. This chapter highlights the research most relevant to our work. For an in-depth discussion of several techniques not covered here, see Brehob's dissertation [13].

In Section 2.1, we explore methods for reducing cache energy consumption, addressing both dynamic and static energy. We emphasize previous papers on cache partitioning for dynamic energy savings and state-preserving techniques for static energy savings. Section 2.2 discusses techniques for improving cache performance, focusing on three main areas: splitting the cache to reduce access latency, improving locality through data placement, and removing misses through data prefetching.

2.1 Reducing Cache Energy Consumption

2.1.1 Dynamic Energy Consumption

Techniques for reducing dynamic energy consumption rely on the fact that the energy dissipated per access is proportional to the cache size. Most of these approaches therefore partition the cache to allow memory references to access smaller structures. Sahuquillo and Pont discussed several of these techniques in their survey paper [92].

Partitioning techniques fit in one of two categories: vertical or horizontal. Vertical partitioning adds a level between the L1 and the processor; these structures provide low-power accesses for data with temporal locality, but typically incur many misses, increasing average observed L1 latency. Su and Despain first proposed line buffers, output latches that store the most recently accessed cache line or lines [102]. On a memory reference, the buffer is checked first and the cache is accessed only on a buffer miss. Ghose and Kamble modify this approach by adding multiple buffers to capture more accesses and accessing the buffers in parallel with the L1 cache [36]. Filter caches [60] perform the same operation on a slightly larger scale, using small direct-mapped caches to capture recent accesses and reduce cache activity. Chang et al. developed a two-level filtering scheme that both exploits temporal locality and uses a partial tag check to determine which ways of a set-associative cache will not hit [20]. One example of a vertical partitioning technique in which the additional level resides between the first and second level caches is the victim cache, which attempts to capture the locality in recently evicted lines [55]. This technique aims primarily to improve performance, not energy consumption.

Horizontal partitioning divides entities at a given level in the memory hierarchy into smaller segments. For example, cache sub-banking [36][102] divides cache lines into smaller segments. Memory references are routed to the proper segment, which is the only one that draws power. Abella and Gonzalez combine sub-banking with data placement techniques to exploit locality and improve energy efficiency [1]. They also vary supply and threshold voltages across the banks to reduce static energy consumption as well. Hu et al. take a slightly different approach with an asymmetric set-associative cache in which each way of the set is a different size [44]. A hit in one of the smaller, faster ways can immediately terminate accesses to the larger ways, thus saving energy per access.

Horizontal partitioning techniques that consider the access behavior of applications can further reduce dynamic energy consumption. Huang et al. route stack accesses to a dedicated structure that is tailored to their reference characteristics and optimized for energy efficiency [46]. Region-based caching [67][69] replaces a unified data cache with heterogeneous caches optimized for global, stack, and heap references. On a memory reference, only the appropriate region cache is activated. Relatively small working sets for stack and global regions allow those caches to be small, dissipating less power per access. The region-based paradigm can be extended to other areas of the memory system; for example, Lee and Ballapuram propose partitioning the data TLB by semantic region [70]. We cover region-based caching in much greater detail in Chapter 4.

We choose to partition the cache horizontally to avoid the performance penalties common in vertical partitioning techniques. Splitting a cache level—in our case, the first level data cache—also allows us the freedom to exploit behavior within a data subset. Most work in this area either fails to consider access behavior or concentrates on stack data because it caches well. We use horizontal partitioning to improve the caching behavior of the heap, the region that typically exhibits the worst locality.

Reconfigurable caches can create virtual partitions to reduce dynamic energy consumption. Ranganathan et al. present a method for dynamically partitioning set-associative caches—for example, a 4-way 1 MB cache can be reconfigured into four direct-mapped partitions of 256 KB, two 2-way partitions of 512 KB each, or two partitions in which one of the partitions is a 3-way, 768 KB cache [89]. They also propose an organization in which the tags are extended to the maximum possible size to accommodate multiple partitions; this organization does not require a set-associative cache. The authors note that one of the partitions could be allocated to a specific task such as prefetching, which essentially foreshadows the work we discuss in Chapter 6. Balasubramonian et al. describe a reconfigurable memory hierarchy that detects phase changes and reconfigures the cache appropriately [7]. The physical cache has only one level but acts as a virtual two-level, non-inclusive cache hierarchy. The size, latency, and associativity of each level are all programmable. This technique improves performance in a typical two-level cache hierarchy; when extended to three levels to address future cache designs, it offers lower energy consumption.

Other reconfigurable caches can dynamically resize the available resources. Albonesi disables inactive cache ways to reduce switching activity and lower dynamic energy dissipation [4]. His approach allows application-specific as well as finer-grained control over cache resources. Motorola’s M-CORE architecture also features a programmable cache that can be dynamically resized to reduce energy consumption [74]. Other customizable options include the write policy and data streaming extensions. Yang et al.

explore both static and dynamic resizing of cache resources to reduce energy consumption [110][111]. They offer the ability to disable either unused sets or unused ways to save energy.

Our work uses caches that are reconfigurable at a higher level than the work discussed above. When partitioning the heap, as discussed in Chapter 5, we maintain two structures—a small cache for high locality data and a larger cache for low locality data. If an application does not require both caches, we can disable the large cache to save energy. This cache uses logic similar to other reconfigurable structures.

Other partitioning methods employ more novel approaches to the reduction of dynamic energy consumption. Some techniques seek to disable the costly tag checks that consume significant amounts of energy in data caches. Petrov and Orailoglu use a reconfigurable cache that allocates a single partition for tagless accesses to predictable blocks [86]. Their work more generally partitions the cache based on the reuse characteristics of data references, concentrating on instructions in nested loops. They use static compiler analysis to identify instructions that exhibit data reuse, either individually (i.e., a single instruction accesses the same address or cache line in multiple loop iterations) or across instructions (i.e., multiple instructions access the same cache line). Ashok et al. employ compiler-managed address speculation to enable tagless cache accesses for power reduction [6]. Their caches use static compile-time information about memory access times and patterns to reduce redundancy in memory accesses.

Hezavei et al. present circuit-level techniques for reducing dynamic energy consumption in SRAM designs [42]. Divided bit lines use shorter, segmented bit lines to decrease latency and bit line capacitance and allow for a split row decoder to further reduce energy. Pulsed word lines minimize the time that the word lines—and therefore the SRAM cells—are active by deactivating the word lines before they make a full voltage swing, once again increasing speed and lowering energy consumption. Isolated bit lines isolate the sense amplifiers attached to the bit lines as soon as they detect a sufficient voltage difference, thus preventing a full voltage swing.

2.1.2 Static Energy Consumption

As transistor feature sizes have decreased in new circuit technologies, managing static energy consumption has become an increasingly important issue. Techniques that reduce static energy in caches must account for the fact that stored data must either remain persistent or be easily recovered. Solutions to the static energy problem encompass both circuit and architectural approaches; our work uses a combination of both.

The simplest circuit technique for static energy reduction is gated-Vdd [88], which turns off unused cache lines to eliminate leakage current. Gated-Vdd effectively reduces static energy at the cost of lower application performance; each time a cache line is turned off, the data it holds is destroyed and must be fetched from the next level of the memory hierarchy. Methods that preserve memory state place inactive lines into a low-power mode; these approaches save less energy than gated-Vdd, but perform significantly better. Nii et al. propose a technique for lowering static energy by dynamically varying the source voltage to the transistor body [79]. This technique does maintain memory state, but the leakage savings are somewhat offset by the increased supply voltage, and the transition time between active and low-power modes may be prohibitively high. Other techniques employ self reverse biasing [10] or voltage scaling [109] to reduce static energy without loss of data.

Architectural management can improve the efficiency of these circuit techniques. Cache decay [56] tracks cache line usage to determine when a line is dead and then gates the supply voltage to turn the line off. The decay interval is approximated by associating a small local counter with each line and incrementing it after a given number of cycles. The line is turned off when the counter saturates; an access to the line resets the counter. Yang et al. present a method for recognizing and disabling unused sets in set-associative instruction caches [110]. Zhou et al. only disable the cache data array, not the tag array [114]. Keeping the tag array active allows them to rapidly determine the performance effect of keeping more lines active and tailor their line-disabling policy accordingly.

Drowsy caching [31][57][58] applies the interval-based technique used in cache decay to state-preserving leakage reduction. Rather than turn inactive lines off after a certain interval, drowsy caches scale the supply voltage to place lines in a low-power state. We discuss this technique in greater detail in Chapter 4, using cache partitioning to allow more precise management of the drowsy policy for lower energy consumption.

Li et al. exploit data reuse in the cache hierarchy to reduce leakage energy [73]. They apply both state-preserving and state-destroying leakage reduction methods to L2 sub-blocks when the data also exists in the L1. They also vary the point at which blocks are turned off and reactivated. The authors evaluate five different methods, varying the level of aggressiveness for leakage reduction as well as the type of technique used.

2.2 Improving Cache Performance

2.2.1 Cache Partitioning

In Section 2.1.1, we discussed research on partitioning the cache to achieve lower energy consumption; a different body of work uses partitioned caches to reduce cache latency. Several papers focus on exploiting the reference characteristics of stack data, which typically has a small working set and displays good locality. Machines like the HP3000 Series II [11], the CRISP processor [9][29], and the Hobbit processor [5] contain caches specifically for stack references; in all cases, the stack cache is the only data cache in the processor. In later work, stack data are routed to cache-like structures customized to exploit their reference characteristics. Cho et al. use a decoupled stack cache that forwards $sp-relative accesses in which the stack pointer does not change and features wider ports to take advantage of the contiguous accesses that are common in that region [25]. The stack value file [68] is a register file treated as a circular buffer that adds extensions to improve performance in the stack region. The architecture recognizes $sp-relative accesses early in the pipeline and morphs these accesses into register moves. The stack value file also contains additional status bits to avoid unnecessary reads and writes when the stack size changes. Although we employ a typical cache for stack data, these techniques are orthogonal to our work and may further improve application performance.

Techniques specifically addressing non-stack data are less common. In their analysis of the CRISP stack cache [29], Ditzel and McLellan note that global variables are also well suited to caching. The pointer cache [26] stores mappings between heap pointers and targets, but the structure is intended as a prefetching aid, not a data cache. Intel's StrongARM SA-1110 processor [48] uses a mini-cache in addition to the main data cache for storing streaming data with little or no temporal locality. Note that region-based caching [67][69] also improves performance slightly; the partitioning scheme allows for reduced associativity and therefore lower access times in each of the regions. Further cache partitioning, as discussed throughout this dissertation, allows us to use smaller caches with lower latency.

Other partitioning techniques aim to improve or ensure good locality in cache accesses. The annex cache [51] is a vertical partitioning scheme similar to the victim cache that filters memory references to reduce conflict misses. All data must essentially prove in the annex cache that they possess sufficient locality to reside in the main cache. Other approaches use dynamic partitions to eliminate conflicts. Dahlgren and Stenstrom discuss a reconfigurable software-controlled cache that can be remapped into virtual areas that cannot conflict [27]. Suh et al. perform a similar task by using analytical models that accurately predict cache miss rates to partition the cache into dedicated per-process regions [104].

The idea of partitioning the cache among multiple processes has become more prevalent with the advent of simultaneous multi-threading (SMT) and chip multiprocessor (CMP) systems. Iyer discusses a method for assigning and enforcing priorities for different data streams in a CMP platform [50]. One priority enforcement method is cache partitioning using both dynamic and static techniques. Kim et al. use partitioning to implement fairness in a CMP between threads sharing the L2 cache [59]. They propose several fairness metrics, correlate them with execution time, and develop an algorithm to optimize the cache behavior. Hsu et al. evaluate multiple performance targets for partitioning shared caches in CMPs [43]. They show that the optimal partitioning varies significantly depending on the performance target and that thread-aware allocation of cache resources is necessary to approximate that optimal partition.

Region-based caching may not work well on these systems, as the presence of multiple applications accessing each region will likely increase conflict in each region cache.

Some studies use separate caches to capture both spatial and temporal locality. Gonzalez et al. present a dual data cache for use with vector data [39]. They use a locality prediction table to steer references to one of the two caches. Lee et al. use a similar split cache with variable fetch and eviction policies [72]. They use a larger fetch size for the spatial cache, a fully associative structure with large cache blocks. In the smaller, direct-mapped temporal cache, they selectively retain lines that demonstrate frequent temporal reuse during program execution.

We address an area that has not been extensively researched in Chapter 6: partitioning to improve a technique that, by itself, improves cache performance. Ranganathan et al. note that a partition can be set aside for prefetched data in a set-associative cache; however, they employ virtual partitions in a unified cache [89]. We treat the new partition as a separate region.

2.2.2 Data Placement

A significant amount of work focuses on remapping data without partitioning the cache. These techniques typically aim to either reduce cache conflicts or improve spatial locality. One method we see for conflict prevention is a form of page coloring, an operating system technique that assigns the same color to two pages if they map to the same location in a physically-indexed cache. Bugnion et al. use compile-time information to allow applications to request a particular coloring policy in the operating system [15]. Sherwood et al. use both compiler and hardware methods to modify page colors for improved locality [93]. Rivera and Tseng take a different approach to reducing conflict misses, modifying the base addresses or dimensions of array structures accessed in loops at compile time [90].

A large number of techniques focus on the placement of data in the cache to improve locality and reduce misses. Many of them perform this task at compile-time, looking at the overall reference behavior of the program to determine which objects fit best together and which should not be mapped to the same sets. Chilimbi et al. reorganize the members of data structures to increase locality [23]. Small structures are split to allow hot fields to fit in the same cache block, while larger structures are rearranged to place fields that share temporal locality close together. Palem et al. also seek to place data with temporal locality close together; they remap the entire data layout of an application at compile time to achieve this goal [83]. Ailamaki et al. focus on database systems, grouping all values of a particular attribute within the same page [3]. We can apply methods similar to these in region-based caches, as shown in Chapter 7. We identify objects for which the locality differs from typical data in that region, and use compiler feedback to direct the data allocation to the correct region.

Cache-conscious data placement [18] works similarly to region-based caching in that it breaks data into stack, global, heap, and constant regions. The authors use profiling to identify temporal relationships between objects and place them appropriately to reduce conflict. They do not rearrange stack and constant data, but the placement of those regions guides the placement of other variables. They also partition objects into popular and unpopular sets, using unpopular data to fill spaces between popular objects.

There are also a significant number of runtime techniques for improving locality. Kistler and Franz focus on reorganizing pointer-based data structures, clustering members that are accessed near the same time and reordering data within cache lines to reduce load latency on a miss [61]. In a later study, they explore dynamic code optimization in concert with their data layout techniques [62]. Johnson and Hwu split the cache into macroblocks—regions of data larger than a single line in which the access behavior is statistically uniform—and use the access patterns of those macroblocks to determine the movement of data within the cache [52]. Their scheme bypasses data that is expected to have little reuse in the cache. They later added a structure to detect spatial locality by tracking accesses to adjacent blocks, fetching multiple blocks when locality is high [53]. Chilimbi explores two different methods for improving data locality. In one study, he identifies hot data streams—reference subsequences exhibiting a high degree of regularity—and uses these streams to cluster data [24]. In a different paper, he and Larus employ a garbage collection utility to gather run-time information about reference patterns [22]. Based on the garbage collection results, they can place data with high temporal affinity together via copying.

Ding and Kennedy use both compile time and runtime optimizations to reorganize data and code for improved locality [28]. Once the structure of the data is known, they insert code that reorganizes that data. In some cases, data accesses are reordered to promote better temporal reuse; in others, the data itself is moved to improve spatial reuse.

Srinivasan et al. explore criticality as an alternate metric for determining data placement. Previous work has shown that some loads are more critical to the performance of programs running on dynamically scheduled processors [98]. The authors later showed that criticality, although a significant program property, is not strong enough to override locality in the organization of a traditional cache hierarchy [99].

Tyson et al. show that selectively allocating cache lines can improve application performance [105]. They show that a small percentage of loads account for the majority of cache misses and mark those references as cacheable/non-allocatable (C/NA), prohibiting them from invoking the cache allocation policy. C/NA loads can be identified statically or dynamically; the dynamic scheme uses a two-bit predictor.

2.2.3 Prefetching

Prefetch techniques attempt to hide the latency of cache misses by anticipating the misses and fetching the data prior to the actual memory reference. Prefetch methods vary widely, and no single technique has been shown to optimize performance in all cases. Vanderwiel and Lilja summarize several prefetch algorithms in their survey paper [106].

Hardware prefetching techniques observe the dynamic behavior of a program and predict prefetch addresses accordingly. The simplest hardware prefetchers are sequential methods that exploit basic spatial locality. Next sequential prefetching (NSP) [96] is the most straightforward prefetch algorithm—on an access to block **n**, prefetch blocks **n+1** through **n+p**, where **p** is the prefetch degree (i.e., number of blocks to prefetch). Smith discusses several variations on this policy, including prefetching on every access, prefetching only on misses, and using a tag bit [37] to filter prefetch accesses. Jouppi uses NSP to fetch data streams into dedicated prefetch buffers that ensure prefetched data do not cause cache conflicts [55]. The caches and buffers are referenced in parallel; on a cache miss that also misses in the buffers, the least recently used buffer is filled starting from the miss address.

More complex techniques can recognize strided access patterns in which the blocks are not necessarily contiguous. These methods must typically maintain a record of reference activity to predict program behavior. Chen and Baer use a reference prediction table (RPT) to track access patterns and generate prefetch requests [21]. Each RPT entry contains a field to indicate the confidence of the corresponding prefetch prediction. When combined with an additional program counter and branch predictor, the RPT can run ahead of the actual program and prefetch more aggressively. Palacharla and Kessler combine Jouppi’s prefetch buffers with a predictor for detecting regular strides [81]. They also add a filter to reduce the number of useless prefetches. Iacobovici et al. present a scheme for handling data streams with multiple distinct strides that follow a regular pattern [47]. They use a two-state table that can track up to four distinct strides.

To handle irregular access sequences like those seen in pointer-based data structures, more complex methods are required. Chen and Baer extend the RPT to correlate their reference predictions with the branch history [21]. Hu et al. use a similar correlating prefetcher that is indexed by cache tag rather than instruction address [45]. This resource-efficient predictor outperforms larger address-based prefetchers. Solihin et al. employ a user-level memory thread to implement their correlating prefetcher [97]. The thread runs on either the DRAM or the memory controller and prefetches data directly into the L2 cache.

Markov prefetchers [54] model the probabilities of addresses occurring consecutively in the miss stream. A full Markov model uses a weighted graph but is too inefficient to realize in hardware, so Joseph and Grunwald approximate the graph with a four-way table using LRU replacement. The MRU way of each set represents the address of highest priority. Sherwood et al. combine a Markov predictor with stream buffers to reduce conflicts [94].

Other specialized techniques focus solely on pointer-based applications. Collins et al. present an architecture that contains a dedicated cache to store pointer transitions [26]. A separate thread runs ahead of the user program and relies on the pointer cache to generate prefetches. This cache can also serve as a value predictor for difficult loads, allowing the runahead thread to proceed more efficiently.

A number of novel schemes attempt to improve the quality of prefetching by combining it with other techniques. Peir et al. use an adaptive cache that identifies unused blocks to use as sites for prefetched data [85]. This scheme allows the cache to closely approximate a global LRU policy—in other words, a fully-associative cache—rather than a set-associative one. Lai et al. use a similar technique to predict dead blocks, but they combine their dead block predictor with a correlating prefetcher that indicates which data to prefetch when another block becomes dead [65]. This method provides effective prefetching for difficult patterns such as pointer references.

Kumar and Wilkerson introduce a spatial footprint predictor that identifies spatial locality in applications that do not prefetch well using standard techniques [64]. The predictor identifies which portions of a line will be used before being evicted. This technique allows cache lines to remain small while accurately predicting which data should fill them. Zhang and Gupta reduce memory traffic by transferring prefetched values in compressed form [113]. They use frequent value compression on the most significant bits of the prefetched data, relying on data in which the prefix contains all zeroes or ones, or on pointer addresses that reference the same block.

Software prefetching techniques typically employ compile-time analysis to detect reference patterns and augment programs accordingly. Callahan et al. use special prefetch instructions to reduce cache misses [19]. They use a simple heuristic to prefetch array variables in program loops—when the array index contains the innermost loop variable, prefetch the next array location. This method was refined to remove prefetches for data likely to be present in the cache. Klaiber and Levy also use prefetch instructions in program loops, although they do not restrict their prefetching to array accesses [63]. The prefetched data are routed to a fully-associative buffer.

Pointer-based prefetching is more difficult to address in software, as prefetches cannot be scheduled before the effective address computation. Cahoon and McKinley successfully handling these accesses by using jump pointers—additional pointers that connect objects that are not directly linked but are often referenced together [17]. The additional pointers improve the quality of prefetching but add overhead that limits the overall performance impact. Roth and Sohi more selectively employ jump pointers, identifying four separate prefetch idioms to handle different types of data structures [91].

Some methods employ both hardware and software to generate effective prefetches. Ortega et al. primarily use software techniques, adding prefetch instructions and aggressively prefetching into registers to bypass load instructions [80]. A hardware assist tracks history to enable stride-based prefetching and to act as a filter that eliminates unnecessary prefetches. Wang et al. take an opposite approach, using compiler hints to regulate an aggressive hardware prefetcher [107]. The hints cover issues such as the number of lines to prefetch and indicate when a reference accesses pointer-based structures.

CHAPTER 3

EXPERIMENTAL FRAMEWORK

This chapter presents the infrastructure used for the experiments discussed in subsequent chapters. In Section 3.1, we describe our simulation environment, highlighting both the models we used and the configuration choices we made. In Section 3.2, we discuss the benchmarks we ran to evaluate our proposed optimizations.

3.1 Simulation Environment

All of our experiments use a modified version of the SimpleScalar ARM target [16]. ARM microprocessors, such as the Intel StrongARM SA-11xx [48][77] and XScale [49] series, are extremely common in embedded devices, including handhelds, cellular phones, and GPS devices. Our simulations use an in-order processor model based on the Intel StrongARM SA-110 [77]; the parameters for the execution engine are shown in Table 1.

To estimate the energy consumption of our benchmarks, we use Wattch [14] to calculate dynamic energy and HotLeakage [112] to calculate static energy. Wattch pre-computes the energy dissipation for various architectural events and then counts the event occurrences to determine the total energy. This simulation module models different degrees of clock gating to allow an assessment of the energy-saving impact of such

Parameter	Value
Issue policy	In-order
Fetch width	1
IFQ size	8
Decode width	1
Issue width	1
Commit width	1
Branch predictor	Not taken
Integer ALU/multiplier	1
FP ALU/multiplier	1
Memory port(s) available to CPU	1

Table 1: SimpleScalar simulation parameters for baseline architectural model, which is based on the Intel StrongARM SA-110 [77]

techniques. We use the `cc2` mode, which assumes aggressive, ideal clock gating—in other words, the circuit dissipates no energy when turned off—and therefore provides an excellent estimate of an application's dynamic energy consumption. The `cc3` mode also models aggressive clock gating, but it assumes that a constant fraction of the dynamic energy is dissipated when a circuit is inactive. Since some of our studies involve techniques for reducing static energy dissipation, we use the more detailed static energy model in HotLeakage. This program contains a detailed drowsy cache model used by Parikh et al to compare state-preserving and non-state-preserving techniques for leakage control [84]. HotLeakage tracks the number of lines in both active and drowsy modes and calculates leakage energy appropriately. It also models the energy of the additional hardware required for drowsy caching. HotLeakage contains device parameters and scaling factors for several different technology sizes; when this tool is integrated with SimpleScalar, Wattch and Cacti use these same values. We use 70 nm technology with an operating voltage and temperature of 0.9 V and 300 K (27° C), respectively. Rather than using the maximum possible frequency for this technology, we slow the clock to 1.7 GHz to allow reasonable cache access latencies and reduce dynamic energy consumption.

Parameter	Value
Line size (all caches)	32 bytes
Baseline L1 data cache configuration	32 KB, direct-mapped, single-ported
Baseline L1 data cache hit latency	0.914 ns (2 cycles)
Stack/global L1 data cache configuration	4 KB, direct-mapped, single-ported
Stack/global L1 data cache hit latency	0.3 ns (1 cycle)
L1 instruction cache configuration	16 KB, 32-way set associative
L1 instruction cache hit latency	2.683 ns (1 cycle, assuming pipelined instruction cache)
L2 cache configuration	512 KB, unified inst./data, 4-way set associative
L2 cache hit latency	6.99 ns (12 cycles)
Main memory latency	88 cycles (first chunk) 3 cycles (inter chunk)
Memory bus configuration	8 bytes wide, fully pipelined

Table 2: Memory system configuration for our simulations. The table contains information for a basic unified cache as well as our region-based cache configurations, in which the heap cache uses the same configuration as the baseline L1 data cache. We use Cacti 3.2 [95] to calculate the access latencies.

Table 2 shows the parameters for our memory model, giving the latencies and configurations for the caches we use throughout our experiments. We use Cacti 3.2 [95] to calculate cache access times. Note that our baseline 32 KB cache, which is the same size as the heap cache in our region-based cache configurations, requires two cycles on a hit. The level 2 cache is used as a common backup storage in all configurations to ensure a fair comparison between the various configurations, as in previous region-based caching studies [67][70].

3.2 Benchmarks

To assess the effectiveness of our proposed changes, we run applications from MiBench [40]. MiBench is a freely available embedded benchmark suite developed at the University of Michigan; this workload is intended as an alternative to the EDN Embedded Microprocessor Benchmark Consortium (EEMBC) suite [30]. Like the EEMBC benchmarks, MiBench reflects the fact that the embedded domain covers a wide

	Description	# dyn. insts	# dyn. loads/ stores	% insts accessing Memory
Automotive				
basicmath	Simple math functions	4.41E+09	1.11E+09	25.1%
bitcount	Tests bit manipulation abilities	1.14E+09	1.83E+08	16.0%
quicksort	Sorts large string array	1.08E+09	1.75E+08	16.2%
susan.corners	Image recognition algorithm; contains different phases to recognize corners and edges and to smooth image	3.15E+07	9.98E+06	31.6%
susan.edges		7.93E+07	2.30E+07	29.0%
susan.smoothing		6.06E+08	1.80E+08	29.7%
Consumer				
jpeg.encode	JPEG image compression/ decompression	1.58E+08	3.93E+07	24.9%
jpeg.decode		3.66E+07	1.12E+07	30.7%
mad	MPEG audio decoder	4.28E+08	1.11E+08	26.0%
tiff2bw	Convert TIFF image to black and white	2.14E+08	5.83E+07	27.3%
tiff2rgba	Convert TIFF image to RGB format	2.60E+08	1.02E+08	39.0%
tiffdither	Dither TIFF bitmap	1.19E+09	2.58E+08	21.7%
tiffmedian	Convert image to reduced color palette	8.32E+08	2.07E+08	24.9%
typeset	Typeset HTML document	9.15E+08	3.17E+08	34.6%
Office				
ghostscript	Postscript language interpreter	1.31E+09	3.76E+08	28.8%
ispell	Spell checker	1.60E+09	4.45E+08	27.8%
rsynth	Text to speech synthesis program	1.45E+09	4.98E+08	34.3%
stringsearch	Searches for given words	6.52E+06	1.46E+06	22.5%
Network[1]				
dijkstra	Dijkstra's shortest path algorithm	4.43E+08	1.25E+08	28.3%
patricia	Routing table application	1.10E+09	2.60E+08	23.6%
Security				
blowfish.encode	Blowfish block cipher encryption/ decryption using variable length key	1.32E+09	3.89E+08	29.5%
blowfish.decode		1.32E+09	3.89E+08	29.5%
pgp.encode	Pretty Good Privacy (PGP) public key sign/verify algorithm	5.67E+07	1.31E+07	23.1%
pgp.decode		1.22E+08	2.83E+07	23.1%
rijndael.encode	Rijndael encryption/decryption—national Advanced Encryption Standard	5.62E+08	1.84E+08	32.7%
rijndael.decode		5.29E+08	1.72E+08	32.6%
sha	Secure hash algorithm	1.88E+08	3.67E+07	19.5%
Telecomm				
adpcm.encode	Variation of Pulse Code Modulation; achieves 4:1 compression of PCM data	8.53E+08	1.00E+08	11.7%
adpcm.decode		7.07E+08	1.00E+08	14.2%
CRC32	32-bit cyclic redundancy check (CRC)	3.25E+09	9.85E+08	30.3%
FFT	Fast Fourier Transform	1.02E+09	2.46E+08	24.1%
FFT.inverse	Inverse Fast Fourier Transform	6.82E+08	1.68E+08	24.6%
gsm.encode	Standard for voice encoding/decoding used in Europe	4.55E+09	1.38E+09	30.3%
gsm.decode		1.84E+09	3.68E+08	20.0%
			AVERAGE	26.1%

Table 3: MiBench applications listed by category. We provide a brief description of each application, the total dynamic instruction count, and the total number and overall percentage of memory references. These applications have similar reference percentages to the MediaBench suite [66]. We run precompiled ARM binaries from the MiBench website and use the large input data sets in all simulations.

[1] The CRC32, sha, and blowfish applications are also relevant to the Network category, but are shown here as part of the Security and Telecommunications suites. [40]

range of applications. It contains six separate categories, each one targeting a different area of the embedded space—automotive and industrial control, consumer devices, network applications, office automation, data security, and telecommunications.

Table 3 lists the applications in the MiBench suite[2], providing a brief description of each application as well as their execution length and memory usage. We run precompiled ARM binaries available from the MiBench web page (http://www.eecs.umich.edu/mibench); all applications were compiled using GCC version 2.95.2 on a Debian Linux 2.2.18 workstation with optimizations enabled [40]. The applications exhibit a fairly wide range of behavior, with dynamic instruction counts ranging from 6.5 million for stringsearch to over 4.5 billion for gsm.encode. We use the large input data sets in all cases. These benchmarks are not memory-intensive, as the third and fourth columns show. On average, about 26% of all instructions access memory. That figure is low compared to the percentage of memory references in the SPEC CPU2000 benchmarks [101], but is comparable to the reference percentage in MediaBench [66], a multimedia and telecommunications benchmark suite. Lee and Tyson show that the SPEC and MediaBench applications average 56% and 24% memory references, respectively, using the SimpleScalar portable ISA (PISA) [67]. In later chapters, we will explore the memory usage of these applications in greater detail.

[2] MiBench includes two applications, lame and sphinx, which we were unable to simulate without errors and therefore have not included in this table or our experiments.

CHAPTER 4

DROWSY REGION-BASED CACHES

Schemes for improving energy consumption typically target either static or dynamic energy, trading a small penalty in one area for significant gains in the other. In this chapter, we present a technique for simultaneously reducing static and dynamic energy consumption in the level 1 data cache. We show that the combination of region-based caching and drowsy caching is particularly effective because each technique amplifies the effect of the other. This research draws from two of our previous publications [32][33].

In Section 4.1, we provide an in-depth look at region-based caching [67][69], the basis for this research. We then present a similar exploration of drowsy caching [31][57][58] in Section 4.2. Section 4.3 shows how the combination of these two techniques provides more benefits than either technique alone. In Section 4.4, we show how region-based caching allows us to tune the aggressiveness of our drowsy caching policy. Section 4.5 summarizes the chapter.

4.1 Region-Based Caching

Region-based caching leverages the reference characteristics of compiler-defined data regions to reduce dynamic energy consumption. Figure 1 shows the different regions common to most architectures; we provide the MIPS and ARM memory maps for

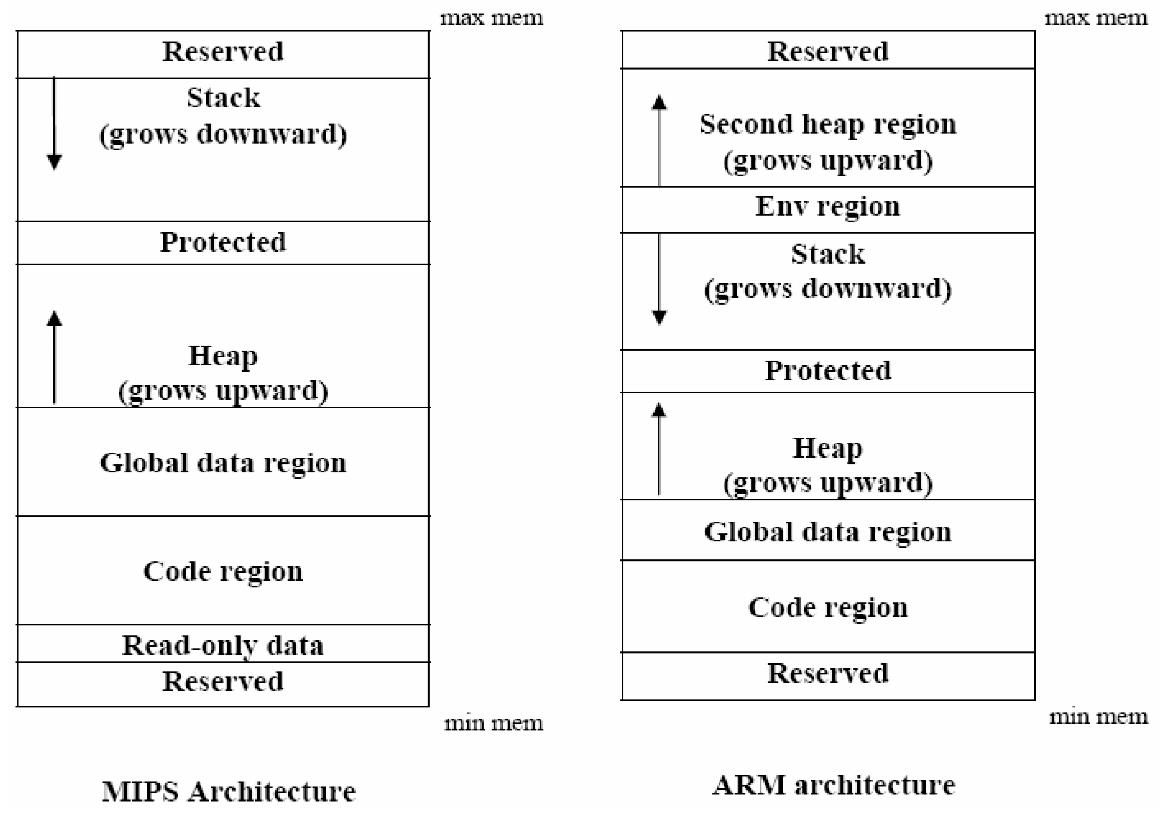


Figure 1: Run-time memory map for the MIPS and ARM architectures (adapted from [69])

comparison. The stack and heap regions contain different sets of dynamically allocated data. Stack data contain the activation records of function calls. The overall size of this region varies with the function call depth, but only a single stack frame is active at any time, keeping the working set small. Objects allocated via functions such as `malloc()` in C reside on the heap, which is usually the region with the largest footprint. These two regions often share the same memory space, as shown in the figure. The global region holds statically allocated data available at all levels of the program; the size of its working set tends to lie between that of the stack and heap regions. The previous work on region-based caching showed that, typically, the stack region holds the data with the most locality, global data has a moderate degree of locality, and data in the heap region has very low locality [67][69][70]. Because we use an ARM-based simulator model, we focus on the ARM memory map in this work. The ARM architecture features two additional regions: the rarely accessed environment (env) region and the text region. The text region must be accessed prior to global references because the base addresses of the global data are compiled as part of the text space. Therefore, global data references require two instructions—a PC-relative load to fetch the base address, and an additional load to read the actual value. The ARM architecture also provides a second heap region, placed after the env region, to prevent collisions between stack and heap data.

Previous work shows that although the stack region is the smallest of the three major semantic data regions, it is usually the most frequently accessed [67][68]. Figure 2 shows the percentage of dynamic memory references to each region for applications in the MiBench benchmark suite. As in the original work [67], about 70% of references are to stack and global data; however, our target applications feature a different access distribution across these regions. Stack references are still the most prevalent, averaging 51% of the dynamic references in MiBench applications. Heap accesses total 23% of the dynamic reference count, while global accesses comprise 18%. The text and env regions are insignificant, accounting for 7% and 0.9%, respectively. By contrast, Lee and Tyson showed a distribution of approximately 40% stack accesses, 30% heap accesses, and 30% global accesses in MediaBench applications [67]. Differences in benchmark suites and instruction set architectures lead to the discrepancies between access percentages.

Note that the reference behavior of these applications varies widely in many cases. For example, the adpcm benchmark references global data almost exclusively; recall that global accesses first require a load from the text region, which explains the high percentage of text references. The tiff benchmarks operate on large amounts of

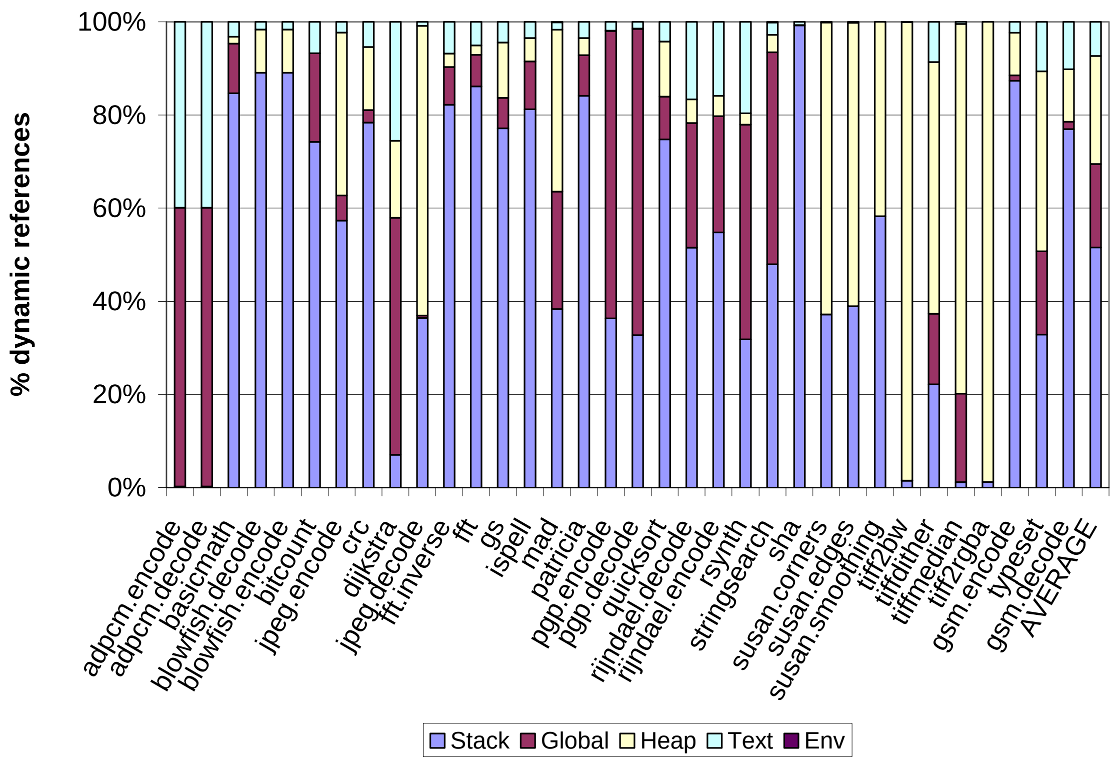


Figure 2: Reference characteristics by region for MiBench benchmark suite

dynamically allocated data, so their reference behavior is dominated by heap accesses—in the extreme case, tiff2rgba, 98.8% of the dynamic references go to the heap region. The sha benchmark is the most stack-bound of our applications, with 99.3% of its dynamic references accessing stack data.

Region-based caching [67][69] leverages the reference characteristics of stack and global data to horizontally partition caches and reduce dynamic energy consumption. As shown in Figure 3, small caches are added at the level of the L1 data cache. Stack and global region accesses are directed to the appropriate caches; all other data accesses go to the L1, as usual. Since most remaining accesses are to heap data, with a small number of

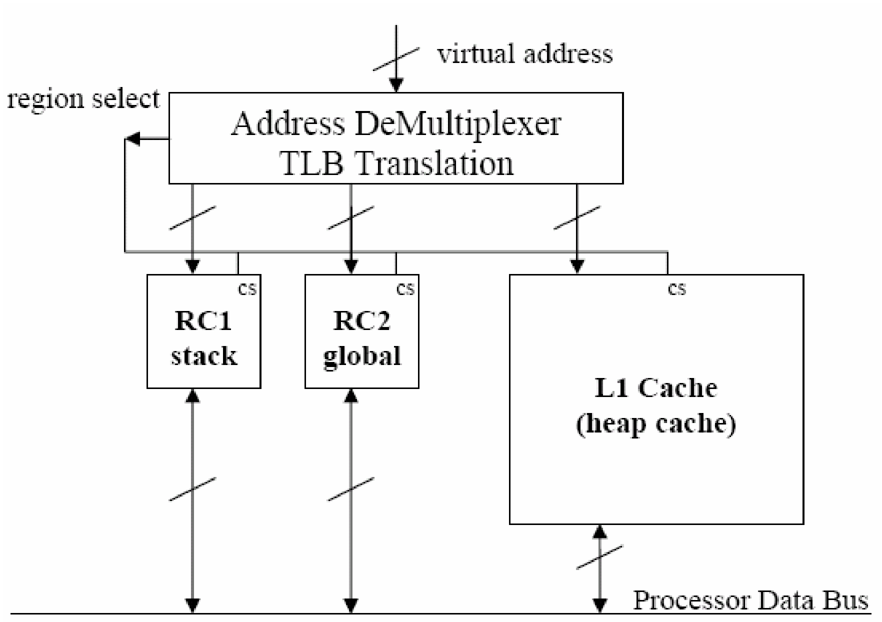


Figure 3: Memory design for region-based caching (from [69])

accesses to text and read-only data, we refer to the L1 as the heap cache in region-based caching. On a memory reference, only the appropriate region cache is activated and draws power.

Because the stack and global regions have relatively small working sets, their accesses can be routed to small caches without a significant performance penalty. Since most data references fall in those two regions, region-based caching significantly reduces average dynamic power per access. Also, splitting the references eliminates inter-region conflicts, thus allowing each region cache to employ lower associativity to reduce access time.

The downside to region-based caching is that it increases static energy consumption, as shown below. Our region caching system uses a 4 KB L1 stack cache, a 4 KB L1 global cache, and a 32 KB heap cache; all three are direct-mapped. We compare this configuration against a baseline direct-mapped 32 KB unified cache. Figure 4 shows

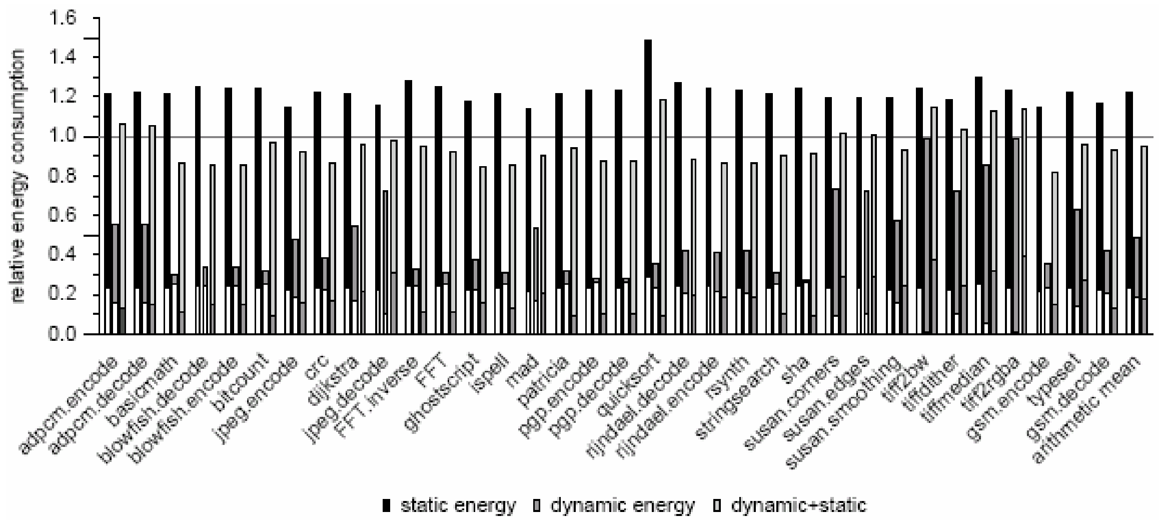


Figure 4: Energy consumption of region-based caches compared to single 32 KB direct-mapped L1 cache. In the first two bars, the white area shows the portion of energy consumption due to the stack and global caches. In the third bar, the darker bottom portion shows what fraction of the total energy is dynamic

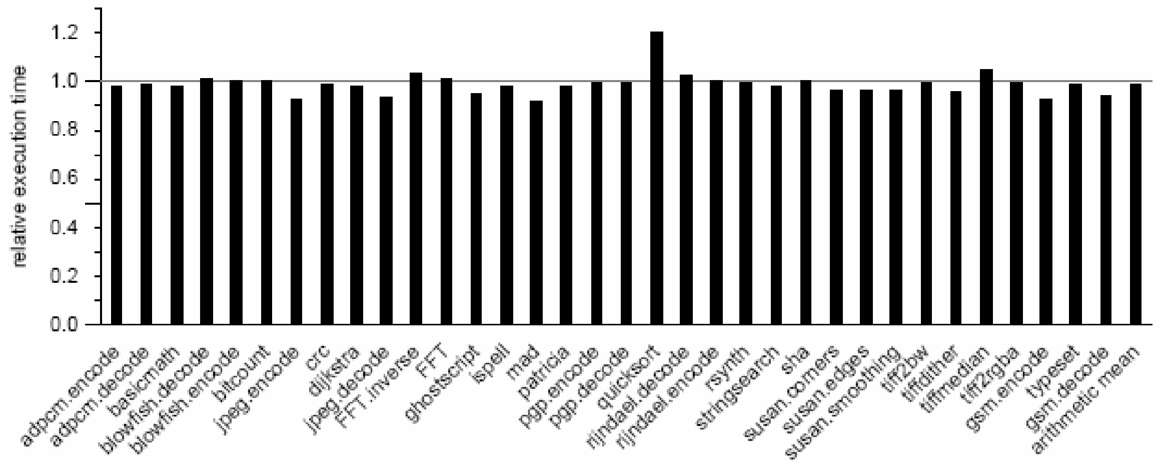


Figure 5: Performance of region-based caches compared to single 32 KB direct-mapped L1 cache

relative energy consumption for the region caches. Three vertical bars are presented for each application, indicating the change in leakage energy, switching (dynamic) energy, and total cache energy. Each of these numbers is normalized to the corresponding value for the baseline; for example, the static energy bars show the ratio of static energy consumption in our region-based caches to the static energy consumption in the baseline. Within the left two bars, the white portion indicates the fraction of energy consumed by the stack and global caches. In the third bar, we show the breakdown of static and dynamic energy contributing to the total cache energy. The first bar shows that leakage energy increases in our region-based caches by an average of 22.4% due to the two extra caches, which together are one fourth the size of the baseline cache. However, this increase is offset by a savings of 51.5% in dynamic energy, resulting in a 5.1% total energy savings compared to the unified baseline cache. Our overall savings are less than Lee and Tyson show in the original region caching study [67]; they report an average power savings of 54%. However, their study assumes 0.35um process technology parameters and therefore ignores leakage power [87], putting our results in line with theirs. These numbers also differ from the results we reported in an earlier paper [33], in which we showed a 23.6% reduction in total energy for region-based caches. In that work, we optimistically assumed single-cycle cache latencies for all caches at a 5.6 GHz clock frequency. For this research, we reduced the clock frequency to 1.7 GHz to provide more realistic memory timing. Since dynamic energy dissipation is proportional to clock frequency, lowering the frequency also lowered the dynamic energy and thus increased the impact of static energy on the overall cache energy.

Figure 5 shows the relative performance impact of region caching on these applications. In most cases, we see small speedups. The increased capacity of our region caching configuration and the faster stack and global caches improve memory latency. Nonetheless, we have not tuned the cache sizes for MiBench applications, and other configurations may yield larger speedups. On average, we experience a speedup of 1.1% for region caches vs. the 32 KB unified baseline cache. A single application, quicksort, suffers a slowdown of 20.1% due to a large increase in global misses.

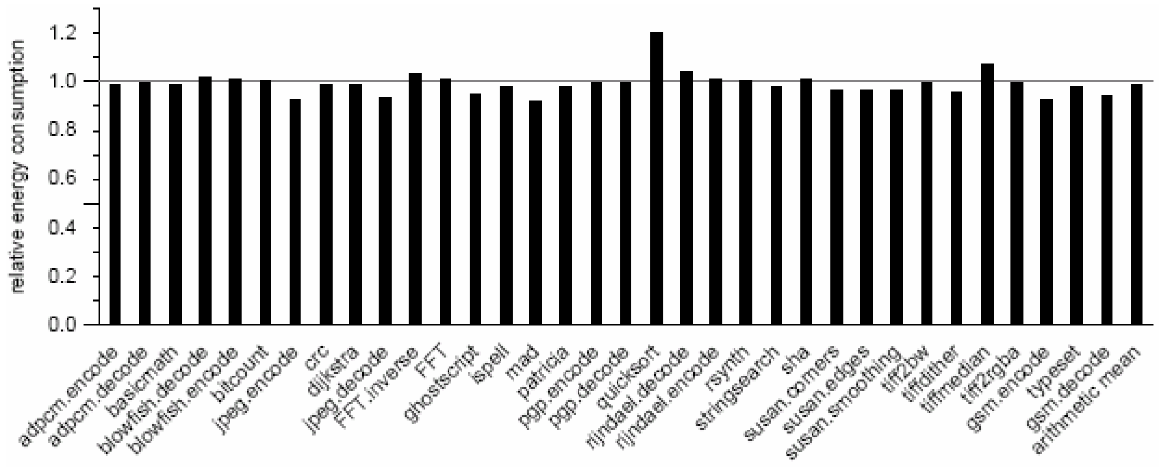


Figure 6: L2 energy consumption for system using region-based caches compared to system using single 32 KB direct-mapped L1 cache

One potential concern is that region-based caching may increase energy consumption in the level 2 data cache if an application's stack and global data do not fit well in the smaller caches. An increase in misses in these regions leads to more L2 accesses and may also raise the program's execution time. Of the 34 applications we studied, 3 experienced more stack misses with region-based caches and 17 experienced more global misses. However, as shown in Figure 6, region-based caching only causes a minimal increase in L2 energy consumption. The misses primarily affect dynamic energy consumption, which increases significantly in some applications—up to 98 times the baseline level (in sha, a stack-intensive benchmark). However, static energy dominates in the level 2 cache, as it is a large, infrequently accessed structure. The static energy consumption is proportional to the program runtime; we see that both experience the same average decrease, 1.1%. Overall, region-based caching causes a 1.0% decrease in the total energy dissipated in the L2 cache.

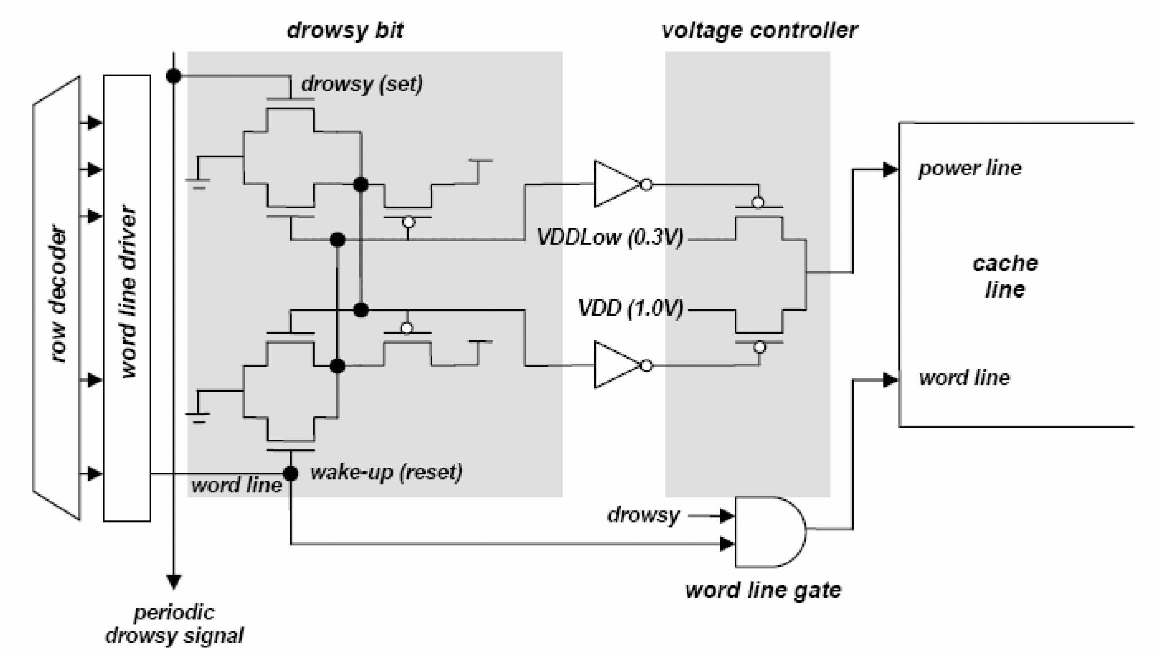


Figure 7: Drowsy cache line (adapted from [58])

4.2 Drowsy Caching

We can use any of the static energy reduction methods discussed in Chapter 2 to combat the static energy increase in region-based caches. We choose drowsy caching [31][57][58] because it is a state-preserving technique and therefore has relatively little impact on performance. We use the dynamic voltage scaling implementation first proposed in Flautner et al [31]. Figure 7 shows the hardware to implement a drowsy cache line—a drowsy bit, a voltage control mechanism, and a word line gating circuit. The voltage controller switches the line's supply voltage between high (active) and low (drowsy) values depending on the drowsy bit state. The word line gate prevents accesses while in drowsy mode, avoiding destruction of a drowsy line's data. When accessed, a drowsy line's bit is cleared, returning the supply voltage to its active value. If tags are kept drowsy, they may need to be awakened, thus increasing wakeup latency. Direct-mapped caches derive no benefit from keeping tags awake since there is only one line per index. We model direct-mapped caches and assume drowsy tags.

Flautner et al [31] present two policies for setting the per-line drowsy bits. In the **simple** policy, all lines become drowsy after a certain number of cycles—the **update interval**. In the **noaccess** policy, only lines not accessed within the interval become drowsy. The **simple** policy reduces leakage power more effectively than the **noaccess** policy, since the former moves lines from active to drowsy more aggressively. However, it is locally oblivious, and may increase execution time when lines soon to be accessed are placed into drowsy mode. Flautner et al. [31] find a minimal difference in performance among policies.

4.3 Drowsy Region-Based Caching

Figure 8 and Figure 9 show simulation results for drowsy caching and region-based caching for a representative subset of the MiBench suite. Here the baseline is again a 32 KB direct-mapped unified cache, but made drowsy with a 4K-cycle update interval. In Figure 8, the left three bars for each application indicate relative total energy for 32 KB unified caches with update intervals of one cycle (always drowsy), 2K cycles, and 8K cycles. The right two bars show relative total energy for drowsy region caches with a 4K cycle update interval. For the first of these, only the heap cache is drowsy, whereas for the second, all three region caches are drowsy. Changing the update interval has little impact on total energy consumption of the unified 32 KB direct-mapped cache. Leakage energy increases as the update interval grows, but dynamic energy remains almost constant. In contrast, going from a drowsy unified cache to a drowsy region cache configuration yields greater energy savings. Making just the heap cache drowsy yields a

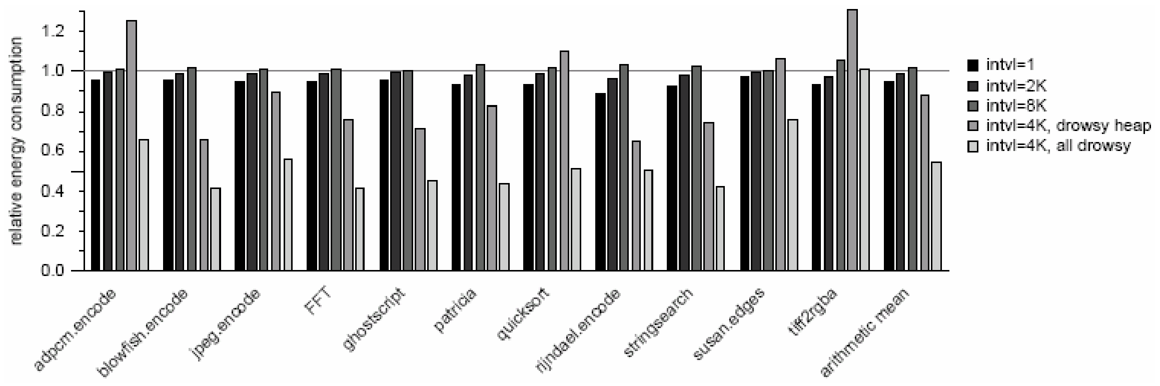


Figure 8: Energy consumption for varying drowsy intervals and drowsy region caches compared to a 32 KB drowsy L1 with 4K-cycle interval

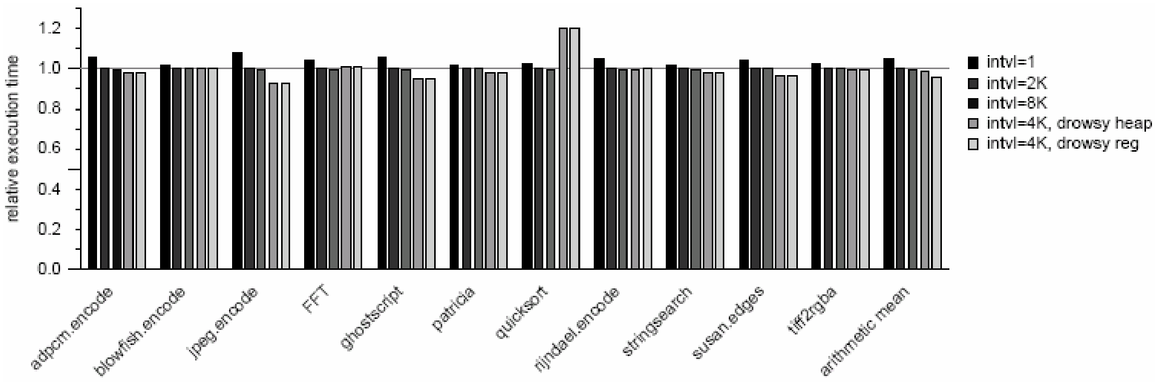


Figure 9: Performance for varying drowsy intervals and drowsy region caches compared to a 32 KB drowsy L1 with 4K-cycle interval

12% energy savings, whereas making all regions drowsy brings total savings up to 45%. As with the unified cache, changing update intervals for the region caches has a minimal effect on total energy.

Figure 9 shows the performance impact of changing update intervals or adding region caches, using the same baseline—a 32 KB direct-mapped unified cache with a 4K-cycle update interval. One visible trend is that performance improves for larger update intervals, at the expense of slightly higher total energy consumption. If region-based caching is not used for these applications, any of the larger intervals represents a reasonable design choice.

Figure 10 and Figure 11 show the relative effectiveness of region caching and drowsy caching both alone and as a combination. The figures compare energy and performance of a baseline 32 KB direct-mapped unified L1 cache to a drowsy version with a 4K-cycle update interval and to our region cache configuration (4 KB stack cache, 4 KB global cache, and 32 KB heap cache) under three different drowsiness schemes—no drowsiness, only the heap cache drowsy, and all three regions drowsy. Figure 10 shows total cache energy compared to the non-drowsy, unified baseline. The combination of region and drowsy caching significantly reduces overall energy consumption. A drowsy heap cache with standard stack and global caches reduces total energy by 63%, and making all region caches drowsy reduces energy by 77%. Drowsy caching alone reduces total energy by 58%; region caching alone reduces total energy by only 5%.

As expected, our results show that drowsy caching has the greatest impact on the heap cache, the largest of the three region caches. However, combining the techniques increases leakage energy over drowsy caching alone, especially when the stack and global caches are not drowsy. When all regions are drowsy, the increased cache capacity leads to an increase of about 10% in leakage energy.

Figure 11 shows that the performance impact of combining techniques is very small. Since drowsy caching has a negligible impact on performance, the runtime penalty is effectively equal to the cost of region caching—about 1%. Performance drops in only a few applications; FFT, quicksort, and rijndael.encode are shown in the figure, but basicmath, dijkstra, FFT.inverse, rijndael.decode, and tiffmedian also suffer performance losses. In all cases, the cause is a dramatic increase in global cache misses.

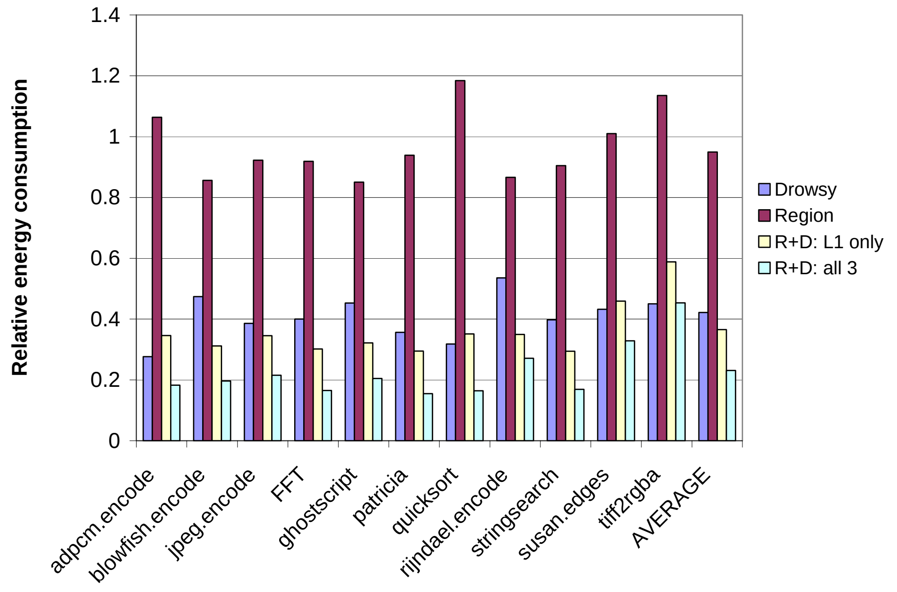


Figure 10: Energy consumption of combined region and drowsy caching

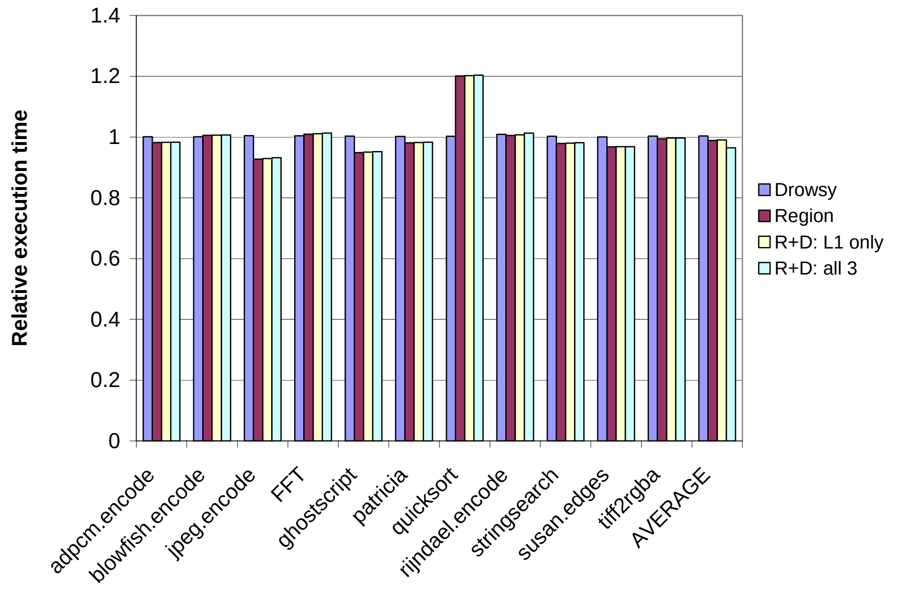


Figure 11: Performance of combined region and drowsy caching

Region and drowsy caching function well together, and our figures demonstrate this, but they do not highlight the techniques' ability to perform better together than individually. In the comparisons performed above, which use a non-drowsy 32 KB direct-mapped cache as the baseline, our experimental results show that the reduction in energy from the combined techniques is greater than the sum of reductions for each technique alone—77% versus 63% (58%+5%). To understand why, we examine how region caching achieves its 5% energy reduction. Region caches are very effective at reducing dynamic energy, achieving a 51% reduction, but this decrease is offset by an increase of 22% in static energy. The total energy saved is thus smaller—5% on average. However, drowsy caches effectively eliminate most static energy dissipation, including the extra static energy from the additional region caches. When we count the number of cache lines not in drowsy mode, the unified drowsy cache averages 9.5 non-drowsy lines per execution cycle versus only 8.2 for drowsy region caches, because the latter organization increases the effectiveness of the drowsy selection hardware. This figure illuminates how the techniques work well together. The removal of inter-region conflicts allows data to remain in the cache longer. The longer a block avoids eviction, the more likely it is to remain drowsy, as the block must return to active mode before being replaced.

4.4 Optimizing Drowsy Intervals

Splitting the caches by region groups together data with similar locality. Highly local data tends to stay active for a short period of time and then become inactive for a much longer stretch. At that point, we can safely place those lines into drowsy mode without their being accessed in the near future. By contrast, in data with little locality, multiple accesses to a line in a short interval are rare, meaning that once a line is accessed, it is unlikely to be accessed again for many cycles. Capitalizing on these traits advocates moving such lines into drowsy mode soon after being accessed.

Region-based caching already allows us to be more aggressive with drowsy caching policies because it splits the reference stream and ensures that each cache only sees a portion of the total data accesses. If we further consider the locality characteristics of each region, we see that we can be very aggressive with the lines in the heap cache, which tend to be referenced rarely, and still relatively aggressive with the lines in the stack cache once we know they have become inactive. Highly aggressive drowsy caching could benefit more from the **noaccess** policy, which would ensure that active lines remain active during the time when they are highly active.

Increasing the aggressiveness of our drowsy caching policy means decreasing the update interval, and this change has a positive effect on the circuit overhead of the drowsy cache as well. A shorter interval implies a smaller cycle counter. If the interval shrinks to one—meaning that a line is put into drowsy mode directly after being accessed—the structure of the drowsy cache line also changes. The drowsy bit is no longer necessary, since each line is either active or drowsy. It is true that such an aggressive policy will increase the number of accesses that must pay the penalty for switching a cache line from drowsy to active. However, the performance penalty should be offset by an equally significant energy reduction.

Figure 12 highlights the difference in energy between typical drowsy caches with a large update interval and aggressive, always drowsy caches. We compare a 4K-cycle update interval to a 1-cycle interval for both a single L1 data cache and for our typical region-based caching configuration, using a non-drowsy 32 KB direct-mapped cache as the baseline. This graph primarily shows the added benefit of combining drowsy and region caches. When the cache configurations are identical, total power consumption is similar across interval sizes. Reduced interval size has a significant impact on leakage power. For a 1-cycle interval, leakage energy is 29% less than it is with a 4K-cycle window, leading to a 1.6% decrease in total energy, as shown in the figure.

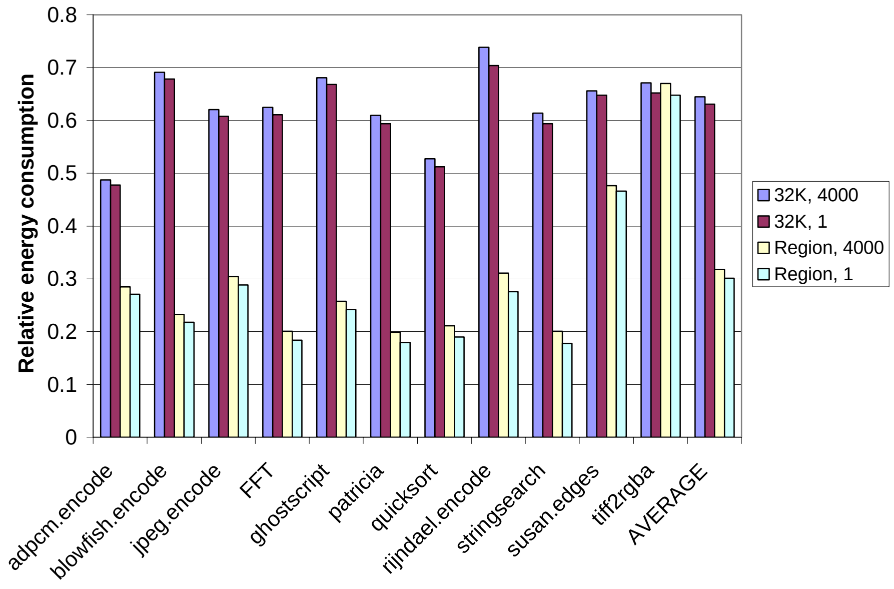


Figure 12: Comparison of large and small windows

As noted above, varying the update interval has little effect on total power for both unified and region caches. However, the effect on performance is somewhat more pronounced. To find the best interval for each cache, we vary the interval from 1 to 8K cycles and plot the resulting energy/performance curves, shown in Figure 13 and Figure 14. All energy and runtime values are averages over all applications. To test region caches, we use a 4K-cycle interval for two of the caches and vary the interval of the third. We choose the interval at the "knee" of the curve—the point at which the speedup decreases more than the energy consumption. As shown in Figure 13, 512 cycles is the best interval for a unified L1 cache. For region caches, as shown in Figure 14, the ideal interval differs for each region—512 cycles for the stack and heap, 256 cycles for the global region.

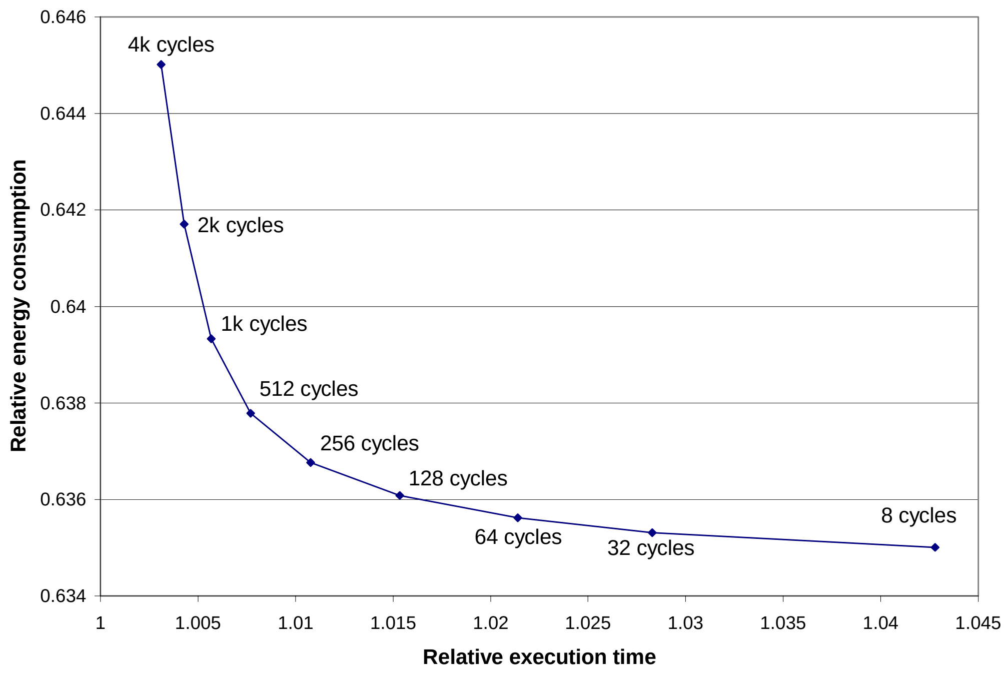


Figure 13: Energy-performance curve for varying drowsy intervals in a unified L1 data cache

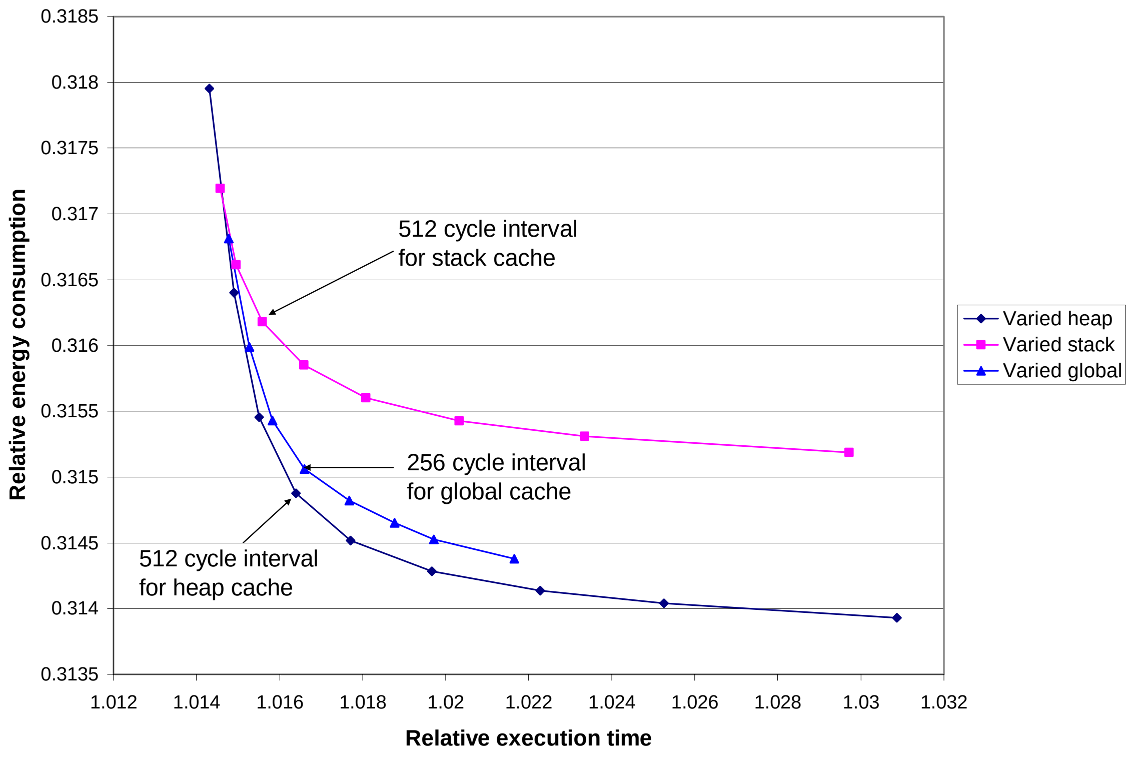


Figure 14: Energy/performance curves for varying drowsy intervals in region-based caches

The data shows that aggressively applying drowsy caching to global data has the smallest effect on performance. Since the global region is accessed the least of the three major regions, it follows that it should experience the fewest wakeups. The surprising result from this data is that heap and stack data impact performance similarly under drowsy caching. Their worst-case penalties are similar—3.1% for the heap, 3.0% for the stack—and their ideal interval is the same. This result suggests that some heap data possess a similar degree of locality to stack data, an idea we explore further in Chapter 5.

Note that we still only require one counter to implement multiple drowsy intervals as long as the intervals remain powers of 2. The structure of binary counters ensures that an **n**-bit counter contains an (**n**-1)-bit counter, an (**n**-2)-bit counter, and so on. For a counter that resets every 2^m cycles (**m** < **n**), all we require is additional logic to recognize when the **m** low-order bits of the **n**-bit counter overflow. With an increased emphasis on wire delays as technology shrinks [2][75][82], there is some question as to which organization makes more sense: a single, central counter sending drowsy signals to all region caches, or a separate counter placed close to each cache to minimize wire length. Investigating this issue is beyond the scope of this dissertation.

Our optimum drowsy intervals for each cache are average intervals and are therefore not tailored to individual application behavior. Given the wide variance in reference behavior shown in Figure 2, we could likely benefit more from application-specific intervals. However, such an approach would require the ability to program the interval counters. We choose to statically define the intervals for all programs to avoid this overhead.

4.5 Summary

In this section, we have shown that the combination of two techniques for reducing memory system energy, region-based caching and drowsy caching, can have a benefit that is greater than the sum of their parts. Both methods achieve significant reductions in their targeted domains—dynamic energy for region-based caches, leakage energy for drowsy caches. Because region-based caching splits the reference stream into groups with similar locality, the activity of the separate caches is well defined. Drowsy caching exploits the periods of inactivity seen in both high and low locality data to remove the static energy penalty inherent in region-based caching. The result is a significant reduction in total energy consumption—as high as 77%—with a minimal performance penalty.

CHAPTER 5

HEAP CACHING STRATEGIES

Region-based caching exploits the locality of stack and global data to reduce energy consumption. However, the heap region, the most difficult region of memory to manage well in a cache structure, limits the effectiveness of this technique. In this chapter, we explore a simple modification to demonstrate the benefits of further specialization: large and small heap caches. Applications that do not need a large cache save energy by using the smaller structure and turning off the larger. The remaining applications can save energy by keeping frequently used "hot" data in the smaller, lower-energy cache. This work was first presented in our second paper on drowsy region-based caches [33], and further explored in later work [34][35].

In Section 5.1, we discuss how to identify hot heap data from a data-centric perspective. We begin with a detailed analysis of heap data characteristics to determine the best heap caching strategy and cache size, and then show how our methods impact energy and performance. Section 5.2 approaches this problem from a different perspective, analyzing the characteristics of memory instructions that access the heap. Section 5.3 summarizes the chapter.

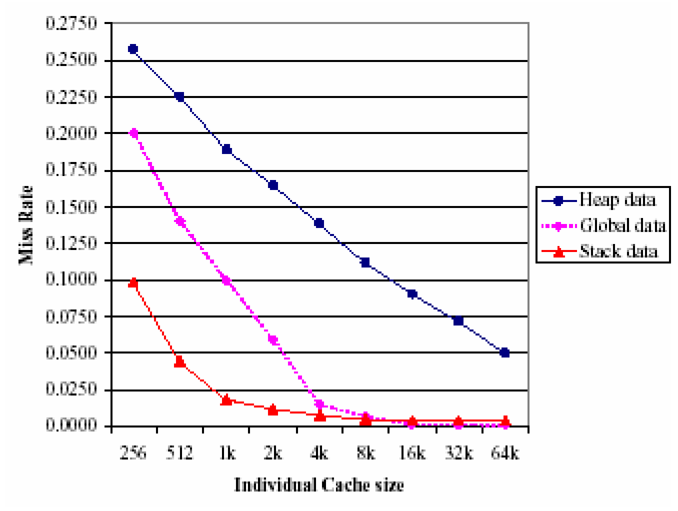


Figure 15: Miss rate by region in MediaBench applications for varying cache sizes and configurations [67]. The stack and global regions display high hit rates in very small caches, but in the heap region, miss rate increases linearly with cache size.

5.1 Data-Centric Heap Caching

The chief difficulty in caching heap data is that they typically exhibit low locality and have a large footprint. Figure 15 [67] plots miss rate by region versus cache size for applications in the Mediabench suite [66]. As the figure shows, stack and global data cache well even in small structures. We can approach a 99% hit rate on stack accesses with a very small cache, and global data only require slightly greater cache capacity. However, the miss rate of heap accesses decreases linearly as cache size doubles, suggesting that heap data possess poor locality and thus do not cache well. Figure 16, taken from Lee and Ballapuram [70], shows the address footprint distribution of different regions in the cjpeg application. Each point represents a cache hit at a given address, with the high order address bits plotted on the y-axis and the low order address bits on the x-axis. As the figure shows, heap references cover a much wider range of unique addresses than either of the other two regions; note that this statistic implies that heap accesses will suffer significantly more compulsory cache misses than stack or global

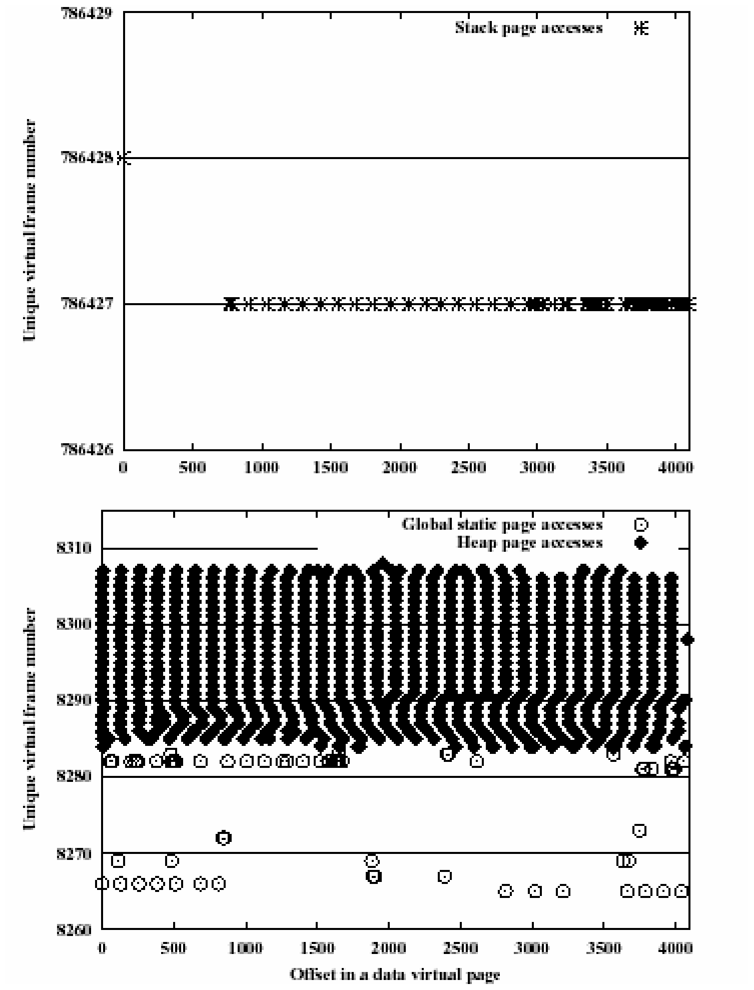


Figure 16: Address footprint distribution of different regions in cjpeg [70]. Each point represents a cache hit at a particular address. The top graph shows stack accesses; the bottom graph, global and heap accesses. Heap references cover a much wider range of unique addresses than either stack or global data.

accesses. Lee and Ballapuram also note that benchmarks from the SPEC2000 integer suite [101] exhibit much larger heap footprints than MiBench applications, suggesting that heap working sets tend to expand faster than working sets of other regions [70].

These figures only tell part of the story. Figure 15 gives the average caching behavior of a group of benchmarks, but gives no insight into the behavior of individual programs. Figure 16 demonstrates how well a particular application caches data in each region, but does not show if these trends hold true for multiple applications. Based on these two figures, it would be easy to assume that all heap data shows similar characteristics. However, we show that the characteristics of heap data vary widely from application to application. The trends we show suggest a customizable solution will be the most effective way to cache heap data.

5.1.1 Heap Data Characteristics

We begin by examining the locality characteristics of heap data. Table 4 assesses the significance of the heap region within each target application, looking at its overall size and number of accesses relative to the other semantic regions. The second and third columns of the table show the number of unique block addresses accessed in the heap cache and the number of accesses to those addresses, respectively. Since our simulations assume 32B cache blocks, 1 KB of data contains 32 unique block addresses. The fourth and fifth columns show this same data as a percentage of the corresponding values for all regions (i.e., the fourth column shows the ratio of unique data addresses in the heap region to all unique data addresses in the application). We can see several cases that bear out the previous assertions about heap data: they have a large footprint and low locality. In these applications, the heap cache accesses occupy a much larger percentage of the overall footprint than of the total accesses. The most extreme cases are applications such as FFT.inverse and patricia in which heap accesses account for over 99% of the unique addresses accessed throughout the programs but comprise less than 7% of the total data accesses. This relationship holds in most applications; heap accesses cover an average of 65.7% of the unique block addresses and account for 29.8% of the total data accesses. In some cases, we see a correlation between footprint size and number of accesses—applications with few heap lines and few accesses, like pgp.encode, and applications

Benchmark	# unique addresses	Accesses to heap cache	% total unique addresses	% total accesses
adpcm.encode	69	39971743	27.6%	39.9%
adpcm.decode	68	39971781	27.0%	39.9%
basicmath	252	49181748	61.2%	4.5%
blowfish.decode	213	39190633	39.0%	10.2%
blowfish.encode	212	39190621	38.9%	10.2%
bitcount	112	12377683	42.7%	6.7%
jpeg.encode	26012	10214537	99.2%	29.3%
CRC32	90	159955061	41.1%	16.7%
dijkstra	347	44917851	19.7%	38.3%
jpeg.decode	1510	7036942	90.2%	62.9%
FFT	16629	15262360	99.2%	8.6%
FFT.inverse	16630	14013100	99.2%	6.3%
ghostscript	59594	56805375	98.0%	15.3%
ispell	13286	28000346	96.5%	6.4%
mad	2123	40545761	82.3%	36.4%
patricia	110010	16900929	99.9%	6.6%
pgp.encode	298	252620	7.4%	1.9%
pgp.decode	738	425414	44.9%	1.5%
quicksort	62770	152206224	66.7%	12.9%
rijndael.decode	229	37374614	31.0%	21.7%
rijndael.encode	236	35791440	40.0%	19.6%
rsynth	143825	104084186	99.2%	21.4%
stringsearch	203	90920	18.2%	6.2%
sha	90	263617	20.9%	0.7%
susan.corners	18479	9614163	97.1%	63.6%
susan.edges	21028	22090676	99.1%	62.3%
susan.smoothing	7507	179696772	97.0%	41.7%
tiff2bw	2259	57427236	92.1%	98.5%
tiffdither	1602	162086279	83.1%	62.8%
tiffmedian	4867	165489090	53.0%	79.8%
tiff2rgba	1191987	81257094	100.0%	98.5%
gsm.encode	302	157036702	68.0%	11.7%
typeset	168075	153470300	98.0%	49.0%
gsm.decode	285	78866326	55.6%	21.5%
		AVERAGE	65.7%	29.8%

Table 4: Characteristics of heap cache accesses in MiBench applications, including total footprint size, total number of accesses, and relative contribution of heap data to the overall data footprint and reference count

with a large percentage of both cache lines and accesses, like tiff2rgba. A few outliers buck the trend entirely, containing frequently accessed heap data with a relatively small footprint; dijkstra is one example.

We see that about half of the applications have a fairly small number of lines in the heap, with 16 of the 34 applications containing fewer than 1000 unique addresses. The adpcm application has the smallest footprint, using 69 and 68 unique addresses—just over 2 KB of data—in the encode and decode phases, respectively. The typical 32 KB L1 heap cache is likely far larger than these applications need; if we use a smaller heap cache, we can dissipate less dynamic power per access with a minimal effect on performance. Since heap cache accesses still comprise a significant percentage of the overall data accesses, this change should have a noticeable effect on the dynamic energy consumption of these benchmarks. Shrinking the heap cache will also reduce its static energy consumption. Previous resizable caches disable unused ways [4][111] or sets [110][111] in set-associative caches; we can use similar logic to simply disable the entire large heap cache and route all accesses to the small cache when appropriate.

Shrinking the heap cache may reduce the energy consumption of the remaining benchmarks, but the resulting performance loss may be too great to tolerate for applications with a large heap footprint. However, we can still gain some benefit by identifying a small subset of addresses with good locality and routing their accesses to a smaller structure. Because we want the majority of references to dissipate less power, we should choose the most frequently accessed lines. The access count gives some sense of the degree of temporal locality for a given address.

Usually, a small number of blocks are responsible for the majority of the heap accesses, as shown in Table 5. The table gives the number of lines needed to cover different percentages—50%, 75%, 90%, 95%, and 99%—of the total accesses to the heap cache. We can see that, on average, just 2.14% of the cache lines cover 50% of the accesses. Although the rate of coverage decreases somewhat as you add more blocks—in other words, the first N blocks account for more accesses than the next N blocks—we still only need 5.84% to cover 75% of the accesses, 13.2% to cover 90% of the accesses,

Benchmark	# unique addresses	% unique addresses needed to cover given percentage of heap cache accesses				
		50%	75%	90%	95%	99%
adpcm.encode	69	1.45%	2.90%	2.90%	2.90%	2.90%
adpcm.decode	68	1.47%	1.47%	1.47%	1.47%	1.47%
basicmath	252	3.97%	25.40%	48.02%	55.56%	61.90%
blowfish.decode	213	0.94%	1.41%	2.35%	26.76%	55.87%
blowfish.encode	212	0.94%	1.42%	2.36%	26.89%	56.13%
bitcount	112	0.89%	1.79%	2.68%	3.57%	3.57%
jpeg.encode	26012	0.10%	0.65%	2.91%	38.19%	87.28%
CRC32	90	2.22%	3.33%	4.44%	4.44%	4.44%
dijkstra	347	0.29%	18.16%	39.19%	49.57%	63.11%
jpeg.decode	1510	4.77%	12.32%	31.85%	44.11%	59.47%
FFT	16629	0.05%	0.14%	4.82%	40.67%	85.33%
FFT.inverse	16630	0.05%	0.14%	13.02%	44.00%	86.51%
ghostscript	59594	0.01%	0.04%	0.56%	6.64%	57.49%
ispell	13286	0.09%	0.23%	0.46%	0.68%	1.29%
mad	2123	1.32%	2.64%	9.70%	14.88%	24.54%
patricia	110010	0.02%	0.06%	0.32%	36.64%	86.03%
pgp.encode	298	0.67%	1.01%	3.69%	6.71%	26.85%
pgp.decode	738	0.27%	0.41%	1.08%	2.30%	29.67%
quicksort	62770	0.02%	0.04%	0.15%	22.08%	49.13%
rijndael.decode	229	1.31%	2.18%	6.55%	31.44%	57.21%
rijndael.encode	236	1.27%	2.97%	7.63%	32.63%	56.78%
rsynth	143825	0.00%	0.00%	0.01%	1.28%	77.33%
stringsearch	203	17.24%	42.86%	59.61%	65.52%	72.91%
sha	90	1.11%	2.22%	3.33%	3.33%	8.89%
susan.corners	18479	0.03%	3.02%	11.04%	14.87%	32.66%
susan.edges	21028	0.02%	4.92%	15.13%	20.22%	30.42%
susan.smoothing	7507	0.01%	0.09%	13.72%	30.25%	44.11%
tiff2bw	2259	10.27%	15.41%	24.26%	29.39%	37.05%
tiffdither	1602	9.43%	19.60%	25.72%	29.59%	40.76%
tiffmedian	4867	4.03%	10.89%	16.72%	20.81%	47.83%
tiff2rgba	1191987	0.04%	0.11%	57.39%	78.69%	95.73%
gsm.encode	302	2.32%	3.97%	5.96%	7.62%	10.60%
typeset	168075	5.55%	15.41%	25.53%	33.02%	60.12%
gsm.decode	285	0.70%	1.40%	4.21%	5.96%	30.53%
AVERAGE (all apps)		2.14%	5.84%	13.20%	24.49%	45.47%
AVERAGE (>1k unique addrs)		1.99%	4.76%	14.07%	28.11%	55.73%

Table 5: Number of unique addresses required to cover different fractions of accesses to the heap cache in MiBench applications. The data show that a small number of lines account for the majority of heap cache accesses, indicating that some of these lines possess better locality than previously believed. This trend is more apparent in applications with large heap cache footprints

24.49% to cover 95% of the accesses, and 45.47% to cover 99% of the accesses. The percentages do not tell the whole story, as the footprint sizes are wildly disparate for these applications. However, the table also shows that in applications with large footprints (defined as footprints of 1000 unique addresses or more), the percentage of addresses is lower for the first two coverage points (50% and 75%). This statistic implies that we can identify a relatively small subset of frequently accessed lines for all applications, regardless of overall footprint size.

Since a small number of addresses account for a significant portion of the heap cache accesses, we can route these frequently accessed data to a smaller structure to reduce the energy consumption of the L1 data cache. Our goal is to maximize the low-power accesses without a large performance penalty, so we need to judiciously choose which data to place in the hot heap cache. To estimate performance impact, we use the Cheetah cache simulator [103] to find a lower bound on the miss rate for a given number of input data lines. We simulate fully-associative 2 KB, 4 KB, and 8 KB caches with optimal replacement [8] and route the N most frequently accessed lines to the cache, varying N by powers of 2. We use optimal replacement to minimize conflict misses and give a sense of when the cache is filled to capacity; the actual miss rate for our direct-mapped hot heap cache will be higher., Table 6, Table 7, and Table 8 show the results of these simulations for 2 KB, 4 KB, and 8 KB caches, respectively. We present only a subset of the applications, omitting programs with small heap footprints and a worst-case miss rate less than 1% because they will perform well at any cache size.

Unfortunately, these simulations suggest little about how to split the heap cache. In most cases, the miss rate rises precipitously for small values of N, but levels off around N = 512 or 1024. This result reflects the fact that most accesses are concentrated at a small number of addresses. However, miss rate alone does not establish the suitability of a given caching scheme for heap data. Applications in which these accesses comprise a high percentage of total data references are less likely to tolerate a high miss rate.

Benchmark	Miss rate for given N value						
	128	256	512	1024	2048	4096	8192
jpeg.encode	0.2%	0.8%	1.8%	2.5%	2.5%	2.5%	2.5%
dijkstra	4.2%	8.0%	8.0%	8.0%	8.0%	8.0%	8.0%
jpeg.decode	0.4%	0.9%	1.7%	2.8%	2.8%	2.8%	2.8%
FFT	0.1%	0.1%	0.2%	0.3%	0.4%	0.8%	1.4%
FFT.inverse	0.1%	0.1%	0.2%	0.3%	0.5%	0.8%	1.5%
ghostscript	0.0%	0.2%	0.3%	0.5%	0.6%	0.6%	0.8%
ispell	0.2%	0.4%	0.4%	0.4%	0.4%	0.4%	0.4%
mad	0.7%	1.6%	2.4%	2.4%	2.4%	2.4%	2.4%
patricia	0.7%	1.3%	1.8%	1.9%	2.0%	2.0%	2.1%
quicksort	0.0%	0.0%	0.1%	0.1%	0.1%	0.2%	0.2%
rsynth	0.0%	0.0%	0.0%	0.0%	0.1%	0.1%	0.1%
stringsearch	1.8%	2.0%	2.0%	2.0%	2.0%	2.0%	2.0%
susan.corners	0.0%	0.0%	0.0%	0.0%	0.1%	0.1%	0.2%
susan.edges	0.0%	0.0%	0.0%	0.0%	0.1%	0.1%	0.2%
susan.smoothing	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
tiff2bw	2.5%	3.8%	4.7%	5.7%	5.7%	5.7%	5.7%
tiffdither	0.4%	0.8%	1.3%	1.6%	1.6%	1.6%	1.6%
tiffmedian	0.5%	1.2%	2.0%	3.4%	3.5%	3.4%	3.4%
tiff2rgba	2.5%	3.8%	4.6%	6.1%	7.1%	7.1%	7.1%
typeset	1.4%	2.6%	2.7%	3.0%	3.4%	4.0%	5.0%

Table 6: Miss rates for a fully-associative 2 KB cache using optimal replacement for different numbers of input addresses, N. These results establish a lower bound for the miss rate when caching these data. Applications shown either have a large heap footprint, which we define as a footprint of at least 1000 unique addresses, or a worst-case miss rate above 1%

Benchmark	Miss rate for given N value						
	128	256	512	1024	2048	4096	8192
jpeg.encode	0.0%	0.3%	0.9%	1.4%	1.5%	1.5%	1.5%
dijkstra	0.0%	2.7%	2.7%	2.7%	2.7%	2.7%	2.7%
jpeg.decode	0.0%	0.3%	0.7%	1.4%	1.5%	1.5%	1.5%
FFT	0.0%	0.0%	0.1%	0.1%	0.3%	0.6%	1.3%
FFT.inverse	0.0%	0.0%	0.1%	0.2%	0.4%	0.7%	1.4%
ghostscript	0.0%	0.0%	0.0%	0.1%	0.2%	0.3%	0.4%
ispell	0.0%	0.1%	0.1%	0.1%	0.1%	0.1%	0.1%
mad	0.0%	0.8%	1.6%	1.6%	1.6%	1.6%	1.6%
patricia	0.0%	0.3%	0.5%	0.6%	0.6%	0.6%	0.7%
quicksort	0.0%	0.0%	0.0%	0.1%	0.1%	0.1%	0.2%
rsynth	0.0%	0.0%	0.0%	0.0%	0.0%	0.1%	0.1%
stringsearch	0.2%	0.5%	0.5%	0.5%	0.5%	0.5%	0.5%
susan.corners	0.0%	0.0%	0.0%	0.0%	0.0%	0.1%	0.1%
susan.edges	0.0%	0.0%	0.0%	0.0%	0.0%	0.1%	0.1%
susan.smoothing	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
tiff2bw	0.0%	2.5%	3.9%	5.0%	5.0%	5.0%	5.0%
tiffdither	0.0%	0.5%	1.1%	1.3%	1.3%	1.3%	1.3%
tiffmedian	0.0%	0.8%	1.3%	2.9%	3.0%	3.0%	3.0%
tiff2rgba	0.0%	2.5%	3.1%	4.6%	5.8%	5.8%	5.8%
typeset	0.0%	0.1%	0.2%	0.5%	0.9%	1.4%	2.3%

Table 7: Miss rates for a fully-associative 4 KB cache using optimal replacement for different numbers of input addresses. Applications are the same set shown in Table 6

Benchmark	Miss rate for given N value						
	128	256	512	1024	2048	4096	8192
jpeg.encode	0.0%	0.0%	0.2%	0.6%	0.6%	0.7%	0.7%
dijkstra	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
jpeg.decode	0.0%	0.0%	0.2%	0.7%	0.7%	0.7%	0.7%
FFT	0.0%	0.0%	0.0%	0.1%	0.3%	0.6%	1.2%
FFT.inverse	0.0%	0.0%	0.0%	0.1%	0.3%	0.7%	1.4%
ghostscript	0.0%	0.0%	0.0%	0.0%	0.1%	0.1%	0.2%
ispell	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
mad	0.0%	0.0%	0.8%	0.9%	0.9%	0.9%	0.9%
patricia	0.0%	0.0%	0.1%	0.2%	0.3%	0.3%	0.3%
quicksort	0.0%	0.0%	0.0%	0.0%	0.1%	0.1%	0.1%
rsynth	0.0%	0.0%	0.0%	0.0%	0.0%	0.1%	0.1%
stringsearch	0.2%	0.2%	0.2%	0.2%	0.2%	0.2%	0.2%
susan.corners	0.0%	0.0%	0.0%	0.0%	0.0%	0.1%	0.1%
susan.edges	0.0%	0.0%	0.0%	0.0%	0.0%	0.1%	0.1%
susan.smoothing	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
tiff2bw	0.0%	0.0%	2.4%	3.6%	3.7%	3.7%	3.7%
tiffdither	0.0%	0.0%	0.6%	0.8%	0.8%	0.8%	0.8%
tiffmedian	0.0%	0.0%	0.2%	1.9%	2.0%	2.0%	2.0%
tiff2rgba	0.0%	0.0%	0.5%	1.7%	3.2%	3.3%	3.3%
typeset	0.0%	0.0%	0.0%	0.0%	0.2%	0.6%	1.3%

Table 8: Miss rates for a fully-associative 8 KB cache using optimal replacement for different numbers of input addresses. Applications shown are the same set shown in Table 6

5.1.2 Split Heap Heuristics

Section 5.1.1 motivates the need for two separate heap caches, one large and one small, to accommodate the needs of all applications. As shown in Table 4, many applications have small heap footprints and therefore do not require a large heap cache; in these cases, we can disable the large cache and place all heap data in the smaller structure. This approach will reduce dynamic energy by routing accesses to a smaller structure and reduce static energy by decreasing the active cache area. Applications with large heap footprints are more likely to require both caches to maintain performance. We showed in Table 5 that most heap references access a small subset of the data; by keeping this hot data in the smaller structure, we can save dynamic energy. In all cases, we can further lower static energy consumption by making the caches drowsy.

In order to gain the maximum benefit from split heap caching, we would like to route as many accesses as possible to a small cache. The Cheetah simulations discussed above indicate that varying the cache size will not have a dramatic effect on performance, so we choose the smallest cache size studied—2 KB—and route the 256 most accessed lines to that cache when splitting the heap. This approach should give us a significant energy reduction without compromising performance. We statically determine which data to route based on a profiling run of the application; in practice, the compiler would perform this task. Our approach assumes the existence of separate heap allocation functions to allocate objects to different regions of memory. Frequently accessed heap structures reside in their own heap, allowing us to maintain the bounds-checking mechanism used in region-based caches to route data to the appropriate cache. The ARM architecture, which contains two regions for heap data, as shown in Figure 1, is particularly well suited to this approach. Other architectures must set aside a portion of the existing heap for highly local data, perhaps reserving the upper addresses of the region for this use.

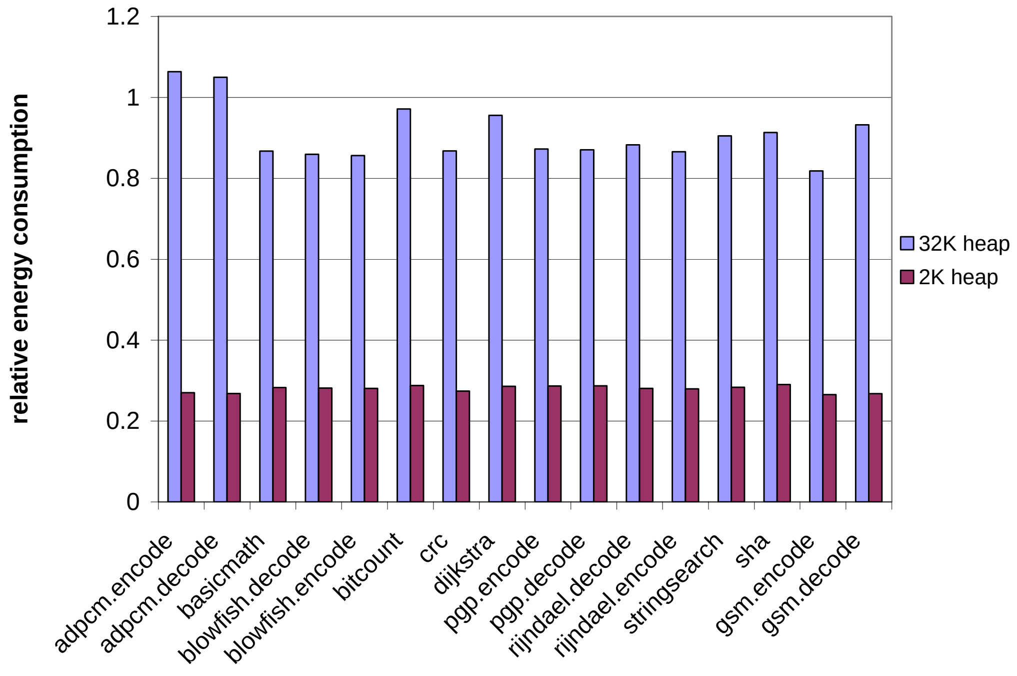
We use a simple heuristic in this work to show the potential effectiveness of our caching strategies. A more refined method that effectively incorporates miss rate estimates as well as footprint size and access percentages would likely yield better results. However, determining an appropriate heuristic is difficult. We explored more complex metrics for determining which data to route with little success. Our initial attempt at heap partitioning used access intervals for each cache line; if a majority of accesses to a line occurred within a small interval, we considered the line "hot" and routed it to the small cache [33]. This method generated prohibitively large profiles, and the heuristic often chose hot data too aggressively, thus dramatically increasing conflicts and reducing performance. Determining the appropriate capacity for the hot heap cache is problematic, however, because that capacity is application-dependent. Determining the right amount of data to reroute requires explicitly simulating different input data sets for the hot heap cache, as we did in the Cheetah simulations shown earlier. This problem clearly requires further investigation.

Although we only consider static data mapping in this dissertation, dynamically mapping lines to the hot heap cache might also yield further benefits. Dynamic mapping would allow us to further customize the caching strategy and exploit the varying behavior of different program phases. The downside to this approach would be the complexity of implementation and the hardware overhead required.

5.1.3 Experiments

Figure 17 and Figure 18 show simulation results for region-based caches using three different heap cache configurations: a large (32 KB) unified heap cache, a small (2 KB) unified heap cache, and a split heap cache using both the large and small caches. We present normalized energy and performance numbers, using a single 32 KB direct-mapped L1 data cache as the baseline. Because all region-based caches are direct-mapped to minimize energy consumption, we use a direct-mapped baseline to ensure a fair comparison. We consider the most effective configuration to be the cache organization with the lowest energy-delay product ratio [38].

For applications with a heap footprint under 1000 lines, the split cache is unnecessary. Figure 17 shows the results from these applications. Figure 18, which shows applications with large heap footprints, adds the energy and performance numbers for the split cache. As expected, using the small heap cache and disabling the large offers the



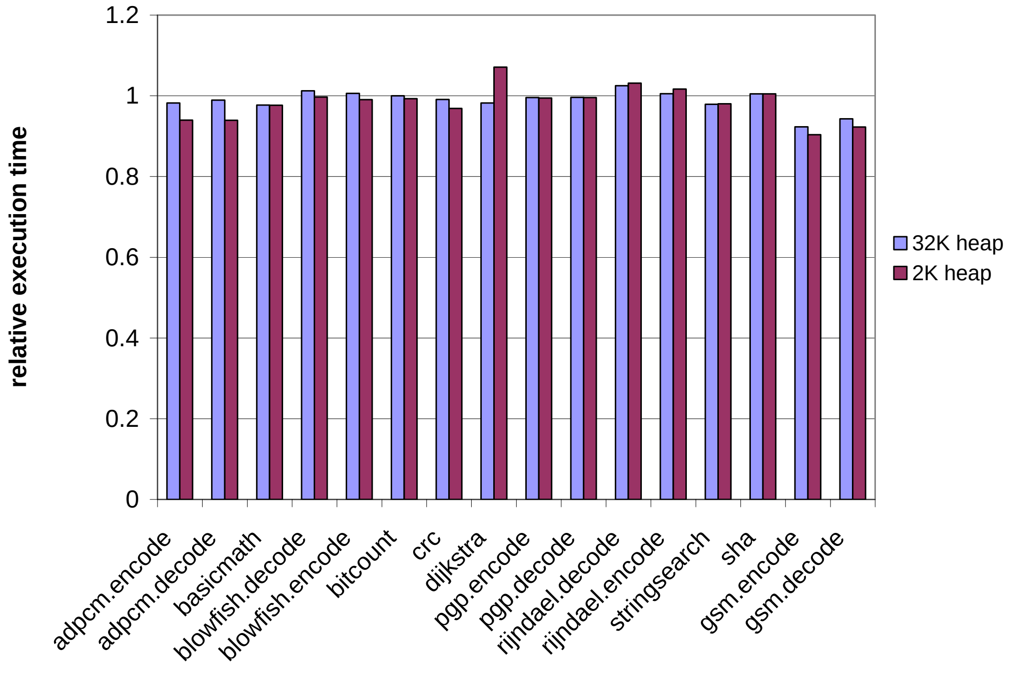
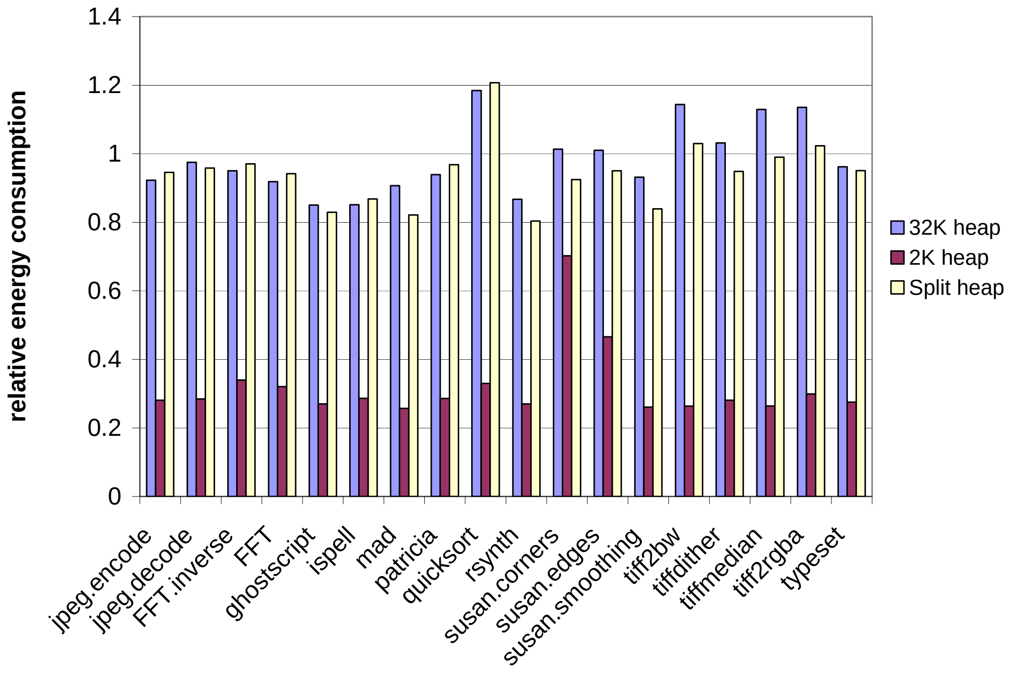


Figure 17: Energy (top graph) and performance (bottom graph) results for MiBench applications with small heap footprints (less than 1000 unique addresses) using region-based caches with large and small unified heap caches. The baseline is a 32 KB direct-mapped unified L1 data cache. Speedups for the large heap cache are due to reduced conflicts between regions



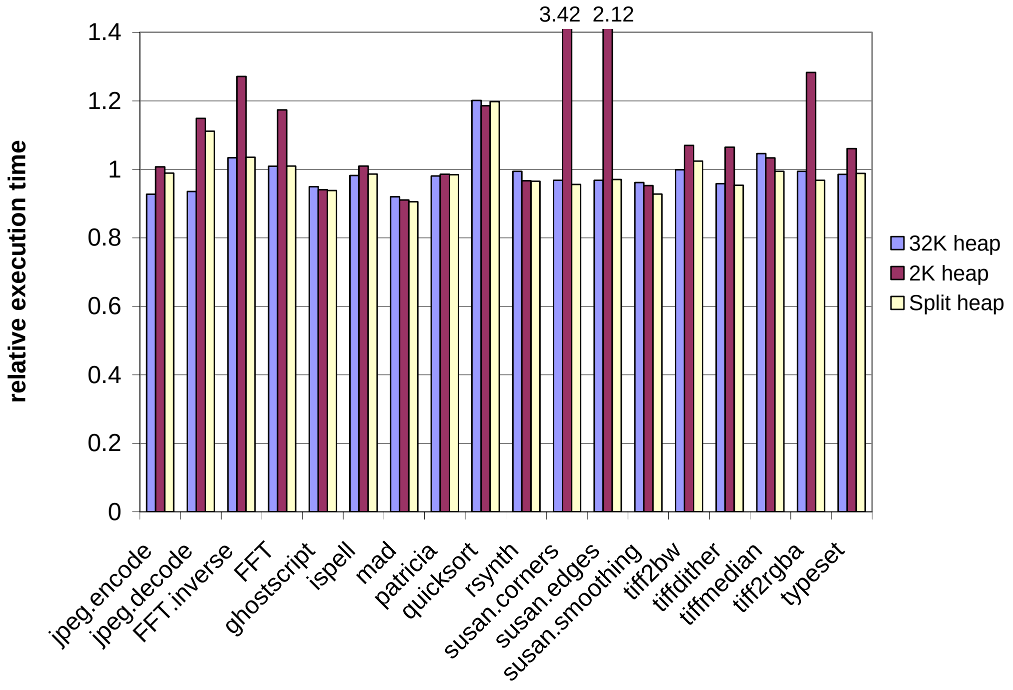
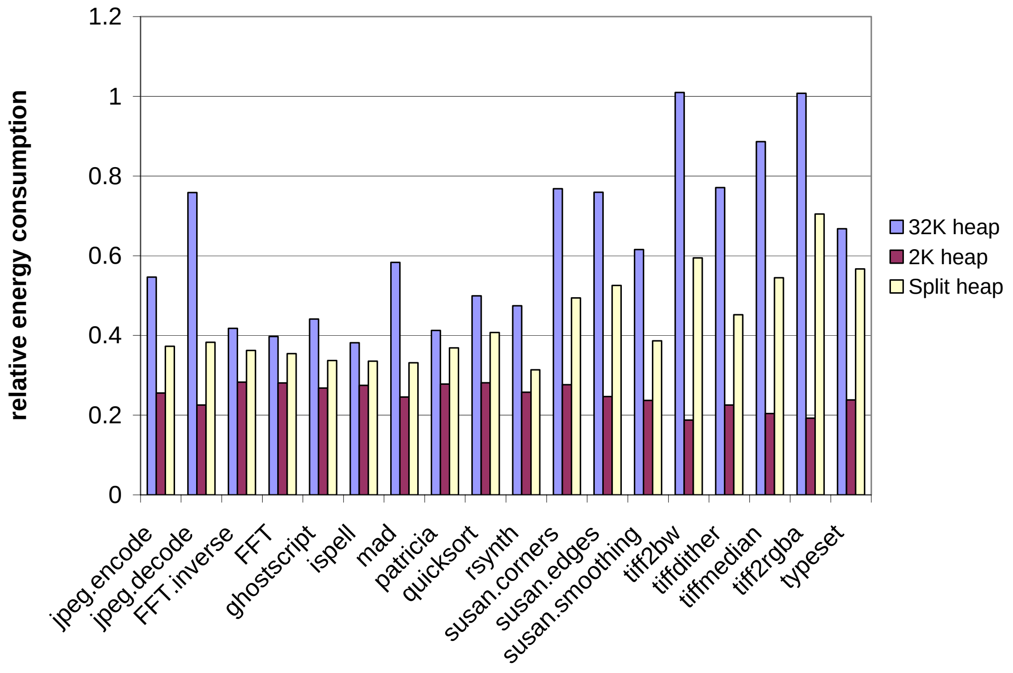


Figure 18: Energy (top graph) and performance (bottom graph) results for MiBench applications with large heap footprints (greater than 1000 unique addresses) using three different heap cache configurations: a large unified heap cache, a small unified heap cache, and a split heap cache employing both large and small caches. The baseline is a 32 KB direct-mapped unified L1 data cache. Speedups for the large heap cache are due to reduced conflicts between regions

best energy savings across the board. Most applications consume over 70% less energy in this case; however, some applications suffer significant performance losses, most notably susan.corners and susan.edges. 20 of the 34 applications in the MiBench suite experience performance losses of less than 1% or slight speedups, including ghostscript, mad, patricia, rsynth, and susan.smoothing—all applications with large heap footprints. This result suggests that heap data in these applications have good locality characteristics and are frequently accessed while present in the cache. Another application, quicksort, suffers significant performance losses for all configurations due to an increased number of global misses, and therefore still benefits most from using the small heap cache. In all of these cases, we gain substantial energy savings with virtually no performance loss, reducing overall energy consumption by up to 86%. Several applications actually experience small speedups, a result of reduced conflict between regions and the lower hit latency for the smaller cache.

For applications that suffer substantial performance losses with the small cache alone, the split heap cache offers a higher-performance alternative that still saves energy. The most dramatic improvements can be seen in susan.corners and susan.edges. With the large heap cache disabled, these two applications run more than twice as slow; with a split heap cache, they experience small speedups. Other applications, such as FFT and tiff2rgba, run close to 30% slower with the small cache and appear to be candidates for a split heap cache. However, the energy required to keep the large cache active overwhelms the performance benefit of a split heap, increasing the energy-delay product.

Figure 19 shows simulation results for drowsy heap caching configurations. In all cases, we use the ideal drowsy intervals derived in [33]—for the unified heap caches, 512



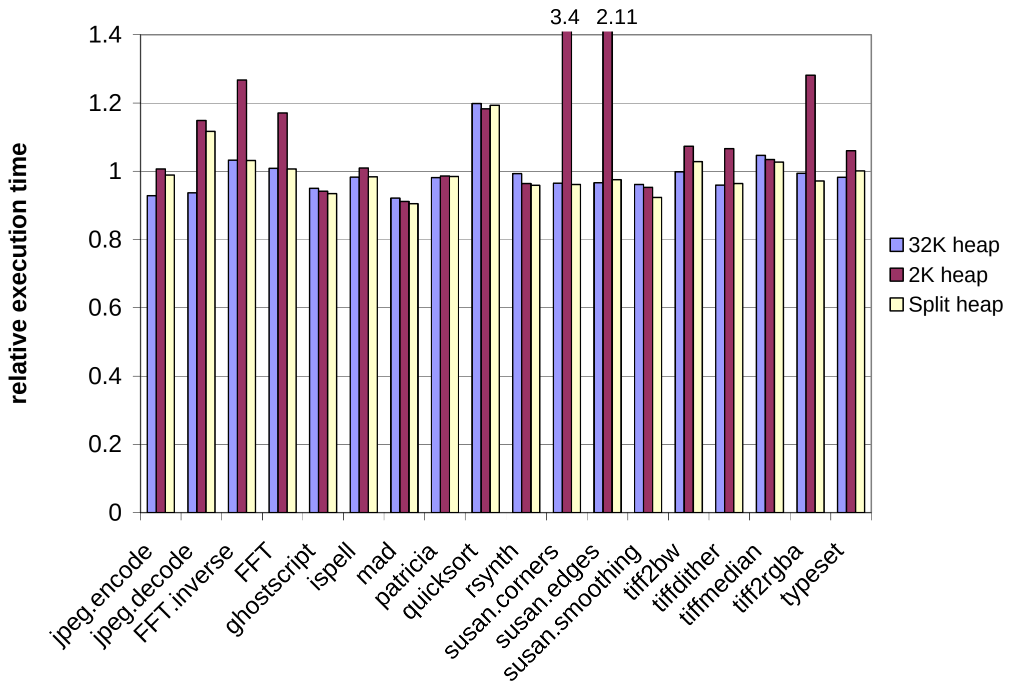
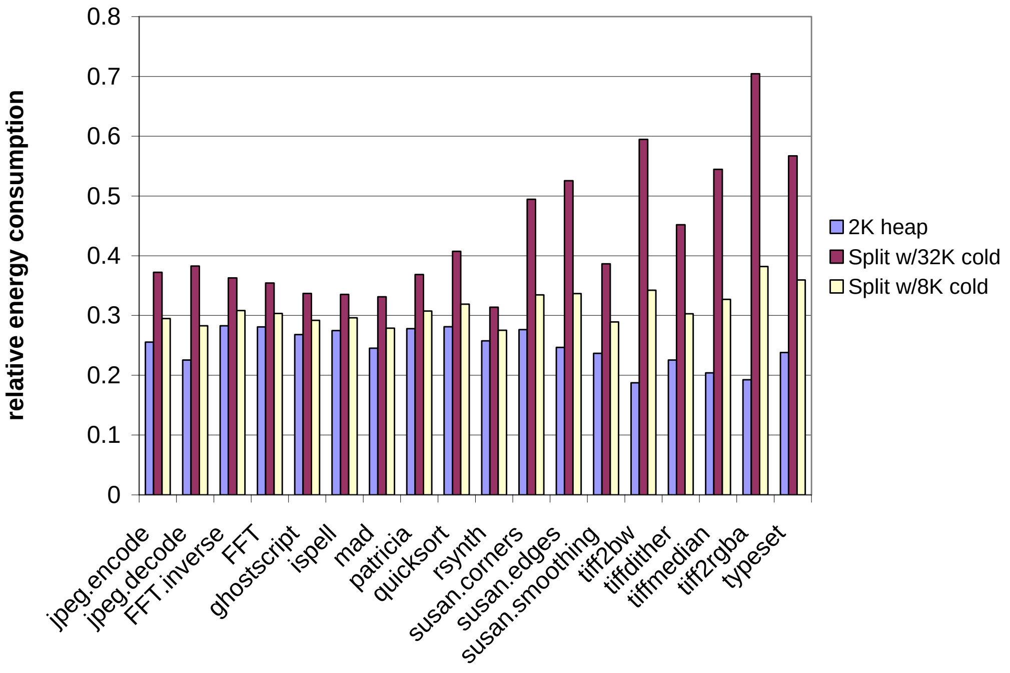


Figure 19: Energy (top graph) and performance (bottom graph) results for MiBench applications with large heap footprints (greater than 1000 unique addresses) using three different heap cache configurations: a large unified heap cache, a small unified heap cache, and a split heap cache employing both large and small caches. The baseline is a 32 KB direct-mapped unified L1 data cache with a 512-cycle drowsy interval; all region caches use ideal drowsy intervals derived in [33]

cycles; for the split heap cache, 512 cycles for the hot heap cache and 1 cycle for the cold heap cache. The stack and global caches use 512 and 256 cycle windows, respectively. We assume a 1 cycle latency for transitions to and from drowsy mode. Note that drowsy caching alone significantly reduces energy for these benchmarks [33].

Although all caches benefit from the static energy reduction offered by drowsy caching, this technique has the most profound effect on the split heap caches. Since the applications with small heap footprints do not require a split cache, the figure only shows the larger benchmarks. Drowsy caching all but eliminates the leakage energy of the large heap cache, as it contains rarely accessed data with low locality and is therefore usually inactive. Since the small cache experiences fewer conflicts in the split heap scheme than by itself, its lines are also less active and therefore more conducive to drowsy caching. Both techniques are very effective at reducing the energy consumption of these benchmarks. Drowsy split heap caches save up to 69% of the total energy, while the small caches alone save between 72% and 81%. Because drowsy caching has a minimal performance cost, the runtime numbers are similar to those shown in the previous figure. The small cache alone and the split heap cache produce comparable energy-delay values for several applications; ispell is one example. In these cases, performance-conscious users can employ a split heap cache, while users desiring lower energy consumption can choose the small unified heap cache.

Shrinking the large heap cache further alleviates its effect on energy consumption. The data remaining in that cache is infrequently accessed and can therefore tolerate an increased number of conflicts. Figure 20 shows simulation results for two different split heap configurations—one using a 32 KB cache for cold heap data, the other using an



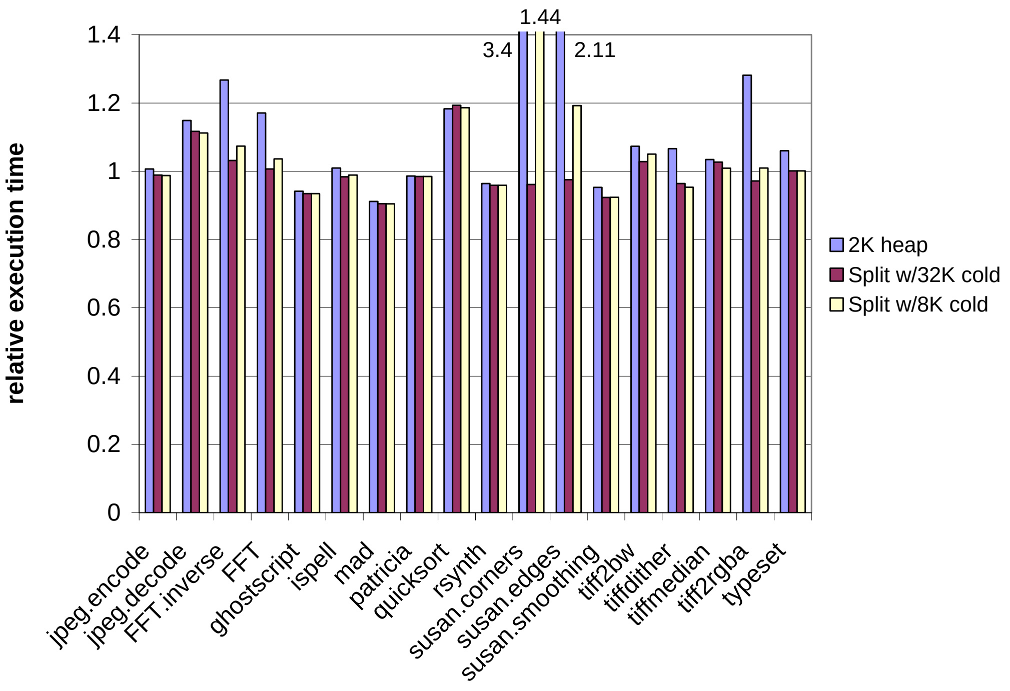


Figure 20: Energy (top graph) and performance (bottom graph) results for MiBench applications with large heap footprints (greater than 1000 unique addresses) using three different heap cache configurations: a small unified heap cache, and split heap caches using either a 32 KB cache or an 8 KB cache for low-locality heap data. The baseline is a 32 KB direct-mapped unified L1 data cache with a 512-cycle drowsy interval; all region caches use ideal drowsy intervals derived in [33]

8 KB cache—as well as the 2 KB unified heap cache. All caches are drowsy. The unified cache is still most efficient for the majority of applications, but shrinking the cold heap cache narrows the gap between unified and split heap configurations. Applications such as susan.corners and tiff2rgba, which contain a number of accesses to the cold heap cache, see the greatest benefit from this modification, with tiff2rgba consuming 32% less energy with the smaller cold heap cache. Overall, these applications save between 62% and 73% of the total energy.

5.2 Instruction-Centric Heap Caching

To this point, we have focused on analyzing heap data to determine how best to cache them. When only a subset of the data displays good locality, we use access frequency to identify hot data to store in a smaller cache. We now approach the same problem from a different angle—rather than looking at the locality characteristics of a particular line, we examine the references themselves. One advantage is that an instruction-based profile is often virtually independent of the program input. Although the data may affect how often a particular instruction executes, most programs follow the same general execution and therefore display the same relative behavior. Choosing hot data through their referencing instructions exploits locality in a different manner. Regularly accessed cache lines have high temporal locality. We cannot necessarily say the same about the targets of frequently executed memory instructions, as each instruction can access many addresses. However, this method effectively leverages spatial locality, as a single load often accesses sequential locations. Tight inner loops of program kernels display this behavior when accessing arrays or streams.

Benchmark	# memory instructions	% memory instructions needed to cover given percentage of heap cache accesses				
		50%	75%	90%	95%	99%
adpcm.encode	171	1.2%	1.8%	1.8%	1.8%	1.8%
adpcm.decode	173	1.2%	1.7%	1.7%	1.7%	1.7%
basicmath	373	1.1%	4.8%	8.6%	10.5%	18.2%
blowfish.decode	325	1.2%	2.2%	2.5%	2.8%	2.8%
blowfish.encode	325	1.2%	2.2%	2.5%	2.8%	2.8%
bitcount	244	0.4%	0.8%	1.2%	1.6%	1.6%
jpeg.encode	1406	1.1%	3.0%	6.5%	8.9%	15.3%
CRC32	329	0.9%	1.2%	1.5%	1.5%	1.5%
dijkstra	383	0.8%	1.0%	1.3%	5.7%	14.4%
jpeg.decode	1192	1.2%	2.6%	4.9%	6.8%	11.8%
FFT	329	5.2%	11.2%	17.0%	20.4%	24.3%
FFT.inverse	327	4.9%	11.3%	17.7%	21.4%	25.1%
ghostscript	7501	0.2%	0.3%	1.5%	4.1%	13.5%
ispell	649	2.3%	4.2%	7.1%	10.8%	18.3%
mad	1043	2.9%	4.5%	7.2%	11.2%	15.0%
patricia	420	3.3%	10.7%	20.5%	23.8%	26.7%
pgp.encode	1119	0.4%	0.5%	2.1%	4.8%	22.5%
pgp.decode	1022	0.4%	0.6%	1.2%	2.7%	17.7%
quicksort	337	2.4%	5.9%	10.4%	12.5%	14.8%
rijndael.decode	540	13.7%	22.2%	27.4%	29.3%	31.1%
rijndael.encode	617	11.2%	18.3%	22.5%	24.1%	25.3%
rsynth	889	2.2%	4.2%	5.5%	7.6%	15.0%
stringsearch	210	1.9%	7.1%	11.9%	15.7%	21.9%
sha	276	1.1%	1.4%	1.8%	1.8%	8.0%
susan.corners	691	4.2%	8.0%	15.6%	18.5%	21.1%
susan.edges	878	6.8%	13.4%	21.5%	25.3%	28.6%
susan.smoothing	517	0.4%	0.6%	0.6%	0.6%	0.6%
tiff2bw	1036	0.4%	0.7%	1.1%	1.4%	1.8%
tiffdither	1314	0.6%	0.9%	2.8%	4.3%	6.8%
tiffmedian	1359	0.7%	1.0%	1.5%	1.8%	3.0%
tiff2rgba	1154	0.8%	1.6%	2.6%	2.9%	3.1%
gsm.encode	736	3.0%	5.3%	7.1%	8.2%	13.5%
typeset	17235	0.6%	1.9%	3.8%	6.7%	16.2%
gsm.decode	555	0.9%	1.4%	2.9%	3.4%	7.6%
	AVERAGE	2.4%	4.7%	7.2%	9.0%	13.3%

Table 9: Number of memory instructions that reference the heap required to cover different fractions of accesses to the heap cache in MiBench applications. As with the data itself, a small number of loads and stores account for the majority of heap cache accesses

5.2.1 Heap Access Characteristics

In Table 5, we showed that a small number of blocks are responsible for the majority of heap accesses. This trend is even more apparent for memory instructions, as shown in Table 9. Just 2.4% of the loads and stores to the heap cache cover 50% of the accesses—a similar figure to the 2.1% of heap addresses required to cover the same percentage of accesses. The numbers do not increase greatly as we look at different coverage points, with approximately 13% of the memory instructions accounting for 99% of the heap references. These results reflect the oft-quoted maxim that programs spend 90% of their time in 10% of the code. Note that the number of instructions accessing the heap cache remains fairly consistent across applications, unlike the size of the heap data footprint. Our studies show that a small percentage of loads and stores access multiple regions.

The data suggest that we can treat heap references in a similar manner to heap data when determining how to cache this region. Because a small number of instructions account for most accesses, we can move their targets to a smaller cache, maintaining a larger cache for the remaining references. Note that only identifying the most frequently executed memory instructions will not sufficiently capture the appropriate accesses. Other memory references that share the same targets must also access the hot heap cache. Choosing appropriate instructions involves an iterative routine that ceases when the set of target addresses overlaps with no remaining references. In practice, we use the method discussed in Section 5.1 to route data to a separate structure once instruction targets are identified; those targets use a separate heap allocator and are placed in their own region.

5.2.2 Experiments

Figure 21 shows some preliminary results from this approach. As in Figure 17 and Figure 18, we compare three non-drowsy cache configurations: a large (32 KB) unified heap cache, a small (2 KB) unified heap cache, and a split cache employing both large and small caches. We use the 128 most executed load instructions as a starting point for routing data between the caches. The figure shows a subset of the MiBench applications,

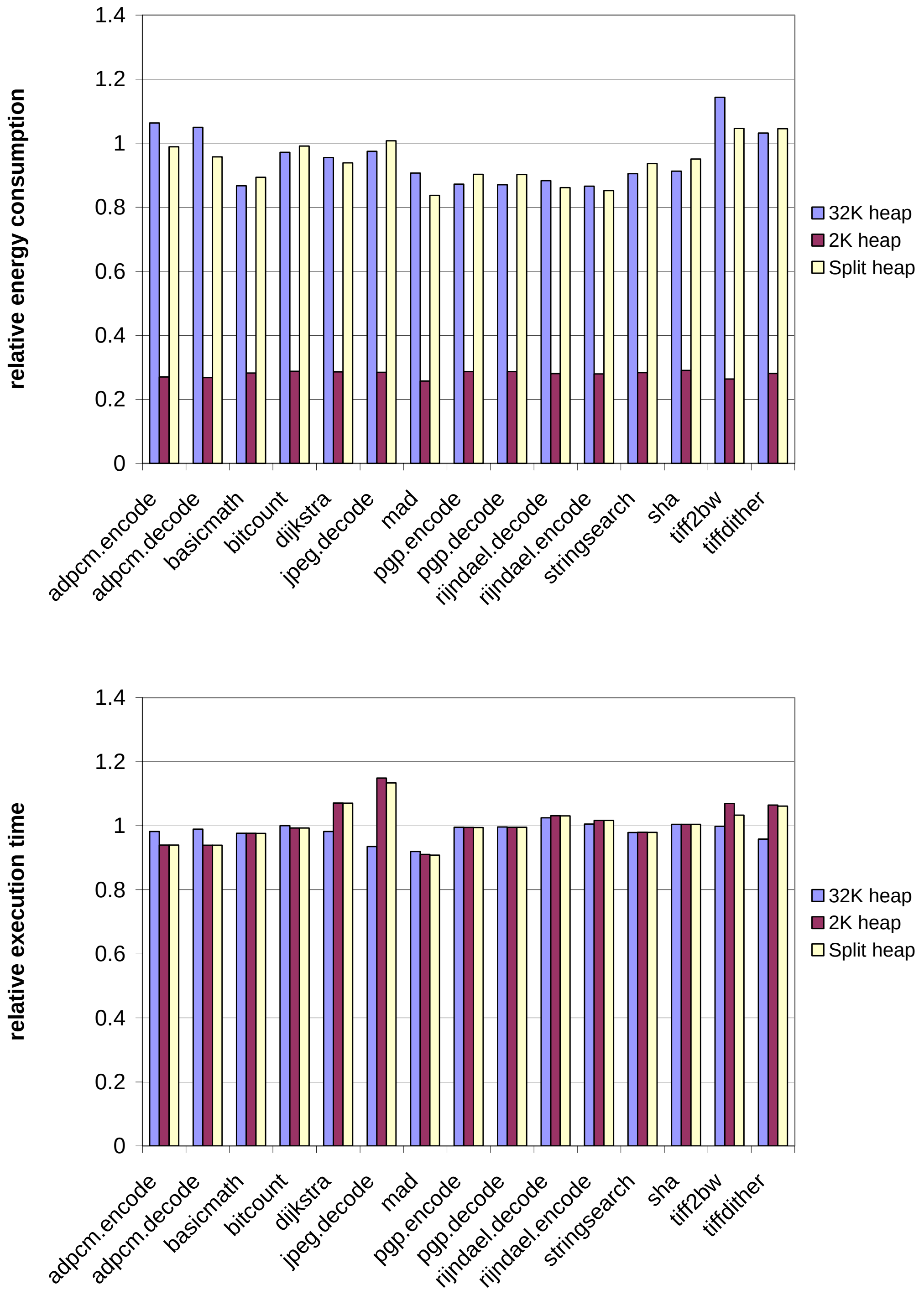
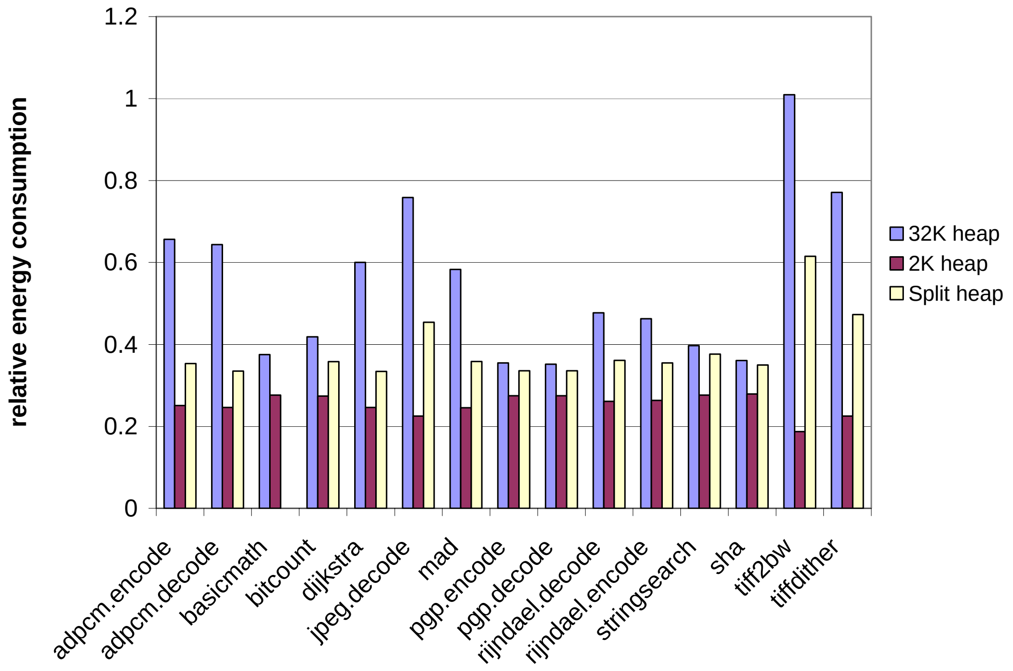


Figure 21: Energy (top graph) and performance (bottom graph) for a subset of MiBench applications using different non-drowsy heap cache configurations. The baseline is a 32 KB direct-mapped unified L1 data cache. The hardware configurations are the same as in Figure 17 and Figure 18, but in the split heap cache, data are routed to the hot heap cache based on the frequency of the accessing instructions, not references to specific blocks



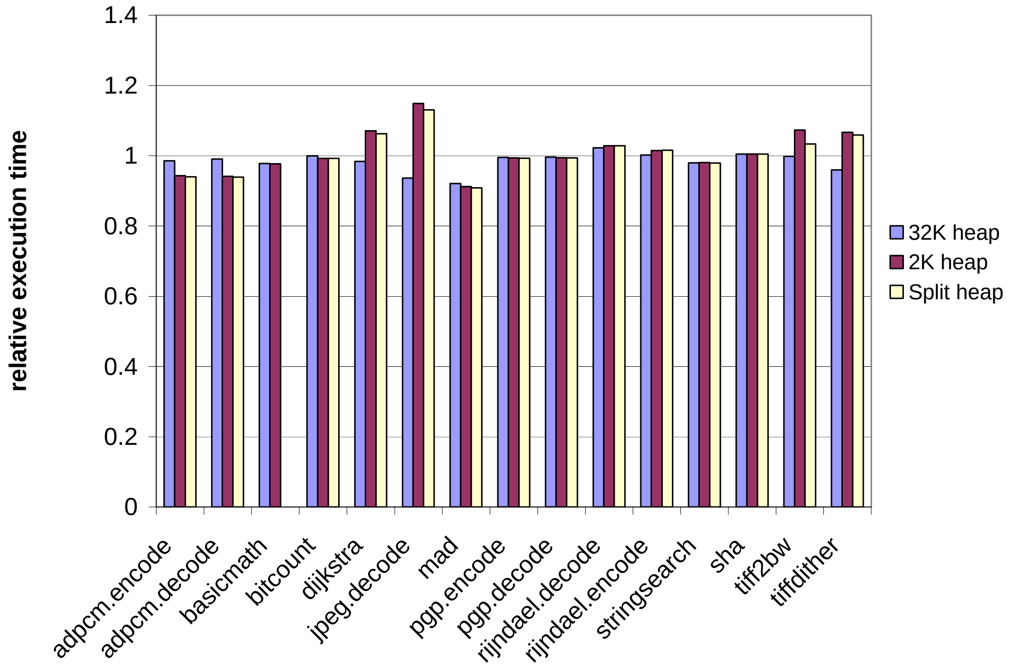


Figure 22: Energy (top graph) and performance (bottom graph) results for a subset of MiBench applications using different drowsy heap cache configurations. The baseline is a 32 KB direct-mapped unified L1 data cache with a 512-cycle drowsy interval. The hardware configurations are the same as in Figure 19, but in the split heap cache, data are routed to the hot heap cache based on the frequency of the accessing instructions, not references to specific blocks

as the space requirements for the memory instruction profiles currently prevents the execution of some of the larger applications. Therefore, the small unified heap cache unsurprisingly represents the ideal design point for the benchmarks shown. For the most part, the results for the split heap configuration are similar to those shown in Figure 17 and Figure 18, with energy savings ranging between 1% and 16%.

In Figure 22, we evaluate drowsy heap cache configurations using the same routing methodology. As with the data shown in Figure 19, the addition of drowsy caching significantly improves the energy consumption of the split cache, leading to comparable results for the small cache and split cache in several cases. Energy savings range from 48% to 67% in the split heap caches for the applications shown.

5.3 Summary

In this chapter, we evaluated a new multilateral cache organization designed to tailor cache resources to the individual reference characteristics of an application. To ensure that all applications perform well, we maintain two heap caches: a small, low-energy cache for frequently accessed heap data, and a larger structure for low-locality data. In most applications, the heap footprint is small and the data possesses good locality characteristics, allowing us to save energy by disabling the larger cache and routing data to the smaller cache. Those applications that do have a large heap footprint can use both heap caches, routing a frequently-accessed subset of the data to the smaller structure. Adding drowsy caching to our split heap cache eliminates most of the static energy dissipation and provides even greater savings.

CHAPTER 6

PREFETCHING WITH REGION-BASED CACHES

Thus far, our focus has been the reduction of energy consumption through intelligent cache partitioning. Partitioning the cache can also improve schemes that target memory system performance, such as prefetching. Data prefetching reduces cache miss effects by anticipating data access patterns and fetching data prior to its use. However, aggressive hardware prefetching methods may over-speculate, caching unnecessary data. In this chapter, we explore prefetching in region-based caches and show how isolating data with predictable access patterns can improve prefetch algorithms.

In Section 6.1, we discuss methods for assessing prefetch effectiveness, focusing on a taxonomy that classifies prefetches into several distinct categories to quantify their impact on cache misses and bus traffic. We then use the taxonomy to evaluate four prefetch algorithms on MiBench applications. In Section 6.2, we use this information to form a new region-based cache organization for prefetching. We then present our experimental results in Section 6.3. Section 6.4 summarizes the chapter.

6.1 Evaluating Prefetch Effectiveness

To be effective, prefetches must be timely and accurate, and have good coverage. However, these goals are often at odds with one another. Data prefetched long before it is accessed can evict active blocks from the cache, creating additional misses. Aggressive, highly speculative prefetch methods can also cache unnecessary data. However, failure to prefetch early enough diminishes prefetch usefulness and adds overhead to the program execution. To improve data prefetching mechanisms, we must determine how well they meet the goals outlined above, what data they prefetch well, and where their inefficiencies lie.

6.1.1 Metrics for Prefetch Effectiveness

A number of metrics exist for measuring the effectiveness of prefetch algorithms. Commonly used metrics include statistics such as misses, traffic, and cycles per instruction (CPI) that assess the effect of prefetching on the entire system. Gross performance measures do give a broad sense of a prefetcher's overall impact, but they provide little insight into whether that algorithm can be improved.

Since prefetching tries to eliminate stall cycles due to cache misses, many papers use metrics that compare prefetch numbers to miss totals. To calculate these figures, prefetches are classified as "good" or "bad." A good prefetch fetches data that is referenced before it is evicted from the cache; a bad prefetch does not. Consider a cache that generates M misses without prefetching and a prefetch algorithm that produces G good prefetches and B bad prefetches in that same cache. The **coverage** of the prefetch algorithm is (G / M)—the ratio of good prefetches to misses. The **accuracy** of the prefetch algorithm is G / (G + B)—the fraction of all prefetches classified as good. These metrics can identify prefetch algorithms that over-speculate. Because aggressive prefetchers base their predictions on reference stream history or simple heuristics, they may choose addresses that eliminate no misses.

These metrics rely on the flawed assumption that every good prefetch replaces a miss. In fact, a prefetch that eliminates a cache miss for its target address may cause a miss for the block the prefetch replaced, leaving the total number of misses unchanged. Such prefetches also increase traffic between cache levels and may even generate additional prefetches, some of which might be classified as "good." The number of good prefetches can therefore exceed the baseline miss count, resulting in a nonsensical coverage value greater than one. We therefore see that ineffective prefetch algorithms can still have good coverage and accuracy.

Srinivasan et al. address these issues with the Prefetch Traffic and Miss Taxonomy (PTMT) [100]. PTMT is an event-driven system that evaluates prefetch algorithms by tracking accesses and replacements for each prefetched block and the block it replaces. By simultaneously simulating two identical caches, one of which uses prefetching (**pf-cache**) and one that does not (**conv-cache**), PTMT can quantify the effects of individual prefetches on cache misses and traffic. The taxonomy identifies ten separate prefetch cases, as shown in Table 10. Those cases are then broadly classified into four categories that replace the simplistic "good/bad" characterization: useful, useless, polluting, and side-effect. Table 11 shows each category and the cases it covers. A **useful prefetch** replaces a miss without increasing traffic. The replaced block is either evicted from the **conv-cache** or prefetched into the **pf-cache** before its next access. **Useless prefetches** do not change the number of misses and increase traffic by one line per prefetch. Some useless prefetches fetch data that remained available in the **conv-cache—**blocks that were evicted from the **pf-cache** by other prefetches. In these cases, no misses can be saved. Other useless prefetches trade a miss to the prefetched block for a miss to the replaced

Case	*pf-cache* outcomes		*conv-cache* outcomes		Extra	
	x (prefetched)	*y (replaced)*	*x (prefetched)*	*y (replaced)*	Traffic	Misses
1	hit	miss	hit	hit	2	1
2	hit	prefetched	hit	hit	1	0
3	hit	don't care	hit	replaced	1	0
4	hit	miss	miss	hit	1	0
5	hit	prefetched	miss	hit	0	-1
6	hit	don't care	miss	replaced	0	-1
7	replaced	miss	don't care	hit	2	1
8	replaced	prefetched	don't care	hit	1	0
9	replaced	don't care	don't care	replaced	1	0

Table 10: Prefetch cases for PTMT classification [100]. PTMT relies on simultaneous simulation of two caches—one with prefetching (**pf-cache**), one without (**conv-cache**)—to determine if each prefetch improves or degrades miss and traffic numbers. The taxonomy classifies prefetches based on the outcome of the next reference to each prefetched and replaced block. The table does not show the 10th PTMT case, side-effect prefetches, because such prefetches only occur in LRU set-associative caches

Category	Cases	Extra Traffic	Extra Misses
Useful	5, 6	0	-1
Useless	2, 3, 4, 8, 9	1	0
Polluting	1, 7	2	1
Side-Effect	10	1	1

Table 11: Prefetch categories that encompass each of the 10 PTMT cases [100]. Useful prefetches replace misses without increasing traffic. Useless prefetches have no effect on the overall miss count and increase traffic by one line per prefetch. Polluting prefetches increase both cache misses and memory traffic. Side-effect prefetches occur only in LRU set-associative caches; these prefetches cause unexpected evictions by reordering the LRU stack

block, thus leaving the miss rate unchanged. The last type of useless prefetch prefetches data that are never referenced before their eviction. **Polluting prefetches** increase both cache misses and traffic, causing an extra miss for both the block they evict and the block they prefetch. The last PTMT category, **side-effect prefetches**, only occur in set-associative caches with LRU replacement. Because prefetched blocks move to the MRU way of their set, they may re-order the LRU stack and evict blocks that would otherwise remain in the cache. Each side-effect prefetch causes an extra miss and an extra line of traffic for the improperly evicted block. Note that the first six prefetch cases qualify as "good" prefetches when calculating coverage and accuracy. However, only two of the six cases are useful prefetches.

PTMT has some limitations. Blocks that are not referenced or evicted before the program completes are not classified. Experiments run on several SPEC CPU2000 benchmarks indicate that unclassified prefetches can account for over 50% of all prefetches [12]. Also, some prefetches—particularly those that occur early in a program run—fill empty, invalid lines and therefore cause no eviction. We can classify these prefetches by determining the correct outcome for an invalid or dead block as follows:

- If the prefetched data is the first valid data in a given block, treat the "replaced block" as if it was replaced in the **conv-cache**, leading to a "don't care" outcome in the **pf-cache**. This prefetch is therefore a case 3, 6, or 9 prefetch.
- If the prefetched block is never referenced or replaced, treat it as if it were replaced in the **pf-cache**, leading to a "don't care" outcome in the **conv-cache**. This prefetch is therefore a case 7, 8, or 9 prefetch.
- If the replaced block is never referenced or replaced, treat it as if it were replaced in the **conv-cache**, leading to a "don't care" outcome in the **pf-cache**. This prefetch is therefore a case 3, 6, or 9 prefetch.

PTMT also does not directly address prefetch timeliness. An access to a prefetched block that is still in-flight is treated as a hit; additional state is necessary to determine if that hit is delayed or not. Blocks that are prefetched too early are replaced in the **pf-cache** prior to their next access, making the corresponding prefetch a case 7, 8, or 9 prefetch.

We do not explicitly consider the effects of prefetch chains within PTMT. Three prefetch cases—case 2, 5, and 8—list "prefetched" as the outcome for the replaced block in the **pf-cache**. Each of these prefetches is therefore chained to another prefetch, which may in turn chain to other prefetches. Srinivasan et al. [100] show that the regular expression (2,5,8)(2)*(1,3) describes all prefetch chains—each case 2, 5, or 8 prefetch starts a chain of at least two prefetches that ends with a case 1 or case 3 prefetch. Any additional prefetches in the middle are case 2 prefetches. Only prefetch chains starting with case 5 prefetches may be useful; all others contain useless or polluting prefetches. The usefulness of such chains depends on their overall length. As shown in the next section, none of our target applications contain any case 5 prefetches, so we can safely ignore chain effects.

6.1.2 Evaluation of Existing Prefetch Algorithms

In this section, we use PTMT to evaluate the effectiveness of several existing prefetch algorithms. We use the following four hardware prefetch mechanisms:

- **Next sequential prefetching (NSP)** [96]: Also known as one block lookahead, NSP relies on spatial locality and predicts that an access to block **x** will be followed by an access to block **x+1**. We prefetch only on a cache miss.
- **Tagged next sequential prefetching (tNSP)** [37]: tNSP is a variation of NSP that associates a tag bit with each block. The bit is initially zero and is set to one on any access; it is reset to zero when the block is evicted. An access that causes the tag to transition from zero to one—the first access to a block after a demand fetch or prefetch—generates a prefetch to the next sequential block.
- Stride prefetching using a **reference prediction table (RPT)** [21]: The RPT is designed to exploit regular accesses in program loops. Figure 23 shows its basic layout. The RPT is a direct-mapped, cache-like structure that is indexed by instruction address. Each entry contains the last referenced address

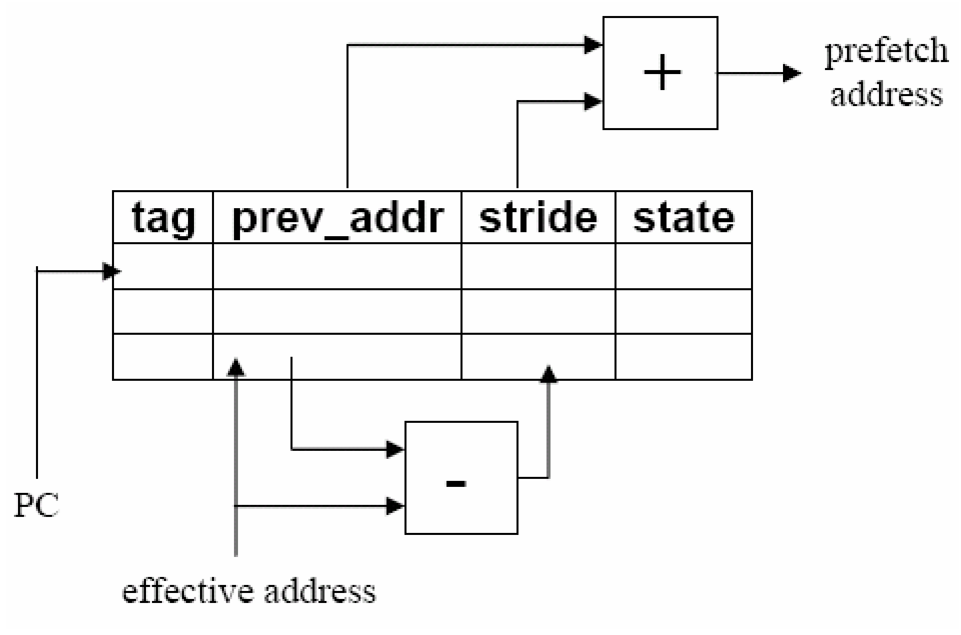


Figure 23: Layout of the reference prediction table (RPT) [21]. The RPT is a cache-like structure indexed by instruction address. For each memory instruction, the RPT tracks the last referenced address (**prev_addr**) and the difference between the last two addresses (**stride**). On a reference, the RPT adds the **prev_addr** and **stride** fields to generate an address prediction and also calculates a new stride. The **state** field tracks the number of consecutive successful predictions and determines if prefetches are issued

(**prev_addr**) and difference between the last two addresses (**stride**), as well as a two-bit state encoding of this instruction's past prefetch history. On each data reference, the RPT computes the actual stride between addresses and compares it to the **stride** field, updating that field on a mismatch. State transitions are based on that comparison; two consecutive mismatches place the predictor in a "no prediction" state. In any other state, the RPT prefetches from address (**prev_addr + stride**). Chen and Baer discuss three different RPT configurations: **basic**, **lookahead**, and **correlating**. The **lookahead** scheme uses an extra program counter (the **LA-PC**) to run ahead and generate prefetches to improve timeliness. A branch predictor facilitates this operation. The **correlated** RPT tracks previous branch history and maintains two

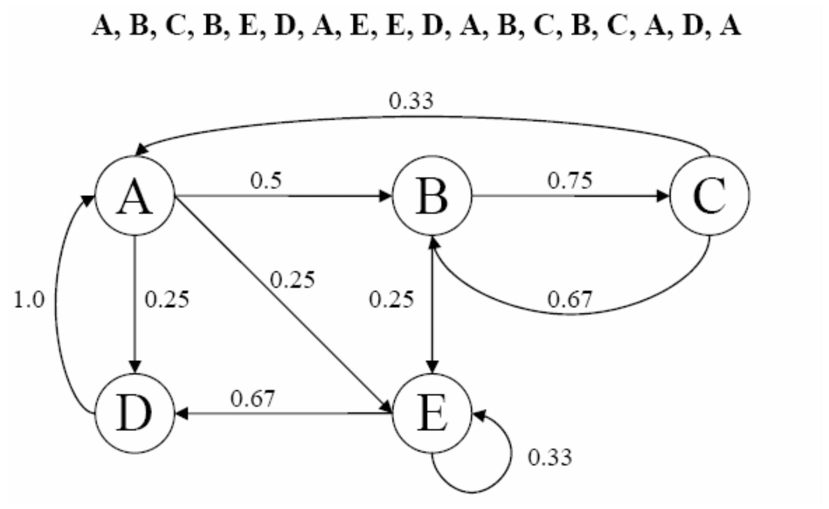


Figure 24: Sample miss address stream and associated Markov graph. Each edge represents the probability of the connected addresses appearing consecutively

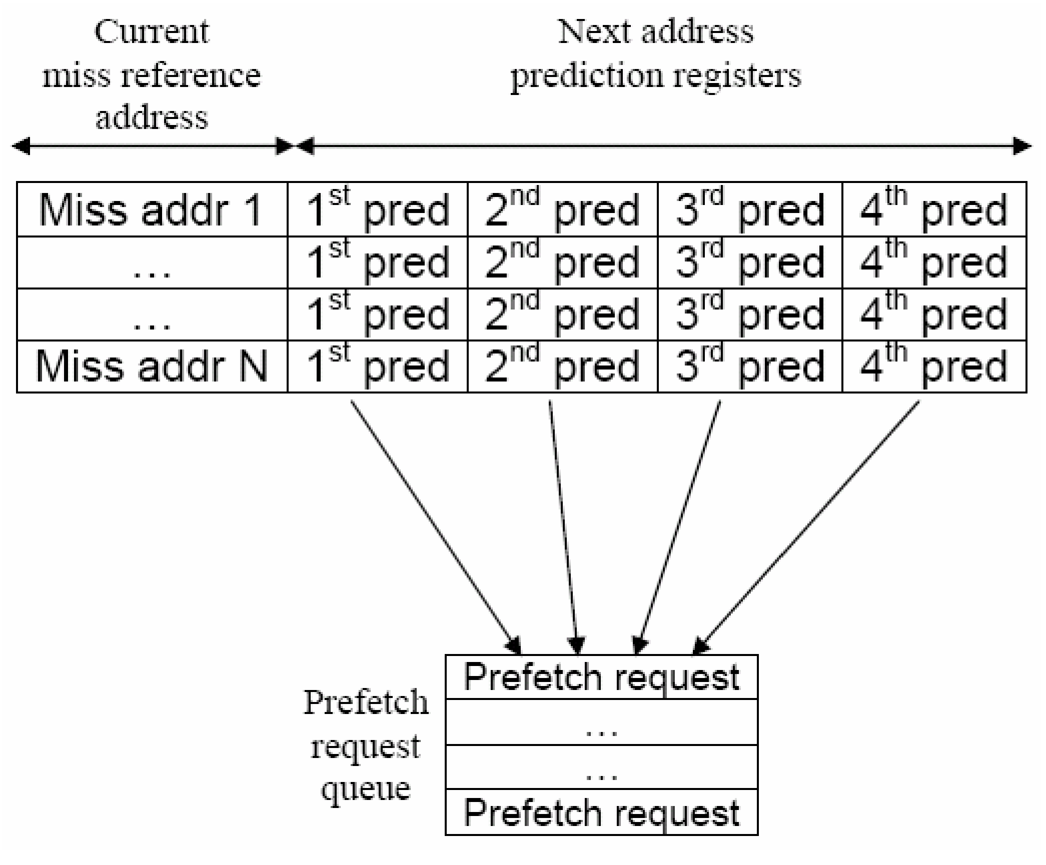


Figure 25: Table used to approximate a Markov graph in hardware using LRU replacement [54]. The MRU way of each set holds the address with the highest transition probability. On a miss, up to four prefetch predictions are issued to the prefetch request queue, with the most likely address given the highest priority. When the memory bus is free, the address at the head of the queue is fetched from the L2 cache

prev_addr and **stride** fields, ostensibly for inner and outer loops. We use the **basic** configuration in this work.

- **Markov prefetching** [54]: A Markov prefetcher uses the miss address stream to generate multiple prefetch predictions. A pure Markov model tracks the probability of two addresses appearing consecutively, as shown in Figure 24. The edges of the graph represent the transition probabilities from one miss address to the next; for example, the probability of a miss to block A being followed by a miss to block E is 0.25. Since maintaining a full Markov graph in hardware is inefficient, Joseph and Grunwald advocate approximating the Markov graph in a table with LRU replacement, with the MRU way of each set holding the address with highest transition probability. As shown in Figure 25, each set corresponds to a single miss address and contains up to four potential prefetch targets. If a miss address is present in the Markov table, up to four prefetch requests are submitted to the prefetch request queue, a prioritized list of potential prefetch targets. The MRU address generates the highest priority request, which can only be superceded by CPU demand fetches. When the memory bus is free, the address at the head of the queue is sent to the L2 cache.

We use a 512-entry RPT, as in Chen and Baer's work [21]; since the **prev_addr** and **stride** fields each require four bytes and each entry uses two state bits, this structure is roughly equivalent to a 4 KB direct-mapped cache. The Markov prefetcher uses a 1 MB table, as in Joseph and Grunwald's paper [54], that resides in the L2 cache. We prefetch directly to the L1 cache, which contradicts the results of several studies. Nesbit and Smith note that prefetching beyond the L2 cache has little benefit because prefetching at higher cache levels tends to increase cache pollution [78]. However, these benchmarks are less memory-intensive than applications used in previous prefetching studies. Also, our cache architecture minimizes cache pollution and allows us to benefit from prefetching at the highest level of the cache.

We begin with the results of our PTMT evaluation, shown in Figure 26 through Figure 29. For all simulations, we used a 32 KB direct-mapped cache with 32-byte blocks. The overhead of tracking prefetch information in some applications led to prohibitively long runtimes. The figures therefore show a subset of the MiBench applications. At this point, only half of the benchmarks run to completion with the RPT, so we only show applications that complete successfully.

The figures show the breakdown of PTMT cases for each prefetch algorithm. Overall, we see that the RPT prefetches best, averaging 78.3% useful prefetches. Because most embedded kernels contain loops that access data streams or arrays with regular strides, the RPT correctly predicts most of these accesses. NSP and tNSP also perform well with MiBench applications, containing 49% and 63.7% useful prefetches, respectively. The Markov prefetcher, which is best suited to irregular access patterns that are typically not found in embedded applications, is the least effective of the four, with only 31.9% useful prefetches. Judging by the raw prefetch numbers, it appears that the Markov prefetcher aggressively fetches too many blocks. The variance in prefetch effectiveness across application argues for an application-specific approach to prefetching.

In Figure 30 through Figure 33, we evaluate the same applications using region-based caches, with mixed results. Because the small caches increase conflict misses in the stack

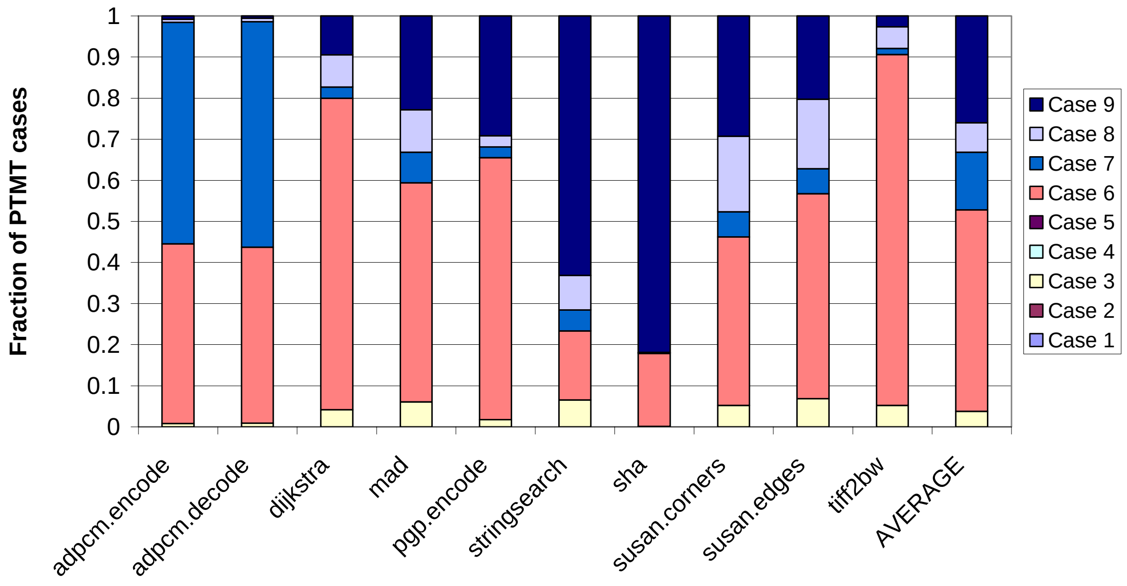


Figure 26: PTMT evaluation for next sequential prefetching (NSP) in a unified 32 KB L1 data cache. The figure shows the fraction of prefetches that fit into each PTMT case for a subset of MiBench applications. Applications in which NSP works effectively contain a higher percentage of case 5 and 6 prefetches

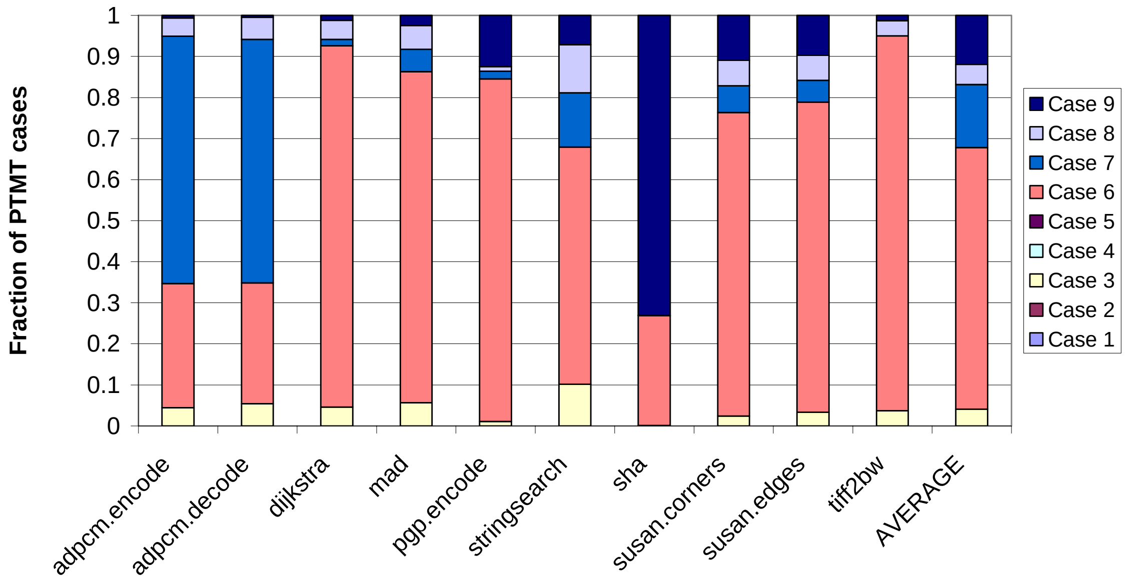


Figure 27: PTMT evaluation for tagged next sequential prefetching (tNSP) in a unified 32 KB L1 data cache. The figure shows the fraction of prefetches that fit into each PTMT case for a subset of MiBench applications. Applications in which tNSP works effectively contain a higher percentage of case 5 and 6 prefetches

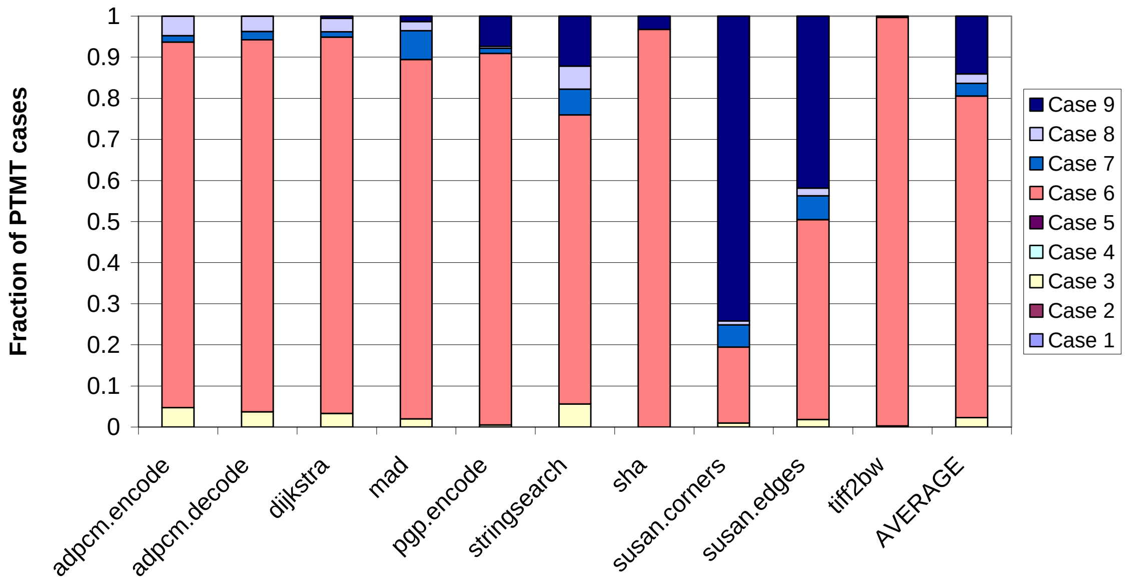


Figure 28: PTMT evaluation for prefetching using a reference prediction table (RPT) in a unified 32 KB L1 data cache. The figure shows the fraction of prefetches that fit into each PTMT case for a subset of MiBench applications. Applications in which the RPT works effectively contain a higher percentage of case 5 and 6 prefetches

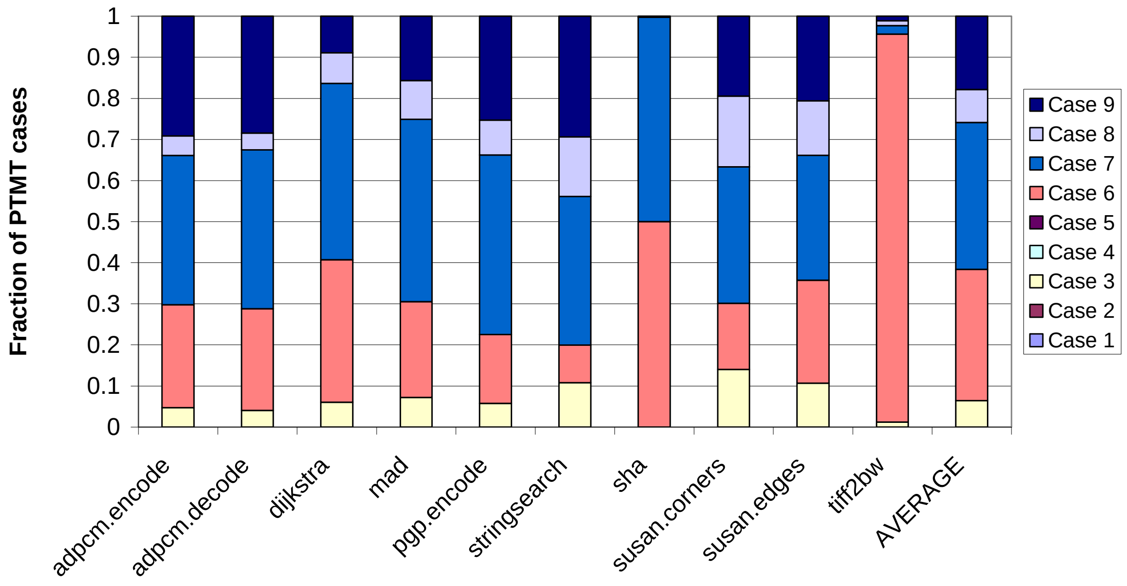


Figure 29: PTMT evaluation for Markov prefetching in a unified 32 KB L1 data cache. The figure shows the fraction of prefetches that fit into each PTMT case for a subset of MiBench applications. Applications in which the Markov prefetcher works effectively contain a higher percentage of case 5 and 6 prefetches

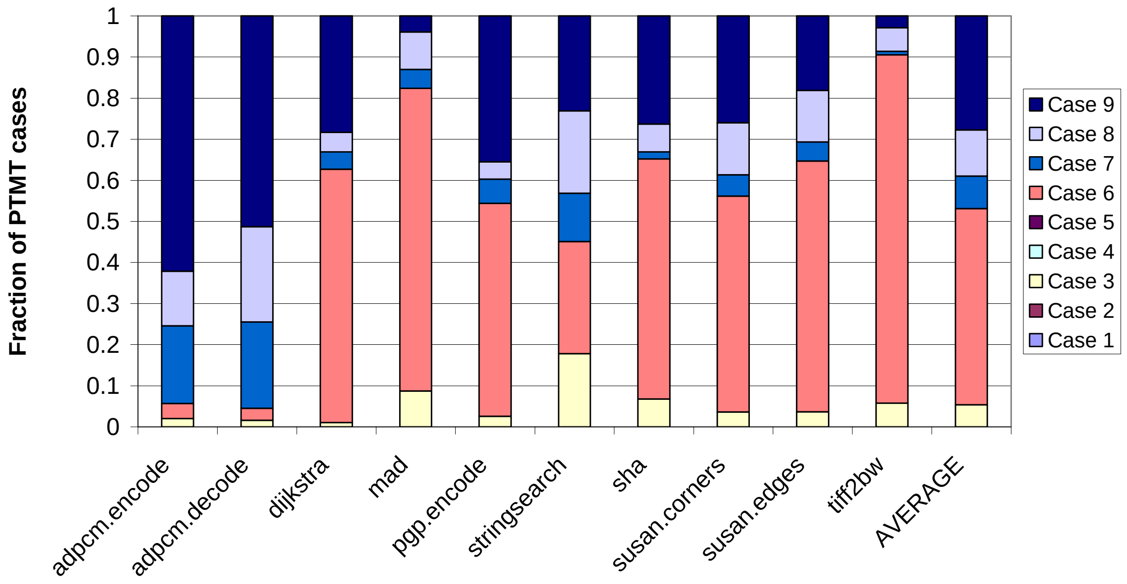


Figure 30: PTMT evaluation for NSP in region-based caches. The figure shows the fraction of prefetches that fit into each PTMT case for a subset of MiBench applications. Applications in which NSP works effectively contain a higher percentage of case 5 and 6 prefetches.

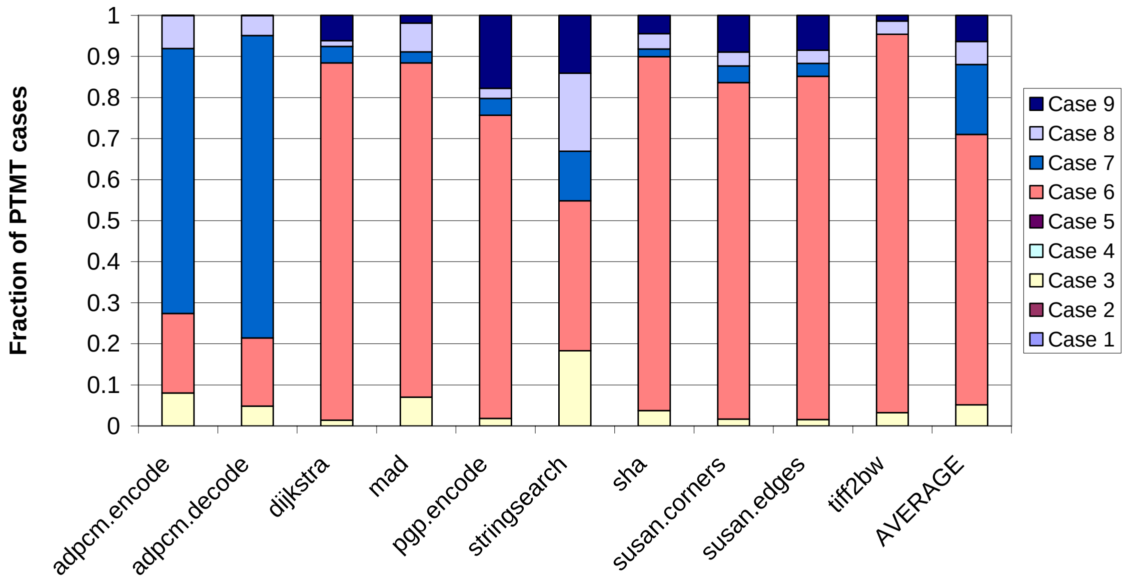


Figure 31: PTMT evaluation for tNSP in region-based caches. The figure shows the fraction of prefetches that fit into each PTMT case for a subset of MiBench applications. Applications in which tNSP works effectively contain a higher percentage of case 5 and 6 prefetches.

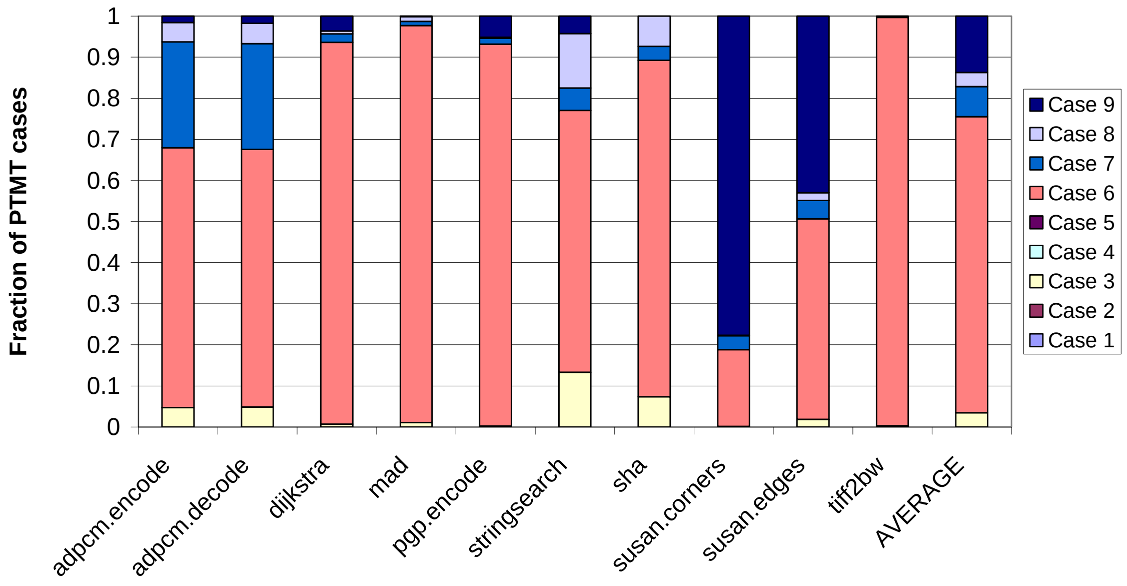


Figure 32: PTMT evaluation with region-based caches for the RPT in region-based caches. The figure shows the fraction of prefetches that fit into each PTMT case for a subset of MiBench applications. Applications in which the RPT works effectively contain a higher percentage of case 5 and 6 prefetches.

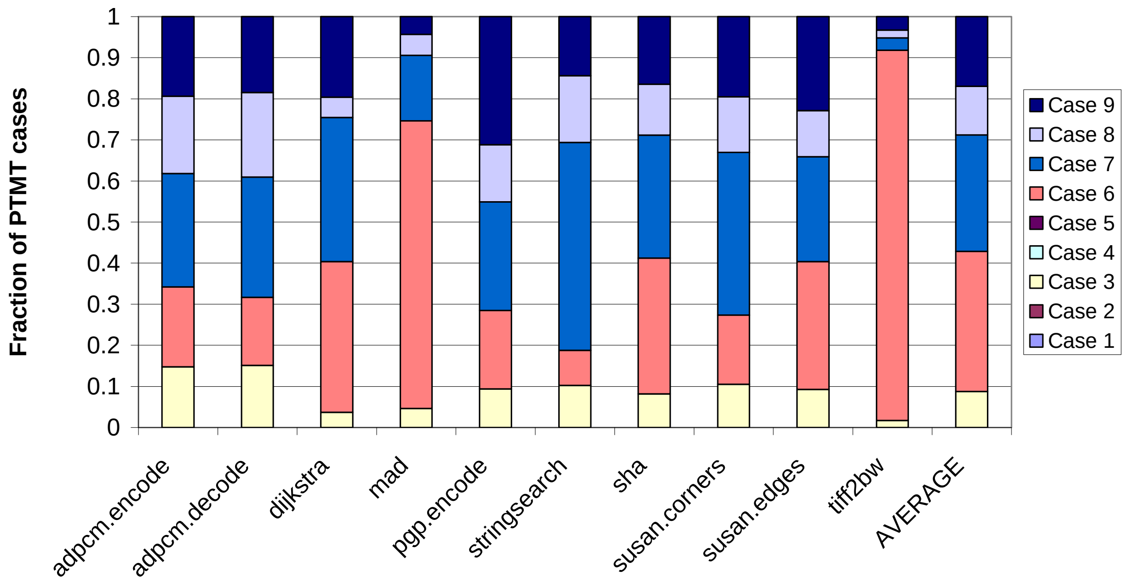


Figure 33: PTMT evaluation with region-based caches for Markov prefetching in region-based caches. The figure shows the fraction of prefetches that fit into each PTMT case for a subset of MiBench applications. Applications in which the Markov prefetcher works effectively contain a higher percentage of case 5 and 6 prefetches.

and global regions, the number of prefetches increases as well. Some applications and algorithms benefit from the partitioned references. For example, sha, which generates few useful prefetches (26.8%) using tNSP in a 32 KB cache, sees mostly useful additional prefetches in region-based caches, raising that percentage to 86.2%. In other cases, the prefetch algorithms become overly aggressive. The adpcm applications experience a drop from 43% to 3% useful prefetches using NSP with the two configurations. On average, tagged NSP (65.9% useful prefetches) and Markov prefetching (34.1% useful) improve slightly with region-based caches, while the NSP (47.8% useful) and RPT (72.1% useful) algorithms are less effective.

To determine which regions prefetch best, we examine the percentage of useful prefetches in each region cache in Figure 34 through Figure 37. Note that some applications show zero bars for the Markov prefetcher. In most cases, that value implies that no prefetches were generated for that cache; the exception is adpcm.decode, in which stack prefetches were generated, but none of them were useful.

For both flavors of next sequential prefetching (NSP and tNSP), the stack experienced the highest percentage of useful prefetches—67.5% and 69.0%, respectively—because stack data typically possess a small, highly local working set. The RPT is most effective (79% useful prefetches) in the global region, implying that structures referenced in a regular pattern reside in the global space. We see that for all mechanisms except the ineffective Markov prefetcher, the percentage of useful stack prefetches is consistently around 66%. The usefulness of prefetches in other regions varies more widely across algorithms.

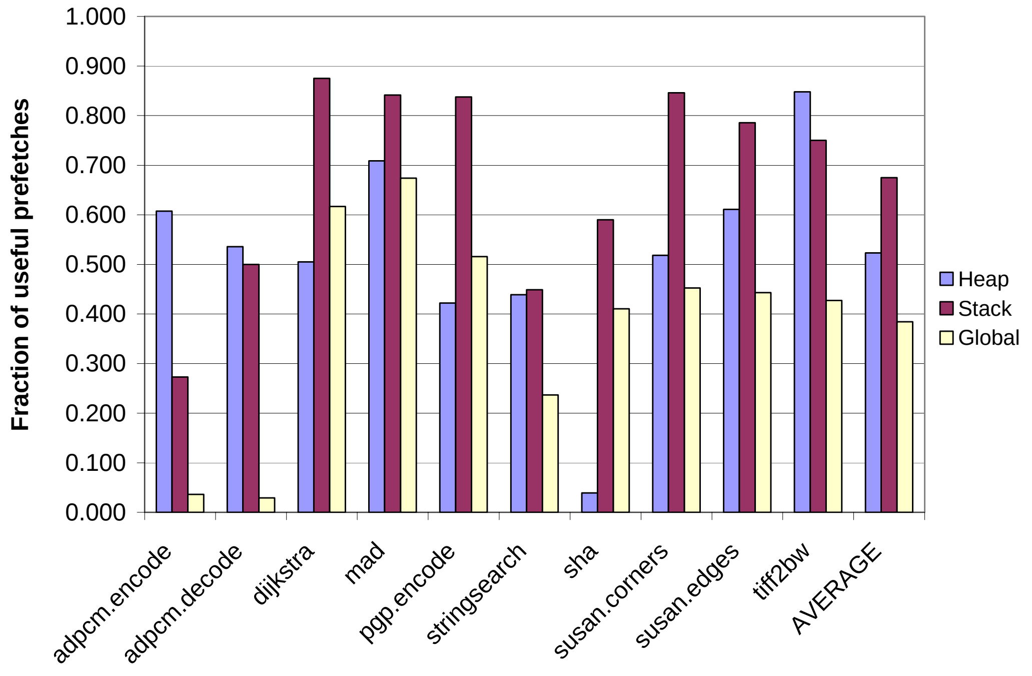


Figure 34: Fraction of useful prefetches in each region cache for NSP. For this algorithm, the stack prefetches most effectively, with close to 70% of stack prefetches classified as useful. However, the percentage of useful prefetches in each region varies dramatically according to the reference characteristics of each application

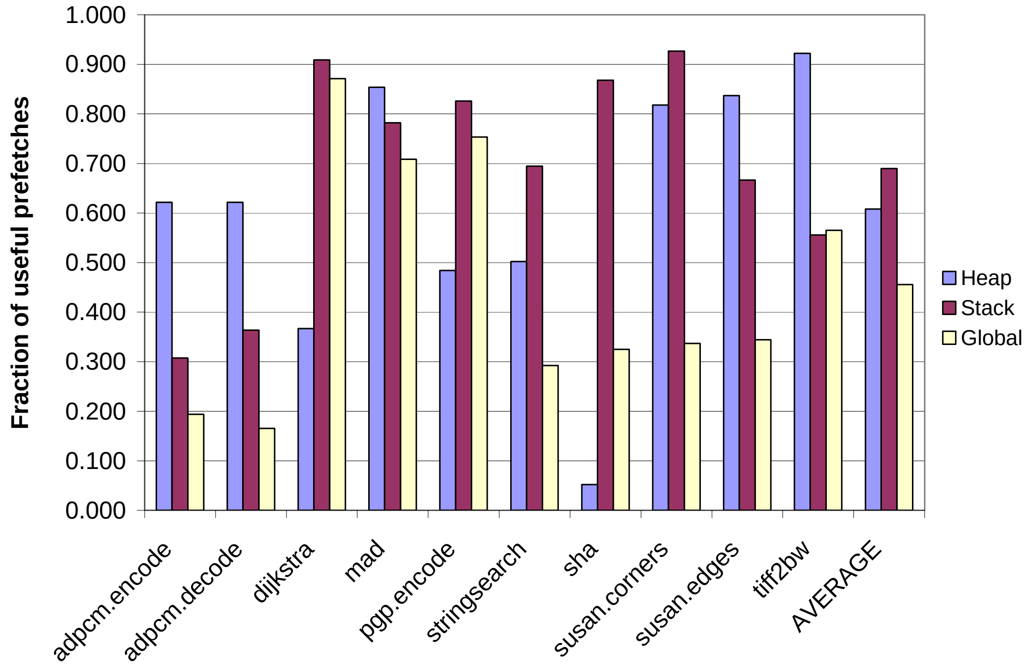


Figure 35: Fraction of useful prefetches in each region cache using tNSP

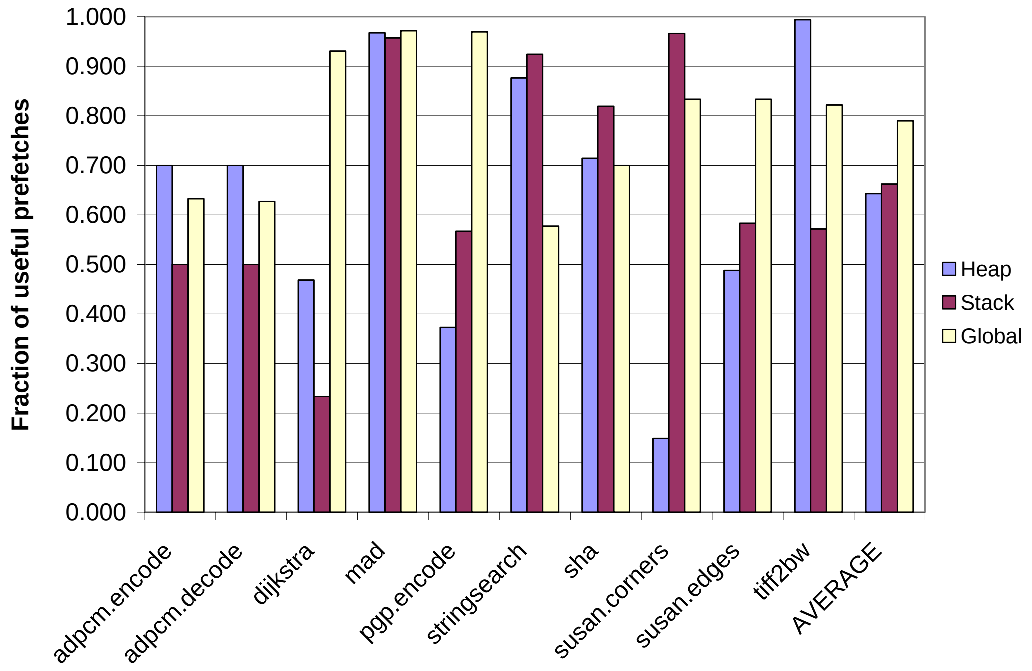


Figure 36: Fraction of useful prefetches per region using the RPT

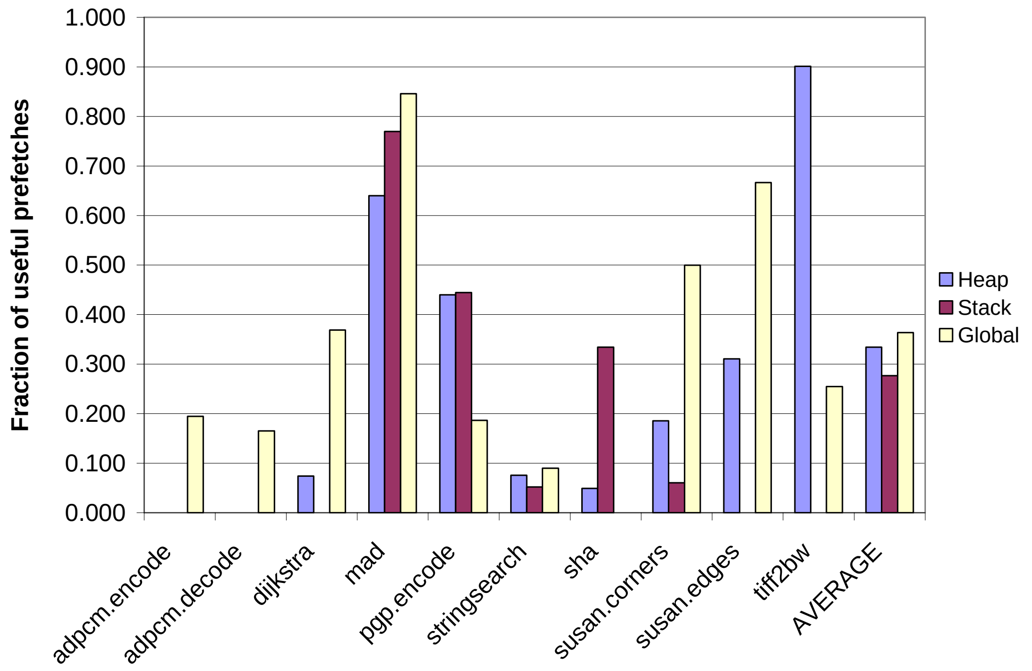


Figure 37: Fraction of useful prefetches per cache using Markov prefetching

6.2 Prefetch Region Implementation

As shown in Section 6.1, some data are better suited to prefetching than others. In some cases, a single region fits well with a prefetch mechanism, as the stack does with next sequential prefetching. In others, the effective subset is spread across regions. However, we do consistently see that only a fraction of prefetches are effective; eliminating the remaining prefetches can undoubtedly improve application performance. The region-based caching simulations suggest that separating conflicting data may improve prefetch effectiveness. We can therefore partition the cache to isolate data that generate the most effective prefetches. For each block, we calculate the percentage of useful prefetches and route blocks with a majority of useful prefetches to a separate cache. This approach excludes data that prefetch poorly and also ensures that prefetched data do not conflict with other regions. A dedicated partition for selectively prefetched data can reduce useless and polluting prefetches, thus limiting cache pollution.

This scheme is somewhat similar to prefetch buffering, as both use structures specifically for prefetched data [55][81][94]. However, prefetch buffers are limited. They typically target data streams and only allow accesses to the head of the buffer. Our scheme uses a cache to store the prefetch targets, which allows random accesses and accommodates any prefetch mechanism. Furthermore, we profile applications to identify prefetchable data and ensure that only those data access the prefetch region. Prefetch buffers can use filters to reduce overly aggressive prefetching, but they typically use heuristics that may not identify useful data.

The major issue with a prefetch region is the routing of accesses to the new cache. One possible approach is to allocate all prefetchable data structures in a separate area of memory. The techniques we present in Chapter 7 can be adapted to move these data to the new region. This approach would allow us to continue using a simple bounds checking mechanism to determine which cache to access. A less desirable alternative is to check both the prefetch cache and appropriate region cache in parallel. This approach, while less complicated at the compiler level, is much less energy efficient and therefore impractical.

6.3 Experiments

In the following sections, we analyze the effects of prefetching on MiBench applications. We focus primarily on performance, using three metrics: overall execution time, miss rate, and memory cycles per instruction (MCPI). As we show in the sections below, the impact of prefetching on overall performance is minimal for these applications because they cache well and use relatively few memory instructions. However, the two memory-specific metrics highlight the ability of these techniques to improve performance in the memory system alone. We believe this work will grow in importance as the gap between memory and processor speeds continues to widen.

6.3.1 Prefetch Mechanisms

Figure 38 shows the effects of prefetching on overall performance, both with and without region-based caches. The baseline is a 32 KB unified L1 cache without prefetching. With the unified L1 data cache, all four prefetchers change the overall performance by less than 1%, usually offering a slight improvement. The Markov prefetcher, which is the least effective, increases application runtime by an average of 0.2%. In the region-based cache configurations, performance improves somewhat due to

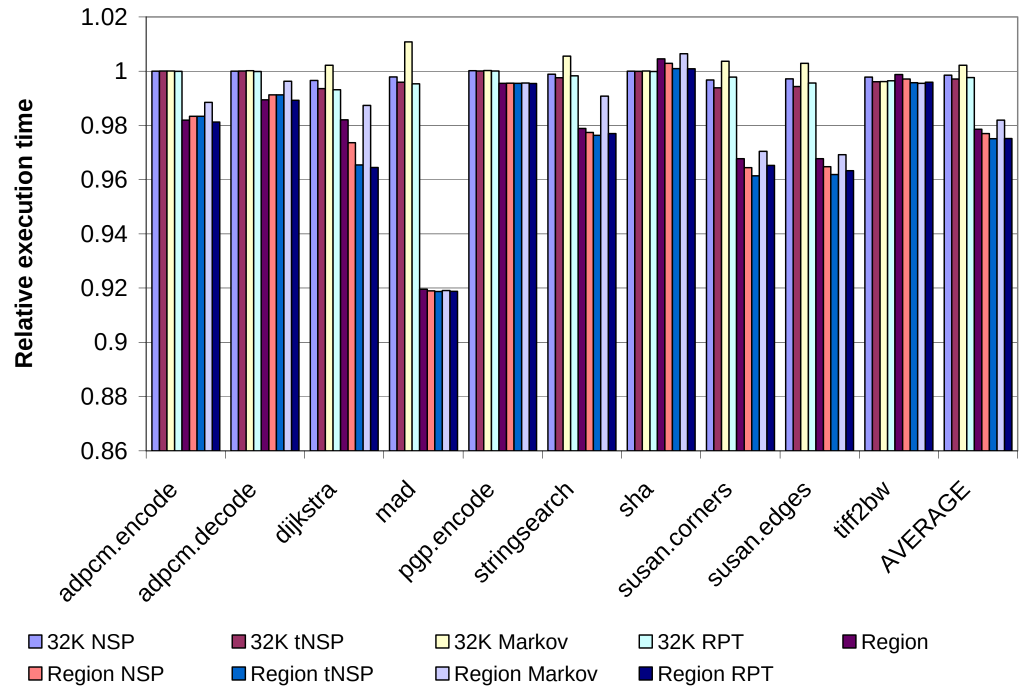


Figure 38: Relative execution time for prefetch mechanisms, with and without region-based caching. The baseline is a 32 KB direct-mapped unified L1 data cache

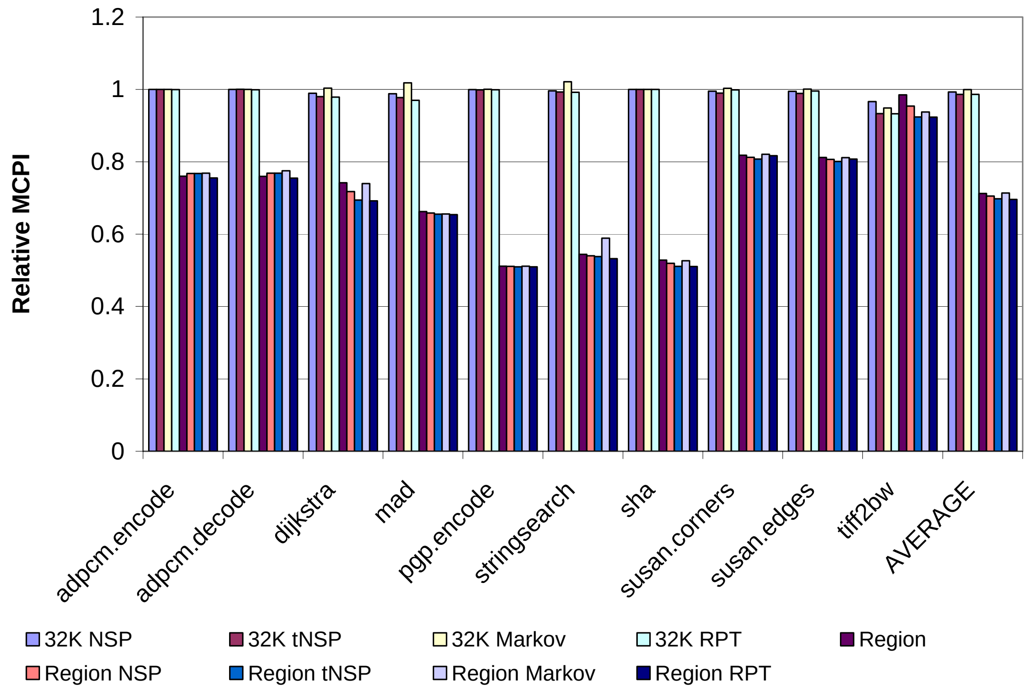


Figure 39: Relative MCPI for prefetch mechanisms, with and without region-based caches. The baseline is a 32 KB direct-mapped unified L1 data cache

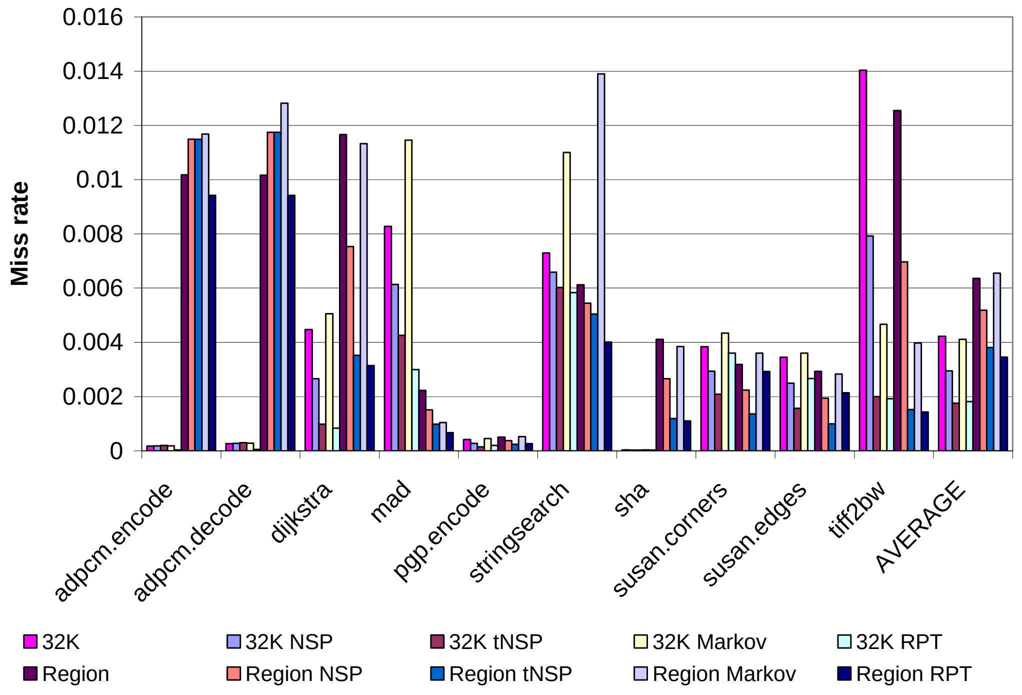


Figure 40: Miss rate for prefetch mechanisms, with and without region-based caching

the reduced latency of the stack and global caches. The prefetch algorithms have a minimal effect, improving performance in most cases by less than 1%. The performance trends are similar for the memory system alone, as shown in Figure 39. The figure shows the MCPI of these applications relative to a 32 KB unified L1 data cache without prefetching. In most cases, prefetching slightly improves memory system performance, with an average MCPI reduction of approximately 1% for both unified and region-based caches. Region-based caching alone reduces MCPI by 29%.

Prefetching offers little performance improvement because the applications cache so well that there are few misses to replace. Figure 40 shows the miss rate of our application subset, with and without prefetching, and also with and without region-based caching. Only one of the ten applications, tiff2bw, has a baseline miss rate worse than 1%, and in several cases, the baseline miss rate is below 0.1%. On average, these applications have a miss rate of 0.4% in a unified 32 KB cache without prefetching. The figure does show that the prefetch mechanisms can remove a significant number of existing misses. NSP lowers the average miss rate to 0.3%, while tNSP and the RPT reduce that figure to 0.2%. In the application with the worst miss rate, tiff2bw, tNSP and RPT reduce the miss rate from 1.4% to 0.2%. Although we may not achieve the same degree of success in more memory-intensive applications, these results show potential for improvement. The region-based caches slightly increase the miss rate for most applications, as stack and global data experience more capacity misses in the smaller caches. In some cases, notably the adpcm applications, we see that prefetching actually causes more misses by placing more pressure on the stack and global caches. On average, all prefetchers except Markov reduce the miss rate in region-based caches, with the RPT effectively reducing misses in all ten applications.

Prefetching also has little effect on data cache energy consumption, as shown in Figure 41. Recall that, given our simulation parameters, static energy dominates L1 cache energy consumption. Since static energy is proportional to program runtime, the minimal change in performance translates to a minimal change in energy. The RPT configurations do consume more energy—on average, 30.5% with a unified L1 cache and 14.5% with region-based caches—because of the additional table. As Figure 42 shows, the effect on the L2 cache energy consumption is even smaller, because static energy consumption comprises an even higher fraction of the energy dissipated by the large, infrequently accessed L2.

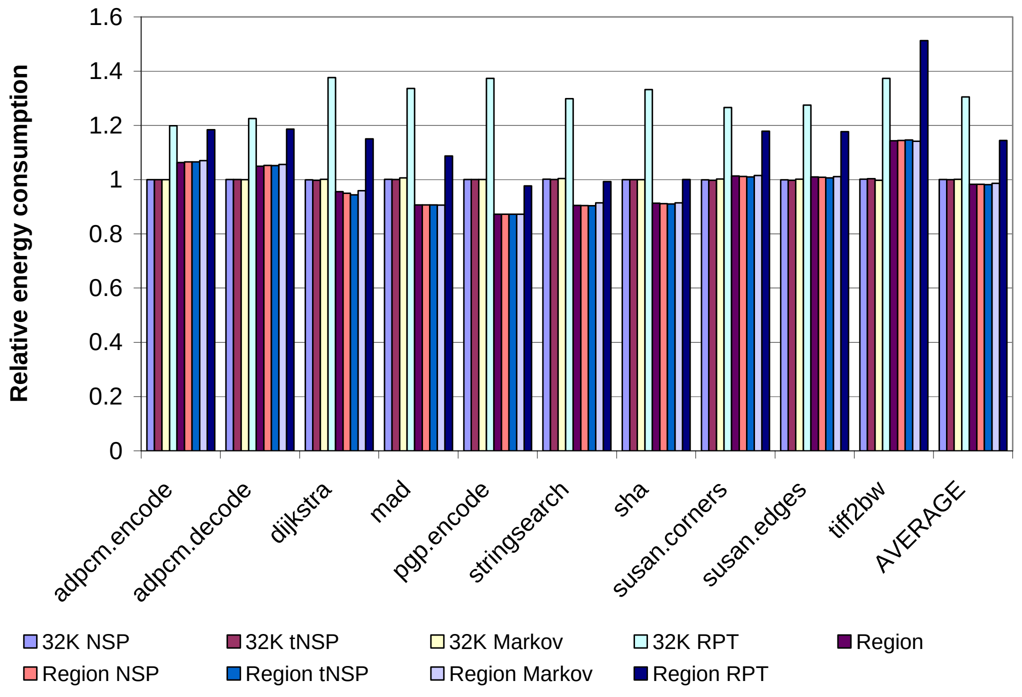


Figure 41: Relative energy consumption of prefetch mechanisms, with and without region-based caching. The baseline is a 32 KB direct-mapped unified L1 data cache

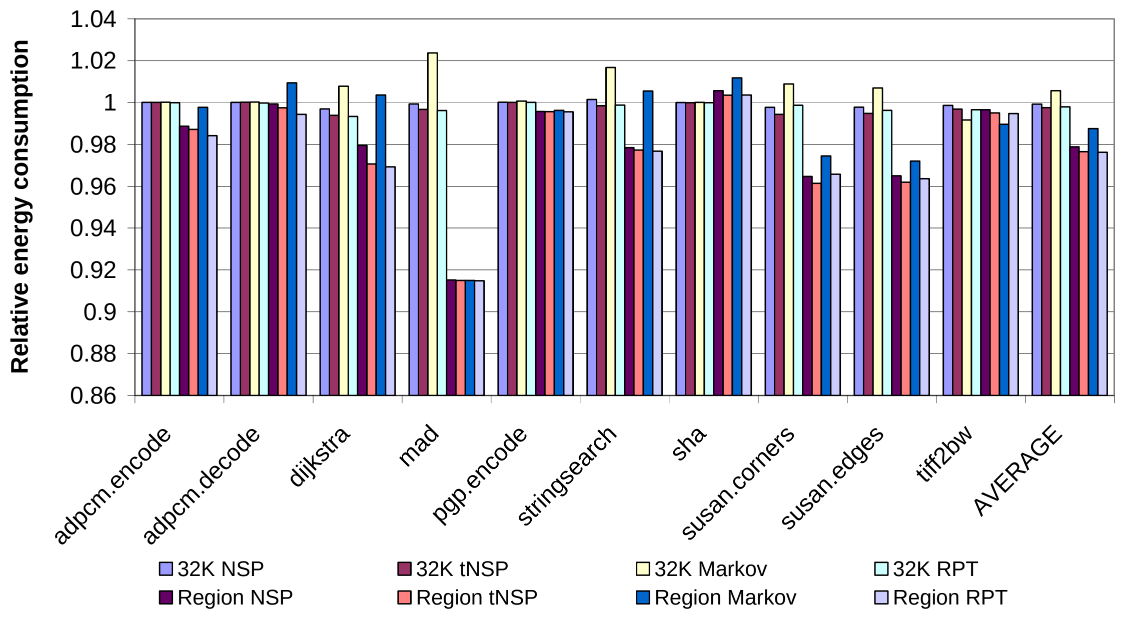


Figure 42: Relative energy consumption of L2 cache for prefetch mechanisms, with and without region-based caching. The baseline configuration uses a 32 KB direct-mapped unified L1 data cache

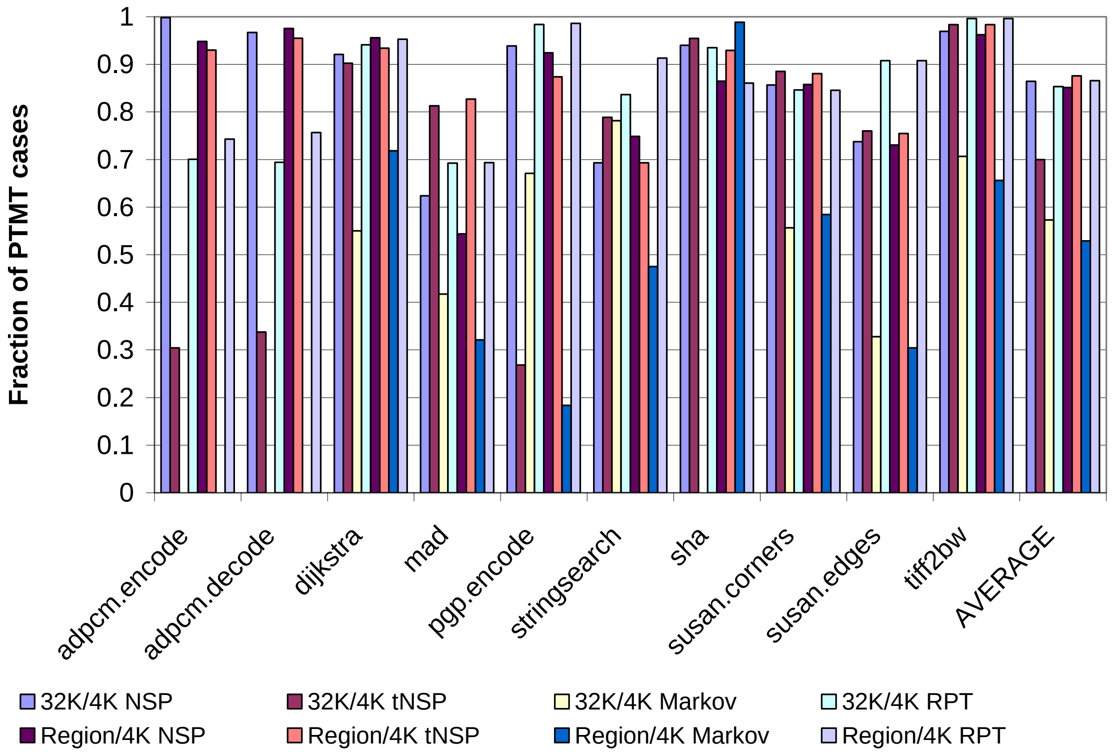


Figure 43: Fraction of useful prefetches using a separate prefetch region, with and without region-based caches. Further cache partitioning dramatically increases the percentage of useful prefetches

6.3.2 Prefetch Region Analysis

We now examine the results of adding a separate prefetch region, as discussed in Section 6.2. In all cases, the prefetch cache is a 4 KB direct-mapped structure, the same size as the stack and global caches. As Figure 43 shows, selectively prefetching to a separate partition increases the fraction of useful prefetches. Some applications show zero values with the Markov prefetcher, indicating that no suitable prefetch candidates exist for those applications. Refining the prefetch target heuristic to include a minimum number of blocks might allow us to improve Markov prefetching in these programs.

In most applications, a clear majority of prefetches are useful, with several applications generating over 90% useful prefetches. NSP experiences the greatest improvement of the four algorithms, with 86.4% and 85.1% useful prefetches in unified and region caches, respectively—an improvement of 37% in each case. For tNSP, useful prefetches increase to 70% (6% improvement) and 87.6% (22% improvement) in unified and region caches. For the RPT, the algorithm with the least room for improvement, useful prefetches increase to 85.3% (7% improvement) and 86.6% (14% improvement). And in the Markov prefetcher, the least effective of the four, partitioning allows the majority of the prefetches to be useful—57.3% (25% improvement) and 52.9% (19% improvement).

Despite the improvement in prefetch quality, the performance of these applications remains relatively unchanged. Figure 44 shows the relative execution times of the application subset using a separate prefetch cache. Most applications remain within 1% of the baseline; however, susan.corners and susan.edges suffer significant performance losses. In Chapter 5, we showed that these same applications ran much slower with a small heap cache due to increased heap conflicts. The small prefetch cache has the same effect.

The relative MCPI results shown in Figure 45 are more promising, showing significant improvement in memory system performance. The prefetch cache allows all applications to use fewer memory cycles per instruction than the baseline. The RPT actually improves the most, with an average relative MCPI of 0.799 in the unified L1 cache—an 18.7% improvement—and 0.596 in the region-based L1 cache—a 9.9% improvement. In the unified and region caches, NSP improves by 13.7% and 2.8%, tNSP improves by 20% and 5.2%, and Markov prefetching improves by 6.7% and 4.3%.

The miss rate numbers shown in Figure 46 are worse than the baseline, as expected. As shown in Figure 40, region-based caches increase the overall miss rate slightly

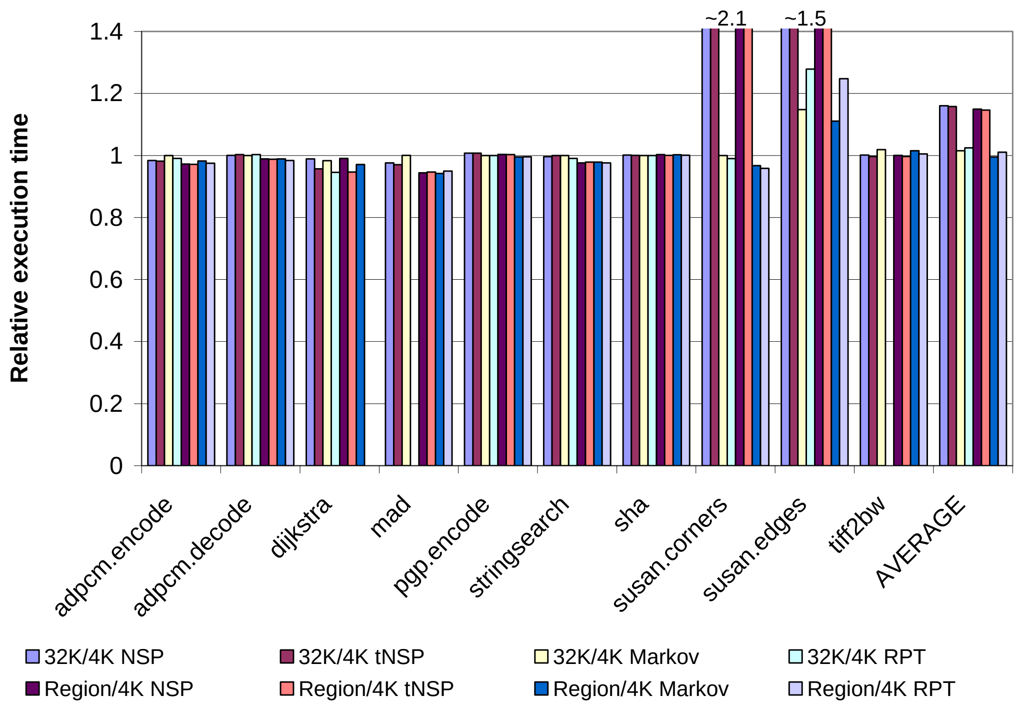


Figure 44: Relative execution time for split prefetch caches, with and without region-based caching. Note that some applications do not complete successfully, leaving blank spaces in the graph. The baseline is a 32 KB direct-mapped unified L1 data cache

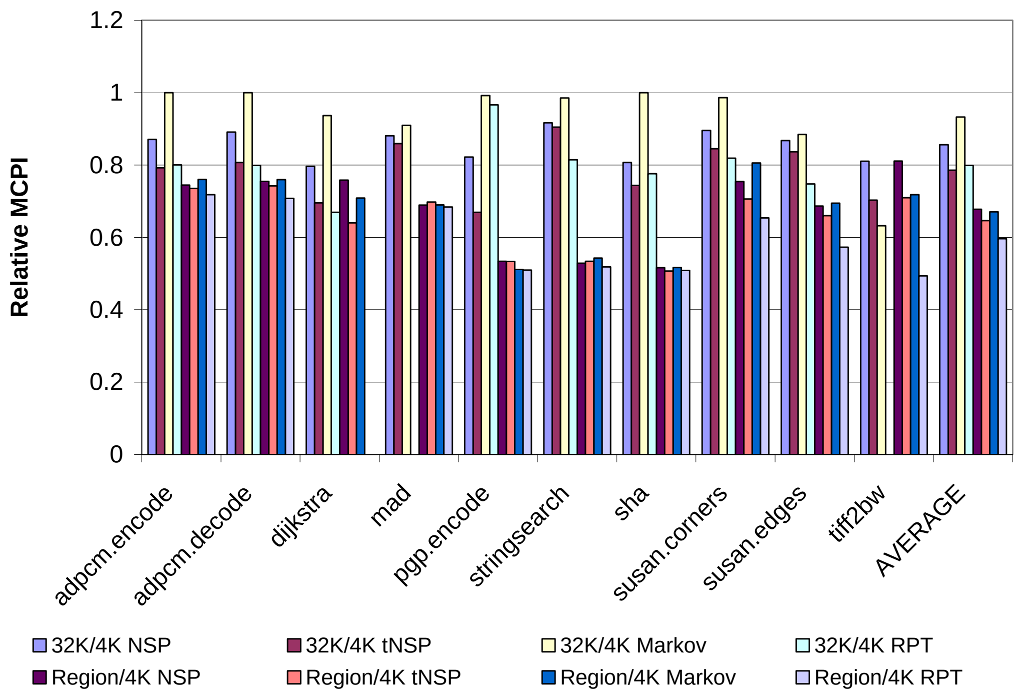


Figure 45: Relative MCPI for prefetchers using split prefetch caches, with and without region-based caching. Note that some applications do not complete successfully, leaving blank spaces in the graph. The baseline is a 32 KB direct-mapped unified L1 data cache

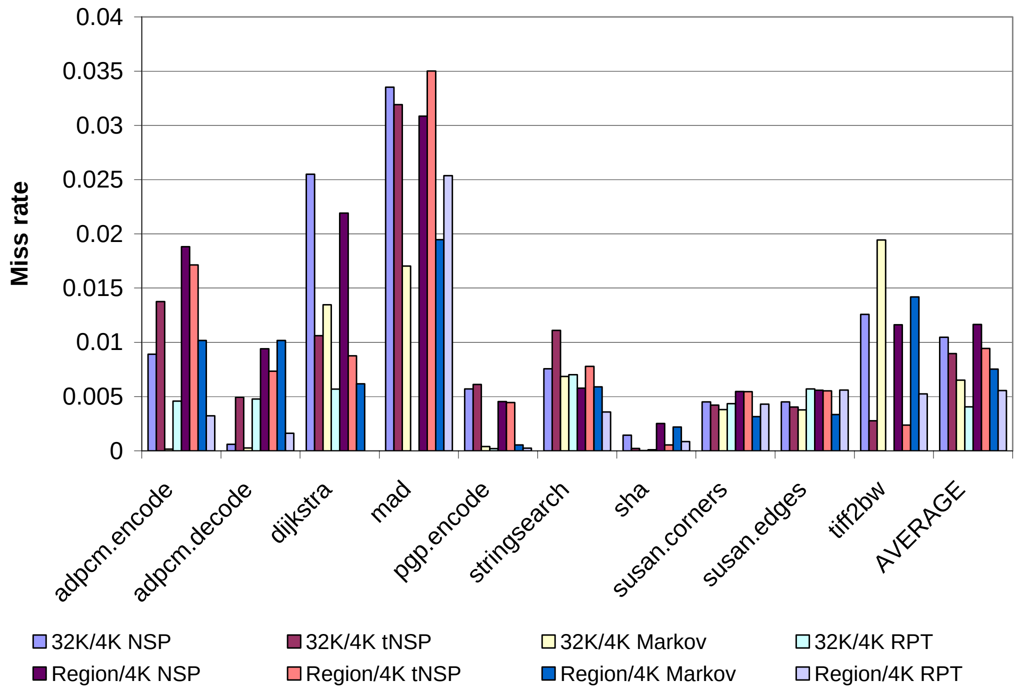


Figure 46: Miss rate for split prefetch caches, with and without region-based caching

because the smaller stack and global caches increase capacity misses; the prefetch cache has the same effect. The overall performance numbers show that these additional misses are not significant. The worst case miss rate—for the *mad* application, using tNSP in region-based caches—is only 3.5%, with an average miss rate between 0.4% and 1.2%.

The additional cache does increase energy consumption, as shown in Figure 47. susan.corners and susan.edges dissipate significantly more energy because of their dramatically increased runtimes. However, when the performance is roughly equivalent to the baseline, the smaller prefetch cache reduces energy consumption due to its lower dynamic energy cost. dijkstra and mad are two examples.

To see the overall effect of the prefetch cache on performance, we show the relative MCPI for the RPT, the most successful prefetch mechanism, in Figure 48. The figure shows four separate L1 data cache configurations: a unified 32 KB cache, a unified cache

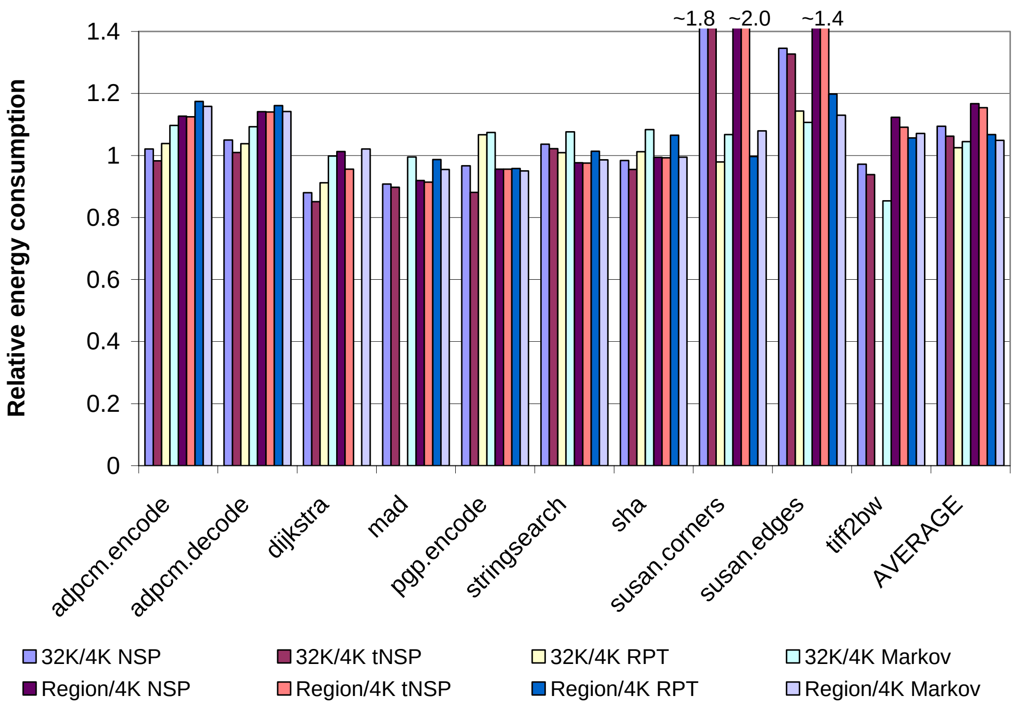


Figure 47: Relative energy consumption for split prefetch caches, with and without region-based caching. The baseline is a 32 KB direct-mapped unified L1 data cache

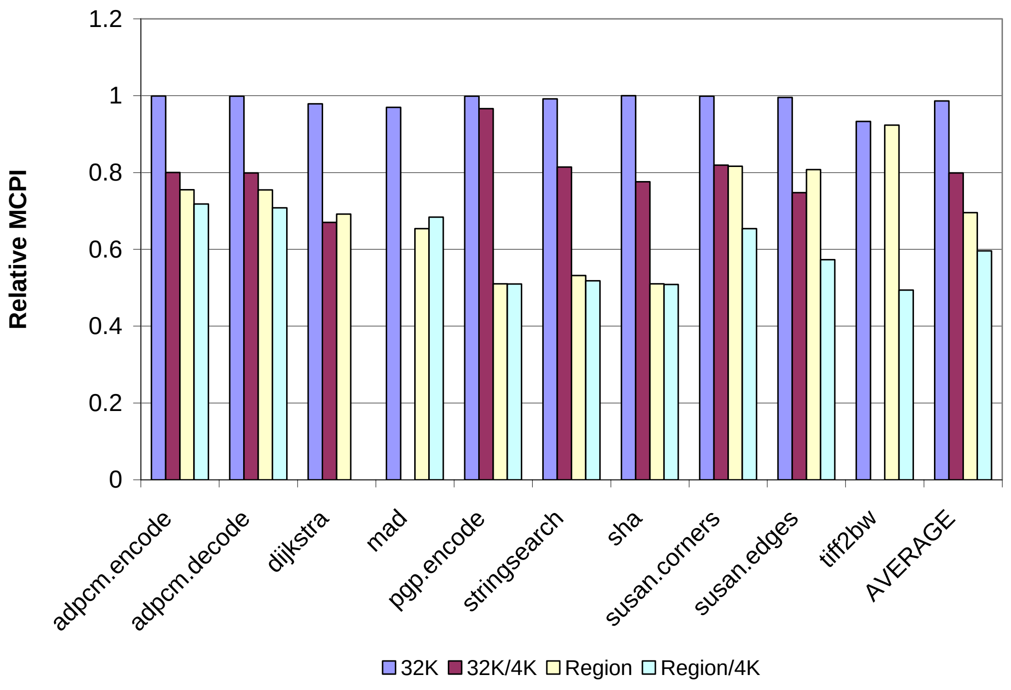


Figure 48: Relative MCPI for the RPT, using four L1 data cache configurations--a single unified 32 KB cache, a unified cache with a 4 KB prefetch cache, region-based caches with 4 KB stack and global caches, and region-based caches with an additional 4 KB prefetch cache

with an additional 4 KB prefetch cache, region-based caches with 4 KB stack and global caches, and region-based caches with the additional prefetch cache. These data show that partitioning the cache increases the benefit of prefetching. Prefetching alone reduces MCPI by just 1.4%; adding the prefetch cache to the unified configuration improves MCPI by an average of 20.1%. Partitioning the cache by region offers a more dramatic improvement of 30.5%, but we again see that the prefetch cache further reduces memory cycles with an average MCPI reduction of 40.4%, a 9.9% reduction over region-based caching alone.

6.3.3 Energy Efficient Prefetching

Because MiBench applications access memory infrequently and cache well, our prefetch schemes have little effect on performance. However, prefetching effectively eliminates misses when those misses are present. If we reduce the cache sizes to lower energy consumption, we can use prefetching to reduce the resulting performance loss. The additional prefetch cache will slightly increase energy consumption but will have a greater effect on the miss rate, providing an energy-efficient cache with good performance. For the following simulations, we shrink the region-based caches to 1 KB for stack and global data, and 4 KB for heap data. The prefetch cache size remains 4 KB. We evaluate the same four prefetch algorithms—NSP, tNSP, Markov, and the RPT.

Figure 49 shows the overall performance for these simulations. Note that smaller caches alone improve performance because all three region caches now have single-cycle accesses. We once again see that susan.corners and susan.edges perform poorly because of increased heap misses. However, in most other cases, prefetching allows us to maintain reasonable performance, offering improvements in many cases. The most

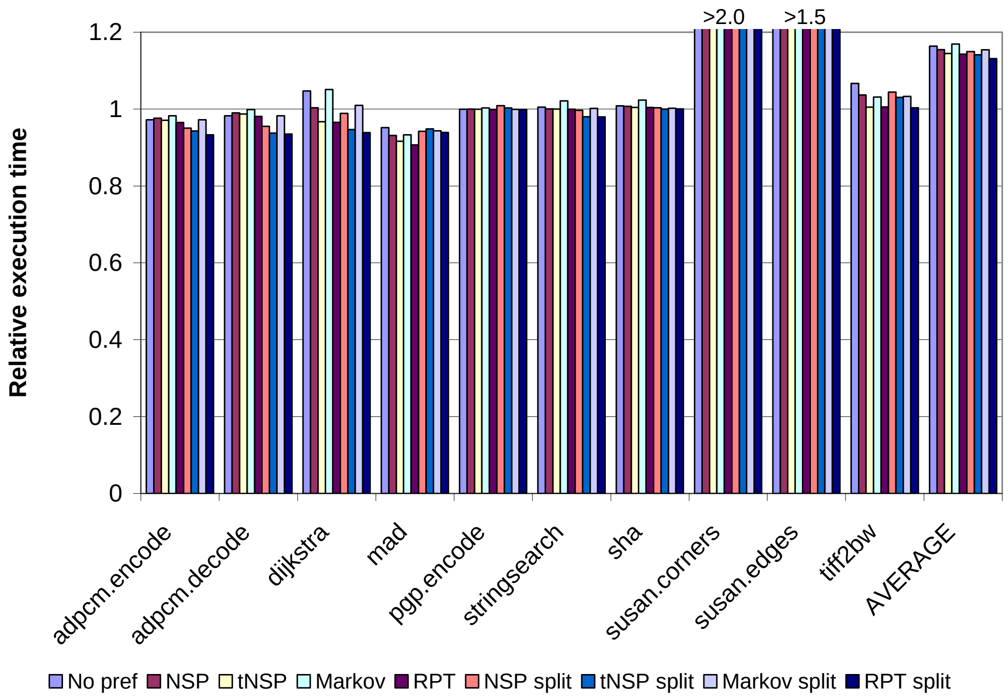


Figure 49: Relative execution time for prefetching in small region cache configurations (1 KB stack and global caches, 4 KB heap cache), with and without an additional 4 KB prefetch cache. The baseline is a 32 KB direct-mapped unified L1 data cache

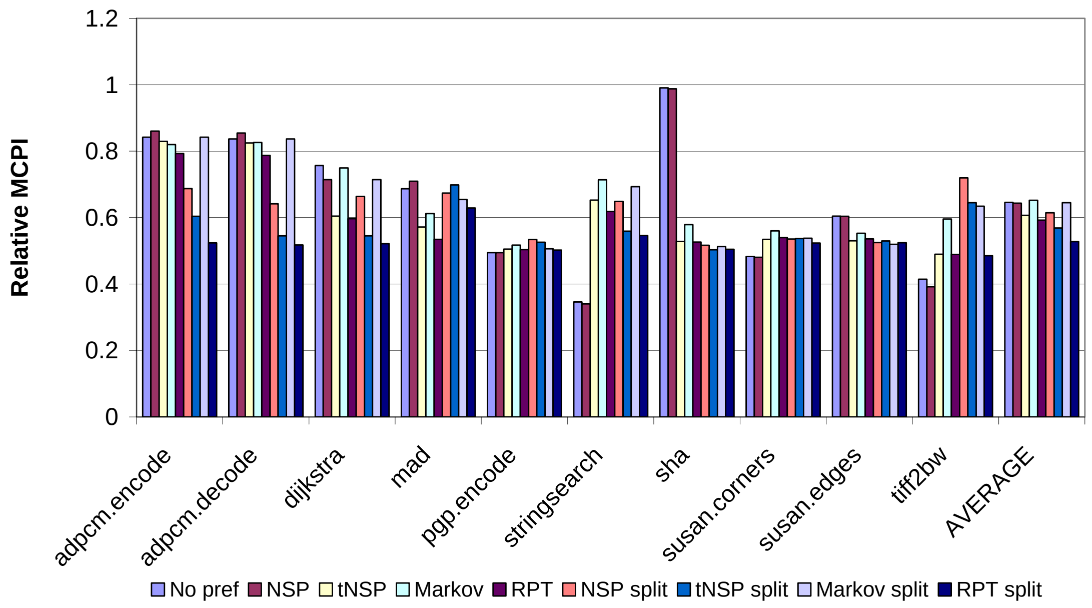


Figure 50: Relative MCPI for prefetching in small region cache configurations (1 KB stack and global caches, 4 KB heap cache), with and without an additional 4 KB prefetch cache. The baseline is a 32 KB direct-mapped unified L1 data cache

dramatic example is dijkstra, which runs 4.7% slower with the small caches. Three of the four prefetchers offer a speedup over the baseline, which becomes more significant when the prefetch cache is added. In the best case, dijkstra runs 6.1% faster than the baseline using the RPT with the prefetch cache.

We see the effect of prefetching on the memory performance of these small caches in Figure 50. Again, the small caches alone improve memory performance with single-cycle accesses; without prefetching, these applications have an average MCPI that is 35.4% lower than a 32 KB unified cache. However, prefetching further reduces memory cycles. In dijkstra, the RPT combined with a prefetch cache reduces MCPI by 47.8%, a 22.5% improvement over the small caches alone. On average, prefetching with the split caches reduces MCPI by 38.5% using NSP, 43.1% using tNSP, 47.2% using the RPT, and 35.5% using Markov prefetching.

As we see in Figure 51, the prefetch cache typically reduces miss rate in these smaller caches. The additional partition further reduces capacity misses for all caches. The best example is adpcm.encode, which has the worst baseline miss rate—5.7%—of any of these applications. Without the prefetch cache, NSP slightly increases the miss rate to 6.0%. The other three prefetchers reduce the miss rate: 5.5% for tNSP, 4.9% for the RPT, 5.3% for Markov prefetching. When the prefetch cache is added, the Markov prefetcher actually performs worse, with a 5.7% miss rate that matches the original value, but the other three prefetchers offer significantly lower rates: 3.1% for NSP, 1.7% for tNSP, and 0.4% for the RPT. As the figure shows, all prefetchers except Markov improve average miss rate, and the prefetch cache lowers the miss rate in all cases.

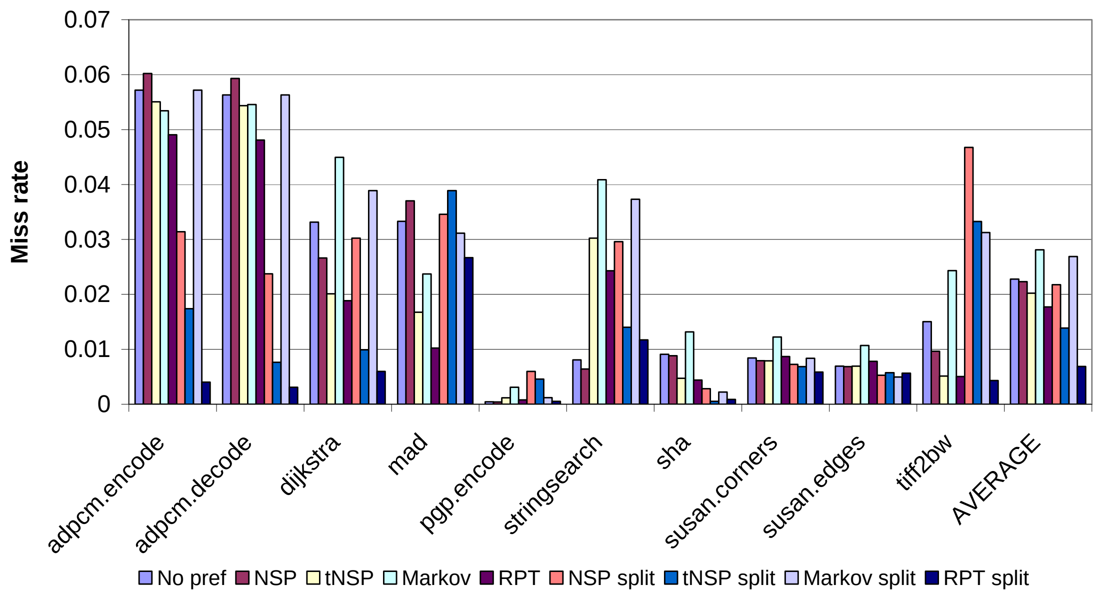


Figure 51: Miss rate for prefetching in small region cache configurations (1 KB stack and global caches, 4 KB heap cache), with and without an additional 4 KB prefetch cache.

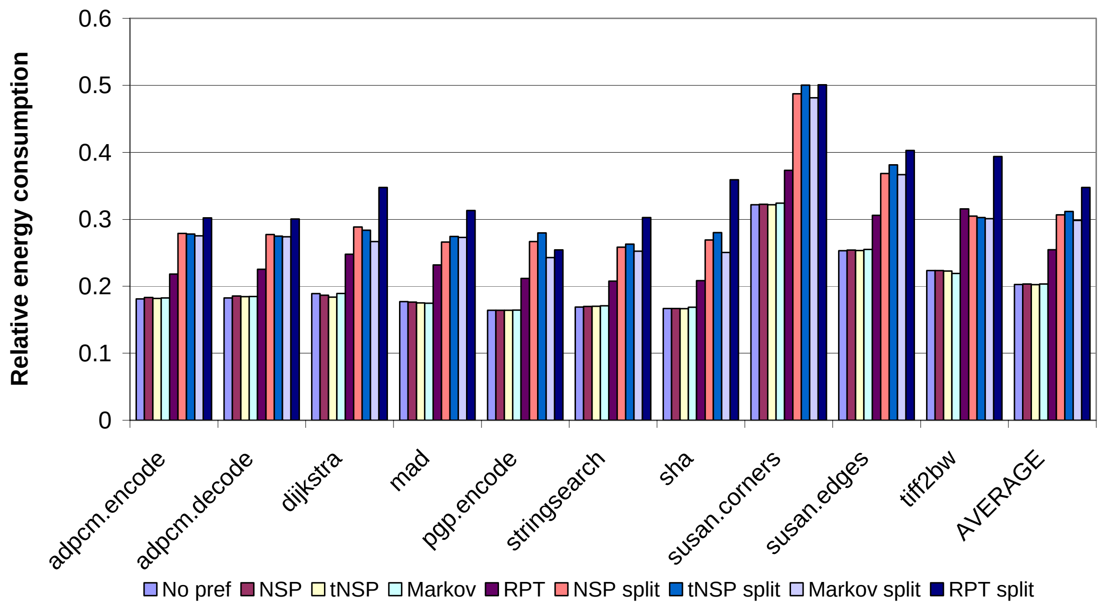


Figure 52: Relative energy consumption for prefetching in small region cache configurations (1 KB stack and global caches, 4 KB heap cache), with and without an additional 4 KB prefetch cache. The baseline is a 32 KB direct-mapped unified L1 data cache

In Figure 52, we see that these schemes save a significant amount of energy. The small caches alone offer the best energy savings, reducing L1 cache energy by 79.7% in these applications. For the same subset, our typical region-based caches consume only

1.7% less energy. As in the larger caches, prefetching increases the energy consumption. The increase is very slight without adding the prefetch cache, as the additional dynamic energy dissipation makes little impact. The RPT is again the exception, as the additional prefetch table reduces the energy savings to 74.5%. With the prefetch cache, we do save less energy, but the consumption is still significantly lower than our unified baseline: 69.3% for NSP, 68.8% for tNSP, 65.2% for the RPT, and 71.2% for the Markov prefetcher.

6.4 Summary

In this chapter, we discussed how partitioning the cache allowed us to improve performance by increasing the effectiveness of data prefetching. Although the effect on program runtime is minimal due to the low memory access frequency and good locality of MiBench applications, prefetching in region-based caches eliminates many of the misses that are present. With the addition of a separate cache for prefetched data, we can refine the effectiveness of our prefetch mechanisms, leading to even greater improvements in memory system performance. Because prefetching can significantly reduce cache misses, we can reduce the cache size to improve energy efficiency.

CHAPTER 7

DATA PLACEMENT IN REGION-BASED CACHES

Region-based caching relies on the idea that programmers use data differently within applications. If a pattern of data usage is repeated at various points throughout a program, that pattern should be coded as a function and the data allocated as local variables. Structures that depend on runtime information are dynamically allocated on the heap. Data that must be visible to multiple functions is allocated in the global region. These different usage patterns across regions lead to varying degrees of locality, which region caches utilize to reduce energy consumption. However, this cache architecture relies on one tenuous assumption: programmers will always place data in the correct regions. If a programmer uses data in an unexpected manner, the semantic regions may possess locality characteristics that are not well-suited to region-based caches, leading to a performance loss. In this chapter, we discuss the ramifications of bad data usage and explore solutions to this problem.

In Section 7.1, we explain how to move misplaced data to the proper region at compile time. In Section 7.2, we analyze a single application, quicksort, to show how data relocation can eliminate performance losses due to bad placement. Section 7.3 summarizes the chapter.

7.1 Moving Data Between Regions

As discussed in Chapter 2, previous work on data movement typically focused on reorganizing data to either improve their locality characteristics or to reduce conflicts [3][18][22][23][24][52][53][62][83]. Our approach targets conflict misses; we want to use the existing locality in certain data sets to determine in which cache they should reside. The targeted data may hurt application performance because their locality is poor or their footprint is large. For example, a sparse local array, which has little locality, should be allocated on the heap instead of the stack. On the other hand, data with good locality are wasted in a large cache. In Chapter 5, we discussed how to identify hot data in the heap and relocate them to a dedicated cache. Finally, an application dominated by references to one region may benefit from moving some references to reduce the footprint size of that region and alleviate pressure on that region cache. For example, if a stack-intensive program uses multiple large local arrays, allocating one of those arrays globally may improve the overall cache hit rate. Note that a single structure may be enough to fill a small cache; an array of thirty-two integers, each of which uses thirty-two bits, consumes 1 KB—one quarter of the stack cache capacity.

Given the appropriate compiler feedback, we can identify those structures for which the locality does not fit the cache and place them in the appropriate region. Low-locality data must reside in the heap cache, the only structure large enough to tolerate random access behavior. Data with good locality can fit in either the stack or global region. Stack data typically has a smaller working set, so the footprint size can determine which region is most appropriate for highly local data. In practice, we believe that most moves will

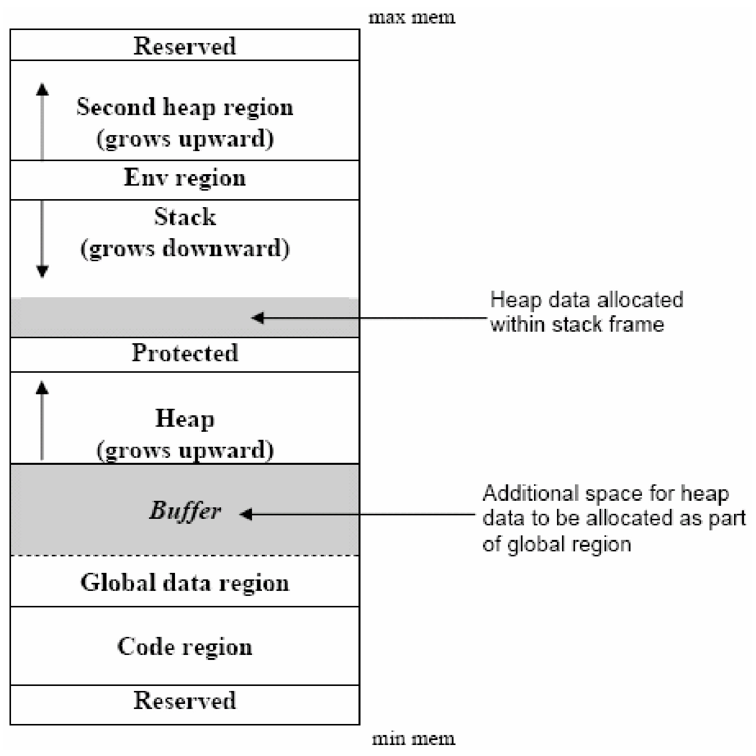

max mem
Reserved
Second heap region (grows upward)
Env region
Stack (grows downward)
Heap data allocated within stack frame
Protected
Heap (grows upward)
Buffer
Additional space for heap data to be allocated as part of global region
Global data region
Code region
Reserved
min mem

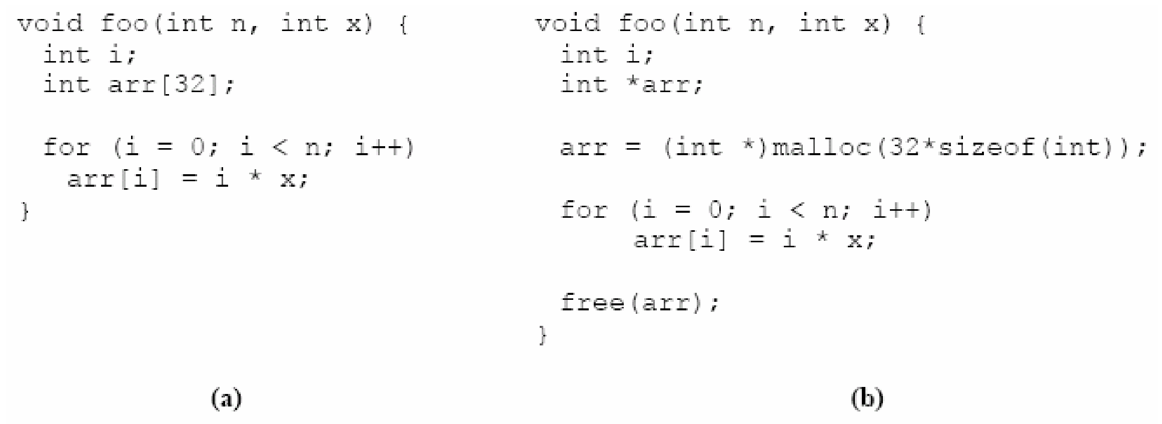
Figure 53: Memory map showing necessary modifications to allow allocation of heap data within stack and global regions. Moving heap data to the global region requires that the base of the heap be moved to a higher address, leaving a buffer in the global area for dynamically allocated data. Moving heap data to the stack requires the function using that data to allocate extra space in its stack frame. Both cases use a second dynamic allocator with the ability to access these regions

involve either heap or stack data. The low frequency of accesses to the global region reduces the impact that sparse global structures would have on system performance. As noted, we have already shown evidence of highly local data in the heap; it stands to reason that, in some applications, the stack possesses poor locality.

Moving data between regions requires code transformations that, in some cases, introduce additional overhead. We discuss the requirements for each region below.

- **Moving heap data:** We first addressed this issue in Chapter 5. In those experiments, we use a second heap allocator to place objects in their own "region" and route them to a dedicated cache. The additional dynamic allocator is necessary because the size of a heap structure is typically unknown at compile time. If the programmer defines a maximum size for all heap items, we can statically allocate that amount of space, but that situation is unlikely and inefficient. Moving hot heap data into existing regions is relatively straightforward. For the global region, statically allocating extra buffer room at the end of the predefined region will allow space to allocate heap objects. As shown in the memory map in Figure 53, this modification is simple when the global and heap regions border one another—the base of the standard heap region must simply be moved to a higher address. Moving heap data to the stack requires the function allocating the data to create a stack frame with enough space to hold the dynamically allocated object.

- **Moving stack data:** Stack data movement depends on the nature of the function accessing those data. Recursive functions must allocate data on the heap, as a global variable referenced by a recursive function would be overwritten on each function call. Non-recursive functions can reallocate local variables in either the heap or global regions. Moving data from the stack to the global region simply requires moving the variable declaration—and therefore its allocation—outside of the function body. Moving a variable from the stack to the heap is slightly more complex and does introduce some overhead, as shown in Figure 54. In this brief example, a local array of integers is reallocated on the heap. Allocating the array requires a call to `malloc()` in the function prologue; the array is de-allocated in the function epilogue using `free()`. The overhead of these calls will negate some of the savings from reduced cache misses. Note that we may move sparse local structures onto the heap even if they are used by non-recursive functions.



```
void foo(int n, int x) {
  int i;
  int arr[32];

  for (i = 0; i < n; i++)
   arr[i] = i * x;
}
```

(a)

```
void foo(int n, int x) {
  int i;
  int *arr;

  arr = (int *)malloc(32*sizeof(int));

  for (i = 0; i < n; i++)
      arr[i] = i * x;

  free(arr);
}
```

(b)

Figure 54: Example function showing the changes required to allocate local variables on the heap. Part (a) shows the original function, which uses an array, `arr`, of thirty-two integers. In part (b), `arr` is allocated on the heap, with the local array replaced by a local pointer. This change incurs the overhead of calls to `malloc()` and `free()` in the function prologue and epilogue, respectively

- **Moving global data:** Global data can be handled similarly to stack data, with no recursive usage to restrict relcation. Moving this data to the stack simply requires that the variable be declared inside a function rather than externally. Moving this data to the heap requires a call to `malloc()`at the beginning of the program and a call to `free()`at the end.

7.2 Benefits of Data Relocation

To demonstrate the benefits of these proposed changes, we use a single application, *quicksort*. This application reads a list of three-dimensional vectors into an array, computes the distance from the origin for each vector, and then sorts the array based on that distance using the `qsort()` library function in `<stdlib.h>`. Unlike most MiBench applications, quicksort experiences a significant performance loss—20.1%—when using region-based caches in place of a unified 32 KB L1 data cache. The cause of this slowdown is a significant increase in stack and global misses, as shown in Table 12.

	Cache configuration		% change in misses
Region	Unified	Region	
Stack	572309	795582	39%
Global	756	300838	397%
Heap	276893	156355	-44%
TOTAL	849958	1252775	47%

Table 12: Misses by region in quicksort for unified and region cache configurations. The percent change in misses is the difference between the two configurations, using the unified configuration as a baseline

The table lists the misses by region for both the unified cache and region cache configurations. We see that, as expected, misses drop significantly for the data accessing the heap cache when the L1 data cache is partitioned. Because the stack and global data access separate caches, these data experience fewer conflicts, leading to a 44% reduction in misses. However, the small stack and global caches significantly increase the number of misses in those regions—by 39% and 397%, respectively. Note that the raw increase in global misses is on the same order of magnitude as the increase in stack misses (300,082 additional global misses, 223,273 additional stack misses). Overall, L1 data cache misses increase by 47%. This increase only slightly affects the L2 cache activity, with L2 misses increasing by 0.4%.

We can use the data movement strategies discussed in Section 7.1 to reduce pressure on the smaller caches. After profiling the program, we can identify which regions experience the most misses and which data structures are primarily responsible for those misses. The compiler can then use the miss profile to allocate data in the appropriate regions. To approximate this process, we profile the references for each cache block and determine which blocks experience the greatest increase in misses. We then manually determine the new addresses for the offending blocks and direct the simulator to reroute those accesses accordingly. This approximation does have some flaws. First, we manually place data without knowing the actual order in which it will be allocated. The

	Cache configuration			% change in misses
Region	Unified	Region	Region + move	
Stack	572309	795582	613430	7%
Global	756	300838	837	11%
Heap	276893	156355	164915	-40%
TOTAL	849958	1252775	779182	-8%

Table 13: Misses by region in quicksort, taking data movement into account. The percent change in misses is the difference between the configuration with data movement and the baseline unified cache. We reallocate problematic global and stack blocks on the heap to reduce conflicts in the smaller caches

actual cache activity—particularly conflicts involving relocated data—may therefore differ from the simulation. Also, we do not consider the overhead involved moving items to the heap, most notably the pointer accesses and calls to `malloc()` and `free()`.

Our analysis of quicksort shows that in the global region, a few blocks account for the majority of the additional misses. For the stack, most of the additional misses are concentrated in a contiguous 1 KB region near the lowest possible stack pointer address, implying that the stack experiences the most conflicts when the program is at its greatest function call depth. We map each of these blocks into an unused heap location to approximate their reallocation. As Table 13 shows, reallocating these problematic blocks on the heap eliminates the extra misses. The stack and global regions do still experience more misses than in the unified cache, but the increases are more reasonable—7% and 11%, respectively. Since the heap now contains more data, the number of heap cache misses is slightly higher with the relocated data than in the basic region configuration. However, that region still experiences 40% fewer misses than it does in the unified cache. Overall, region-based caches with appropriate data movement reduce the number of L1 cache misses by 8%.

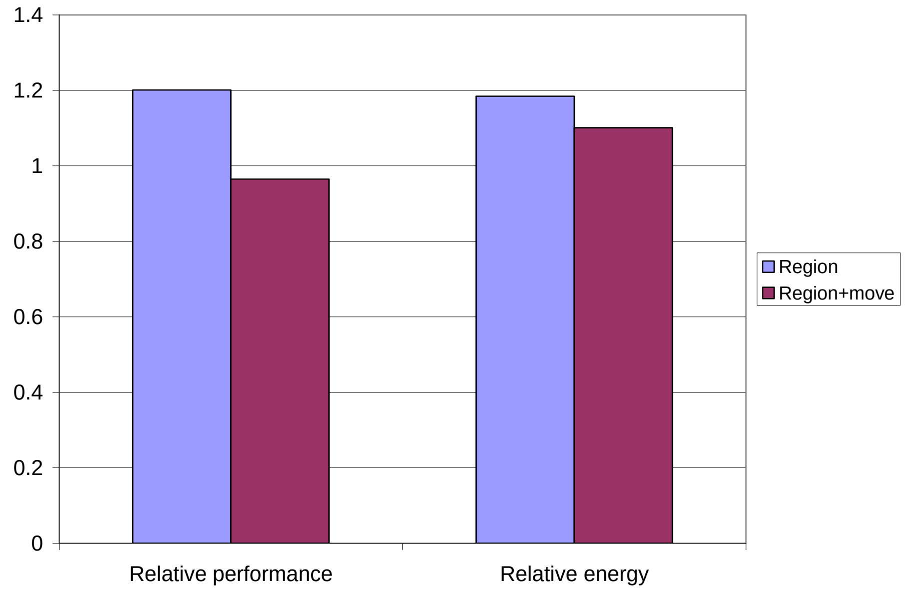


Figure 55: Relative energy and performance values for region-based caches with and without data movement. The baseline is a 32 KB direct-mapped cache. Relocating problematic stack and global blocks to the heap reduces conflict misses and improves the overall performance. However, the relative energy consumption remains high because the relocated blocks access the heap cache, which dissipates more dynamic energy per access than the smaller stack and global caches

Figure 55 shows the relative performance and energy values for quicksort using region-based caches with and without data movement. The baseline is a direct-mapped, 32 KB unified L1 cache. The leftmost bars show the relative performance for these two configurations. As expected, removing the additional misses dramatically improves application performance, leading to a modest speedup of 3.5% with the appropriate data placement. The one downside to this strategy is that L1 data cache energy consumption remains higher than the baseline value. Recall that region-based caches decrease dynamic energy per access by directing most accesses to smaller structures; the tradeoff is higher static energy dissipation per cycle due to the additional cache capacity. With standard placement, the longer runtime leads to a 48.7% increase in static energy over the baseline, causing the application to consume 18.4% more total energy in the L1 cache.

With the new placement, the static energy consumption is only 20.6% higher than the baseline. However, since references to the relocated blocks access the heap cache, not the smaller stack or global caches, we fail to achieve the expected dynamic energy reduction. The new placement leads to a 18.4% decrease in dynamic energy, significantly less than the 63.9% decrease that standard region placement provides. The overall L1 energy consumption is 10.1% higher than the baseline with this configuration.

7.3 Summary

In this chapter, we discussed the problems that arise when data locality characteristics do not match the region in which they are allocated. Identifying these misplaced data and reallocating them in the appropriate region is relatively straightforward given the appropriate compiler feedback. As our case study of the quicksort application shows, these techniques can eliminate unnecessary misses caused by bad data usage, allowing all applications to reap the performance benefits of region-based caches.

CHAPTER 8

CONCLUSIONS

8.1 Summary of Contributions

This dissertation explores techniques for reducing energy consumption and memory latency through intelligent cache partitioning. Using region-based caches as a starting point, we show how splitting the cache allows us to increase the effectiveness of well-known cache optimizations. We also perform a detailed analysis of data reference characteristics in MiBench, our target application suite, to discover further opportunities for improvement.

We show in Chapter 4 that drowsy caching and region-based caching, two complementary techniques for reducing data cache energy, are each more effective when combined. Partitioning the cache by region allows us to tune the drowsy caching policy to the locality characteristics of each subset. In turn, drowsy caching can remove the static energy penalty due to the additional region caches. The result is a cache with low static and dynamic energy consumption that performs comparably to less energy-efficient configurations.

In Chapter 5, we debunk the assumption that all heap data possess poor locality. Our analysis identifies applications in which the entire region caches well, and we find that in other applications, a subset of the heap data exhibits good locality. We use this knowledge to further reduce cache energy consumption by tailoring the cache resources to the demands of each application. We maintain two heap caches, one small, one large, and disable the larger cache when possible for energy savings. When both caches are active, we can save up to 73% in cache energy consumption; when the large cache is inactive, our maximum savings increase to 81%.

In Chapter 6, we shift our focus to memory system performance. Although we find little room for improvement in applications that cache extremely well, we demonstrate how splitting the cache can help us prefetch data more effectively. Classifying individual prefetches allows us to identify data that prefetch well and route their accesses to a dedicated cache, thus increasing the percentage of useful prefetches. We also show that our approach to prefetching allows us to tolerate a smaller cache that produces more conflicts but saves energy consumption. By further partitioning region-based caches, we can reduce MCPI by up to 40% or save over 70% in energy consumption.

Chapter 7 offers a discussion on improving data locality by relocating misplaced data to the appropriate region. We show how, given the appropriate compiler feedback, we can identify and reallocate these data with a small amount of overhead. A case study of the quicksort application demonstrates the effectiveness of data relocation for eliminating unnecessary cache misses.

8.2 Future Directions

One possible area for future work is the refinement of our heap caching policy. We believe that the instruction-based selection of hot heap data may ultimately hold more promise than the data-centric approach and plan to explore this topic further. We also wish to investigate dynamic techniques to identify hot heap data so that we can account for differing behavior across program phases. Finally, the studies we ran using Cheetah suggest we can significantly lower the heap cache miss rate by reducing conflicts within it. Improving the data layout in that cache could significantly improve performance.

We believe there is also ample room to further explore prefetching in region-based caches. We would like to evaluate our prefetch methods in memory-intensive applications to see if the promise shown in Chapter 6 extends to applications that do not cache as well. In addition, we want to develop prefetch algorithms that are tailored to the reference characteristics of a particular region. The stack shows the most promise, as its accesses are regular and easily predictable. We can avoid write misses when a stack frame is first allocated by validating the cache lines covered by that frame; since the program will write new data into those locations, the contents of the cache do not matter. Actual prefetches are necessary once the stack grows larger than its dedicated cache; at this point, new frames overwrite old data, and we can prefetch that data when the offending frame is deallocated to ensure that the stack experiences fewer misses as it returns from function calls. Further analysis of reference characteristics in other regions may yield prefetch strategies for those areas as well.

A final direction for future work is to implement the compiler support required for each of these hardware techniques. This dissertation assumes the existence of a compiler that, using profile-directed feedback, can identify data locality characteristics and reallocate that data appropriately, whether it be to a second heap, a separate prefetch region, or one of the existing regions. In reality, this tool does not exist, and designing it is a non-trivial task worthy of substantial research.

BIBLIOGRAPHY

[1] Jaume Abella and Antonio Gonzalez. Power Efficient Data Cache Designs. *Proceedings of the 21st International Conference on Computer Design*, pp. 8-13, San Jose, California, October 2003.

[2] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures. *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pp. 248-259, Vancouver, British Columbia, June 2000.

[3] Anastassia Ailamaki, David J. DeWitt, and Mark D. Hill. Data Page Layouts for Relational Databases on Deep Memory Hierarchies. *International Journal on Very Large Databases*, Vol. 11, No. 3, pp. 198-215, November 2002.

[4] David H. Albonesi. Selective Cache Ways: On-Demand Cache Resource Allocation. *Proceedings of the 32nd International Symposium on Microarchitecture*, pp. 248–259, Haifa, Israel, November 1999.

[5] P. V. Argade, et al. Hobbit: A High-Performance, Low-Power Microprocessor. *Compcon Spring '93, Digest of Papers*, pp. 88-93, February 1993.

[6] Raksit Ashok, Saurabh Chheda, and Csaba Andras Moritz. Cool-Mem: Combining Statically Speculative Memory Accessing with Selective Address Translation for Energy Efficiency. *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 133-143, October 2002.

[7] Rajeev Balasubramonian, David Albonesi, Alper Buyuktosunoglu, and Sandhya Dwarkadas. Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures. *Proceedings of the 33rd International Symposium on Microarchitecture*, pp. 245-257, Monterey, California, December 2000.

[8] Laszlo A. Belady. A Study of Replacement Algorithms for a Virtual-Storage Computer. *IBM Systems Journal*, Vol. 5, No. 2, pp. 78-101, 1966.

[9] Alan D. Berenbaum, Brian W. Colbry, David R. Ditzel, R. Don Freeman, Hubert R. McLellan, Kevin J. O'Connor, and Masakazu Shoji. CRISP: A Pipelined 32-bit Microprocessor with 13-kbit of Cache Memory. *IEEE Journal of Solid-State Circuits*, Vol. SC-22, No. 5, pp. 776-782, October 1987.

[10] Azeez J. Bhavnagarwala, Stephen V. Kosonocky, Michael Immediato, Dan Knebel, and Anne-Marie Haen. A Pico-Joule Class, 1 GHz, 32 kB x 64b DSP SRAM with Self Reverse Bias. *Proceedings of the IEEE Symposium on VLSI Circuits*, pp. 251-252, Kyoto, Japan, June 2003.

[11] Russell P. Blake. Exploring a Stack Architecture. *IEEE Computer*, Vol. 10, No. 5, pp. 30-39, May 1977.

[12] Jayaram Bobba, Michelle Moravan, and Umair Saeed. TAP: Taxonomy for Adaptive Prefetching. Project report for CS/ECE 752: Advanced Computer Architecture I, University of Wisconsin-Madison, December 2004.

[13] Mark W. Brehob. On the Mathematics of Caching. Doctoral Dissertation, Michigan State University, 2003.

[14] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. *Proceedings of the 27th International Symposium on Computer Architecture*, pp. 83-94, Vancouver, British Columbia, June 2000.

[15] Edouard Bugnion, Jennifer M. Anderson, Todd C. Mowry, Mendel Rosenblum, and Monica S. Lam. Compiler-Directed Page Coloring for Multiprocessors. *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 244-255, Cambridge, Massachusetts, October 1996.

[16] Doug C. Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin-Madison, June 1997.

[17] Brendon Cahoon and Kathryn S. McKinley. Data Flow Analysis for Software Prefetching Linked Data Structures in Java. *Proceedings of the 2001 14th International Conference on Parallel Architectures and Compiler Techniques*, pp. 280-291, Barcelona, Spain, September 2001.

[18] Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. Cache-Conscious Data Placement. *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 139-149, San Jose, California, October 1998.

[19] David Callahan, Ken Kennedy, and Allan Porterfield. Software Prefetching. *Proceedings of the 4th International Conference on Parallel Architectures and Compiler Techniques*, pp. 40-52, Santa Clara, California, April 1991.

[20] Yen-Jen Chang, Shanq-Jang Ruan, and Feipei Lai. Design and Analysis of Low-Power Cache Using Two-Level Filter Scheme. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 11, No. 4, pp. 568-580, August 2003.

[21] Tien-Fu Chen and Jean-Loup Baer. Effective Hardware-Based Data Prefetching for High-Performance Processors. *IEEE Transactions on Computers*, Vol. 44, No. 5, pp. 609-623, May 1995.

[22] Trishul M. Chilimbi and James R. Larus. Using Generational Garbage Collection to Implement Cache-Conscious Data Placement. *Proceedings of the 1st International Symposium on Memory Management*, pp. 37-48, Vancouver, British Columbia, October 1998.

[23] Trishul M. Chilimbi, Bob Davidson, and James R. Larus. Cache-Conscious Structure Definition. *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 13-24, Atlanta, Georgia, May 1999.

[24] Trishul M. Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 191-202, Snowbird, Utah, June 2001.

[25] Sangyeun Cho, Pen-Chung Yew, and Gyungho Lee. Decoupling Local Variable Accesses in a Wide-Issue Superscalar Processor. *Proceedings of the 26th International Symposium on Computer Architecture*, pp. 100-110, Atlanta, Georgia, May 1999.

[26] Jamison Collins, Suleyman Sair, Brad Calder, and Dean M. Tullsen. Pointer Cache Assisted Prefetching. *Proceedings of the 35th International Symposium on Microarchitecture*, pp. 62-73, Istanbul, Turkey, November 2002.

[27] Fredrik Dahlgren and Per Stenström. On Reconfigurable On-Chip Data Caches. *Proceedings of the 24th International Symposium on Microarchitecture*, pp. 189-198, Albuquerque, New Mexico, November 1991.

[28] Chen Ding and Ken Kennedy. Improving Cache Performance in Dynamic Applications through Data and Computation Reorganization at Run-Time. *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 229-241, Atlanta, Georgia, May 1999.

[29] David R. Ditzel and H. R. McLellan. Register Allocation for Free: The C Machine Stack Cache. *Proceedings of the 1st International Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 48-56, March 1982.

[30] The EDN Embedded Microprocessor Benchmark Consortium. http://www.eembc.org.

[31] Krisztián Flautner, Nam Sung Kim, Steve Martin, David Blaauw, and Trevor Mudge. Drowsy Caches: Simple Techniques for Reducing Leakage Power. *Proceedings of the 29th International Symposium on Computer Architecture*, pp. 147-157, May 2002.

[32] Michael J. Geiger and Gary S. Tyson. Reducing Static Power Dissipation in Region-Based Caches. *Proceedings of the 1st Watson Conference on Interaction between Architecture, Circuits, and Compilers (P=ac^2)*, pp. 55-62, Yorktown Heights, New York, October 2004.

[33] Michael J. Geiger, Sally A. McKee, and Gary S. Tyson. Drowsy Region-Based Caches: Minimizing Both Dynamic and Static Power Dissipation. *Proceedings of ACM Computing Frontiers*, pp. 378-384, Ischia, Italy, May 2005.

[34] Michael J. Geiger, Sally A. McKee, and Gary S. Tyson. Beyond Basic Region Caching: Specializing Cache Structures for High Performance and Energy Conservation. *Proceedings of the 1st International Conference on High Performance Embedded Architectures and Compilers (HiPEAC '05)*, pp. 102-115, Barcelona, Spain, November 2005.

[35] Michael J. Geiger, Sally A. McKee, and Gary S. Tyson. Specializing Cache Structures for High Performance and Energy Conservation in Embedded Systems. To appear in *Transactions on High Performance Embedded Architectures and Compilers*, Vol. 1, No. 1.

[36] Kanad Ghose and Milind B. Kamble. Reducing Power in Superscalar Processor Caches using Subbanking, Multiple Line Buffers and Bit-Line Segmentation. *Proceedings of the 1999 International Symposium on Low Power Electronics and Design*, pp. 70-75, San Diego, California, August 1999.

[37] J.D. Gindele. Buffer Block Prefetching Method. *IBM Technical Disclosure Bulletin*, Vol. 20, No. 2, pp. 696-697, July 1977.

[38] Ricardo Gonzales and Mark Horowitz. Energy Dissipation in General Purpose Microprocessors. *IEEE Journal of Solid State Circuits*, Vol. 31, No. 9, pp. 1277-1284, September 1996.

[39] Antonio Gonzalez, Carlos Aliagas, and Mateo Valero. A Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality. *Proceedings of the 9th International Conference on Supercomputing*, Barcelona, Spain, pp. 338-347, July 1995.

[40] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. *Proceedings of the 4th IEEE Workshop on Workload Characterization*, pp. 3-14, Austin, Texas, December 2001.

[41] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, third edition, 2002.

[42] Jeyran Hezavei, N. Vijaykrishnan, M. J. Irwin. A Comparative Study of Power Efficient SRAM Designs. *Proceedings of the 10th Great Lakes Symposium on VLSI*, pp. 117-122, Evanston, Illinois, March 2000.

[43] Lisa R. Hsu, Steven K. Reinhardt, Ravishankar Iyer, and Srihari Makineni. Communist, Utilitarian, and Capitalist Cache Policies on CMPs: Caches as a Shared Resource. *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, Seattle, Washington, September 2006.

[44] Zhigang Hu, Stefanos Kaxiras, and Margaret Martonosi. Improving Cache Power Efficiency with an Asymmetric Set-Associative Cache. *Workshop on Memory Performance Issues (in conjunction with ISCA-28),* June 2001.

[45] Zhigang Hu, Margaret Martonosi, and Stefanos Kaxiras. TCP: Tag Correlating Prefetchers. *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, pp. 317-326, Anaheim, California, February 2003.

[46] Michael Huang, Jose Renau, Seung-Moon Yoo, and Josep Torellas. L1 Data Cache Decomposition for Energy Efficiency. *Proceedings of the 2001 International Symposium on Low Power Electronics and Design*, pp. 10-15, Huntington Beach, California, August 2001.

[47] Sorin Iacobovici, Lawrence Spracklen, Sudarshan Kadambi, Yuan Chou, and Santosh G. Abraham. Effective Stream-Based and Execution-Based Data Prefetching. *Proceedings of the 18th International Conference on Supercomputing*, Saint-Malo, France, pp. 1-11, June 2004.

[48] Intel Corporation. "Intel StrongARM SA-1110 Microprocessor Developer's Manual." http://developer.intel.com/design/strong/manuals/278240.htm.

[49] Intel Corporation. "The Intel XScale Microarchitecture Technical Summary." http://www.intel.com/design/intelxscale/xscaledatasheet4.htm.

[50] Ravi Iyer. CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms. *Proceedings of the 18th Annual Conference on Supercomputing*, pp. 257-266, Malo, France, June 2004.

[51] Lizy Kurian John and Akila Subramanian. Design and Performance Evaluation of a Cache Assist to implement Selective Caching. *Proceedings of the 1997 IEEE International Conference on Computer Design*, pp. 510-518, Austin, Texas, October 1997.

[52] Teresa L. Johnson and Wen-Mei W. Hwu. Run-time Adaptive Cache Hierarchy Management via Reference Analysis. *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 315-326, Denver, Colorado, June 1997.

[53] Teresa L. Johnson, Matthew C. Merten, and Wen-Mei W. Hwu. Run-time Spatial Locality Detection and Optimization. *Proceedings of the 30th International Symposium on Microarchitecture*, pp. 57-64, Research Triangle Park, NC, December 1997.

[54] Doug Joseph and Dirk Grunwald. Prefetching Using Markov Predictors. *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 252-263, Denver, Colorado, June 1997.

[55] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 364-373, Seattle, Washington, June 1990.

[56] Stefanos Kaxiras, Zhigang Hu, and Margaret Martonosi. Cache decay: Exploiting generational behavior to reduce cache leakage power. *Proceedings of the 28th International Symposium on Computer Architecture*, pp. 240-251, Göteborg, Sweden, June 2001.

[57] Nam Sung Kim, Krisztián Flautner, David Blaauw, and Trevor Mudge. Drowsy Instruction Caches: Leakage Power Reduction using Dynamic Voltage Scaling and Cache Sub-bank Prediction. *Proceedings of the 35th International Symposium on Microarchitecure*, pp. 219-230, Istanbul, Turkey, November 2002.

[58] Nam Sung Kim, Krisztián Flautner, David Blaauw, and Trevor Mudge. Circuit and microarchitectural techniques for reducing cache leakage power. *IEEE Transactions on VLSI*, Vol. 12, No. 2, pp. 167-184, February 2004.

[59] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques*, pp. 111-122, Antibes Juan-les-Pins, France, September 2004.

[60] Johnson Kin, Munish Gupta, and William H. Mangione-Smith. Filtering Memory References to Increase Energy Efficiency. *IEEE Transactions on Computers*, Vol. 49, No. 1, pp. 1-15, January 2000.

[61] Thomas Kistler and Michael Franz. Automated Data-Member Layout of Heap Objects to Improve Memory-Hierarchy Performance. *ACM Transactions on Programming Languages and Systems*, Vol. 22, No. 3, pp. 490-505, May 2000.

[62] Thomas Kistler and Michael Franz. Continuous Program Optimization: A Case Study. *ACM Transactions on Programming Languages and Systems*, Vol. 25, No. 4, pp. 500-548, July 2003.

[63] Alexander C. Klaiber and Henry M. Levy. An Architecture for Software-Controlled Data Prefetching. *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 43-53, Toronto, Ontario, May 1991.

[64] Sanjeev Kumar and Christopher Wilkerson. Exploiting Spatial Locality in Data Caches using Spatial Footprints. *Proceedings of the 25th International Symposium on Computer Architecture*, pp. 357-368, Barcelona, Spain, June 1998.

[65] An-Chow Lai, Cem Fide, and Babak Falsafi. Dead-Block Prediction and Dead-Block Correlating Prefetchers. *Proceedings of the 28th International Symposium on Computer Architecture*, pp. 144-154, Göteborg, Sweden, June 2001.

[66] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: A Tool for Evaluating Multimedia and Communications Systems. *Proceedings of the 30th International Symposium on Microarchitecture*, pp. 330-335, Research Triangle Park, North Carolina, December 1997.

[67] Hsien-Hsin S. Lee and Gary S. Tyson. Region-Based Caching: An Energy-Delay Efficient Memory Architecture for Embedded Processors. *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pp. 120-127, San Jose, California, November 2000.

[68] Hsien-Hsin S. Lee, Mikhail Smelyanski, Chris J. Newburn, and Gary S. Tyson. Stack Value File: Custom Microarchitecture for the Stack. *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, pp. 5-14, Monterrey, Mexico, January 2001.

[69] Hsien-Hsin S. Lee. Improving Energy and Performance of Data Cache Architectures by Exploiting Memory Reference Characteristics. Doctoral Dissertation, The University of Michigan, 2001.

[70] Hsien-Hsin S. Lee and Chinnakrishnan S. Ballapuram. Energy Efficient D-TLB and Data Cache using Semantic-Aware Multilateral Partitioning. *Proceedings of the 2003 International Symposium on Low Power Electronics and Design*, pp. 306-311, Seoul, Korea, August 2003.

[71] Hsien-Hsin S. Lee, Joshua B. Fryman, A. Utku Diril, and Yuvraj S. Dhillon. The Elusive Metric for Low-Power Architecture Research. *Workshop on Complexity-Effective Design*, San Diego, California, June 2003.

[72] Jung-Hoon Lee, Shin-Dug Kim, and Charles Weems. Application-Adaptive Intelligent Cache Memory System. *ACM Transactions on Embedded Computing Systems*, Vol. 1, No. 1, pp. 56-78, November 2002.

[73] Lin Li, Ismail Kadayif, Yuh-Fang Tsai, Narayanan Vijaykrishnan, Mahmut Kandemir, Mary Jane Irwin and Anand Sivasubramaniam. Leakage Energy Management in Cache Hierarchies. *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, pp. 131-140, Charlottesville, Virginia, September 2002.

[74] Afzal Malik, Bill Moyer, and Dan Cermak. A Programmable Unified Cache Architecture for Embedded Applications. *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pp. 165-171, San Jose, California, November 2000.

[75] Doug Matzke, "Will Physical Scalability Sabotage Performance Gains?" *IEEE Computer*, Vol. 30, No. 9, pp. 37-39, September 1997.

[76] Sally A. McKee. Reflections on the Memory Wall. *Proceedings of ACM Computing Frontiers,* Ischia, Italy, April 2004.

[77] James Montanaro, et al. A 160-MHz, 32-b, 0.5-W CMOS RISC Microprocessor. *Digital Technical Journal,* Vol. 9, No. 1, pp. 49-62, January 1997.

[78] Kyle J. Nesbit and James E. Smith. Data Cache Prefetching Using a Global History Buffer. *IEEE Micro,* Vol. 25, No. 1, pp. 90-97, January 2005.

[79] K. Nii, et al. A low power SRAM using auto-backgate-controlled MT-CMOS. *Proceedings of the 1998 International Symposium on Low Power Electronics and Design,* pp. 293-298, Monterey, California, August 1998.

[80] Daniel Ortega, Eduard Ayguadé, Jean-Loup Baer, and Mateo Valero. Cost-Effective Compiler Directed Memory Prefetching and Bypassing. *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, pp. 189-198, Charlottesville, Virginia, September 2002.

[81] Subbarao Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. *Proceedings of the 21st International Symposium on Computer Architecture*, pp. 24-33, Chicago, Illinois, April 1994.

[82] Subbarao Palacharla, Norman P. Jouppi and J.E. Smith. Complexity-Effective Superscalar Processors. *Proceedings of the 24th Annual International Symposium on Computer Architecture,* pp. 206-218, Denver, Colorado, June 1997.

[83] Krishna V. Palem, Rodric M. Rabbah, Vincent J. Mooney III, Pinar Korkmaz, Kiran Puttaswamy. Design Space Optimization of Embedded Memory Systems via Data Remapping. *ACM SIGPLAN Notices,* Vol. 37, No. 7, pp. 28-37, July 2002.

[84] Dharmesh Parikh, Yan Zhang, Karthik Sankaranarayanan, Kevin Skadron, and Mircea Stan. Comparison of State-Preserving vs. Non-State-Preserving Leakage Control in Caches. *Proceedings of the 2nd Annual Workshop on Duplicating, Deconstructing, and Debunking*, pp. 14-24, San Diego, California, June 2003.

[85] Jih-Kwon Peir, Yongjoon Lee, and Windsor W. Hsu. Capturing Dynamic Reference Behavior with Adaptive Cache Topology. *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems,* pp. 240-250, San Jose, California, October 1998.

[86] Peter Petrov and Alex Orailoglu. Performance and Power Effectiveness in Embedded Processors—Customizable Partitioned Caches. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 20, No. 11, pp. 1309-1318, November 2001.

[87] Fred Pollack. New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies. MICRO-32 Keynote Speech, November, 1999. http://www.intel.com/research/mrl/library/micro32keynote.pdf.

[88] Michael Powell, Se-Hyun Yang, Babak Falsafi, Kaushik Roy, T.N. Vijaykumar. Gated-Vdd: A circuit technique to reduce leakage in deep-submicron cache memories. *Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 90-95, Rapallo, Italy, July 2000.

[89] Parthasarathy Ranganathan, Sarita Adve, and Norman P. Jouppi. Reconfigurable Caches and Their Application to Media Processing. *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pp. 214-224, Vancouver, British Columbia, June 2000.

[90] Gabriel Rivera and Chau-Wen Tseng. Data Transformations for Eliminating Conflict Misses. *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 38-49, Montreal, Quebec, June 1998.

[91] Amir Roth and Gurindar S. Sohi. Effective Jump-Pointer Prefetching for Linked Data Structures. *Proceedings of the 26th International Symposium on Computer Architecture,* pp. 111-121, Atlanta, Georgia, May 1999.

[92] Julio Sahuquillo and Ana Pont. Splitting the Data Cache: A Survey. *IEEE Concurrency*, Vol. 8, No. 3, pp. 30-35, July-September 2000.

[93] Timothy Sherwood, Brad Calder, and Joel Emer. Reducing Cache Misses using Hardware and Software Page Placement. *Proceedings of the 1999 International Conference on Supercomputing*, pp. 155-164, Rhodes, Greece, June 1999.

[94] Timothy Sherwood, Suleyman Sair, and Brad Calder. Predictor-Directed Stream Buffers. *Proceedings of the 33rd International Symposium on Microarchitecture*, pp. 42-53, Monterey, California, December 2000.

[95] Premkishore Shivakumar and Norman P. Jouppi. Cacti 3.0: An Integrated Cache Timing, Power, and Area Model. *Compaq Western Research Lab Technical Report 2001/2*, August 2001.

[96] Alan Jay Smith. Cache memories. *ACM Computing Surveys*, Vol. 14, No. 3, pp. 473-530, September 1982.

[97] Yan Solihin, Jaejin Lee, and Josep Torellas. Using a User-Level Memory Thread for Correlation Prefetching. *Proceedings of the 29th International Symposium on Computer Architecture*, pp. 171-182, Anchorage, Alaska, May 2002.

[98] Srikanth T. Srinivasan and Alvin R. Lebeck. Load Latency Tolerance in Dynamically Scheduled Processors. *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pp. 148-159, Dallas, Texas, December 1998.

[99] Srikanth T. Srinivasan, Roy Dz-ching Ju, Alvin R. Lebeck, and Chris Wilkerson. Locality vs. Criticality. *Proceedings of the 28th International Symposium on Computer Architecture*, pp. 132-143, Göteborg, Sweden, June 2001.

[100] Viji Srinivasan, Edward S. Davidson, and Gary S. Tyson. A Prefetch Taxonomy. *IEEE Transactions on Computers*, Vol. 53, No. 2, pp. 126-140, February 2004.

[101] Standard Performance Evaluation Corporation. SPEC CPU2000 Benchmarks. http://www.spec.org/cpu2000/, 2000.

[102] Ching-Long Su and Alvin M. Despain. Cache Designs for Energy Efficiency. *Proceedings of the 28th Hawaii International Conference on System Science*, pp. 306-315, Maui, Hawaii, January 1995.

[103] Rabin A. Sugumar and Santosh G. Abraham. Efficient Simulation of Multiple Cache Configurations Using Binomial Trees. Technical Report CSE-TR-111-91, CSE Division, University of Michigan, 1991.

[104] G. Edward Suh, Srinivas Devadas, and Larry Rudolph. Analytic Cache Models with Applications to Cache Partitioning. *Proceedings of the 15th International Conference on Supercomputing*, pp. 1-12, Sorrento, Italy, June 2001.

[105] Gary Tyson, Matthew Farrens, John Matthews, and Andrew Pleszkun. A Modified Approach to Data Cache Management. *Proceedings of the 28th International Symposium on Microarchitecture*, pp. 93-103, Ann Arbor, Michigan, December 1995.

[106] Steven P. Vanderwiel and David J. Lilja. Data Prefetch Mechanisms. *ACM Computing Surveys*, Vol. 32, No. 2, pp. 174-199, June 2000.

[107] Zhenlin Wang, Doug Burger, Kathryn S. McKinley, Steven K. Reinhardt, and Charles C. Weems. Guided Region Prefetching: A Cooperative Hardware/Software Approach. *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pp. 388-398, San Diego, California, June 2003.

[108] William A. Wulf and Sally A. McKee. Hitting the Memory Wall: Implications of the Obvious. *ACM SIGArch Computer Architecture News*, Vol. 23, No. 1, pp. 20-24, March 1995.

[109] Masanao Yamaoka, Yoshihiro Shinozaki, Noriaki Maeda, Yasuhisa Shimazaki, Kei Kato, Shigeru Shimada, Kazumasa Yanagisawa, and Kenichi Osada. A 300MHz 25µA/Mb Leakage On-Chip SRAM Module Featuring Process-Variation Immunity and Low-Leakage-Active Mode for Mobile-Phone Application Processor. *Proceedings of the 2004 IEEE International Solid-State Circuits Conference*, pp. 494-495, San Francisco, California, February 2004.

[110] Se-Hyun Yang, Michael D. Powell, Babak Falsafi, Kavshik Roy, and T.N. Vijaykumar. An Integrated Circuit/Architecture Approach to Reducing Leakage in Deep-Submicron High-Performance I-Caches. *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, pp. 147-158, Monterrey, Mexico, January 2001.

[111] Se-Hyun Yang, Michael Powell, Babak Falsafi, and T.N. Vijaykumar. Exploiting Choice in Resizable Cache Design to Optimize Deep-Submicron Processor Energy-Delay. *Proceedings of the 8th International Symposium on High-Performance Computer Architecture,* pp. 147-158, Boston, Massachusetts, February 2002.

[112] Yan Zhang, Dharmesh Parikh, Karthik Sankaranarayanan, Kevin Skadron, and Mircea Stan. HotLeakage: A Temperature-Aware Model of Subthreshold and Gate Leakage for Architects. Technical Report CS-2003-05, University of Virginia Department of Computer Science, March 2003.

[113] Youtao Zhang and Rajiv Gupta. Enabling Partial Cache Line Prefetching Through Data Compression. *Proceedings of the International Conference on Parallel Processing,* pp. 277-285, Kaohsiung, Taiwan, October 2003.

[114] Huiyang Zhou, Mark Toburen, Eric Rotenberg, and Tom Conte. Adaptive mode-control: A static-power-efficient cache design. *Proceedings of the 14th International Conference on Parallel Architecture and Compilation Techniques*, pp. 61-70, Barcelona, Spain, September 2001.