IMPROVING ENERGY AND PERFORMANCE OF DATA CACHE ARCHITECTURES BY EXPLOITING MEMORY REFERENCE CHARACTERISTICS

by

Hsien-Hsin Sean Lee

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
2001

ABSTRACT

IMPROVING ENERGY AND PERFORMANCE OF DATA CACHE ARCHITECTURES BY EXPLOITING MEMORY REFERENCE CHARACTERISTICS

by
Hsien-Hsin Sean Lee

Chairperson: Gary S. Tyson

Minimizing power, increasing performance, and delivering effective memory bandwidth are today's primary microprocessor design goals for the embedded, high-end and multimedia workstation markets. In this dissertation, I will discuss three major data cache architecture design optimization techniques, each of which exploits the data memory reference characteristics of the applications written in high-level languages. Through a better understanding of the memory reference behavior, we can design a system that executes at higher performance, while consuming less energy, and delivering more effective memory bandwidth.

The first part of this dissertation presents an in-depth characterization of data memory references, including analysis of semantic region accesses and behavior of data stores. This analysis leads to a new organization of the data cache hierarchy called Region-based Cachelets. Region-based Cachelets are capable of improving memory performance of embedded applications while significantly reducing dynamic energy consumption, resulting in a 50% to 70% improvement in energy-delay product efficiency using this approach.

Following this, I will discuss a new cache-like structure, the Stack Value File (or SVF), which boosts performance of general purpose applications by routing stack data references to a separate storage structure optimized for the unique characteristics of the stack reference substream. By utilizing a custom structure for stack references, we are able to increase memory level parallelism, reduce memory latency, and reduce off-chip memory activity. The performance can be improved by 24% by implementing an 8KB SVF for a processor with a dual-ported L1 cache.

Finally, I will address memory bandwidth issues by proposing a new write policy called Eager Writeback which can effectively improve overall system performance by shifting the writings of dirty cache lines from on-demand to times when the memory bus is less congested. It lessens the criticality of on-demand misses and improves performance by 6% to 16% for the 3D graphics geometry pipeline.

To my parents, Mr. C.-H. Lee and Mrs. C.-H. Liko.

ACKNOWLEDGEMENTS

This dissertation seemed to be an endless pilgrimage for me in the beginning. Without mentors and friends, I could not reach my destination. It is my privilege to have Gary Tyson as my thesis advisor. Gary shares with me his many research ideas, presentation skills, friendship, and house building tips, not only makes my Ph.D. research fruitful but also enjoyable. I am greatly indebted to Ed Davidson, who guided me into the dome of computer architectures in 1993. He also served as my first research advisor and provided me a research assistantship from 1993 to 1995. I also want to express my maximum gratitude to my former manager also a principal architect at Intel Corporation, Vladimir Pentkovski, who taught me how to be skeptical of anything we read and how to become a good researcher. I thank Todd Austin, Richard Brown, and Steven Reinhardt for serving in my thesis committee and providing many precious comments in my work. I also thank Matt Farrens and Chris "CJ" Newburn for their insightful discussions during our cooperation. I thank Jesse Fang for the Intel Fellowship during my last year at Michigan.

There are many friends at Michigan who continue to keep me sane from various stages of my wonderful two-episode doctorate journey. People from episode one (1992 to 1995), Chih-Chieh Lee, Chien-Chang Liu, Jude Rivers, Tien-Pao Shih, I-Cheng Chen, Perry Wang, John-David Wellman, Chuan-Hua Chang, Hung-Kuei Ku, Jia-Jang Liou, Eric Hao, Eric Boyd, Po-Yung Chang, Waleed Meleis, Alexandre Eichenberger, Karen Tomko, Gheith Abandah and Shih-Hao Hung. People from episode two (1999-2001): Viji Srinivasan, Stevan Vlaovic, Murali Annavaram, Mikhail Smelyanskiy, Suhwan Kim, Wei-Fen Lin, Brian Hutsell, Nagarajan Kandasamy, Ramu Pyreddy, Lisa Wu, Xun Liu, Shao-Lun Li and Shuo-Hung Hsu.

Special thanks to my college roommates Shyh-Kang Chang and Chi-Wei Wang, who convinced me to abandon my original plan of going to Wisconsin and join their adventures at Michigan in order to share an apartment again.

I thank all of my brilliant Intel friends including Emile Hsieh, Wen-Hsun Han, Youfeng Wu, Shih-Wei Liao, Hong Wang, Paul Zagacki, Chung-Chi Wang, Steve Tsai, Trung Diep, Deep Buch and Mike Dwyer, whom I learned a lot from. In particular, my best friend and also my closest partner in the entire Pentium III development, Hsien-Cheng "Emile" Hsieh, who passed away just 3 days after I finished my oral defense and left me so much grief and regret not even having a chance to share my accomplishment with him. Emile, the work we had done together will be the best treasure of my life and you will always live in my heart.

I thank my high school and college buddies, Bin-Hung Lin, Yung-Fa Chou, Chien-Liang Lin, Hao-Chao Hong, Hsiao-Tzu Ni, Wan-Thai Hsu and Tsung-Lin Cheng, who give me hope and inspiration constantly and galvanize me with their wild ideas once a while.

Finally, I would like to thank my family for their unconditional support. I thank my lovely wife Esther Liu, how lucky I am to have you with me for all these years. I thank my younger sister Hui-Yin Lee who always cheers me up during my ups and downs. I thank my parents Ching-Hsiung Lee and Chung-Hua Liko for giving me love and providing whatever I need since day one.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF APPENDICES

CHAPTER 1

INTRODUCTION

Following the rapid progression and innovation in the semiconductor industry, the microprocessor market is gradually converging into two major segments: The first segment is driven by traditional high-performance microprocessors used in information servers, scientific computation, simulation-based engineering and ever complicated content-rich multimedia and graphics-oriented applications. The second segment is growing from the emerging application market in embedded and special purpose processors, e.g. digital signal processing (DSP) processors. These processors are omnipresent in our daily life, from mobile computers, personal digital assistants (PDA), multi-purpose handsets, tablet computers, etc., to invisible microcontrollers embedded in household appliances. The design focuses of this processor category are faster design cycle for time-to-market, low energy consumption, more constrained die area budget, and flexible integration capability, all while achieving reasonable performance, making them much different from a traditional high-performance processor design.

Processor architects and researchers in the past decade have been primarily focused on the following design space: instruction level parallelism, memory level parallelism, memory throughput, and low energy designs. Evaluation metrics emphasis were also varied from cycles per instruction (CPI), instructions per cycle (IPC) to more recent energy-delay product (EDP), to emphasize different design goals for different market segments at different times.

Entering the 1990s, the per-transistor cost was no longer a roadblock to complex designs due to significant advancements in semiconductor manufacturing and design technology, processor architects were given an enormous number of transistors on a single processor chip to devote to extracting instruction level parallelism (ILP) and enhancing the locality of memory references. Superscalar architectures and RISC design philosophy [60] were accepted as the fundamental elements for building high-performance ILP processors. In the domain of a dynamic ILP processor design, the goal is to maximize the number of instructions supplied per cycle from the front-end pipeline. Innovative microarchitectural technologies towards this end were proposed and several of them have been implemented in commercial microprocessors to provide an effective instruction supply to the execution core; these include aggressive branch predictors [81][84][104][120], trace caches [54][86][95][115], value predictors [46][79] and simultaneous multithreading [38][109] . All of these technologies aim at increasing instruction throughput at an extra hardware cost, either from a single program or from multiple concurrent program threads. In addition, new types of instructions such as predicate instructions [55][31][33], memory hint [34][96], or branch hint [33] instructions were incorporated into recent instruction set architectures. Meanwhile, several ILP limit studies [20][69][73][88][112] were published to address the limits and potential issues of inherent instruction level parallelism, primarily for integer applications that were known for their high frequency of branch instructions and scanty instruction level parallelism.

Memory subsystem designs have also been extensively studied including new cache organizations and replacement policies for more efficient and effective instruction and data management. In the early days of single-issue processor designs, traditional caches were implemented for reducing ever higher memory latency. Non-blocking caches were later proposed in [67] to increase execution concurrency of the instructions that are independent of in-flight cache misses. As the deviation between processor core speed and DRAM speed increases, multi-level caches are becoming increasingly popular. Today, several commercial high performance microprocessors are designed with three level caches [39][40][50][110]. Even though memory latency cannot be completely eliminated, researchers have developed intelligent hardware and software-based data prefetching mechanisms [23][26] as a means to improving memory level parallelism by latency hiding. Recently researchers have started to investigate new techniques for exploiting memory reference locality from the perspective of data temporality, through hardware and/or software modifications. Techniques proposed in [49][61][93] classify data based on their temporality of prior reference patterns and then deposit data of different characteristics into distinct cache/memory structures. Throughout this dynamic classification, these techniques are able to alleviate the effects of conflict misses and improve hit ratios by preferentially retaining temporal data, i.e. data exhibiting greater likelihood of reuse, in the main cache. Several software techniques were also proposed to improve data layout. They usually rely on statistics of program profiling or affinity region information provided by programmers to re-structure data in virtual memory space. Based on this information, compilers can generate a data layout with better reference locality. For example, cache-conscious data placement, proposed in [22], profiles run-time data affinity for a program and then rearranges data at linker time to achieve better reference locality. Similarly, array grouping techniques [100] mitigate local cache misses and interprocessor communication latency by packing non-consecutive references together for parallel applications.

Another major issue in the memory subsystem design is *bandwidth*. Most of the multimedia, telecommunication and future emerging applications stream a large amount of data to/from memory. Network-centric applications are essentially bandwidth-bound. These data streams can easily saturate memory bus bandwidth. These new applications are the major driving force behind developing novel high memory bandwidth techniques. The Accelerated Graphics Port (AGP) [29] was proposed for desktop processors to enhance the memory bandwidth between system memory and a graphics processor, primarily for delivering graphics commands and texture bitmaps. High bandwidth Rambus DRAM (RDRAM) [36], Synchronous-Link DRAM (SLDRAM) [57] and Double Data Rate DRAM (DDR-DRAM) are now accepted as new industry standards for future system main memory devices. Given a fixed amount of memory bandwidth, architects can also potentially improve system performance through more intelligent bandwidth management mechanisms that lead to more effective bandwidth utilization.

The philosophy of this dissertation is to address these design issues (energy consumption, high performance and high memory bandwidth) by analyzing and exploiting characteristics of data memory reference streams. By examining these characteristics, we propose architectural solutions to attack each problem individually. In the first part of this dissertation, we present an in-depth characterization of data memory references, including analysis of semantic region references and behavior of data stores. The first part of the analysis leads to two new cache structures designed to exploit the characteristics of high-level programming language semantics which, by convention, partitions run-time code and data into multiple separate virtual memory regions. The Region-based Cachelets design, our first novel cache architecture, is capable of improving memory performance of embedded applications while significantly reducing dynamic energy consumption. The second technique, the Stack Value File, can increase the performance of general purpose applications by routing stack data references to a separate register file-like storage structure optimized for the unique characteristics of the stack reference substream. This technique can increase memory level parallelism, reduce memory latency, and reduce off-chip memory activity. Then, we analyze the data reference behavior in a set-associative cache for data streaming applications. We present a novel technique called Eager

Writeback that effectively exploits the data reference features and improves the system performance by shifting dirty cache line writebacks and balancing memory bandwidth accordingly.

1.1 Contributions

This dissertation makes the following key contributions:

1. It investigates data memory reference behavior through the partitioning provided by programming language semantics [74]. The resulting characterization and its in-depth analysis quantitatively demonstrate the information content carried in memory address streams for each data region.

2. It applies the characterization methodology to some emerging media and communication applications. Based on the stack and global data memory reference characteristics, a new design concept called *Region-based Cachelets* [76] is proposed for low-energy embedded processor designs. This new design scheme physically divides the data cache into parallel cachelets based on the regions defined by high-level programming languages. We show that the region-based cachelet design can reduce power consumption substantially without compromising execution performance for embedded processors.

3. It examines the memory performance impact of programming language semantics and addressing modes. A new technique, called *Stack Value File (SVF)* [75] is proposed that uses a directly indexed circular register file effectively exploit the unique characteristics shown in stack memory reference stream.

4. It presents an *Eager Writeback cache* [77] for alleviating memory bandwidth constraints based on our analysis of data cache replacement streams in a set-associative cache. Eager Writeback is a modified writeback cache replacement scheme for achieving more effective utilization of memory bandwidth, thereby improving system performance for today's memory bound applications.

1.2 Organization

This dissertation is organized as follows. First, the simulation infrastructure and benchmarks used throughout this dissertation are described in Chapter 2. Chapter 3 discusses the distinct reference characteristics demonstrated in each data memory region for these applications. Additionally, we present some observations on dirty writeback behavior in graphics and data streaming applications. Chapter 4 overviews the dynamic power/energy dissipation issues on caches. Through data region characterization for media and communication benchmarks, we achieve power savings by re-organizing the first level cache into variable sized region cachelets that can more efficiently exploit the memory reference characteristics produced by programming language semantics. Chapter 5 describes the Stack Value File design for high performance processors. This technique exploits stack reference characteristics to increase memory level parallelism. Chapter 6 presents the Eager Writeback technique for improving memory bandwidth utilization and thereby improving system performance. Chapter 7 concludes this dissertation and discusses future research directions.

CHAPTER 2

EXPERIMENTAL FRAMEWORK

This chapter describes the experimental framework used throughout this dissertation, including our baseline processor simulator and benchmark suites. The detailed simulation models used in each individual study (such as new microarchitectural additions, instruction set architecture, machine issue width, cache configuration, memory subsystem modeling, etc.) will be detailed in each corresponding chapter.

2.1 Processor Simulator and Tool Suites

The infrastructure of our simulation environment used throughout this dissertation is based on the *SimpleScalar tool set* originally developed at the University of Wisconsin at Madison [19]. SimpleScalar was implemented to be a portable and extensible tool for microarchitecture research. This tool set includes a processor simulator and several modified GNU tools and utilities including a *gcc*-based compiler that is capable of generating target binary to run on the SimpleScalar instruction set architecture, or the SimpleScalar *Portable ISA* (PISA). The PISA encoding and addressing modes are almost identical to the MIPS ISA [64]. The processor simulator is able to simulate both the Compaq Alpha ISA [102] and PISA. The processor simulator is an execution-driven simulator. Besides functional simulation and cache simulation, the simulator is also capable of simulating a dynamic ILP processor with speculative and out-of-order execution support. For dynamic scheduling machine simulation, the simulator models a Register Update Unit (RUU) that combines the functions of reservation stations (RS) and the re-order buffer (ROB) necessary for supporting out-of-order execution. An in-depth description of the RUU mechanism is in [105]. All in-flight instructions are allocated inside the RUU that maintains correct data flow between dependent instructions. Functional unit binding, instruction dispatch, data dependency resolution and retirement are all managed by the RUU. Once an instruction is ready to be committed, it is removed from the RUU and retired into the architecture state.

In this dissertation, we also study the aspects related to dynamic power dissipation in Chapter 4. The power dissipation modeling and projection is based on another tool called *Wattch* developed at Princeton University [18]. *Wattch*, an add-on extension to the original *Simplescalar* simulator, provides performance simulation as well as both static and dynamic power projection through running a single-pass of the microarchitecture-level simulation.

Benchmark	Source	Application
099.go	C	Go chess game
124.m88ksim	C	MC88100 chip simulator
126.gcc	C	GNU c compiler 2.5.3
129.compress	C	UNIX compression utility
130.li	C	Xlisp interpreter
132.ijpeg	C	Image compression/decompression
134.perl	C	Perl interpreter
147.vortex	C	Object-oriented database
101.tomcatv	FORTRAN	Vectorized mesh generation
102.swim	FORTRAN	Shallow water equation
103.su2cor	FORTRAN	Monte-Carlo method
104.hydro2d	FORTRAN	Navier Stokes equation
107.mgrid	FORTRAN	3D potential field
110.applu	FORTRAN	Partial differential equation
125.turb3d	FORTRAN	Turbulence modeling
141.apsi	FORTRAN	Weather prediction
145.fpppp	FORTRAN	Gaussian series of quantum chemistry
146.wave5	FORTRAN	Maxwell's equations

Table 2.1: SPEC CPU95 benchmark

2.2 Benchmarks

2.2.1 SPEC

The SPEC benchmark suites are published by the *Standard Performance Evaluation Corporation*, a non-profit consortium whose members include researchers, industrial hardware vendors and software vendors. The main purpose of this benchmark is to provide a common basis for characterizing the performance of different workstation-level microprocessors. High performance computer industries typically use SPEC numbers as the major indicator to quantify their machine performance and compare performance with machines from different vendors. For a processor under development, architects project and publish the SPEC target performance numbers in order to demonstrate the performance objective as well as evaluate and justify their new machine design. Every 3 to 5 years, the SPEC consortium will revise the benchmark contents to reflect a more representative collection for the state-of-the-art applications.

The SPEC CPU95 (SPEC95) benchmark suite [37] has been widely used by researchers and architects for the last five years. In some of our earlier research work in this dissertation, we performed the simulations using SPEC95 benchmark. The SPEC95 binaries used in this dissertation were compiled using the SimpleScalar GCC compiler that generates code in the PISA format. Table 2.1 describes each benchmark program. The SPEC CPU95 benchmark was retired at July, 2000.

The SPEC CPU2000 benchmark [53] is the most recently published SPEC benchmark as the successor of the retired SPEC CPU95 benchmark. We primarily use their integer benchmark suite to perform our research for studying high performance processors that are not targeted for scientific computing. The binaries of the SPEC CPU2000 benchmark used in this dissertation were compiled using the Compaq Alpha compiler with optimization level 3 and automatic loop unrolling enabled. The input files for our experiments are either taken from reference input set or training input set as explained inside the chapters. Table 2.2 describes the attributes of each benchmark program.

2.2.2 Mediabench

The Mediabench benchmark suite was collected and bundled by researchers at the University of California at Los Angeles [72]. Mediabench was designed for bridging the gap among the compiler community, embedded processor architects and embedded software developers. The programs from

Benchmark	Source	Application
164.gzip	C	Compression
175.vpr	C	FPGA circuit placement and routing
176.gcc	C	C programming language compiler
181.mcf	C	Combinatorial optimization
186.crafty	C	Game playing: chess
197.parser	C	Word processing
252.eon	C++	Computer visualization
253.perlbmk	C	PERL programming language
254.gap	C	Group theory, interpreter
255.vortex	C	Object-oriented database
256.bzip2	C	Compression
300.twolf	C	Place and route simulator
168.wupwise	FORTRAN 77	Quantum chromodynamics
171.swim	FORTRAN 77	Shallow water modeling
172.mgrid	FORTRAN 77	Multi-grid solver
173.applu	FORTRAN 77	Parabolic/Elliptic PDEs
177.mesa	C	3D OpenGL graphics library
178.galgel	FORTRAN 90	Computational fluid dynamics
179.art	C	Image recognition / Neural networks
183.equake	C	Seismic wave propagation simulation
187.facerec	FORTRAN 90	Image processing: face recognition
188.ammp	C	Computational chemistry
189.lucas	FORTRAN 90	Number theory / primality testing
191.fma3d	FORTRAN 90	Finite-element crash simulation
200.sixtrack	FORTRAN 77	High energy nuclear physics accelerator design
301.apsi	FORTRAN 77	Meteorology: pollutant distribution

Table 2.2: SPEC CPU2000 benchmark

Benchmark	Application
cjpeg	Discrete Cosine Transform Image Compression
djpeg	Discrete Cosine Transform Image Decompression
mpeg2encode	MPEG2 video encoder
mpeg2decode	MPEG2 video decoder
rawcaudio	Speech compression using ADPCM standard
rawdaudio	Speech decompression using ADPCM standard
g721encode	Voice compression using G.721 standard
g721decode	Voice decompression using G.721 standard
pgpencode	Data encryption and signing using RSA, IDEA and MD5
pgpdecode	PGP decoding exercising RSA, IDEA and MD5
pegwitencode	Public key encryption and authentication
pegwitdecode	Public key decryption and authentication
gs	Ghostscript
mesa.texgen	Mesa 3D OpenGL library (Textured Teapot)
mesa.osdemo	Mesa 3D OpenGL library (Draw Polygons with Z-buffering)
mesa.mipmap	Mesa 3D OpenGL library (Texture mapping)
rasta	Speech recognition
epic	Data compression using wavelet decomposition and Hoffman coding
unepic	Epic decoding wavelets and Huffman coding

Table 2.3: Mediabench benchmark

Benchmark	Application
Doom	3D shoot'em-all game
POVray	Persistence of Vision Ray-Tracer
xanim	Multiformat animation/video/audio viewer
xlock	X-window screen saver

Table 2.4: X benchmark

the Mediabench benchmark suite represent the workloads for a variety of emerging multimedia and communication applications. These applications are also recognized as the application software and algorithms commonly used in personal telecommunication, mobile and PDA devices. Table 2.3 describes the algorithm for each application.

The binaries were compiled using the SimpleScalar *gcc* compiler in PISA format. The optimization level 3 and automatic loop unrolling were turned on in the compilation.

2.2.3 X Benchmark

The X benchmark suite was collected by Todd Austin [98]. It consists of four applications representing different graphics algorithms as listed in Table 2.4. *DOOM*, a popular video game developed by *id software* corporation, uses a polygon-based rendering algorithm. *POVray* is a public domain ray tracing package developed for generating photo-realistic images on a computer. The third application *xanim* is an animation viewer which processes an MPEG-1 data stream to display an animated sequence. The final application, *xlock*, a popular X-window screen saver, renders a 3D polygonal object on the screen.

2.2.4 3D Geometry Pipeline

Today's polygon-based 3D graphics engine is composed of two major components, a 3D geometry processing pipeline and a rasterization pipeline. The 3D geometry processing pipeline as shown in Figure 2.1 is representative of a very frequently used algorithm in most polygon-based rendering engines. 3D geometry processing, consisting of intensive floating-point operations on a large quantity of vertex data from memory, handles vertices from the object model database. It maps the vertices from the world coordinate space to viewer's space, i.e. the display. It also computes the interaction between the light sources and their effect on each vertex for generating the shading intensity to be used during rasterization. These functions are typically done on the host processor. Once the vertices are transformed and lit, they are sent to the rasterization pipeline for scan-converting them into pixels on the display. The rasterization pipeline interpolates color values for each interior pixel within a polygon. Rasterization is typically performed by a dedicated graphics accelerator.

This pipeline consists of three nested loops wrapped by two outer loops which iterate through frames and 3D objects in the world space. The first innermost loop processes vertices for each 3D object assuming the entire object is modeled by a single triangle strip or a triangle fan. A triangle strip or fan as illustrated in Figure 2.2 is an object representation using a pre-ordered triangulation scheme. It is supported by popular graphics libraries such as OpenGL [118]. Using such an ordering, only one incremental vertex needs to be specified to describe a new triangle. For example, T1 is constructed by V1, V2 and V3. T2 is constructed by V2, V3 and V4. It is a more compact representation, therefore it requires fewer number of operations performed in the geometry pipeline. The basic functionality performed inside the innermost loop includes *transformation*, *lighting*, and *command output*. Figure 2.3 illustrates the control flow chart in block diagrams.

The *transformation* function projects the new location of each vertex on screen through a 4x4 matrix multiplication and a viewport transformation. The *lighting* function calculates the interaction

```
MINI-GEOMETRY()
  while ( frames )
          for ( objects in each frame )
               for ( every 4 vertices )
                    /* Transformation */
                    $tx = m11 * InV[]_x + m21 * InV[]_y + m31 * InV[]_z + m41;$
                    $ty = m12 * InV[]_x + m22 * InV[]_y + m32 * InV[]_z + m42;$
                    $tz = m13 * InV[]_x + m23 * InV[]_y + m33 * InV[]_z + m43;$
                    $w = m14 * InV[]_x + m24 * InV[]_y + m34 * InV[]_z + m44;$
                    $OutV[]_{rw} = 1/w;$
                    $OutV[]_{tx} = X_{offset} + tx * OutV[]_{rw};$
                    $OutV[]_{ty} = Y_{offset} + ty * OutV[]_{rw};$
                    $OutV[]_{tz} = tz * OutV[]_{rw};$
                    /* Texture coordinates copying */
                    $OutV[]_{tu} = InV[]_u;$
                    $OutV[]_{tv} = InV[]_v;$
                    /* Lighting Loop */
                    $ID_r = ID_g = ID_b = 0.0;$
                    for ( every light source )
                         $dot = LDir[]_x * InV[]_{nx} + LDir[]_y * InV[]_{ny} + LDir[]_z * InV[]_{nz};$
                         $ID_r = ID_r + Ambient_r + Diffuse_r * dot;$
                         $ID_g = ID_g + Ambient_g + Diffuse_g * dot;$
                         $ID_b = ID_b + Ambient_b + Diffuse_b * dot;$
                         $OutV[]_{cd} = ((int)ID_r << 24)|((int)ID_g << 16)|((int)ID_b << 8|\alpha);$

               /* Device driver loop */
               for ( each transformed and lit vertex )
                    /* Assume Tri-Strip triangles */
                    /* Copy entire OutV records to graphics AGP memory */
                    $GfxCommand[vertex - 2] = OutV[vertex - 2];$
                    if $(even - numberedvertex)$
                       $GfxCommand[vertex] = OutV[vertex];$
                       $GfxCommand[vertex - 1] = OutV[vertex - 1];$
                    else
                       $GfxCommand[vertex - 1] = OutV[vertex - 1];$
                       $GfxCommand[vertex] = OutV[vertex];$
```

Figure 2.1: Algorithm of the mini-geometry kernel

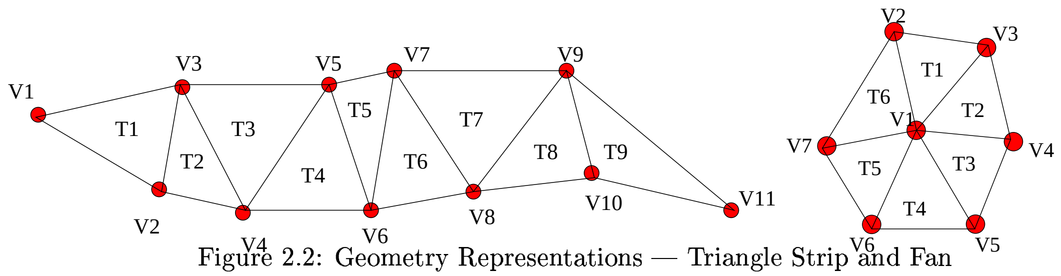


Figure 2.2: Geometry Representations — Triangle Strip and Fan

of each vertex with light sources and generates the color intensity for each vertex. This calculation involves a dot product between the light direction vector and the vertex normal vector using a Phong illumination model [113]. A single parallel light source with diffuse only components is assumed in the lighting model. For a parallel light source, per-vertex normal transformations can be replaced by an inverse transformation of the light source location on a per-scene basis, thus eliminating a large number of computations for generating light direction vector of each vertex. A color packing conversion then packs four single-precision floating-point RGBA color intensities into a packed 4-byte integer. The instruction set architecture of interest is assumed to support four wide SIMD computation, similar to Intel's Pentium III Processor.

After finishing with all the vertices in one object, a loop imitating the functionality of a device driver is invoked (the *command output* function). This driver loop breaks one triangle strip into individual triangles and copies these transformed and lit vertices to the uncacheable and/or write-combinable graphics memory, e.g. Advanced Graphics Port memory space. We adopt the online driver model [91][121] advocated by Intel's Pentium III architects in the pipeline. It possesses the advantage to copy the post-transformed and lit vertices into graphics memory on the fly concurrently

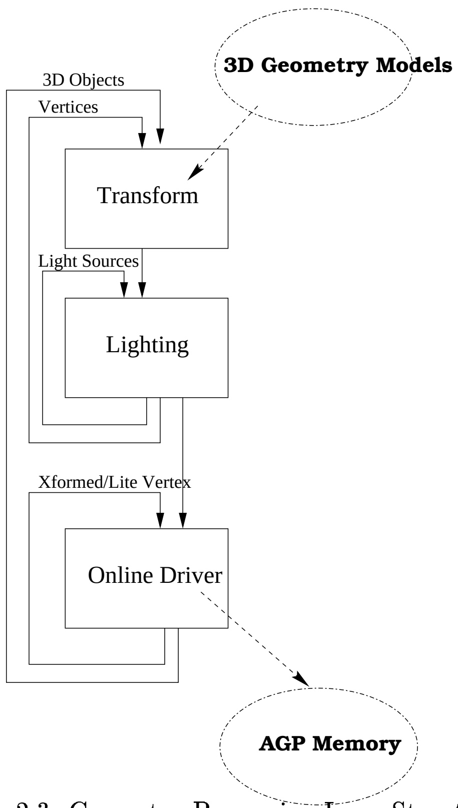


Figure 2.3: Geometry Processing Loop Structure

overlapped with the rest of the geometry processing. There is no need for allocating extra memory space to an intermediate buffer for device driver, therefore, overall performance is improved.

2.2.5 Streaming Kernel

```
STREAMING()
  float $array_A[MAX]$, $array_B[MAX]$;
  for $(m = 0; m < loop; m++)$
       for ( $array_A[i] \in$ each set of L2 cache )
            write $array_A[i]$ to way 0;
            write $array_A[i + 1 * 8 * set\_size]$ to way 1;
            write $array_A[i + 2 * 8 * set\_size]$ to way 2;
            write $array_A[i + 3 * 8 * set\_size]$ to way 3;
       for ( $array_A[j] \in$ each cache line in L2 cache )
            read $array_A[j]$;
            compute $array_A[j]$;
            write $array_A[m]$;
       for ( $array_B[k] \in$ each set of L2 cache )
            read $array_B[k]$ into way 0;
            read $array_B[k + 1 * 8 * set\_size]$ into way 1;
            read $array_B[k + 2 * 8 * set\_size]$ into way 2;
            read $array_B[k + 3 * 8 * set\_size]$ into way 3;
            write $array_A[m]$;
```

Figure 2.4: Algorithm of the Streaming Kernel

The algorithm of *Streaming* kernel is presented in Figure 2.4. This algorithm mimicking multimedia data streaming consists of three inner loops that exercise the L2 cache. The first loop writes data into $array_A[]$. The second loop reads data from $array_A[]$, performs some floating-point computation and passes the results to inner loop invariant array elements. Finally the third loop accesses a new array ($array_B[]$), displacing elements of $array_A[]$ from the cache.

This program is designed to represent the typical behavior of many emerging streaming applications. As pointed out previously, however, in order to highlight the memory system in a uniprocessor, no actual computational work is performed per data read.

CHAPTER 3

MEMORY REFERENCE CHARACTERIZATION

3.1 Virtual Memory Space Partitioning

The run-time storage allocation of a program is conventionally subdivided into several non-overlapped regions: the generated target code, static and dynamic data objects [2]. The mobility of data objects depends on the specification of each programming language. For example, all data objects in *Fortran* can be allocated statically at compilation time into pre-determined storage areas for the sizes of their data objects are static and known prior to execution. Therefore, even activation records can be allocated during program compilation. However, for languages such as *C* or *Pascal* that support data allocation under program control, the locations of these data objects are manipulated dynamically by a run-time system. These programming language semantics provide the capability for flexible data manipulation such as dynamic data object allocation and recursive procedure calls. Thus, the sizes of data structures of a program can vary as needed under different run-time scenarios. In addition, static binding of activation records can no longer be used in *recursive* procedural calls. Conventionally, memory references for modern microprocessors fall into the following categories — code, read-only data (literals), global static data, stack data and heap data; according to the regions of memory they access and the access method used. The stack data region is used for the dynamic allocation of activation records; the heap data region is used for dynamic data object allocation, while code (i.e. program binary), read-only data and global static data are statically allocated by the compiler and linker.

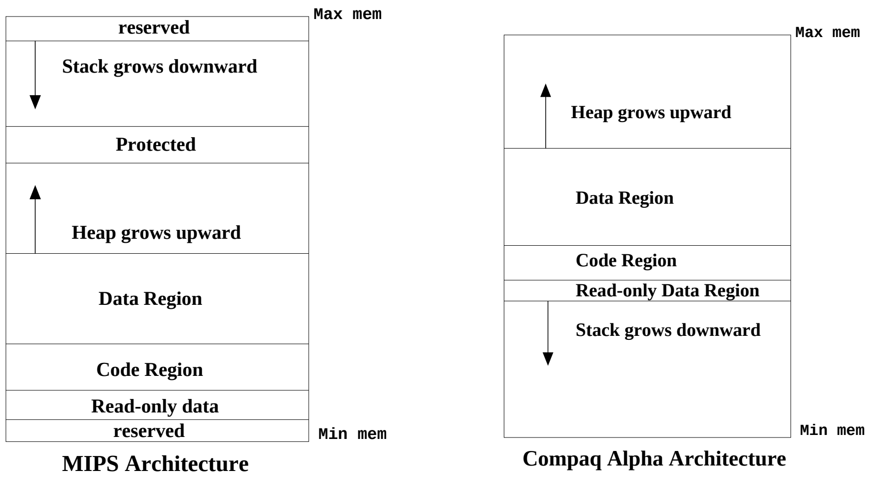


Figure 3.1: Run-time Memory Subdivision for MIPS and Compaq Alpha Architectures

Figure 3.1 illustrates the regions from virtual memory partitioning used by the MIPS architecture [64] and the Compaq Alpha architecture [16]. For the MIPS architecture, a system-defined amount of space is allocated to the stack, which grows down. The top of stack (TOS[1]) dynamically maintains the size of the stack, which forms a bound on address references to the stack. The bottom address range, allocated during compilation, includes read-only data, the instruction code region and the global static data region. Memory is dynamically allocated at run-time by the program from the heap, which grows upwards from the middle address range. For both architectures, and most other architectures, newly allocated stack memory grows from higher addresses to lower addresses, thus new stack space can be typically allocated via a "push" operation ; heap memory allocation grows upward.

3.2 Memory Reference Distribution

3.2.1 Distribution of Memory Instructions

The majority of data memory references fall into the stack, global static data and heap regions, although some applications that intensively use strings, constants, such as data formatting, could access read-only data quite frequently. For example, using *printf()* in C often access a user-defined read-only literal for the format string (first argument). To understand the memory reference behavior by region, the distributions of run-time data memory accesses were profiled using the following two benchmark suites, the latest SPEC CPU2000 integer benchmark (SPECint2000) and Mediabench benchmark, described in Section 2.2. The Compaq Alpha ISA represents a 64-bit architecture while the PISA is 32-bit. Both processors are RISC-like machines that follow the RISC philosophy of a load/store architecture. Hence, nothing but load and store instructions may access memory.

Figure 3.2 and Figure 3.3 show the profiling statistics of memory references by regions. The data in both figures are normalized to the total number of memory instructions. For the SPECint2000 profiling data, an average of 42% of the instructions executed access memory. Stack references account for an average of 56% of all memory accesses, while global static data references account for about another 21% and a majority (20%) of the remaining accesses are to the heap. Only 3% memory accesses are to the read-only data region.

A slightly different behavior from the SPECint2000, the Mediabench contains an average of 24% of the instructions accessing memory. Out of this 24% memory instructions, stack references account for an average 39.1% of all memory references, while global static data references and heap references average about 30% each of the total memory references. Data read from the read-only region are rather insignificant (about 0.1%) in Mediabench. Also note that *djpeg*, *mpeg2encode*, *epic* and *unepic* skew these averages with an extraordinarily large portion of heap accesses.

3.2.2 Distribution of Data Cache Occupancy

As demonstrated in the previous section, stack references account for the majority of the memory references while global static and heap data account for approximately equal proportions of the rest of the memory activities. In this section, we will quantify the cache residency behavior by using a physical data cache structure to characterize their respective occupancy ratio for each data region at run-time.

Figure 3.4, Figure 3.5, Figure 3.6, and Figure 3.7 show normalized data occupancy ratios for a direct-mapped, two-way, four-way, and eight-way set associative cache using SPECint2000 benchmark; each way of these caches consists of a 16KB content addressable memory. For instance

[1]The TOS in this example actually maintains the *lowest* stack memory address rather than the top memory address. TOS is thus used as a conventional nomenclature in this case.

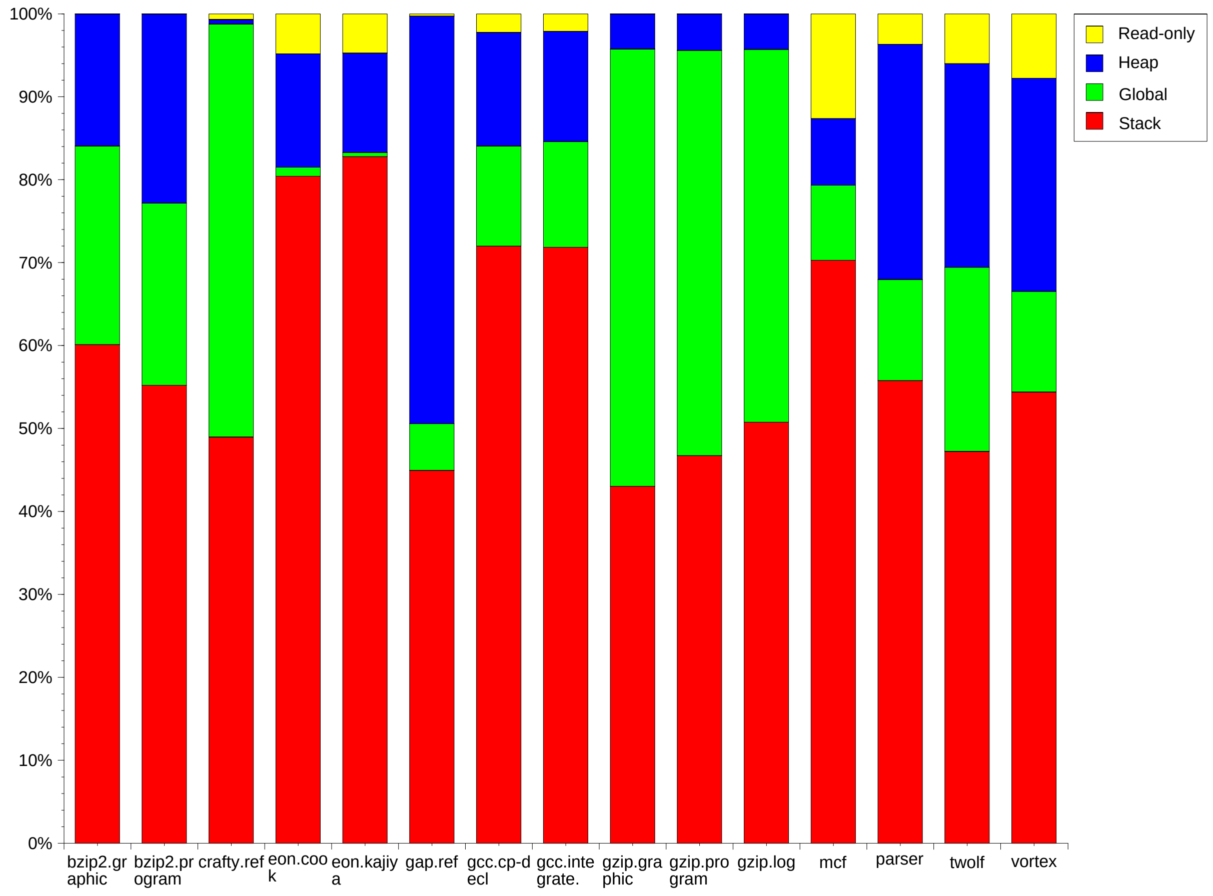


Figure 3.2: Percentage of Memory Access Regions for SPECint2000

in Figure 3.6, a four-way set associative first level cache is used in our profiling experiment. There are four stacked bars for each program of the SPEC2000int benchmark. In this figure, way0 or the *Most Recently Used* (MRU) cache line, is drawn in the leftmost hand side of the four bars; while way3 or the *Least Recently Used* (LRU) line is drawn in the rightmost side. The occupancy count for each memory region in each entry of the cache is calculated as described in the following. Each time when a data cache access is initiated, we take a snapshot for all cache ways of the set being accessed and increment the corresponding region occupancy counter for each way based on the region of the data that reside in that particular cache line entry.

For each stacked bar, we averaged the region occupancy counters across entire cache sets and normalized each counter for all four data regions: read-only, heap, global static and stack data, from top to bottom. Overall, it is observed that most of the stack data can be found in way0, i.e. the MRU state; only 4.4%, 3.0% and 2.6% of the time stack data are residing in way1, way2 and way3. Nevertheless, heap data show relatively opposite behavior from stack data. Only 20.6% of the time, heap are found in MRU state. They spent more of their lifetime in other cache ways: 73.8% for way1, 81.3% for way2, and 83.5% for way3. For some of the benchmark programs such as 186.crafty and 164.gzip, global static data occupy a longer period of time in way1, way2 and way3. This characteristics corresponds to the heavy memory references in global static data in Figure 3.2. 176.gcc also shows slightly different behavior, mainly the stack data. In most of the other benchmark programs, stack region is only heavily accessed in way0. Once after they were sent from MRU state down to less recently used ways, they were seemingly replaced out of the cache without being re-accessed temporally. For 176.gcc, however, the stack data show some temporal locality during their

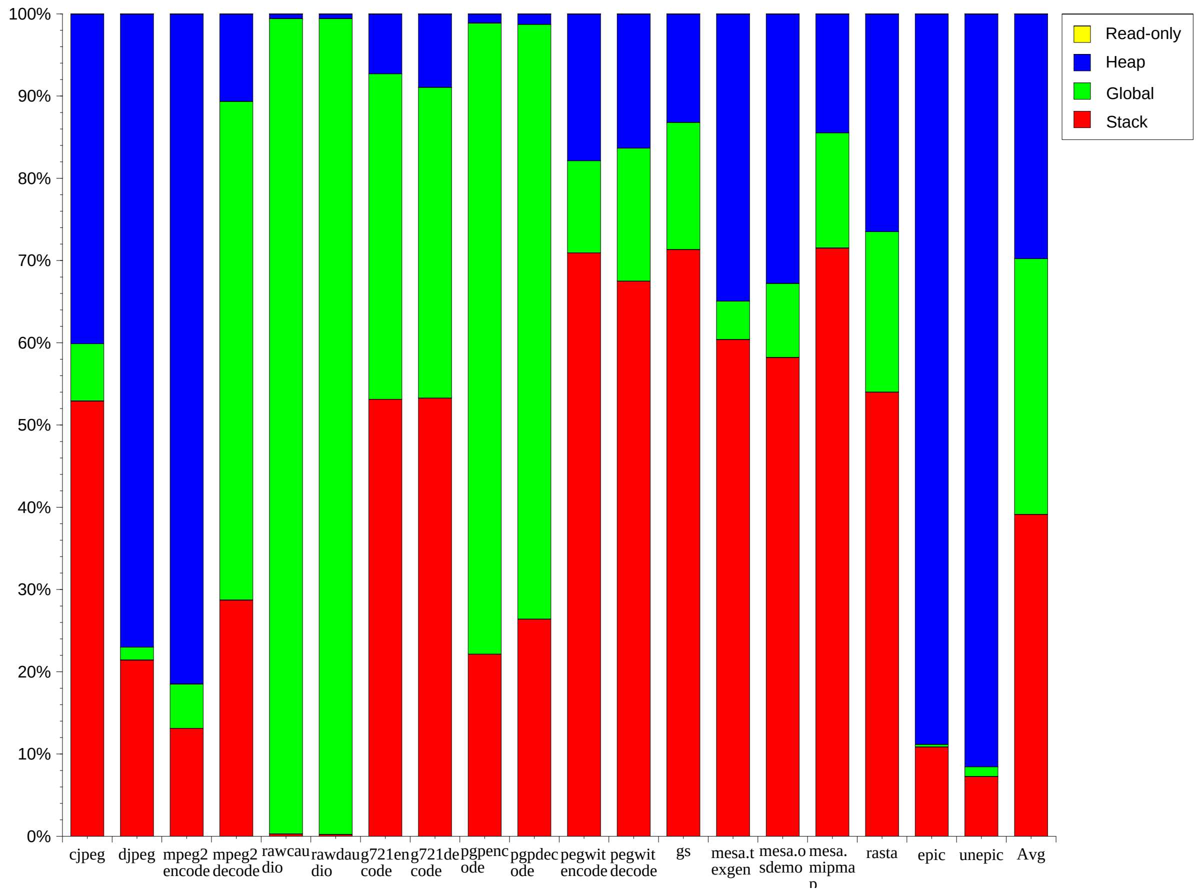


Figure 3.3: Percentage of Memory Access Regions for Mediabench

lifetime in all cache ways. Approximately 30% of the time, they are found in way1, way2 and way3.

As we increased the size of the cache by increasing the number of ways in the cache from Figure 3.5 to Figure 3.7, it is noticed that the heap data tend to occupy the non-MRU ways even more frequently in most of the benchmark programs. In other words, the heap data contain very little temporal locality in these integer programs.

The occupancy information collected from this characterization suggests that most of the stack data would reside in MRU states of cache lines while most of the LRU lines contain heap data. It implies that heap data are less frequently or unlikely to be promoted again to the MRU state once they have been demoted to non-MRU states. In fact, this provides some useful information for heuristics that exploit generational behavior in caches to either reduce power consumption or enhance cache hit ratio. For example, Kaxiras et al. [65] proposed a technique that reduces static leakage power by gating off supply voltage V_{dd} of the *decayed cache lines*, namely, those lines that are very unlikely to be reused in the near future.

3.3 Analysis of Regional Reference Characteristics

3.3.1 Weighted Reference Distance

We have shown that stack data outstrip the other data in terms of reference activity and cache occupancy. In this section, we will investigate the average cache line locality, including temporal and spatial locality, for each region. Our goal here is to understand how often each individual cache

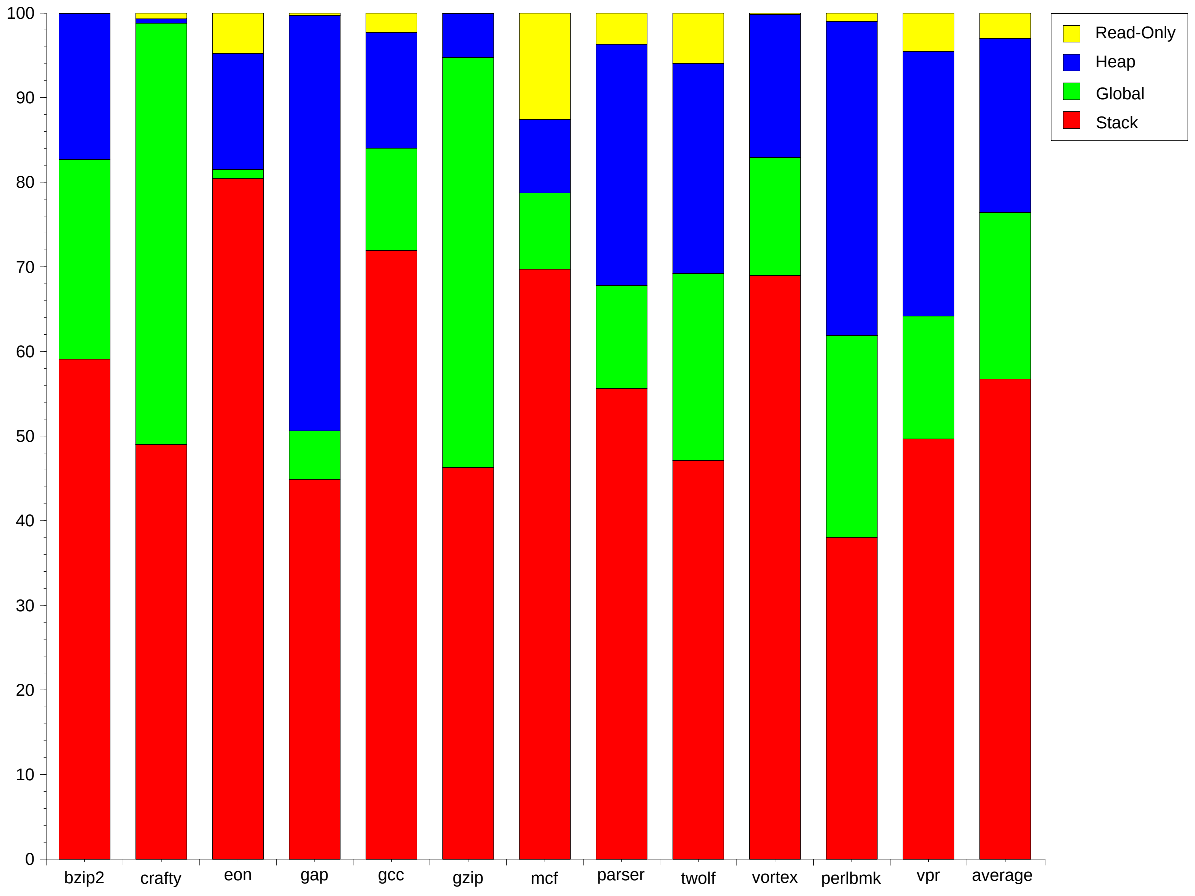


Figure 3.4: Data Occupancy Ratios in Each Region for a Direct-Mapped 16KB Cache

line of a particular data region is re-accessed during execution. To measure the reference locality in a quantitative manner, we define and calculate the *weighted reference distance* for each data region. We assume that $\Psi^n_{(R,\ell)}$ denotes the total number of memory references, accumulated from all data regions, at the time when the n^{th} memory instruction accesses cache line ℓ in region R. The weighted reference distance, denoted by $Dist(R)$ for region R is computed as follows. Given there are L cache lines accessed within the lifetime of a program, and there are N_ℓ references for each cache line ℓ within region R. The objective is to obtain an average distance between two successive accesses to the same cache line, for all cache lines in an infinitely large cache. As long as the memory access sequence is the same, i.e. no out-of-order effect, the only cache parameter that could change $Dist(R)$ is the cache line size. The formula is shown below.

$$Dist(R) = \frac{\sum_{\ell=1}^{L}\sum_{n=1}^{N_\ell}(\Psi^n_{(R,\ell)} - \Psi^{n-1}_{(R,\ell)})}{\sum_{\ell=1}^{L}(N_\ell - 1)}$$

Figure 3.8 illustrates the weighted reference distance for each data region using SPECint2000 benchmark. The cache line size used in this experiment is 32 bytes. Note that the distance on the y-axis, represented by the number of memory references in-between 2 consecutive accesses to the same cache line, is plotted on a log scale. As the figure shows, the weighted reference distance of stack cache lines is mostly below 100 and has a harmonic mean value of 66. The only outlier is 126.gcc that exhibits a longer $Dist(R)$, close to 10,000. In contrast, the weighted reference distance of heap cache lines shows a much greater $Dist(R)$ for all the benchmark programs. The $Dist(G)$ of the global static data spread in-between those of stack and heap data yet are closer to the $Dist(S)$ of stack data.

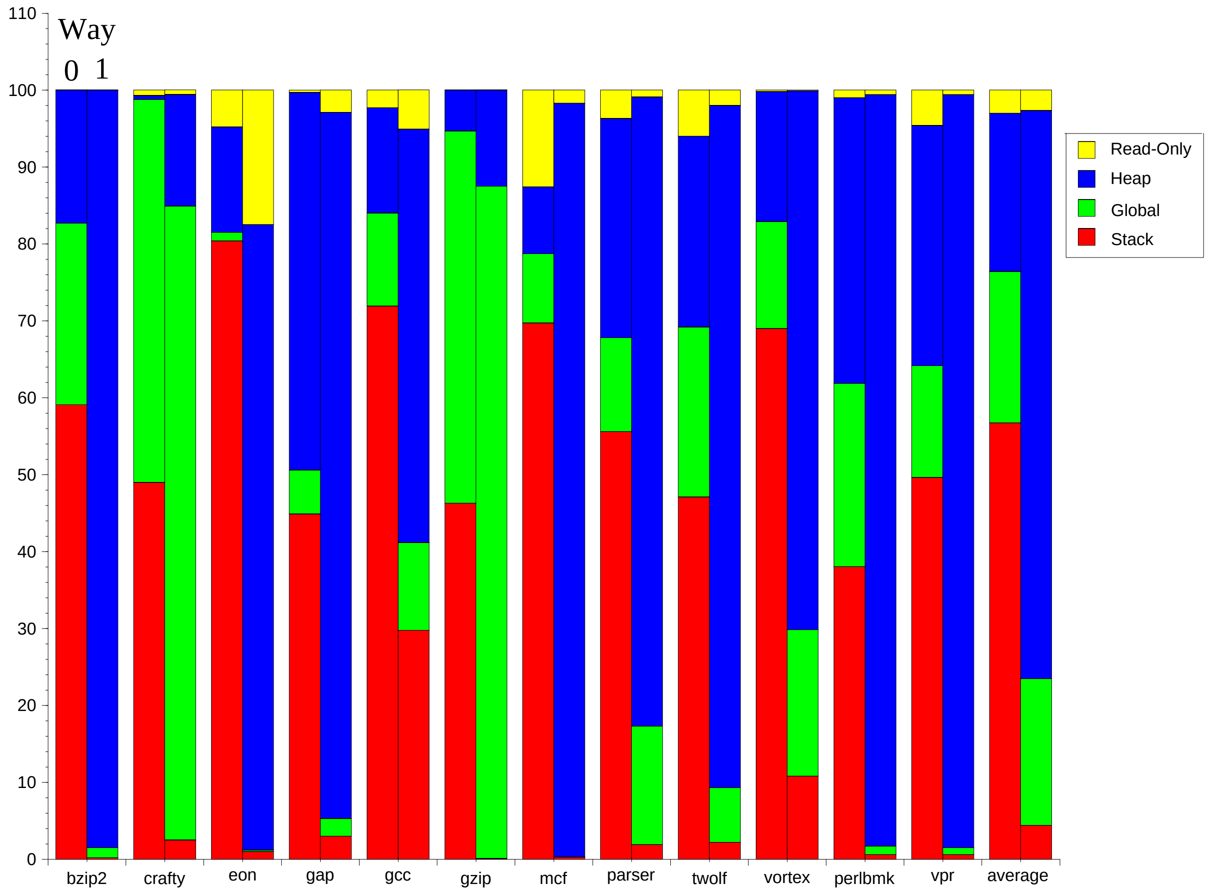


Figure 3.5: Data Occupancy Ratios in Each Region for a 2-way 32KB Cache

The respective harmonic means of the weighted reference distance are 66, 693, 26,387, and 1,499 for stack, global static, heap and, read-only data regions. This distance can be regarded as a metric of reference locality or re-access distance for cache lines in each region under the circumstances that no replacement has ever occurred in the cache.

3.3.2 Unique Memory Block References by Region

Figure 3.9 and Figure 3.10 show the unique memory block (or cache line) references by regions for 256.bzip2 and mpeg2encode respectively. The intention of this characterization is to understand the distribution of cache line accessed in each region, measured by the number of references for each unique memory block during execution. A unique memory block is essentially distinguished by the frame address of a cache line, i.e. the tag address plus the index address. All cache lines are tagged and sorted by their numbers of references. On the x-axis, each curve representing each data region is plotted on a log scale from the most referenced cache line, or the *hottest* cache line, starting from a unique memory block identification number 1 at the left, to the least referenced cache line. The y-axis plots the number of accesses on a log scale as well. Using Figure 3.9 as an example, the curves of stack, global static and read-only data all drop quite steeply. For instance, after the top 100 most referenced stack cache lines, the numbers of accesses to each of the rest stack cache lines are all well below 10,000 times. However, the heap cache lines demonstrate a very distinct behavior from the other regions. The trend of the heap curve follows very closely to that of the "all accesses" curve. In other words, after the top 1,000 most referenced cache lines, most of the

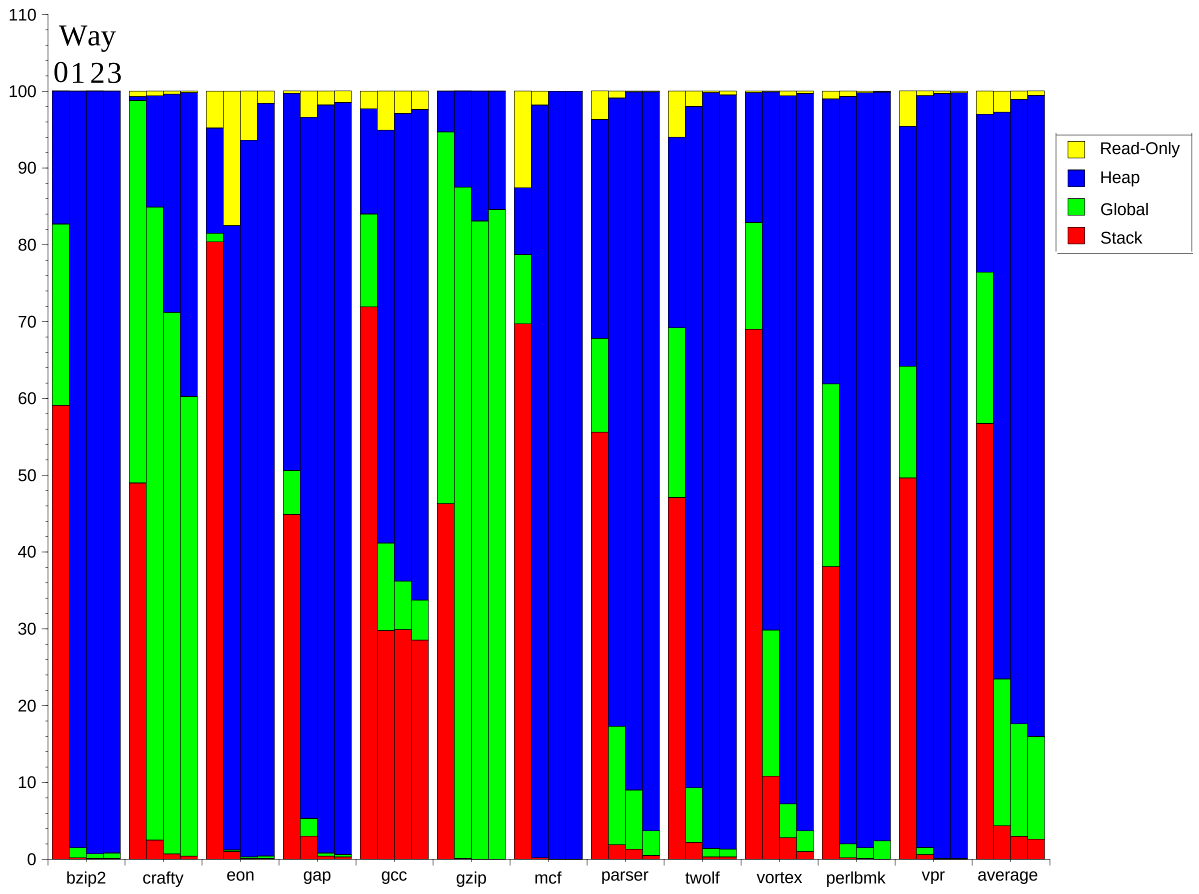


Figure 3.6: Data Occupancy Ratios in Each Region for a 4-way 64KB Cache

rest of the references, spread across a large number of cache lines, go to the heap. Heap data, by and large, occupy a large cache memory image at run-time. Without regard to conflict misses, it is seemingly true that architects are designing an ever larger cache for facilitating heap data. The other SPEC2000int and Mediabench applications all demonstrate a very similar trend as depicted in Appendix A and Appendix B.

3.3.3 Information Content of Cache Frame Addresses

In this section, we study the information content inherent to the data references of programs written in high-level languages. Information theory [99] has been widely used to measure the information complexity of computing and communication systems. Hammerstrom and Davidson [51] developed several estimation techniques based on information theory for analyzing the information content of memory addressing. In order to understand the relative information content, or *entropy*, of data access streams for each region, we apply the fundamental concepts of information theory to the data addresses referenced, using the cache frame address as a random variable.

Let $P(A_i)$ be the probability of a cache frame address, A_i, being referenced during a program's lifetime. The information $I(A_i)$ carried by each A_i is then computed as $\log_2^{\frac{1}{P(A_i)}}$. For example, assuming that only 3 memory addresses are referenced in a program and their respective reference probabilities are $\frac{1}{2}$, $\frac{1}{4}$, and $\frac{1}{4}$. Then minimum information represented by number of bits for each of these 3 addresses can be compressed and communicated by 1 bit, 2 bits and 2 bits. For a zero-

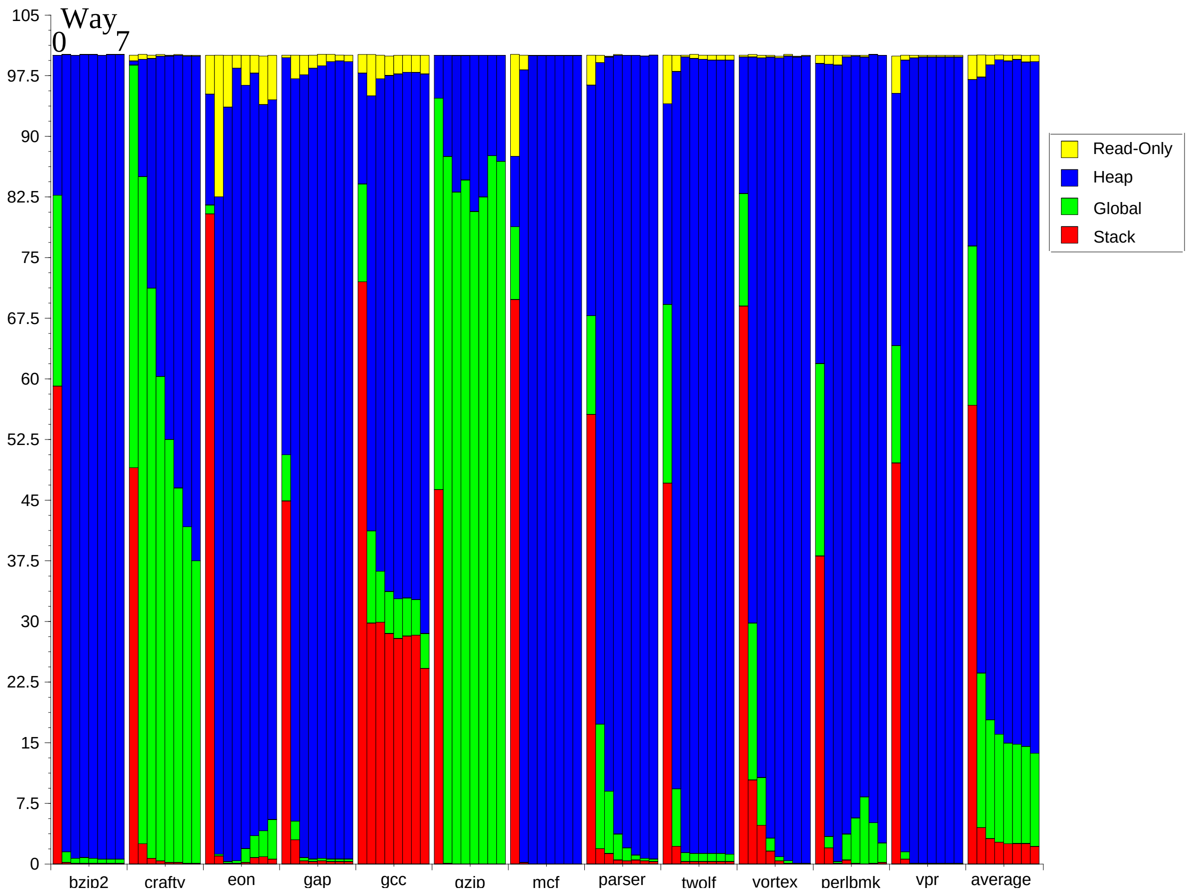


Figure 3.7: Data Occupancy Ratios in Each Region for an 8-way 128KB Cache

memory source [1] system[2], the information content or entropy carried for each cache frame address is $P(A_i) * I(A_i)$. Therefore, the average entropy $H(A, R)$ of each data region R is defined as:

$$H(A,R) \doteq \sum_{A_i \in R} P(A_i) * \log_2^{\frac{1}{P(A_i)}} = -\sum_{A_i \in R} P(A_i) * \log_2^{P(A_i)}$$

Using this equation, we calculate and plot the $H(A, R)$ for each region R in Figure 3.11. According to prior analysis and discussion, it is expected that the entropy of heap data addresses can be much higher than those of the other regions. As shown, the average entropy $H(A, Heap)$ is approximately 3 times higher than $H(A, Stack)$ and $H(A, Static)$. Namely, heap data addresses consist of much richer information than the other two major data address regions. We now discuss how this could impact the cache sizes.

Consider two discrete caches for data storage in region X and Y with their minimum cache sizes of C_X and C_Y to satisfy their data capacity. We define a metric called the *expected capacity ratio*, ECR, to be $\frac{C_X}{C_Y}$. The entropy $H(A, R)$ represents the minimum number of bits needed to convey information needed for region R. Therefore, the expected capacity ratio of these two caches can be represented as follows.

$$ECR_{\frac{X}{Y}} = \frac{2^{H(A,X)}}{2^{H(A,Y)}} = 2^{(H(A,X)-H(A,Y))}$$

[2]A zero-memory source is an information system in which the successive information or symbols emitted are statistically independent.

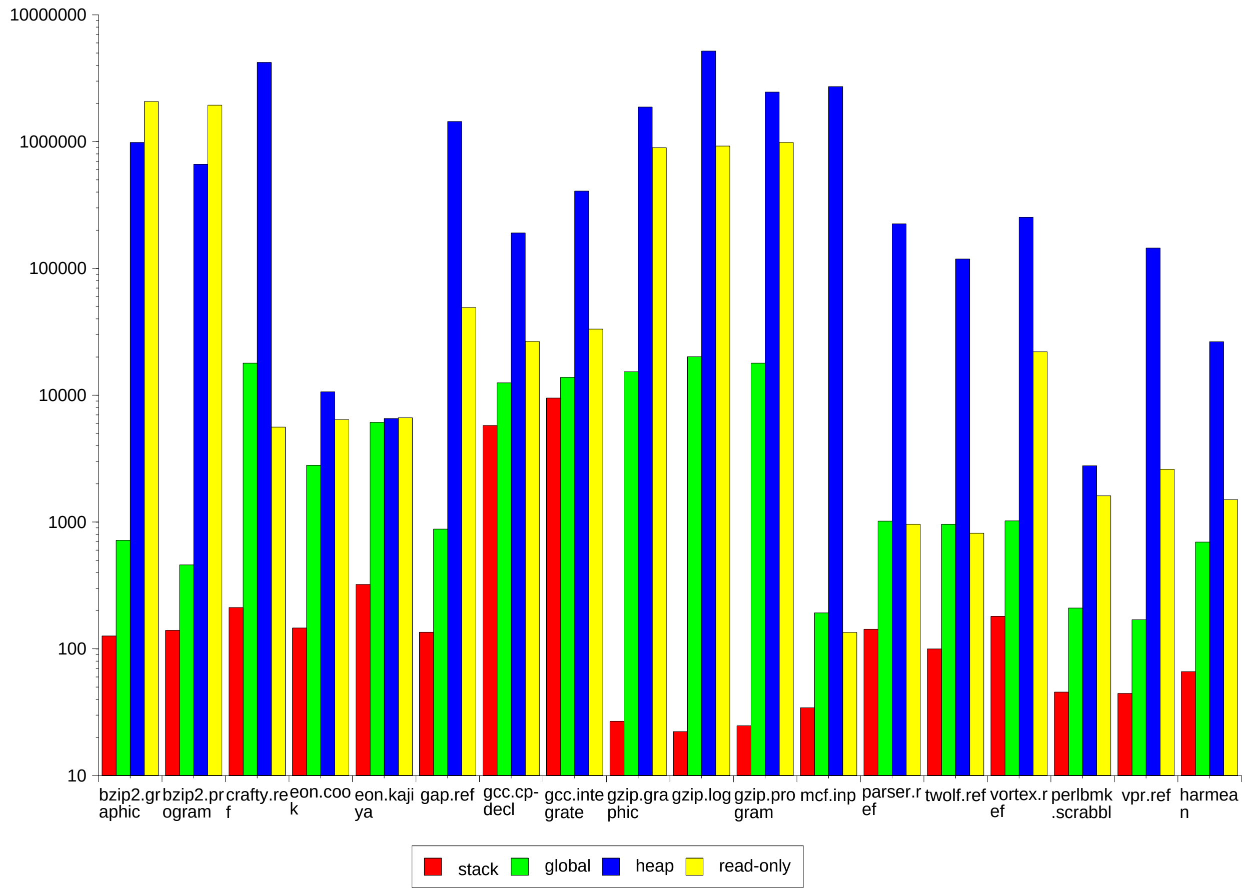


Figure 3.8: Weighted Reference Distance of Cache Lines by Regions

Let $H(A, X)$ of region X be a multiple α (alpha) of $H(A, Y)$ of region Y, where the size C_Y is equal to 2^N. Then we can derive

$$ECR = 2^{N(\alpha-1)}$$

Figure 3.12 plots the expected capacity ratio with varied α and N. For instance, when α is 3 (which is approximate to what we measured in the benchmark) and assume a 1KB stack cache (i.e. N=10) can satisfy information for stack region. According to Figure 3.12, we need roughly a 1MB heap cache in order to satisfy the needed address information in heap region. Note that the y-axis in the graph uses a log scale. Thus when α increases linearly, the ECR actually grows exponentially in N. That is, the effective cache size expands exponentially even if the entropy increases only linearly.

Combined Figure 3.11 and Figure 3.12, we get a picture of the relationship between information content with respect to their expected cache capacities. As the results show, to achieve the same effect, the heap region needs an exponentially larger storage capacity than the stack and global static data regions.

3.4 Behavior of Cache Line Writes

We explore the characteristics of cache line writes in this section. Cache line writes change the architectural state of memory image and require extra logic to maintain memory consistency. Two write policies are popularly employed in modern microprocessor architectures for the general memory types — writeback and write-through. There are also other memory types available, e.g.

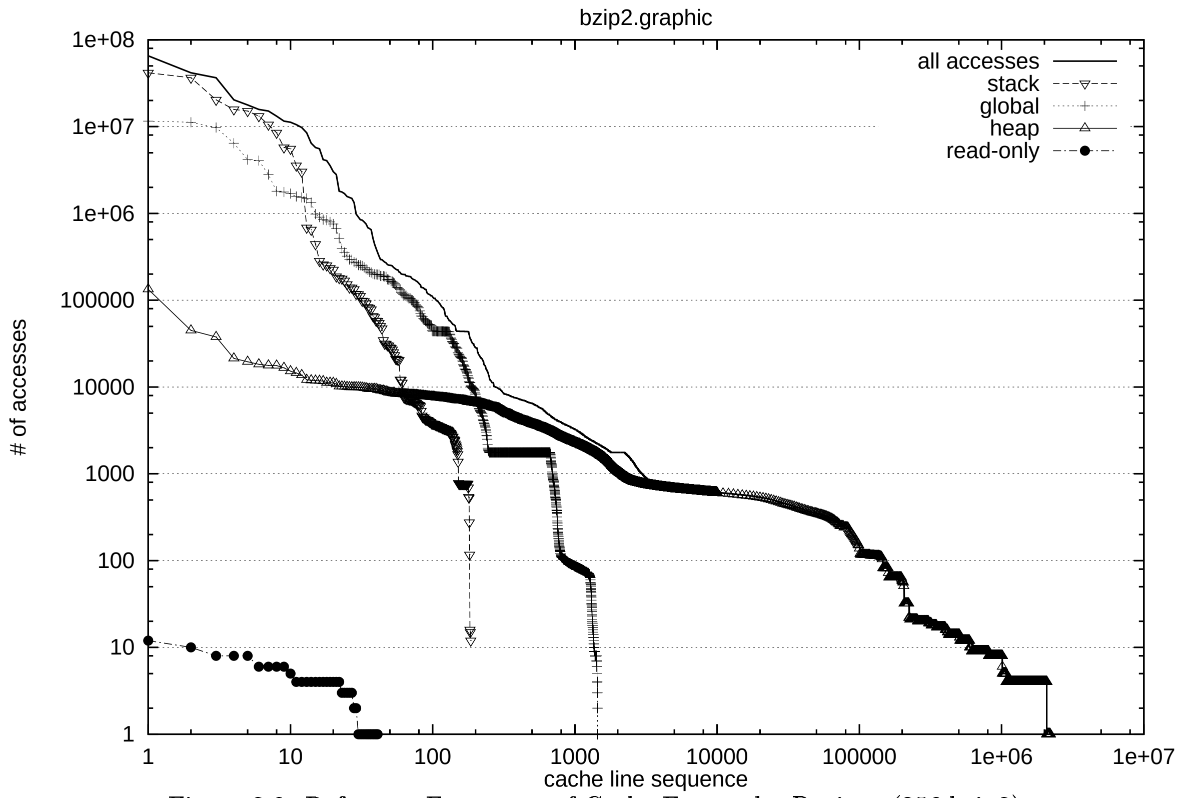


Figure 3.9: Reference Frequency of Cache Frames by Regions (256.bzip2)

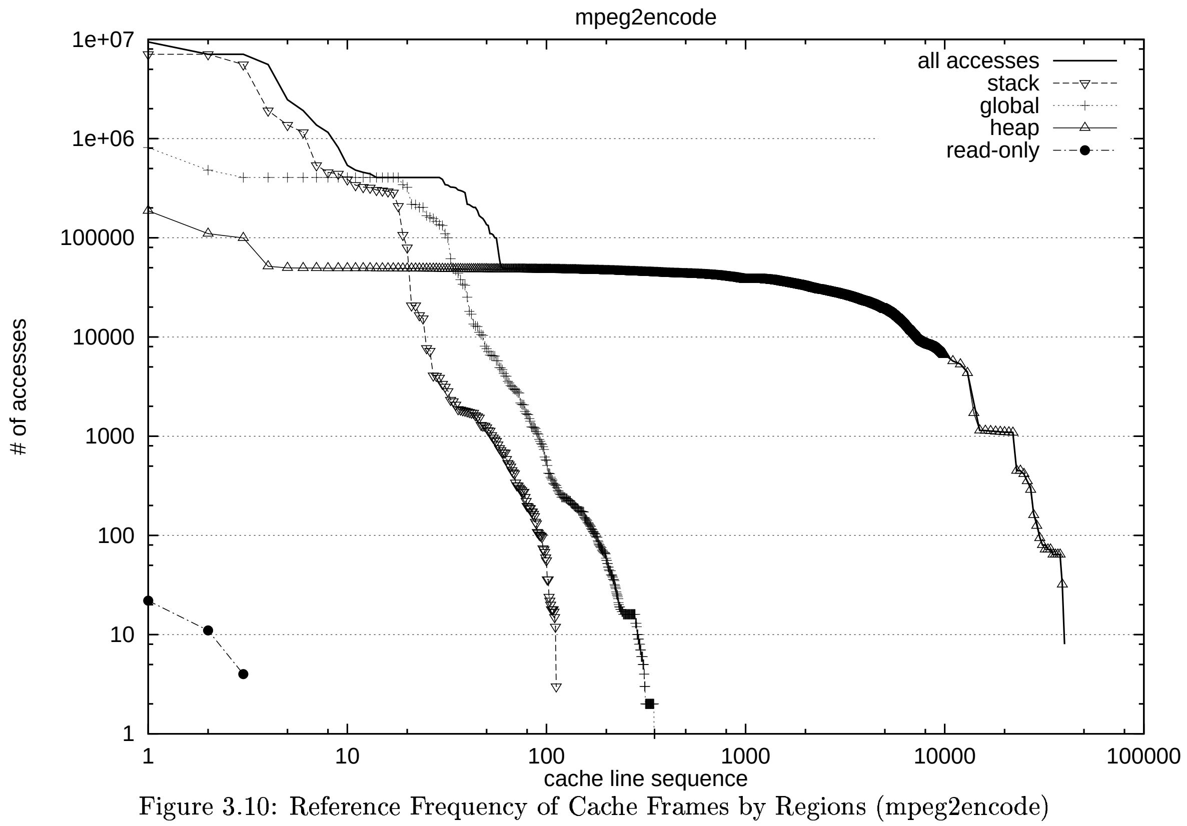


Figure 3.10: Reference Frequency of Cache Frames by Regions (mpeg2encode)

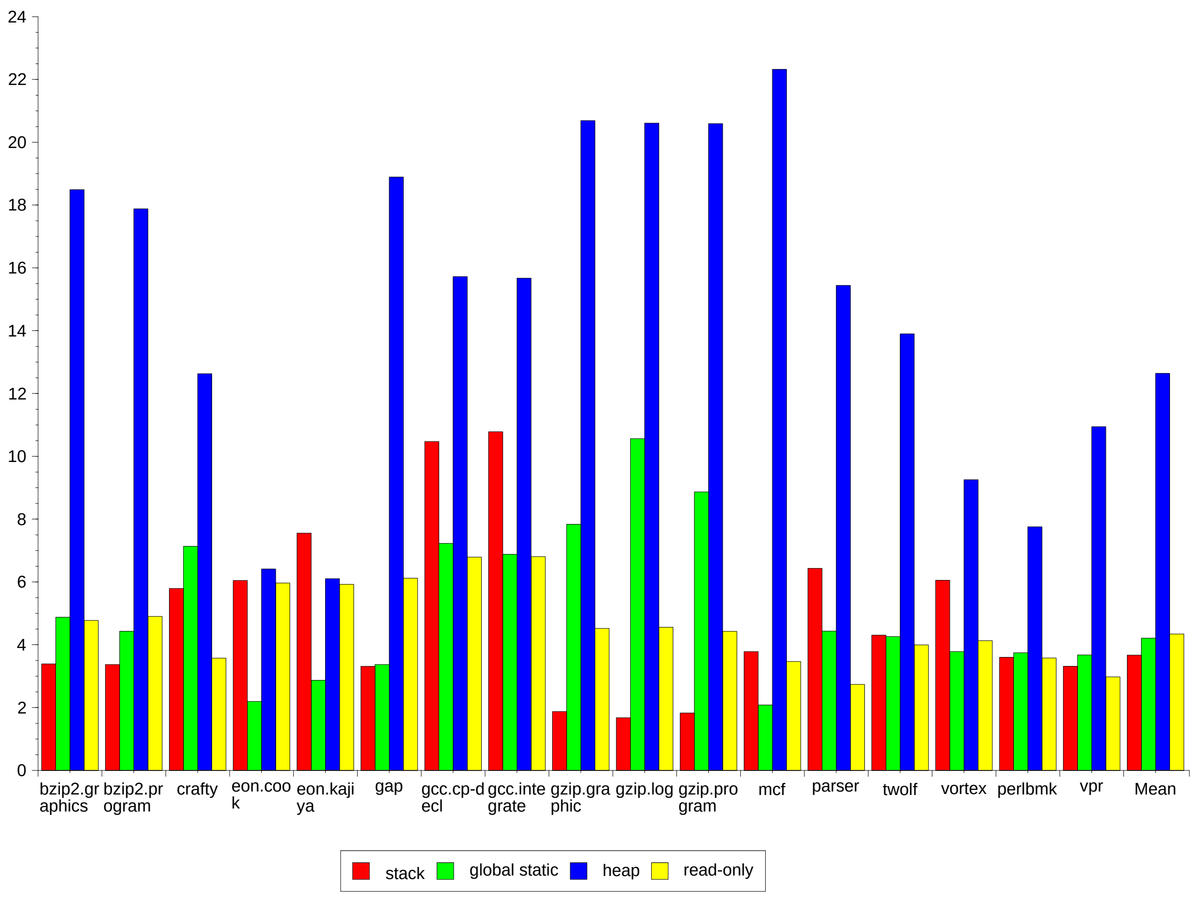


Figure 3.11: Entropy of Cache Frames

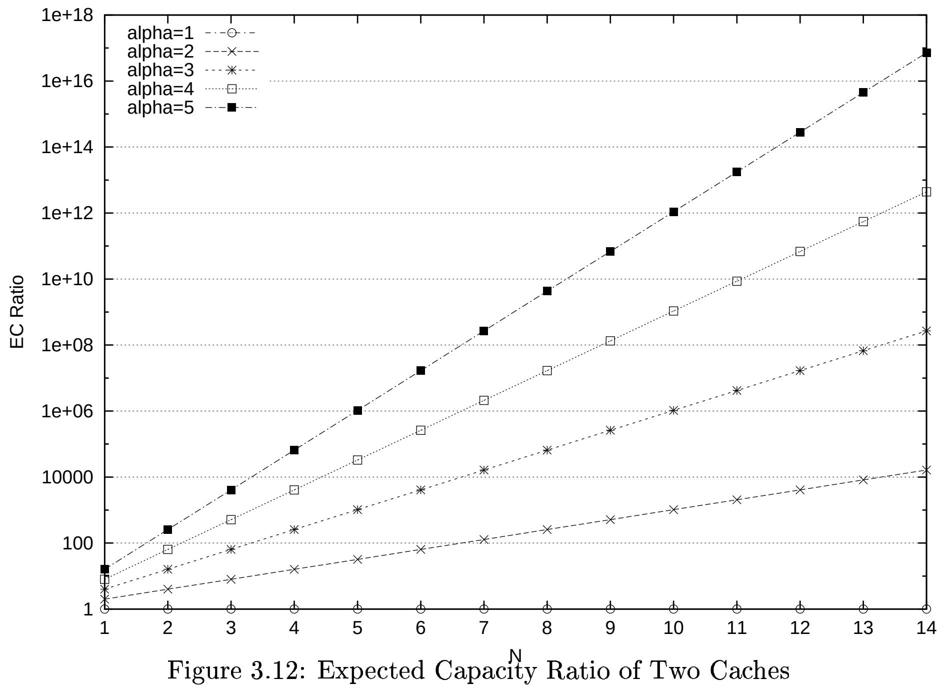


Figure 3.12: Expected Capacity Ratio of Two Caches

uncacheable, speculative write-combining or write protected [32] designed for special data manipulation. A *write-through* cache simply writes through the entire memory subsystem upon a data write to maintain a consistent memory state. It could throttle the memory bus, as it generates too much bus traffic, if memory bandwidth is limited. Instead of updating the entire memory hierarchy every time a cache line write occurs, a *writeback* cache designed to alleviate excessive traffic by associating a dirty bit to each cache line. Whenever a data write occurs to a memory location found in the cache, the dirty bit is set. This dirty cache line will not be propagated down to the lower level memory hierarchy until it is replaced due to a conflict miss or a context switch. Due to its updating mechanism, the lower level memory will not always contain a consistent state, therefore, a snooping protocol needs to be implemented when other agents on the system, e.g. a shared-memory machine, need a copy of this dirty data.

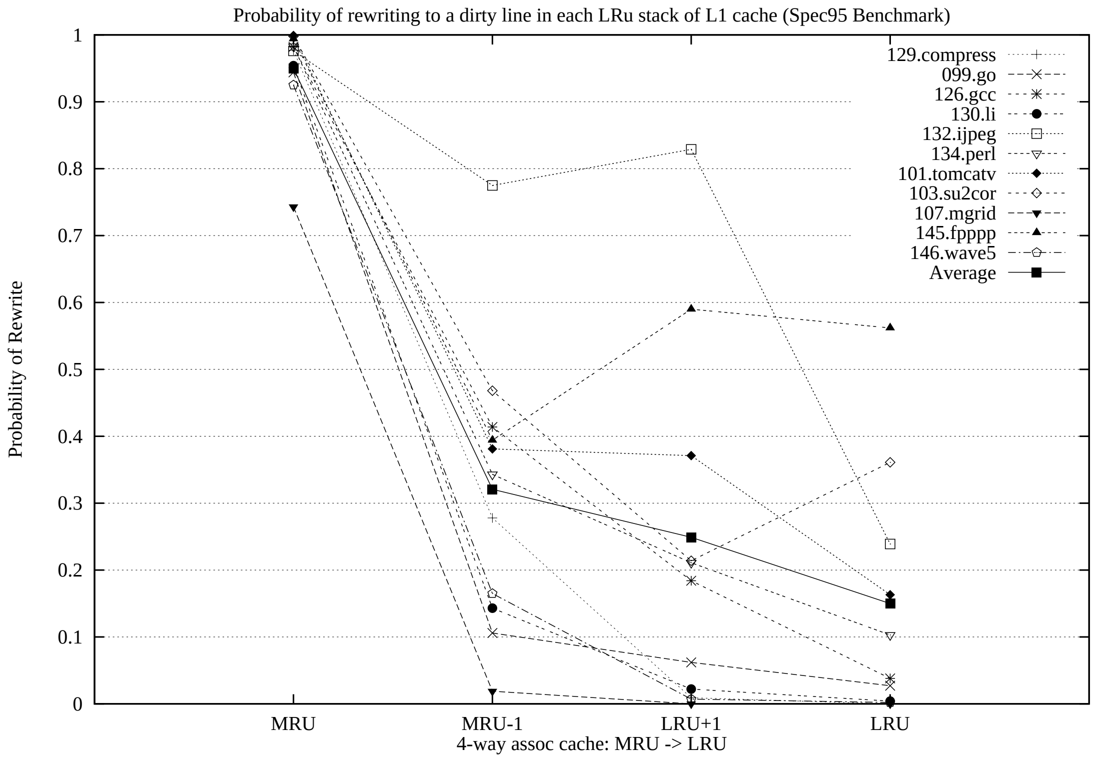
The goal of the analysis here is to investigate the characteristics of "liveliness[3]" of a dirty cache line. We examined the probability of rewriting a dirty line in a set-associative cache when it was in a given state (MRU through LRU) for the SPEC95 benchmark and four applications from the X benchmark suite. The probability, $P_{re-dirty}(\zeta)$, is defined in the following. Given an *N-way* set-associative cache in which there are N cache lines for each cache set. Assume a *Least Recently Used* (LRU) replacement policy — is employed in the cache. Conceptually, we can correspond a usage state in terms of "recency" to a particular cache line in each set. For instance, the cache line that is just accessed will be identified as the *Most Recently Used* (MRU) line. And the line used to be in MRU state is then dribbled one state down to the state right next to MRU, or "*MRU-1*" state for short. We denote the number of entrances of a dirty cache line l that enters a particular usage state, ζ, as $\Phi_\zeta(\ell)$. The probability of rewriting a dirty line for a particular state ζ is then computed in the following formula.

$$P_{re-dirty}(\zeta) = \frac{\sum_{\ell=1}^{L}(\Phi_\zeta(\ell) - 1)}{\sum_{\ell=1}^{L}\Phi_\zeta(\ell)} = \frac{\sum_{\ell=1}^{L}\Phi_\zeta(\ell) - L}{\sum_{\ell=1}^{L}\Phi_\zeta(\ell)}$$

Using a four-way set-associative data cache for the experiment, our results indicate that cache lines that have been marked dirty and reach the LRU state are very unlikely to be written to again before they are evicted. As shown in Figure 3.13 and Figure 3.14, we show the probability of a line that was marked dirty being written to again as it moves from the MRU state to the LRU state for both L1 and L2 caches. The data cache hierarchy consists of a four-way 16KB L1 data cache and a four-way 512KB unified L2 cache. The line size of these caches is 32 bytes. The graph on the top of Figure 3.13, for example, shows that in the L1 cache the average probability (the solid line) of a dirty line in the LRU state being re-written is 0.15, while the similar probability for a dirty line in the MRU state is 0.95. The X benchmark in Figure 3.14 demonstrates an even more obvious trend. The probabilities of re-dirtying lines in the LRU state are even much lower in the L2 cache — in fact, close to 0 as shown in the graphs on the bottom of Figure 3.13 and Figure 3.14. It suggests that a dirty line that enters the LRU state is seemingly "dead", i.e. unlikely to be updated again, from the viewpoint of memory consistency with respect to the rest of the memory hierarchy.

These figures indicate there are some programs (such as *fpppp* and *su2cor*) that have a fairly high probability of writing to dirty lines after they have entered the LRU state, however. In order to further evaluate these cases, we looked at the ratio of the number of times a dirty line in the LRU state is written to, normalized to the number of times a dirty line in the MRU state is written to. The results are presented in Figure 3.15, which shows that while the probabilities may be high, the actual number of these occurrences is negligible compared to the rewriting that occurs when a line is in other states (MRU, MRU-1, etc.). This property can be well exploited and lead to a more efficient cache replacement policy design, which we will discuss in details in Chapter 6.

[3]By liveliness, we mean the probability of a cache line to be accessed after it is brought into the cache.



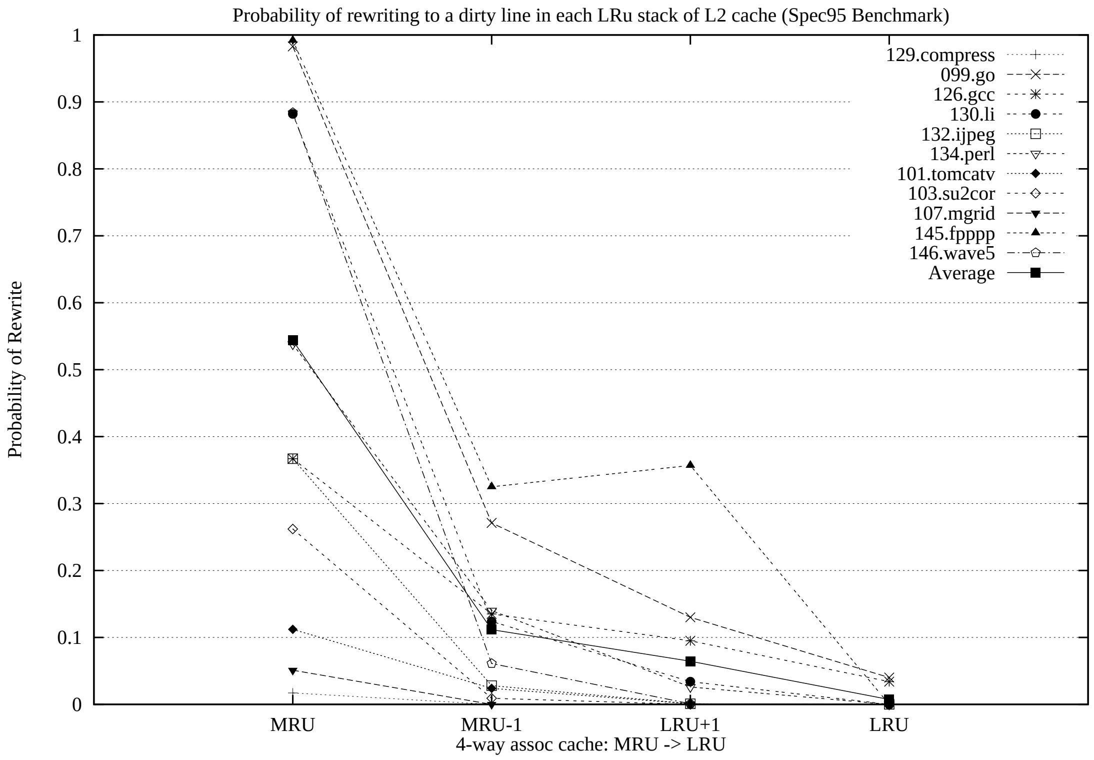
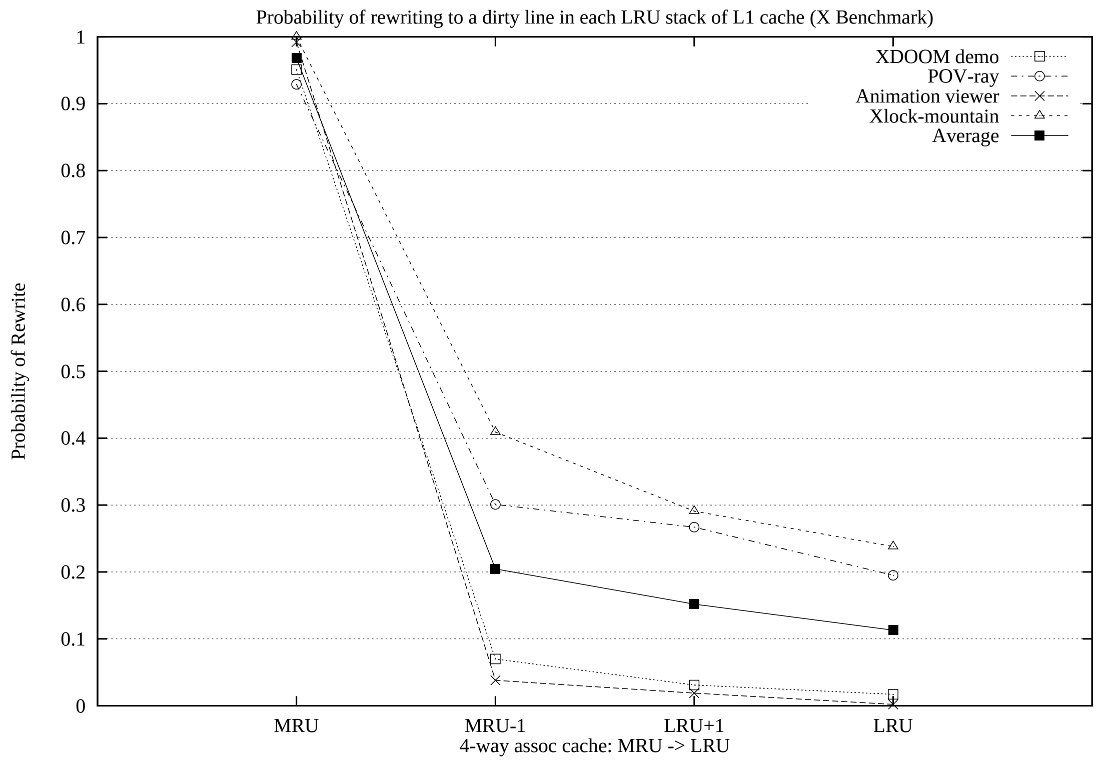


Figure 3.13: Probability of writing to a dirty line in each LRU stack of L1 and L2 caches (SPEC95)



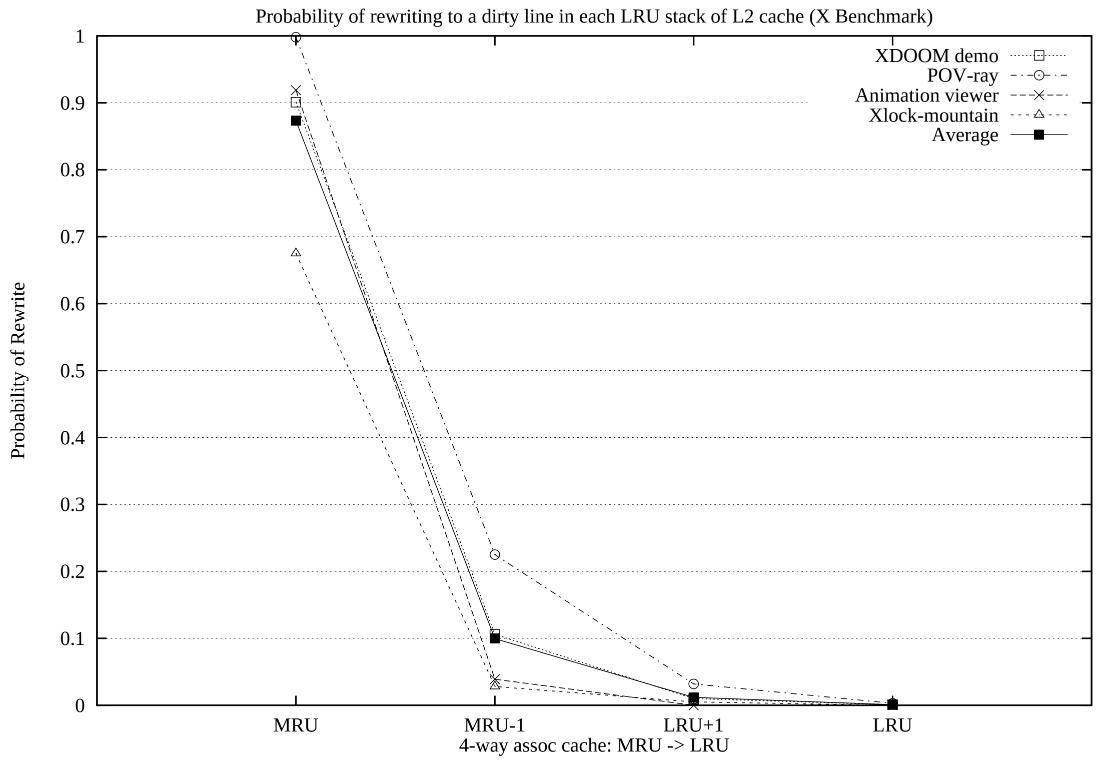


Figure 3.14: Probability of writing to a dirty line in each LRU stack of L1 and L2 caches (X benchmark)

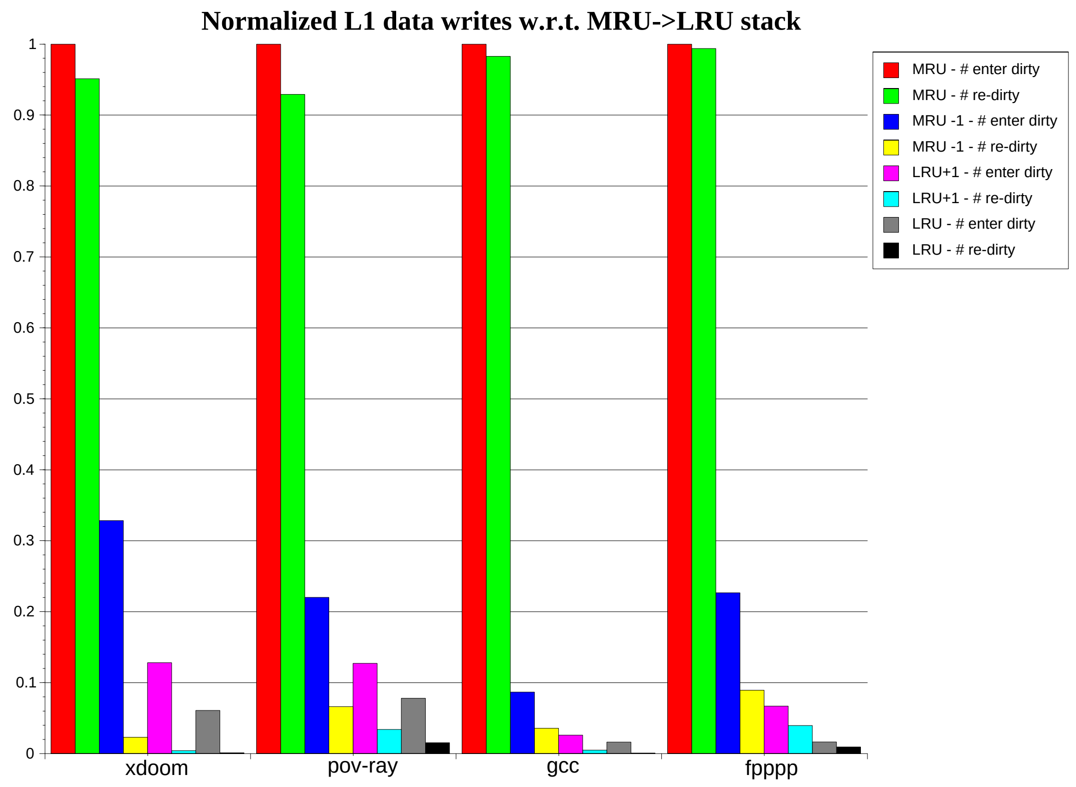


Figure 3.15: Normalized number of writes and rewrites to a dirty line in each MRU-LRU stack

3.5 Chapter Summary

In this chapter, we performed a thorough analysis that characterizes the behavior of data memory references. In the beginning of this chapter, we discussed the concept of virtual memory partitioning by high level programming languages applied to instruction set architectures. We then showed that the stack data convey much less information content than the other major regions, global static data and heap data, in modern application software. We also showed that a larger data cache design is primarily targeted at the more unpredictable heap data while the heap data do occupy the majority of space in the data caches. This leads to a less cost-effective utilization of the cache resource. In the second part of this chapter, we investigated the cache line write behavior and found that dirty cache lines demonstrate an interesting property that can be further exploited for potential improvement in the write policies employed by modern microprocessors.

In the following chapters, we study, analyze, and propose viable technologies that exploit these particular memory reference characteristics to improve processor architectures with respect to energy consumption, performance and memory bandwidth.

CHAPTER 4

LOW ENERGY REGION-BASED CACHELETS

As the feature size of IC process continues to shrink following Moore's Law, manufacturing cost per transistor is decreasing dramatically. While more sophisticated microarchitectural features are being integrated into future generation high performance processors to improve performance, power density (measured in capacitance per unit die area) is increasing, making it necessary to supply large currents and making it more difficult to dissipate heat from the chip [87]. Reducing power requirements has not been the highest priority goal in developing microprocessors targeted at desktop or high-end server market. However, as notebook computers, hand-held computing, mobile and personal telecommunication devices are getting more popular, power is no longer a secondary goal in the process of microprocessor design. As embedded processors gain overall market share, processor designers are targeting more resources to meet high performance requirements while simultaneously reducing power consumption. Researchers from different disciplines including circuits, logic, devices, architectures and even operating systems as well as compilers, are investigating new low-power technologies.

Power dissipation in the memory subsystem constitutes a major portion of the overall power dissipation in these embedded processors [11][45][63][82]. Advances in instruction compression algorithms [78], mixed mode instruction encoding, or compressed instruction encoding such as the ARM Thumb architecture extensions [97] have reduced the power consumption of each instruction fetched and delivered in the instruction cache, but these techniques cannot reduce the power dissipated in the data cache(s).

Several architectural level techniques have been proposed to reduce power consumption in data caches [13][47][66][107][108]. Generally, these techniques achieve power reduction by partitioning the data cache into smaller components. Each data reference accesses a smaller memory structure to achieve power reduction if the desired data are located in the smaller structure. This partitioning can reduce the power required to perform a data access, but the same partitioning generally increases average access latency, caused by an increased miss rate, leading to longer execution time.

Most high-performance processors already employ a split first-level cache structure to partition the instruction code and data into distinct caches. Based on our design philosophy of segregating memory regions described previously, we propose a further partitioning of the data cache into stack, global static and heap regions in this chapter. *Region-based Cachelets* can effectively reduce power by re-directing the stack and global static data accesses into smaller separate cache structures. Region-based Cachelets can achieve this power reduction without increasing average memory latency and execution time. This is due to the high temporal and spatial locality exhibited by stack and global static data references as discussed in Chapter 3. We examine several region-based cachelets designs and quantify their advantages with respect to performance and power efficiency in this chapter.

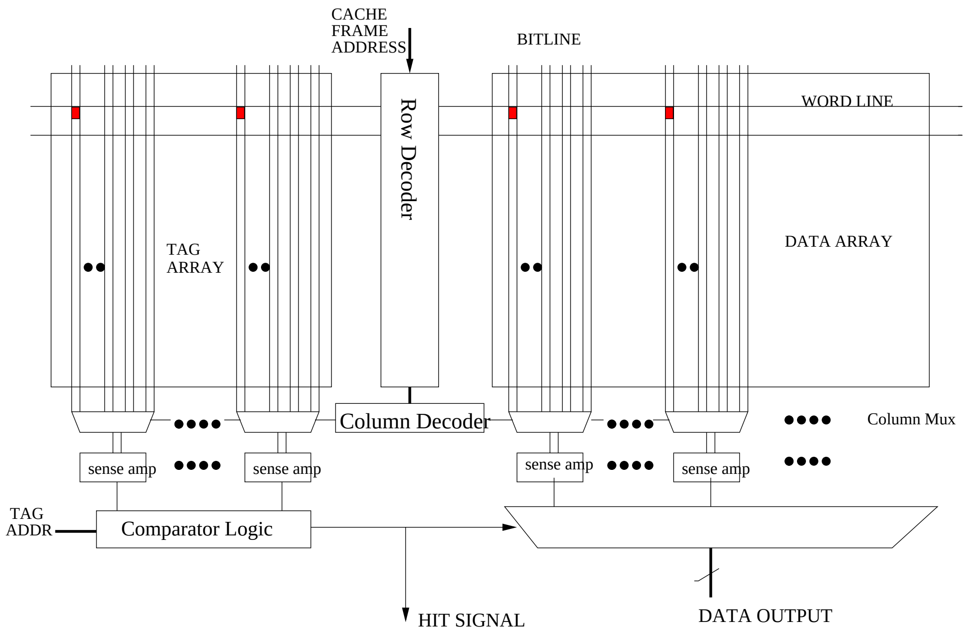


Figure 4.1: Cache Memory Organization

4.1 Cache Structures and Power Dissipation

4.1.1 Overview of Cache Structures

Cache size continues to expand in each generation of modern microprocessor design. Because of its effectiveness in improving memory performance and its regularity and density, today's cache space constitutes a significant portion of a microprocessor die budget. A typical cache structure consists of the following primary elements — address tag arrays, data arrays, row and column decoders, sense amplifiers and comparison logic for matching address tags. The cache power consumption can be modeled as a high level function of the following parameters: the size of tag arrays, the size of read/write ports, the length of wordlines and bitlines, the sizes of the decoders and sense amplifiers. Figure 4.1 illustrates a common cache structure used in contemporary general purpose microprocessors for instructions and data storage. For each cache access, the decoder decodes the reference address and then enables the appropriate row. Only the corresponding decoded row of the wordlines in the address tags and data arrays will be driven. Data from the memory cells of the enabled wordline will be sent across the dual bitlines ($bitline$ and $\overline{bitline}$) to the multiplexers and the requested data will be filtered out through the bitlines to the sense amplifiers. Each sense amplifier detects the bitline changes and amplifies the data signal accordingly. If there is a match in address tag, then its corresponding cache line stored in the data arrays will be sent to output, otherwise, a cache miss occurs and data will be retrieved from the next level of memory hierarchy.

4.1.2 Power Models

Total power consumption of CMOS circuits can be primarily attributed to the following three sources: static leakage dissipation, dynamic short-circuit dissipation and dynamic switching dissipation as summarized in the equations below.

$$P_{total} = P_{leakage} + P_{sc} + P_{switching}$$

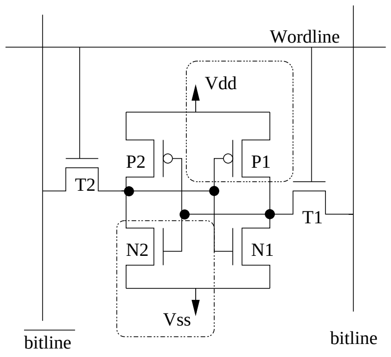


Figure 4.2: A 6-transistor CMOS SRAM cell

$$P_{total} \;\; = \sum_{i=1}^{n} I_{leak,i} * V_{dd} \;+\; I_{sc} * V_{dd} \;+\; \alpha_{0\to 1} C_L V_{dd}^2 f_{clk}$$

The first component in these equations, leakage power — a static power dissipation, is the product of the device leakage current and the supply voltage. Even though there is no direct path between power (V_{dd}) and ground (V_{ss}), there is tiny leakage current flowing through each device. The leakage current is due to the reverse-biased leakage between the substrate and the diffusion regions of a CMOS device, where parasitic diodes are formed. The leakage current is mainly determined by fabrication technology. As deep-submicron CMOS process technology advances and supply voltage is reduced, leakage power dissipation is ever worsening because sub-threshold leakage current is increased when transistors threshold voltage is reduced. Recently, researchers [65][89] have proposed novel techniques at the architectural level to cut off supply voltage V_{dd} through resizing cache structures for leakage power reduction.

The second component, short-circuit power dissipation, occurs during the brief period when both the *NMOS*-transistor and the *PMOS*-transistor are simultaneously active, generating a current pulse from supply (V_{dd}) to ground (V_{ss}). This brief period is correlated to the transition time of an input signal. A longer rise or fall time of the input signal results in a longer active short-circuit path.

The switching power dissipated dynamically is required to change capacitor state — charging or discharging capacitors when state changes from logical 0 to 1 or vice versa. This is the largest component of the total chip power consumption in modern digital CMOS circuits. The elements of the dynamic power equation include the average switching probability α, the capacitive load (C_L), the supply voltage (V_{dd}), and the clock frequency (f_{clk}). Except for clock buffers, the α value typically is much smaller than one. Note that the power dissipation is proportional to the square of supply voltage. Generally, designers can effectively reduce dynamic power consumption by reducing the supply voltage although this exacerbates the leakage power consumption as mentioned above. Reducing load capacitance or switching frequency can reduce dynamic power dissipation as well.

In the following sections, we will concentrate on the dynamic power consumption of cache structures by assuming that the static leakage power and short-circuit transient power dissipation can be completely ignored.

4.1.3 Power Modeling in Cache Structures

Consider a conventional six-transistor CMOS SRAM cell as shown in Figure 4.2 as one bit for each cached datum. Unlike the pseudo-NMOS implementation, the SRAM cell does not consume standby power, except for leakage current, when no read/write operations occur; when a 1 is stored in the cell, only the transistors encompassed by the dashed boxes are active during static time. Since there exists no direct path that conducts current from V_{dd} to V_{ss}, only leakage power is dissipated.

At the beginning of a read operation, both bitlines are precharged to V_{dd}. Then the read operation is started by asserting the wordline that enables the corresponding NMOS pass transistors — T1 and T2. If a one is stored, then the $\overline{bitline}$ will be discharged through T2 and N2. During the course of a write operation, the *bitline* and $\overline{bitline}$ are charged to the desired values, then the wordline is asserted to activate T1 and T2 for writing data into the cell.

As outlined in Section 4.1.1, the cache power sources are primarily originated from decoders, wordline drive and bitline discharge for both tag and data arrays, and sense amplifiers. We use the Wattch tool [18] that adopts model parameters such as transistor sizes defined in CACTI 2.0 [92]. The power modeling for each wordline and each bitline are similar. The entire single wordline capacitance is a sum of the diffusion capacitance of the wordline driver from the decoder, the gate capacitance of the pass transistors across each memory cell and the metal wire capacitance. The bitline capacitance is computed analogously. The following equations describe the components of these capacitive loads.

$$C_{wordline} = C_{diff-wordline\_driver} + C_{gate-pass\_Nt} * num\_bitlines + C_{metal} * wordline\_len$$

$$C_{bitline} = C_{diff-precharge} + C_{diff-pass\_Nt} * num\_wordlines + C_{metal} * bitline\_len$$

To estimate the power of sense amplifiers, a model proposed in [122] was used. The typical sense amplifier is an inverter with the input that connects to the bitline. The equation below shows the power modeling for each sense amplifier.

$$P_{SA,inv} = \frac{V_{dd}}{8} * I_{dsat} \qquad where \quad I_{dsat} = 0.5mA$$

We assume that 0.35μm process technology parameters are used in this study. Under this assumption the leakage current power can be ignored [87]. The P_{sc} component is typically small and there exists design and fabrication technologies [24] to minimize the short-circuit current, I_{sc}. The dominant component of the total power dissipation is $P_{switching}$, i.e., transitions that charge or discharge the load capacitance [59][90]. Generally, smaller cache structures induce less dynamic power because of the shortening of wordlines and bitlines that contain less wire capacitance. It also could have a smaller decoder and a smaller number of sense amplifiers.

4.2 Locality of Data Cache Regions

Although the region-based caching technique generally works for high-performance processors, embedded processors used in mobile devices are more power-conscious due to limited battery life. Therefore, we focus our implementation for embedded processors and use the Mediabench benchmark suite for our experiments in this chapter. To recapitulate the results shown in Section 3.2.1, applications from Mediabench contain an average of 24% of instructions accessing memory. Stack references average 40% of all memory references, while global static data references and heap references average about 30% each of the total memory references.

To understand the access locality of a cache line brought into the L1 cache, we calculated the number of cache line hits prior to a line eviction. We refer to the total number of access hits prior to cache line eviction as the *life span* of a cache line. Figure 4.3 illustrates the average life span of a cache line in each data region. In this experiment, all the data regions compete in a single monolithic L1 data cache. Simulations were performed for cache sizes from 256B to 64KB with fully associative cache (represented by FA with solid lines) and direct-mapped cache (represented by DM with dashed lines.) The y-axis plots the cache line life span on a log scale. For most cache configurations and applications, the stack cache lines show the greatest life span, the heap cache lines have the shortest lifespan, and the global static cache lines fall between. For example, for a

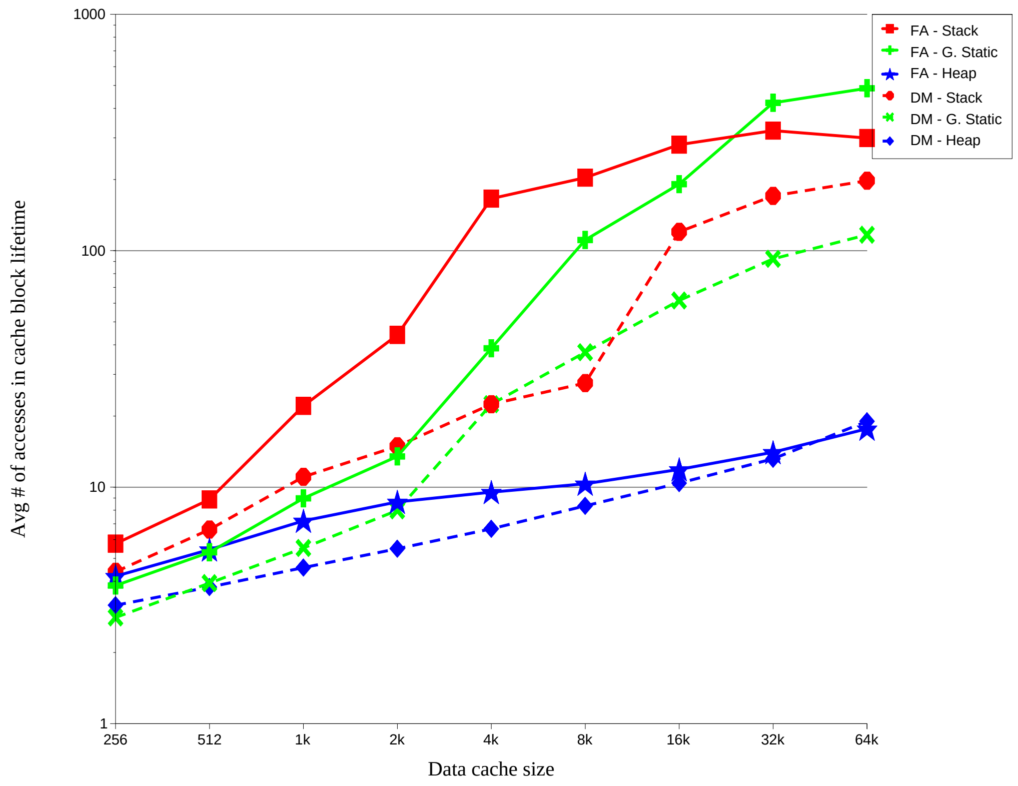


Figure 4.3: Average Life Span of Cache Lines by Regions for Mediabench

fully associative 4KB L1 cache, stack cache lines has an average life span of 166 — i.e., each line was re-accessed an average of 165 times prior to eviction. In contrast, heap cache lines show an average life span of only 9.5.

Figure 4.4 shows the miss ratios for a spectrum of cache sizes from 256B to 64KB assume that a dedicated cache is allocated for each individual memory region. These data show that the stack data consistently demonstrate the best cache locality for a given cache size. Furthermore, the miss rate approaches 1% for a very small (2KB) stack cache. The heap data show the worst locality with a miss rate decreasing *linearly* as the cache size doubles, reaching 5% at a 64KB heap-cache. It also exhibits an observation that large cache designs are primarily effective for heap data only. As expected, the miss rate of global static references falls between the stack and heap approaching a 1% miss rate at a relatively small (4KB) global-static-cache configuration. These experiments show that by partitioning the cache structure into three components — a small stack cache, a small global static cache and a larger cache for heap and others — a majority of memory references access small cache structures (40% stack cache, 30% global static cache) while retaining a high hit rate; since the caches are small, they consume less power; since the hit rates are high, they provide good performance with low access latency.

4.3 Region-based Cachelets

Recent energy reduction techniques proposed in architectural level cache designs can be classified into two primary schemes, vertical partitioning and horizontal partitioning. The basic idea of these partitioning techniques is to reduce power dissipation by referencing a smaller storage structure. For the vertical partitioning as shown in the left hand side block diagram of Figure 4.5, which employs a multi-level cache hierarchy, an extra level of storage structure is added nearest to the processor. This

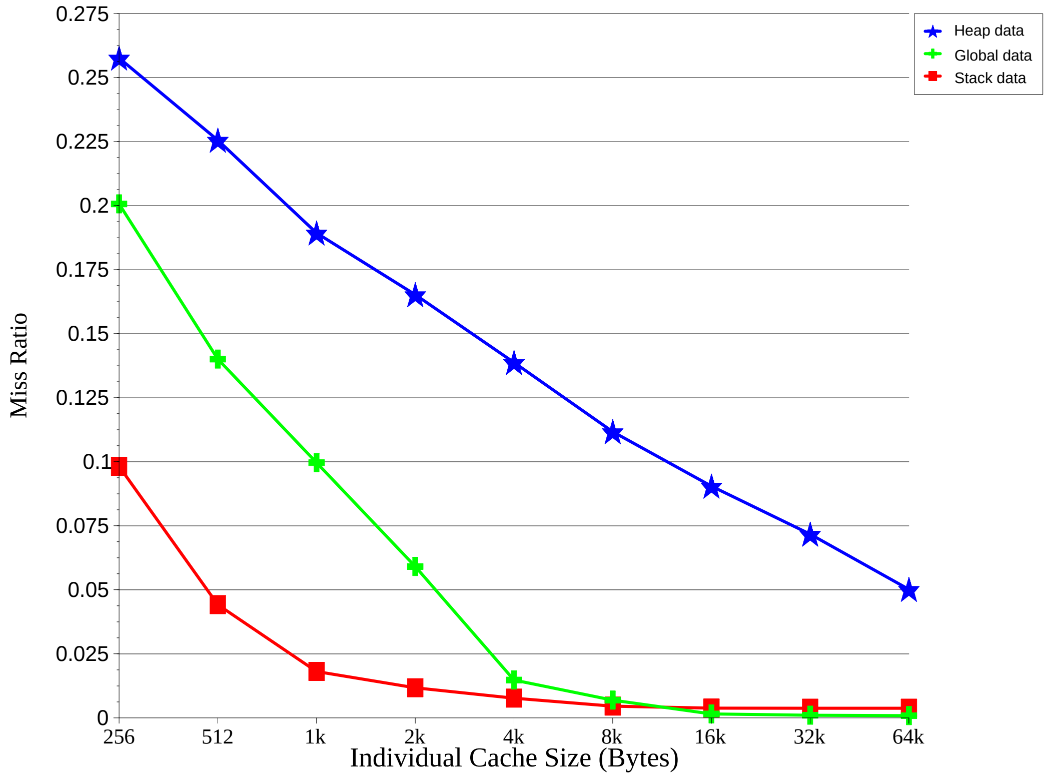


Figure 4.4: Average Memory Region Miss Rates for Mediabench

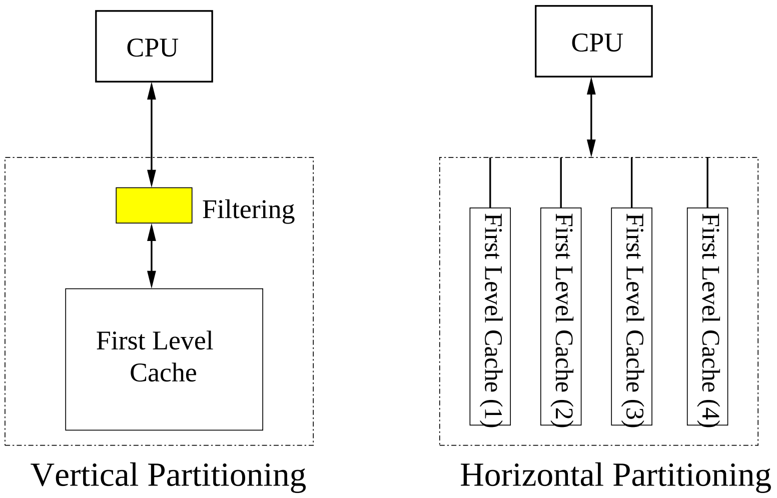


Figure 4.5: Low-power Cache Partitioning Scheme

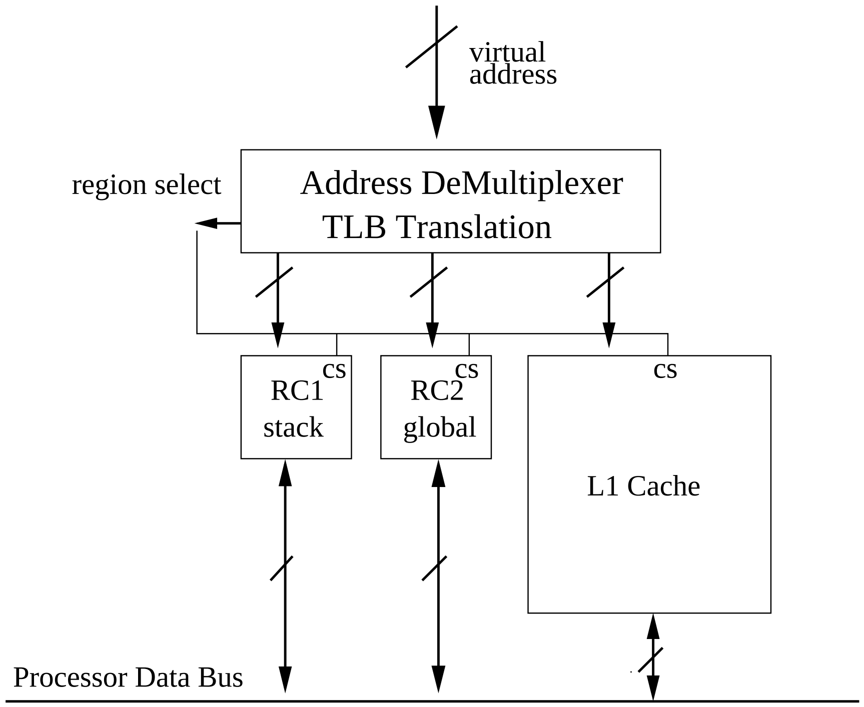


Figure 4.6: Region-based Cachelets

extra level storage can be a line buffer [47][107] or a small filter cache [66]. These extended structures capture short-term data locality and would consume less dynamic power when the requested data can be found in the small buffer or cache. However, the hit rate for this small structure is often relatively low and each miss requires another L1 data cache access after this miss is determined; this increases the effective latency of an L1 access since the L1 access request is delayed.

An alternative to vertical partitioning is to perform a horizontal cache partitioning. Horizontal partitioning as shown in the right hand side of Figure 4.5 involves slicing each cache set into smaller segments (e.g., cache sub-banking [47][108]). The processor accesses (and powers up) only the line segment that is being referenced (requiring additional early address decode circuitry), saving power by not driving data paths in the cache that are not referenced. This approach is orthogonal to vertical partitioning.

Region-based Cachelets, our proposal in this chapter, is yet another horizontal partitioning design method that can reduce power dissipation of data caches more effectively by exploiting the nature of memory allocation conventions and memory reference characteristics. The basic idea of this approach is to partition data references based on semantically defined memory regions into distinct cachelets. Data exhibiting high degree of utilization and locality, e.g. stack data or global static data as discussed in Section 4.2, can be filtered out from the regular cache. Figure 4.6 sketches one implementation of a region-based cachelets design in block diagram. In this example, two horizontally partitioned region-based caches are added alongside of the regular L1 cache — one for stack data and one for global static data. All heap and other memory references are sent to the L1 cache as normal. All cache miss fill requests and evictions are directed to the next-level caches or DRAM memory. The region cache is activated (drawing power) only when a memory reference is made to its respective memory region. Note that the stack and global static region cachelets, based on the requirements of target applications in the embedded processor, can be built much smaller than the L1 cache.

The region-based cachelets design provides several benefits. First, line conflicts are eliminated among semantic regions since different region accesses are routed to different structures. This makes it more feasible to implement each cache with lower associativity; particularly the stack cache since the active region of the stack is generally a single contiguous section at the top of stack [75]. A direct mapped stack-cache generally has no more conflict misses than a fully associative stack cache [75]. Second, building smaller separate caches provides more flexibility in increasing overall data storage than enlarging a single cache. For instance, to enlarge a 32KB direct-mapped cache, one needs to either double the cache size to 64KB or opt for a possibly higher latency multi-way cache, e.g. a 5-way 40KB cache. Finally, as mentioned earlier, a smaller cache dissipates less dynamic power when accessed. Since about 70% of the references hit in the stack and global static data regions, the overall data cache power consumption can be significantly reduced when the sizes of those caches are made small (but large enough to retain a high hit ratio). We will quantify the performance impact in our analysis in Section 4.5.

This chapter does not intend to investigate the implementation details for the region-based cachelets, yet we will discuss some of these implications. Since there are multiple caches at each level of memory hierarchy within our implementation (the first level cache in our study), it could add extra complexity in the cache snooping protocols for supporting a shared-memory multiprocessor system. The first design option is to choose between virtual address cachelets and physical address cachelets. A physical cache implementation makes the design of cache snooping protocols less complicated. It maintains cache coherency between different processes using physical memory addresses to avoid address aliasing issues. Many commercial microprocessors implemented physical caches. An alternative is to design a virtual cache, which can eliminate the TLB translation latency if the access is a cache hit. Without a process ID number associated with each cache line, however, a virtual cache is required to be entirely flushed every time a context switch occurs because of aliasing issues. Nevertheless, the region demultiplexing is based on virtual address space in our region-based cachelets scheme. To enable a physical cache design, the address demultiplexing (to determine which cachelets to go to) should be determined prior to an address translation. In such a design, either each TLB entry should possess ID bits to identify which physical cache the current cache access should be routed to, or each region implements their own TLB.

Another implementation issue is how do we determine which particular region a virtual address falls into. The least expensive way is to hardwire the virtual address into some approximate partitioning in the address demultiplexing hardware if a *private virtual memory* is supported. Since the virtual address partitioning of each region for most of the ISA design is relatively distant from each other as shown in Figure 3.1, therefore, we could mask out the lower bits of a virtual address to segregate the semantic address space. A more accurate partitioning would rely on information provided by the operating system for each process it invokes. This requires a few specialized control registers that store the base address of each semantic region, then the processor partitions the address streams based on the contents of these system level registers.

4.4 Simulation Framework

4.4.1 Machine Models and Simulators

In this study, *Wattch* is used to evaluate relative performance and power dissipation for different processor design configurations by integrating the region-based caching mechanism into *Wattch*. The power modeling of this study was previously discussed in Section 4.1.2.

Our baseline machine model resembles the Intel StrongARM SA-110 microprocessor [82]. The Intel StrongARM SA series have been widely adopted in set-top boxes, Internet terminals and PDA devices such as the Compaq iPAQ Pocket PC [30]. The microarchitecture of our baseline machine model is a single-issue in-order processor with a conventional five-stage pipeline. The processor contains a unified 32KB on-chip level-one cache. The size and associativity of the L1 cache and its corresponding access latency were varied according to the access timing information gathered from CACTI 2.0 [92]. In order to perform a fair comparison in both performance and power dissipation, a four-way 512KB level-2 cache is incorporated for both the baseline machine and the region-based cachelets machine[1]. The caches are blocking caches that will stall instruction execution followed by any cache miss. All the caches are single-ported.

The *Wattch* tool estimates power at the architectural level by storing the event occurrences of each functional unit during simulations. We assume that a simple clock gating [45] technique is applied to each cache module; therefore, each cache is activated only when an access is requested — zero power dissipation otherwise. The device capacitances used in *Wattch* are similar to those published in [117]. The power consumption models of each cache consider typical components of a cache array structure including tag arrays, address decoder, wordline drive, bitline drive, and sense amplifiers as we had discussed in Section 4.1.3.

4.4.2 Energy-Delay Product Metric

In [48], Gonzales and Horowitz argue that the widely used metric, **energy**, measured in $Watt/MIPS$ or $Watt/SPEC$, is not an ideal metric for evaluating the efficiency of a machine design. By simply reducing supply voltage or load capacitance, energy can be reduced at the expense of increasing circuit delay. For such a design, a lower energy processor would also have lower performance. Instead of using the **energy** metric, they advocate using the **energy-delay product** (EDP), calculated by $Watt/SPEC^2$, as the metric for an efficient system design. The EDP considers both performance and energy simultaneously in a design. To achieve an energy efficient design without compromising performance, a design should attempt to minimize the EDP. If a processor trades off performance for energy, then its EDP will be unlikely to decrease.

Our results show the EDP of a given machine relative to that of the baseline machine model as the comparison metric (in addition to performance and power). The following equations describe how we compare the EDPs of two machines. For a target machine A, a better design will reduce its EDP ratio with respect to that of a base machine. In other words, the goal of an energy-delay efficient system design should minimize the EDP Ratio, i.e. $\frac{EDP_A}{EDP_{Baseline}}$.

$$E = \frac{Watt}{MIPS} = W * Delay$$

$$EDP = E * D = W * (Delay)^2$$

$$\frac{EDP_A}{EDP_B} = \frac{W_A * (Delay_A)^2}{W_B * (Delay_B)^2} = \frac{W_A}{W_B} * \frac{1}{(Speedup_{\frac{A}{B}})^2}$$

$$\therefore \quad EDP\ Ratio = \frac{EDP_A}{EDP_B} = \frac{Power\ Reduction_{\frac{A}{B}}}{(Speedup_{\frac{A}{B}})^2}$$

4.5 Simulation Results and Analysis

The Mediabench benchmark suite [72] was used in this study. Section 2.2.2 contains a detailed description of Mediabench. All the simulations were run to completion except for *mpeg2decode* and *gs* that exit after 600 million instructions to reduce simulation time.

We present our simulation results and analyze them in this section. First we evaluate one region-based configuration, comparing that configuration with alternative conventional cache configura-

[1]DRAM memory power is not modeled in the *Wattch* toolset, an L2 is simulated as a common backing storage for all machine models. We evaluate dynamic power consumption for all levels of cache hierarchy.

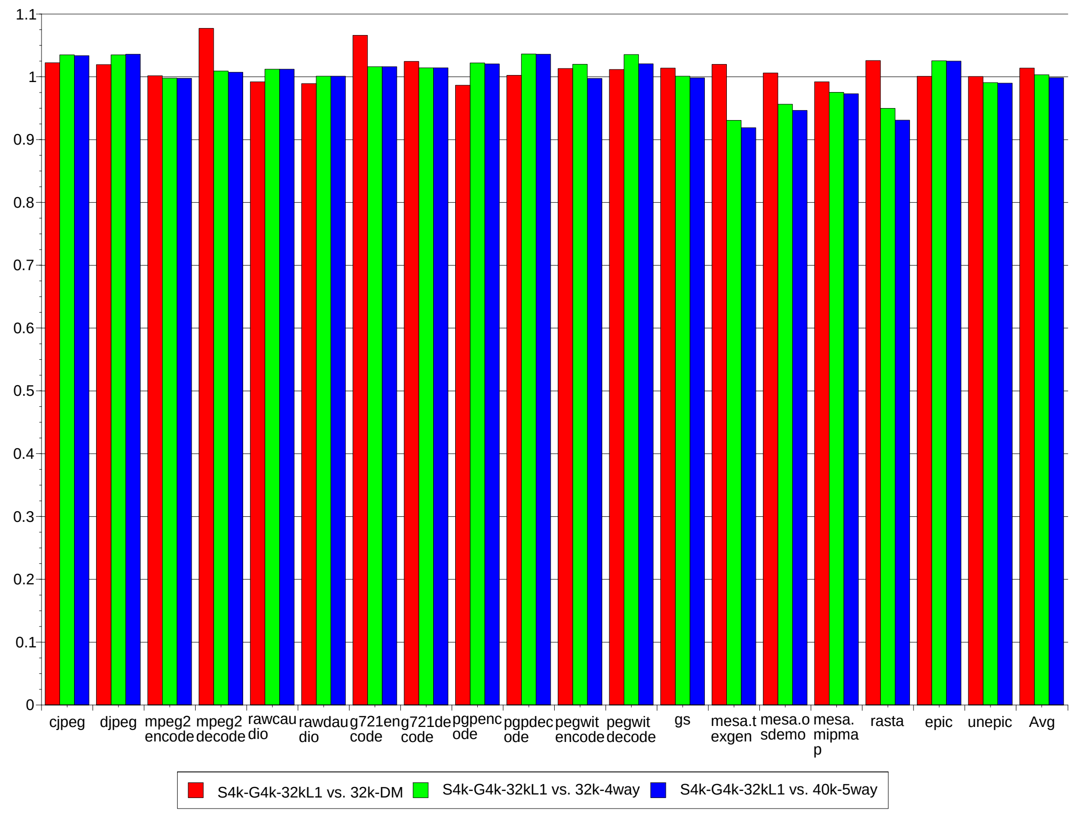


Figure 4.7: Performance Comparison of Machines with Region-based Cachelets

tions. Then we more fully explore the design space for different region-based caches configurations. Finally we evaluate a compound design by incorporating a filter cache into our region-based cachelets.

4.5.1 Comparisons with Baseline Design

In this set of experiments, we compare the power dissipation and performance of region-based cachelets with a 4KB direct-mapped stack cache, a 4KB direct-mapped global static cache and a 32KB conventional L1 cache. Each cache has a single cycle access latency. This design is compared to three baseline machine designs in these experiments. The first baseline cache uses a 32KB direct-mapped L1 cache with single cycle access latency. The second baseline cache has a 4-way 32KB L1 cache. The third baseline cache expands the cache size to 40KB by increasing the associativity to five ways. Both multi-way caches have a two-cycle latency. As mentioned earlier, we used the timing information gathered from CACTI 2.0 [92] to determine cache access time for each cache configuration. Both 4-way and 5-way 32KB caches had access timing exceeding the 7ns target necessary to achieve single cycle access on our target architecture (they were 11ns and 12ns respectively). The purpose of using a 5-way 40KB cache is to match up the cache capacity of our region-based cachelets in order to perform an apple-to-apple comparison. As mentioned in Section 4.4.1, we add a 512KB L2 cache for all configurations as a common backing storage.

Figure 4.7 shows the performance comparison of our region-based cachelets design with regular cache designs. For the Mediabench applications, the region-based cachelets design performs almost on par or slightly faster than the regular cache designs. It reduces performance between 4% to 7% in *mesa* and *rasta* when compared to the 4-way and 5-way cache designs. For the same L1 size, it is simply because stack and global static data increase much locality moving from the 4KB cache to

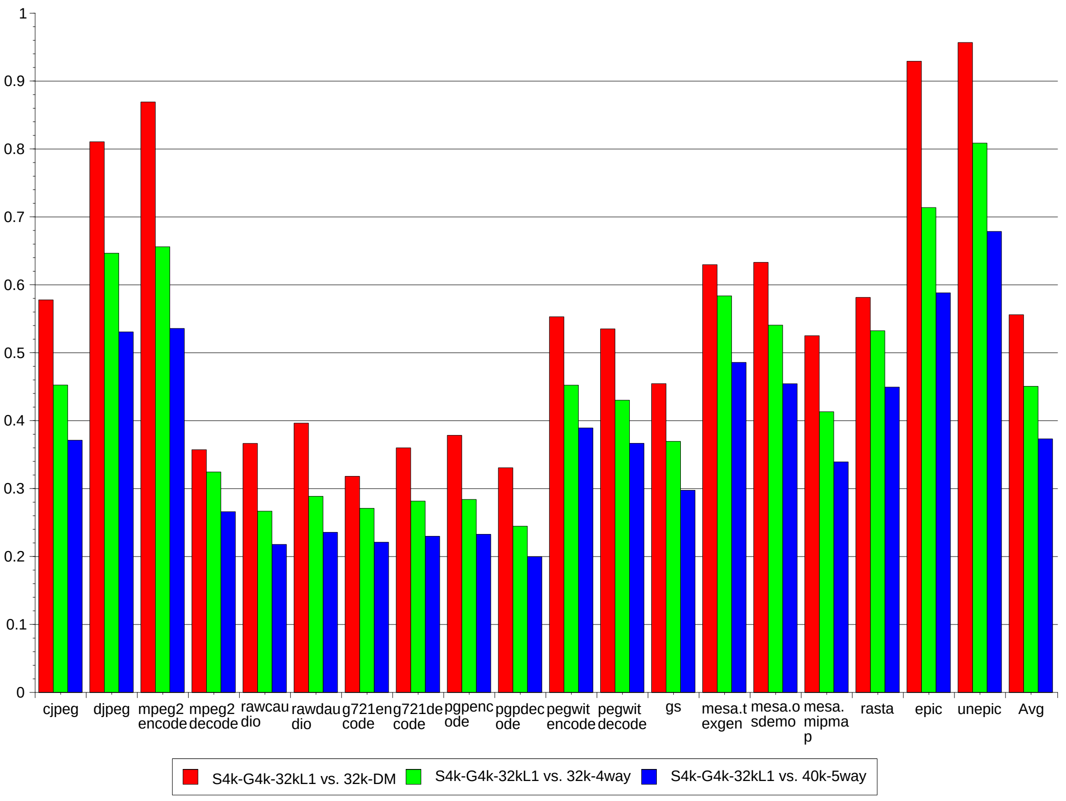


Figure 4.8: Power Dissipation Comparison of Machines with Region-based Cachelets

the 32KB cache. Performance increases relative to all baseline cache design for *cjpeg*, *djpeg* and *epic*. For the 32KB configurations this can be due to the increased overall cache size. There is a relative performance improvement of about 3% for these applications with respect to the 40KB cache as well. This speed-up primarily comes from reducing the access time to one cycle and secondarily from reducing set conflicts among different data regions.

Figure 4.8 demonstrates the power dissipation of data references in all caches. The average relative power dissipation of the region based cache is significantly reduced to 56%, 45%, and 37% of the 32KB DM, 32KB 4-way and 40KB 5-way designs respectively. The major power reduction occurs for stack and global static references that were re-routed to the smaller stack and global static cachelets. Power savings are significantly lower for *unepic*, *epic*, *mpeg2encode* and *djpeg*. This is due to the unusually high occurrences of heap accesses shown earlier in Figure 3.3.

Combining the results in Figure 4.7 and Figure 4.8, the Energy-Delay Product Ratios of the region-based cachelets design versus the baseline machines are plotted in Figure 4.9. This plot is normalized to the EDP of the baseline designs; lower EDP ratios occur when the region-based cachelets scheme is the better cache design — the lower the EDP, the better the design. The average EDP ratio of the region-based cachelets is 0.54 compared to a 32KB direct-mapped baseline cache, 0.45 compared to a 32KB 4-way L1 baseline cache, and 0.37 compared to the alternate 40KB 5-way L1 cache design.

These experiments indicate that a region-based cachelets design consisting of a 4KB stack cache, a 4KB global static cache and a 32KB L1 cache will achieve the same execution performance as a 40KB, 5-way cache while achieving a much more energy efficient implementation.

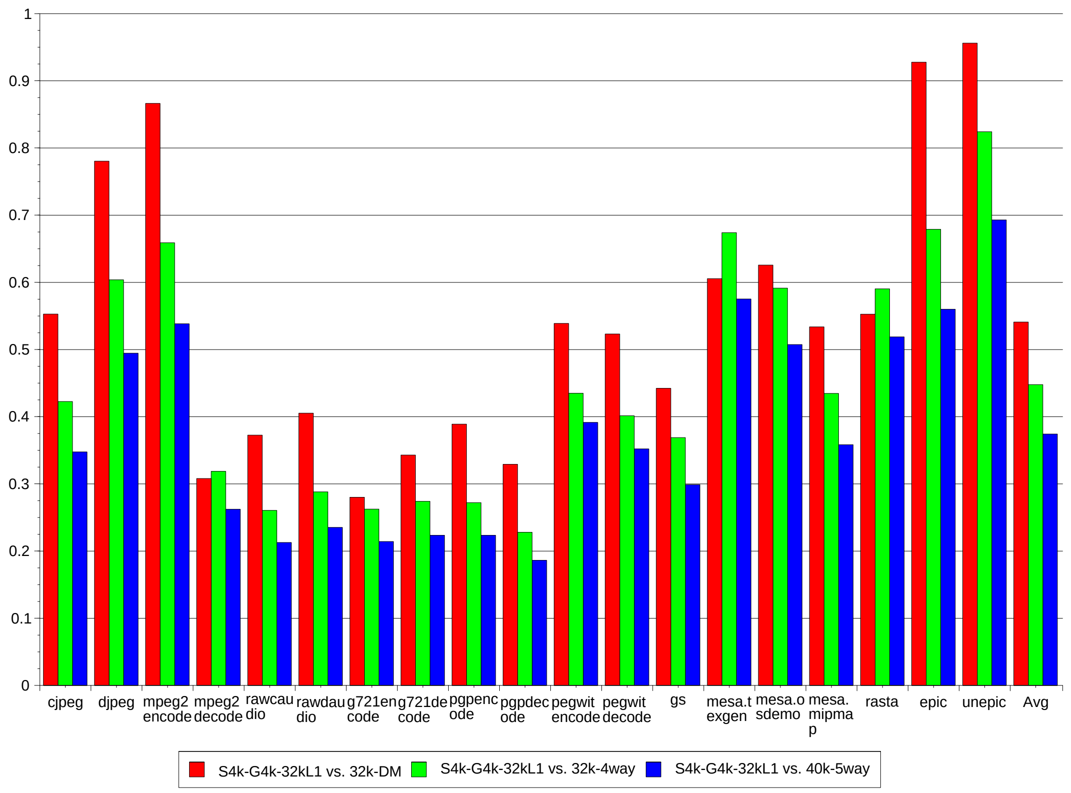


Figure 4.9: Energy-Delay Product Comparison of Machines with Region-based Cachelets

4.5.2 Exploiting Design Space of Region-based Cachelets

In this section, a spectrum of region-based cachelets design choices is investigated. In all comparisons, we use the 40KB, 5-way cache presented in Section 4.5.1 as the baseline for comparison. We examine seven different region-based cachelets configurations in Figure 4.10, varying the sizes of the stack and global static regions. Each cache configuration uses a 32KB, direct-mapped L1 cache (represented as *dm*), except for the leftmost bar which uses a 32KB, 4-way conventional L1 cache (represented as *4w* in the symbol). We use the following naming conventions in the figure. The *SmGn* symbols represent the size of region-based cachelets: an *m*KB Stack cache and an *n*KB Global static cache; when G is absent, there is no global-static region cache and global static data are routed to the L1 cache. For example, the rightmost configuration *S2G2-dmL1* consists of a 2KB stack cache and a 4KB global static cache and a 32KB, direct mapped L1 cache.

Figure 4.10 shows the average performance speedup, power reduction and energy-delay product ratio for Mediabench. Table 4.1 lists the energy-delay product ratio for each application in the benchmark (used to calculate the average). The 2KB stack cache and 2KB global static cache (S2G2-dmL1) demonstrates the best design in EDP ratio. It consumes only one third of the power in a 5-way 40KB counterpart while achieving 99% of the execution performance. Only three of the applications have EDP ratios above 0.50 while 7 of the 19 applications have EDP ratios of less than 0.20. This shows that the overall performance and energy efficiency of region-based cachelets is significantly better than the design alternatives studied. Region-base caching reduces the power dissipation by routing data references to small, special purpose cache structures. High hit rates are maintained because the routing algorithm exploits known characteristics of high-level language programs. These hit rate translate into high performance execution, while retaining the power dissipation advantage.

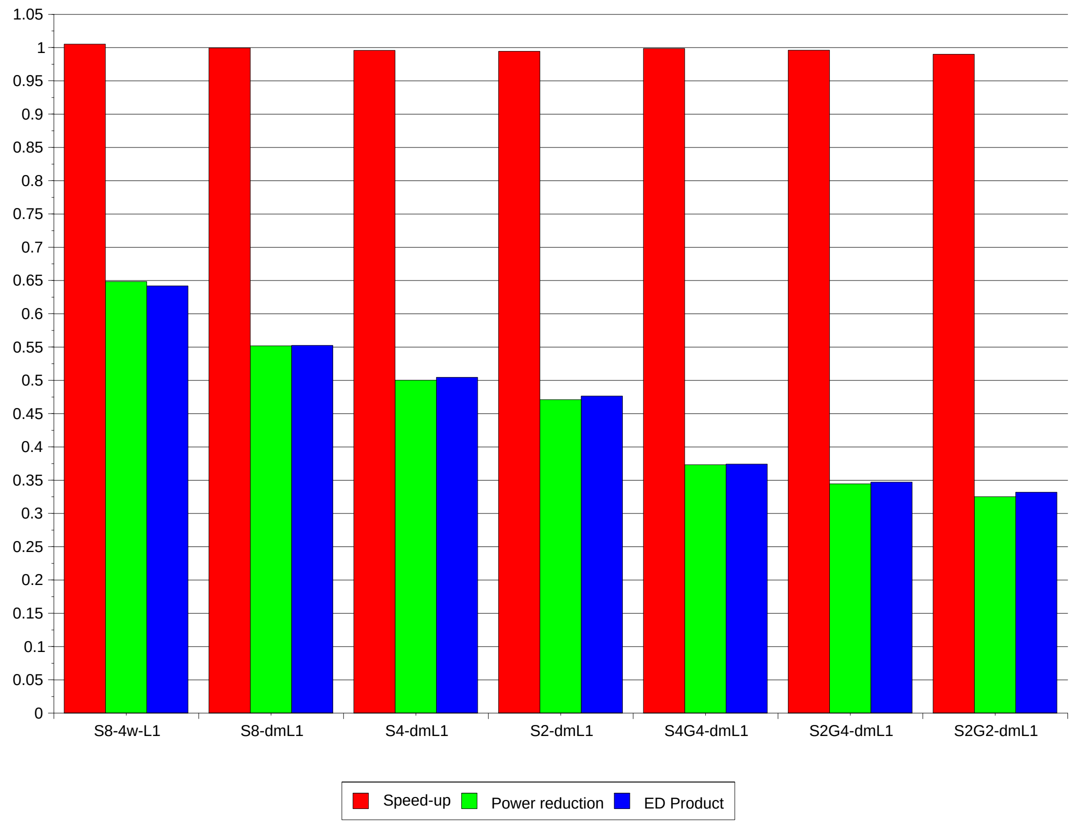


Figure 4.10: Average Performance, Power and EDP for Different Region-based Cachelets Machines (Baseline: 5-way 40KB L1)

EDP	S8-4w-L1	S8-dmL1	S4-dmL1	S2-dmL1	S4G4-dmL1	S2G4-dmL1	S2G2-dmL1
cjpeg	0.553	0.448	0.381	0.337	0.348	0.304	0.298
djpeg	0.717	0.530	0.517	0.483	0.495	0.477	0.475
mpeg2encode	0.738	0.583	0.561	0.548	0.538	0.526	0.523
mpeg2decode	0.661	0.509	0.467	0.443	0.262	0.237	0.188
rawcaudio	0.815	0.570	0.570	0.569	0.213	0.212	0.143
rawdaudio	0.815	0.580	0.580	0.579	0.235	0.235	0.165
g721encode	0.551	0.611	0.523	0.470	0.214	0.167	0.129
g721decode	0.550	0.508	0.426	0.376	0.223	0.176	0.139
pgpencode	0.702	0.512	0.482	0.471	0.223	0.205	0.150
pgpdecode	0.674	0.499	0.462	0.441	0.186	0.165	0.105
pegwitencode	0.564	0.501	0.420	0.348	0.391	0.331	0.388
pegwitdecode	0.549	0.471	0.390	0.344	0.352	0.305	0.415
gs	0.484	0.484	0.384	0.316	0.299	0.233	0.237
mesa.texgen	0.547	0.681	0.599	0.588	0.575	0.564	0.559
mesa.osdemo	0.582	0.645	0.553	0.500	0.507	0.455	0.446
mesa.mipmap	0.499	0.444	0.432	0.454	0.358	0.376	0.365
rasta	0.611	0.700	0.612	0.571	0.519	0.478	0.470
epic	0.774	0.576	0.561	0.552	0.560	0.551	0.551
unepic	0.809	0.707	0.697	0.692	0.693	0.687	0.653
Average	0.642	0.552	0.504	0.476	0.374	0.347	0.332

Table 4.1: Energy-Delay Product for Mediabench with Different Region-based Cachelets

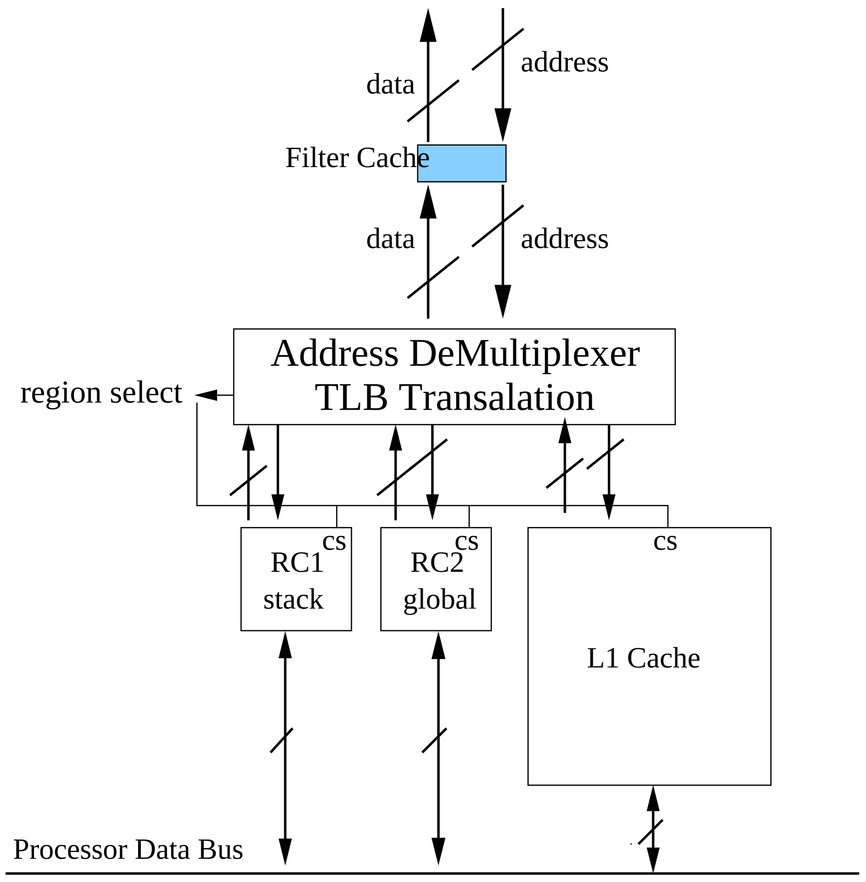


Figure 4.11: Combining Filter Cache with Region-based Cachelets

4.5.3 Combining a Filter Cache and Region-based Cachelets

This region-based cachelets technique proposed in this chapter needs not be a stand-alone technique. In practice, it can be applied along with other proposed low-energy techniques, e.g. *filter cache* [66], to further reduce energy consumption. In this section, we will investigate the design impact, in terms of performance and power, when a filter cache is combined with our region-based cachelets design.

As discussed in Section 4.3, the filter cache is one variation of vertical partitioned cache scheme. The filter cache is a small cache structure inserted in-between the processor and the first level cache in order to *filter out* some memory references that demonstrate high access locality. In these original study by Kin et al. [66], they extensively studied a 128 byte and a 256 byte filter cache with a variety of line sizes and associativities. Since the filter cache itself introduces one extra level into the existing memory hierarchy, the access latency is increased whenever a filter cache miss occurs; meanwhile, power consumption is also exacerbated due to the extra cycling of the filter cache.

As illustrated in Figure 4.11, the filter cache scheme can be analyzed by inserting an extra small cache or line buffer in-between the processor and our region-based cachelets. For each data memory access, the filter cache is accessed concurrently with the address de-multiplexing. If the desired data cache line is found in the filter cache, we can save the trip to the larger size region-based cachelets. Otherwise, we continue the access in the region-based cachelets. The filter cache is a unified cache, namely, each cache line can accommodate all kinds of data regardless of their semantic regions.

In the following experiments, we assume the filter cache has a unit cycle access latency. The filter

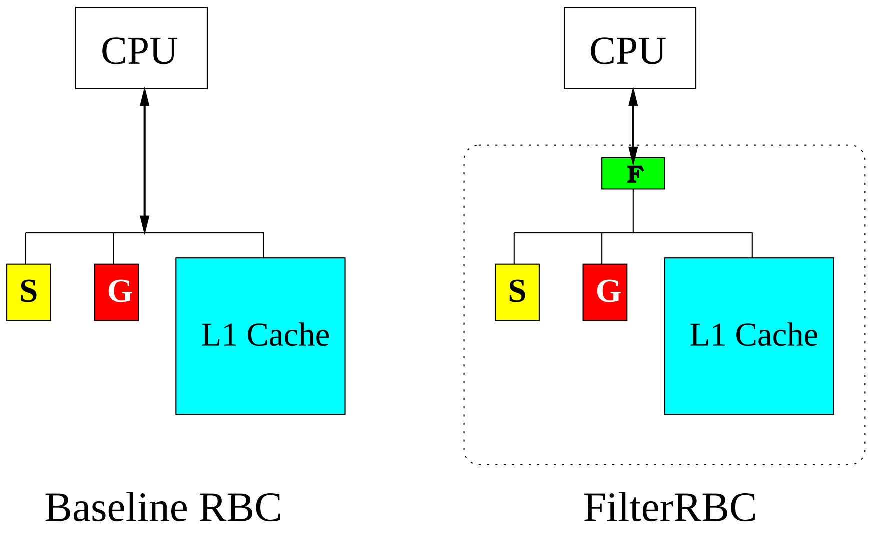


Figure 4.12: Region-based Cachelets versus a Combined Filter Cache with Region-based Cachelets

cache is a direct-mapped cache with a 32-byte cache line size. As studied in [66], fully-associative filter caches actually consume more power than the cases with filter caches. Our preliminary study using a fully-associative cache agree their observation. We use a region-based cachelets machine with the same configuration *S4G4-dmL1* as discussed in Section 4.3 for our baseline machine model. Figure 4.12 shows the block diagrams of our baseline region-based cachelets machine and the same machine with a vertically partitioned filter cache inserted.

Figure 4.13 shows the performance impact of a cache design with a combined filter cache and region-base cachelets. The bar chart shows the performance speedups of machine models with various filter cache sizes, from left to right — 128 bytes to 8KBytes, compared to an *S4G4-dmL1* baseline machine. Since smaller filter caches cannot satisfy the entire working set size, we would expect performance slowdown rather than speedup in many cases, in exchange for the power reduction. For all the cases, the performance reaches more than 90% of the baseline machine. For example, with a tiny 128-byte direct-mapped filter cache, we reach an average 94% of the baseline performance. With larger filter caches, the performance gap is reduced. With a large filter cache, the system could outperform the baseline in some cases because the stack or global static data working set can fit into the larger filter cache (8KB) well while their individual specialized cachelets (4KB stack + 4KB global static cachelets) cannot satisfy. The benchmark program *mesa.mipmap* is one of the examples, which actually shows 3% speedup over the baseline model with an 8KBytes unified filter cache.

Figure 4.14 illustrates the simulated power reduction by incorporating a filter cache. An interesting observation is that the power was reduced when we increased the filter cache size up to 4KB. When a 1KB or 2KB filter cache was used in the experiments, the average power consumption is minimized. Beyond this sweet spot, the power consumption trend is reversed. There are some cases such as *pegwitencode* that show worse power dissipation when a 4KB filter cache is used, mainly due to additional accesses to the region-based cachelets caused by filter cache misses. Combining the performance and power figures, we plot the energy-delay product bars in Figure 4.15. Given the *S4G4-dmL1* region-based cachelets design, a 2KB filter cache appears to be best design configuration despite trading off an additional 1% performance degradation.

Notice that there is no best design configuration for all. The best design is associated with the characteristics of target applications. For an embedded processor design that focuses on a specific application market segment, a true design win requires an in-depth analysis of the application. An

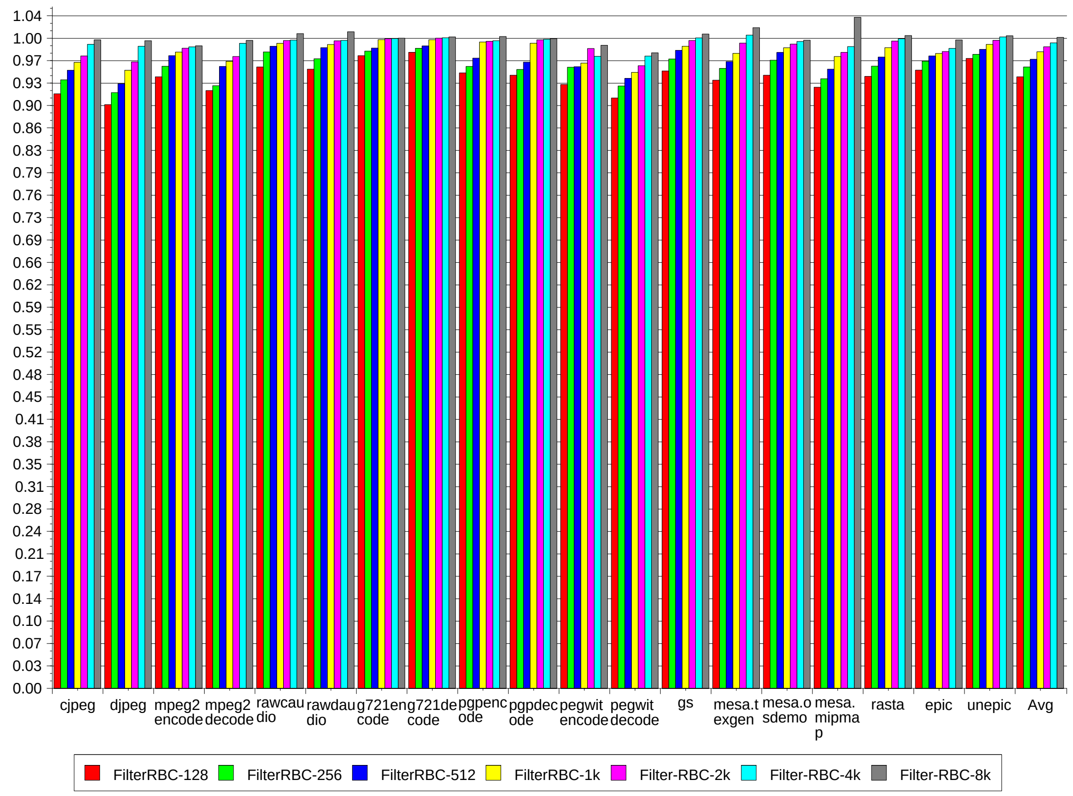


Figure 4.13: Performance of a Combined Filter Cache with Region-based Cachelets (Baseline: S4k-G4k-32K with a 512KB L2 = 1.0)

optimal design is then chosen to suit the needs of both performance and power reduction.

4.5.4 Design Options with a Filter Cache

4.5.4.1 Unified Filter Caches

More analyses were performed in order to evaluate various scenarios for region-based cachelets when a filter cache is present. In Section 4.5.3, we have shown that a filter cache can be inserted on top of a region-based cachelets design to further improve energy consumption. In this section, we evaluate two different data cache architectures, that implement a unified filter cache. Below the filter cache, one adopts a conventional monolithic five-way 40KB L1 data cache, the same one described in Section 4.5.2, while the other one uses an *S4G4-dmL1* region-based cachelets design. The motivation is to understand the benefits of region-based cachelets over a conventional design, when both contain a filter cache. Figure 4.16 shows the block diagrams of our comparison. The left hand side of the figure depicts a conventional cache with a filter cache and the right hand side depicts region-based cachelets with a filter cache.

Figure 4.17 illustrates the attainable speedups obtained by using region-based cachelets as opposed to a conventional monolithic data cache when a unified filter cache is applied to both. In this figure, *(FC+RBC)* represents a memory hierarchy that is composed of a filter cache on top of a region-based cachelets design. *FC* simply represents a conventional monolithic cache with a filter cache. The bars represent eight different direct-mapped filter caches sized from 128 bytes to 8KB. In average, the region-based cachelets improve performance insignificantly, at most 3.2%, when the size of the filter cache is smaller. There are some outliers, such as *mesa.texgen* or *rasta*, which actually

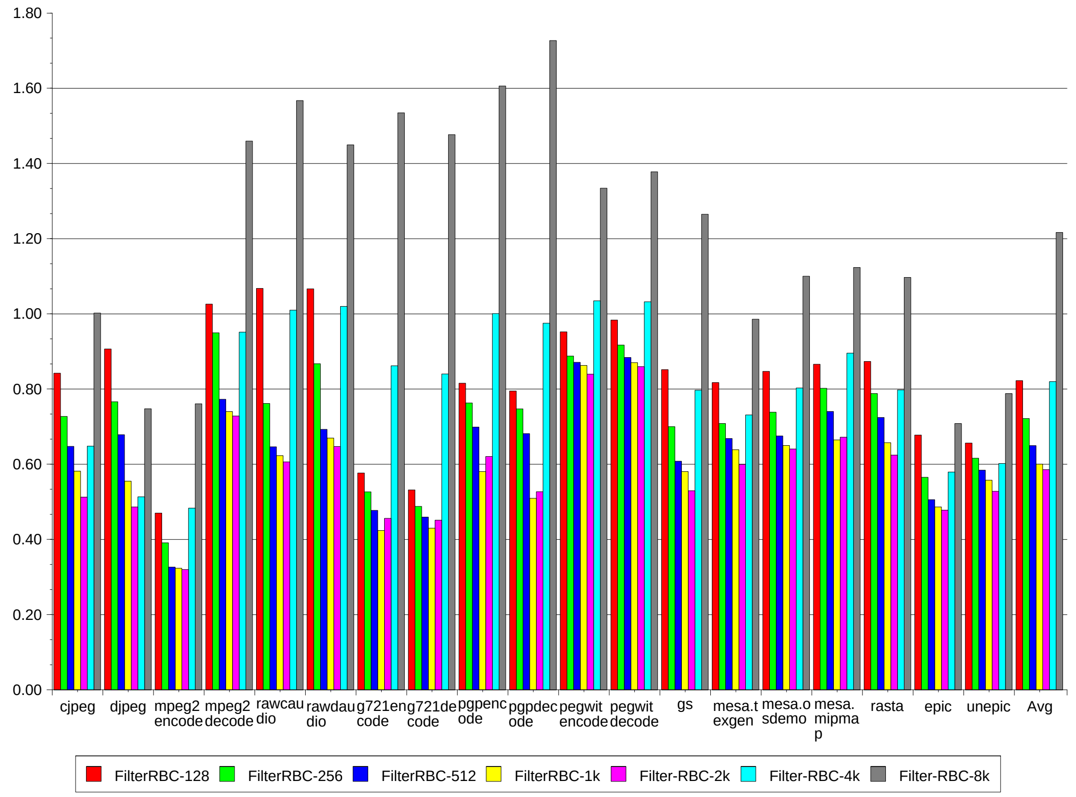


Figure 4.14: Power Reduction of a Combined Filter Cache with Region-based Cachelets (Baseline: S4k-G4k-32K with a 512KB L2 = 1.0)

show performance degradation in the range of 3% to 9%. The major cause, as discussed above, is due to the 4KB capacities of the stack and global static caches that are unable to accommodate the entire working set sizes of these applications which fit nicely into a 40KB monolithic data cache, nevertheless. When the filter cache is large, the region-based cachelets scheme has lower performance than a conventional cache design. As the 0^{th} level cache increases, the cache organizations in the 1^{st} level cache play a less significant role in the overall performance. As a result, the speedups continue to decrease as the filter cache is gradually enlarged. When the speedup drops below one, a majority of the frequently used data can be captured by the filter cache while data working set fits into a larger monolithic cache better than specialized yet smaller region-based cachelets.

Figure 4.18 compares relative power consumption of a cache architecture with a conventional cache design and region-based cachelets, both with a filter cache. A region-based cachelets design clearly demonstrates the usefulness of smaller filter caches in terms of additional power reduction through filtering a majority of cache references into smaller cachelet structures. This advantage, however, diminishes when the size of the filter cache is about 1KB in some of our experiments. By and large, different region-based cachelets configurations show roughly 20% to 50% power reduction in addition to what can be obtained from a filter cache implementation when the size of the filter cache is reasonably small. When the size of the filter cache is equal to or more than 4KB, the region-based cachelets are less effective than a conventional monolithic cache. This is caused by additional L2 cache power consumption in region-based cachelets when the stack and global cachelets cannot accommodate their region data. Consider both performance and power reduction, Figure 4.19 evaluates the energy-delay products of distinct design points. It shows that a filter cache with region-based cachelets is a better design for most cases when the filter cache is smaller than 4KB.

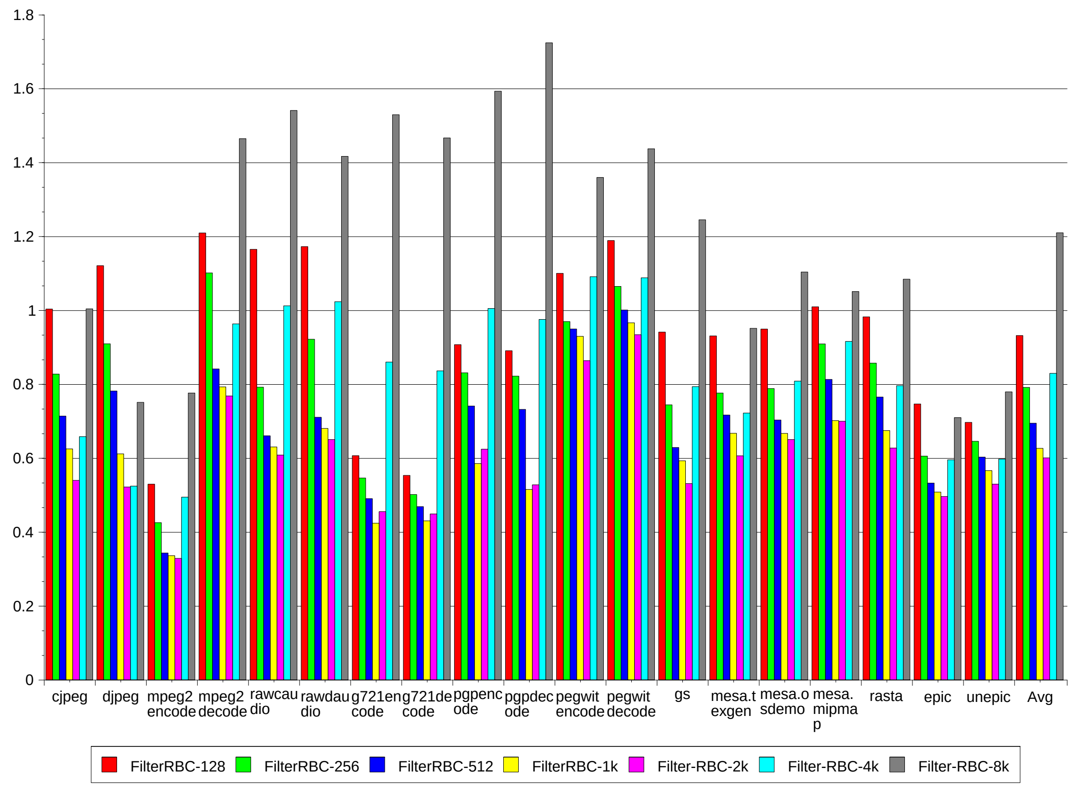


Figure 4.15: Energy-Delay Product of a Combined Filter Cache with Region-based Cachelets (Baseline: S4k-G4k-32K with a 512KB L2 = 1.0)

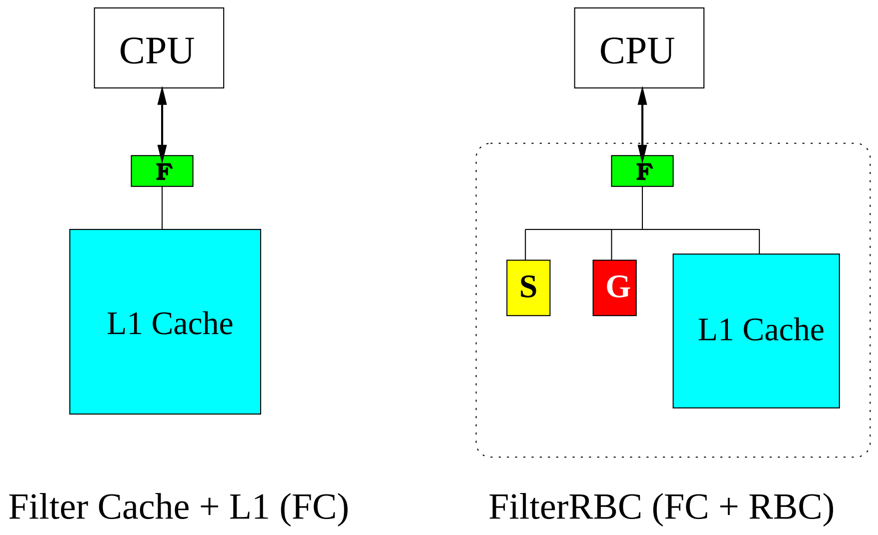


Figure 4.16: Filtered L1 Cache versus Filtered Region-based Cachelets

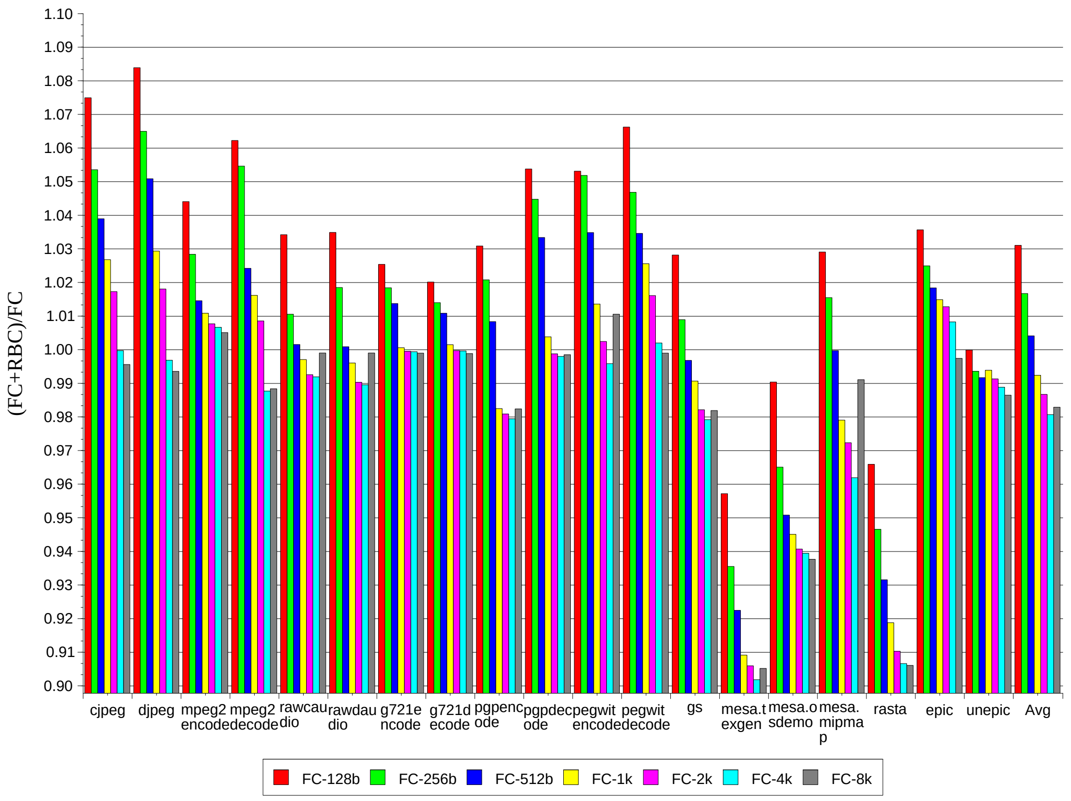


Figure 4.17: Performance Speedup of a Filter Cache with Region-based Cachelets versus a Filter Cache with a Conventional Cache

4.5.4.2 Dedicating Filter Caches to Semantic Regions

As shown in Figure 3.4 in Section 3.2.2, the majority of the cache space appears to be occupied by stack data. When the size of a unified filter cache approaches or is greater than the size of a region-based stack cachelet, the filter cache could be ineffective because what is missed in the filter cache is very likely to be missed in the next level stack cachelet. In such a cache organization, the filter cache is less useful for filtering stack data. If we leave the stack cachelet as is and dedicate the filter cache to the larger L1 cache (mainly for heap data), we might be able to achieve a more effective filtering effect to reduce power consumption. In this section, we investigate the design opportunities for dedicating the filter cache to a particular region.

Figure 4.20, Figure 4.21 and Figure 4.22 evaluate the performance speedup, power reduction and energy-delay product for such designs. The numbers were averaged for the entire Mediabench and normalized to the baseline case, the leftmost bar. The baseline, represented by *Filter L1* in these charts, is a filter cache with a five-way 40KB monolithic conventional cache. Along with the *RBC — Unified Filter*, they are the same cache configurations presented in Section 4.5.4.1. We exploit three different region filter caches, each of them is exclusively dedicated to one semantic region, including *RBC — Heap Filter* for heap data, *RBC — Stack Filter* for stack data, and *RBC — Global Filter* for global data. Seven filter cache sizes varied from 128 bytes to 8KB were used in these experiments. The latencies of data cache accesses that bypass the dedicated filter cache, i.e. accessing a different region, can be reduced. However, it consumes more power than a unified filter cache with region-based cachelets as shown in Figure 4.21.

When the size of the filter cache increases up to 4KB or 8KB, equal to or greater than the sizes of the stack and global cachelets (4KB in these cases), a dedicated filter cache to heap data appears to

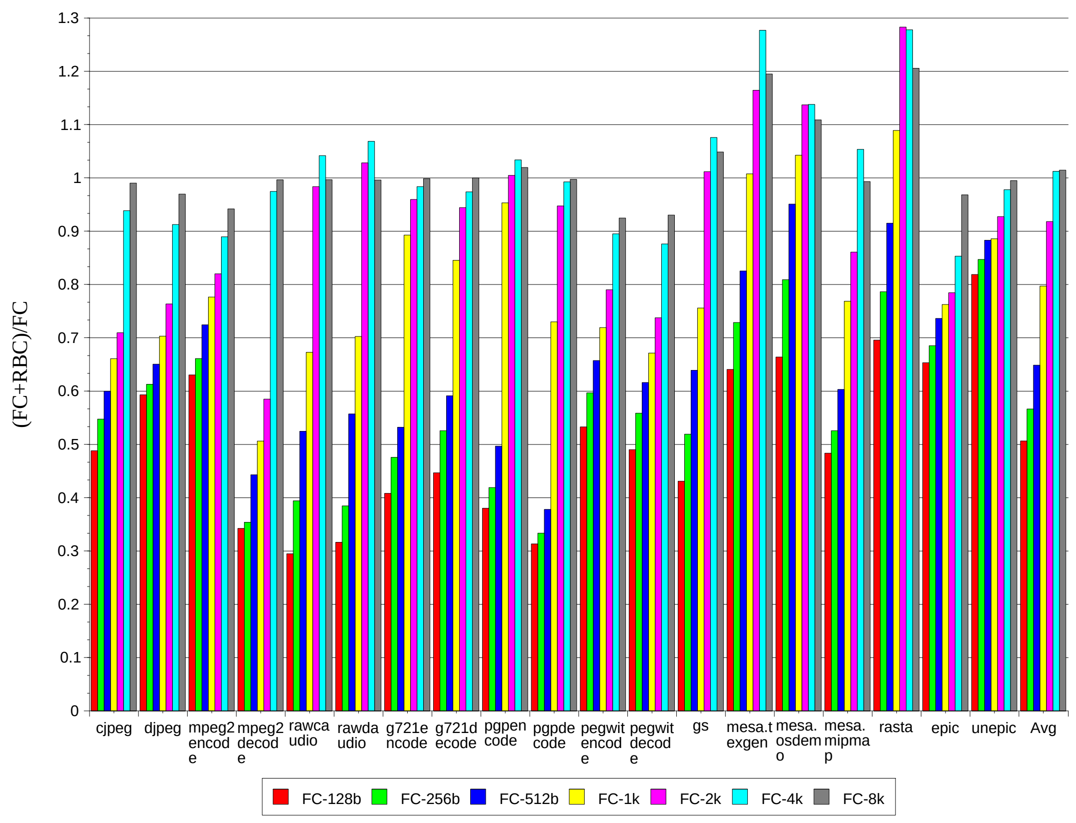


Figure 4.18: Power Comparison of a Filter Cache with and without Region-based Cachelets

be a better design as shown in Figure 4.22. In these cases, the stack and global cachelets themselves behave like filter caches, thus it seems redundant to have a unified filter cache in front of the stack and global cachelets. Instead, if we dedicate the filter cache to heap data only, we can achieve the maximum power reduction as shown in Figure 4.21.

For the energy-delay products, when the filter cache is less than 2KB, a unified filter cache *RBC — Unified Filter* appears to be the best design among the others except for the 128-byte filter cache. While the filter cache approaches 4KB or larger, the region-based cachelets with a dedicated heap filter cache (*RBC — Heap Filter*) apparently shows the best design.

4.6 Related Work

Low-power IC design techniques can be classified into several levels of design space from system level, architecture, logic, to transistor level. Frenkil in [45] presented an overview of research activities at each level. We briefly overview architecture level techniques for low-power cache design.

Power dissipation is generally proportional to the size of the SRAM array structure. Researchers and embedded processor architects have been studying designs employing smaller structures for the majority of the cache accesses to reduce power dissipation.

Early machines such as the HP3000 Series II [17] has an integrated stack cache as an extension to main memory. Since the machine does not have a data cache, the stack cache functions as a tiny direct-mapped cache with FIFO replacement policy for stack references. The CRISP [15][41] processor developed at Bell Labs adopted a complete memory-to-memory instruction set architecture and simple addressing modes to avoid the overheads of procedure calls. The design offloads the burden of register allocation on the top of the stack from the compiler to the hardware by incorpo-

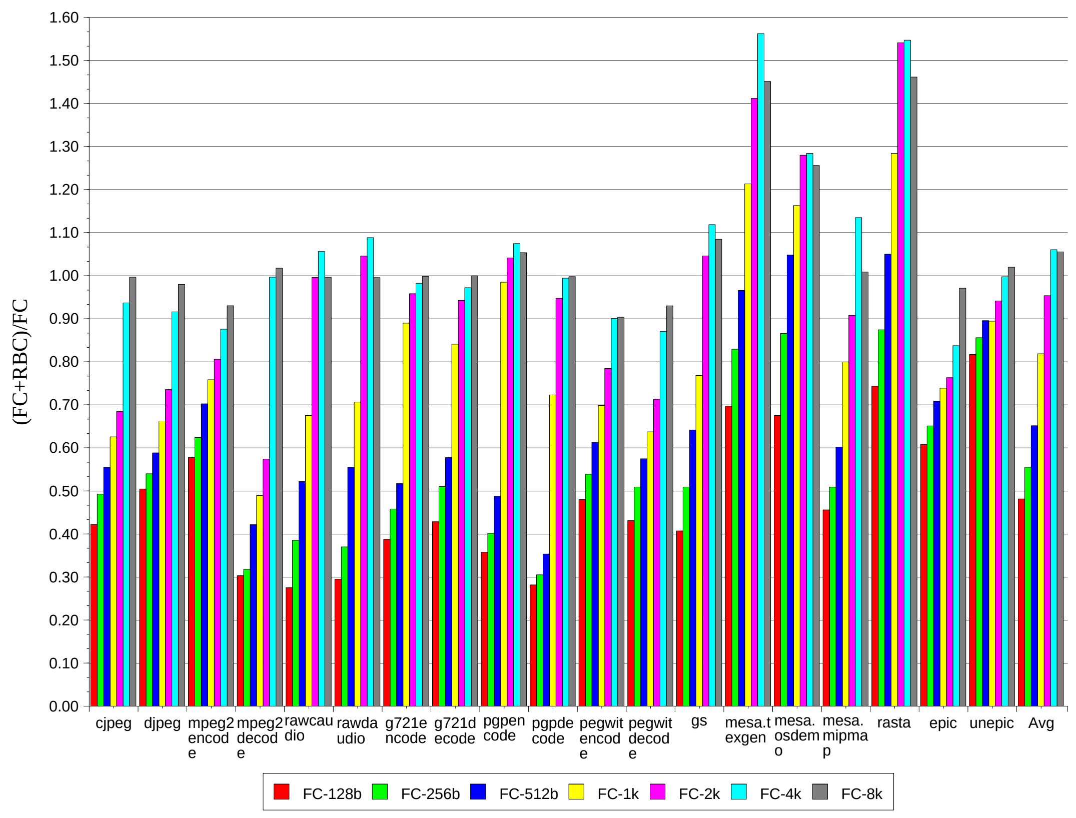


Figure 4.19: Energy-Delay Product Comparison of a Filter Cache with and without Region-based Cachelets

rating a 32-entry stack cache. The stack cache is the processor's only data cache. More recently, researchers [28][75] have proposed similar methods that incorporate a large stack-cache memory structure to alleviate the cost of multi-ported cache designs. Their goal was to improve performance of wide issue superscalar processors — not to reduce power dissipation.

Line buffers (or block buffering) and sub-banking [47][107] have been proposed to reduce power. To exploit spatial locality and reduce power, line buffers hold most recently accessed cache lines for potential hits by subsequent accesses. The cache is not exercised when a cache access hits in the line buffers. Kin et al. described a similar technique [66] by inserting a very small *filter cache* as the first-level (L0) cache to the CPU. The filter cache design approach sacrifices cache performance in exchange of power-saving as the filter cache has poorer data locality. In Kin's study, by employing direct-mapped 256-byte filter I-cache and filter D-cache, the power consumption is reduced by 58% while reducing performance by 21%.

Sub-banking is similar to column multiplexing [119] known to RAM designers for reducing the number of sense amps. Only the sub-banks that contain the requested data are accessed. As a result, power is reduced by eliminating unnecessary accesses. The first microprocessor in the StrongARM family [82], the SA-110, employs a sub-banking mechanism by enabling only 4 ways of its 32-way cache for each cache access. The processor also incorporates a cache sub-block castout mechanism to minimize data going out to memory, thereby reducing unnecessary power consumption.

Intel's StrongARM SA-1110 processor [58], based on the SA-110 core, implements a mini-cache in addition to the main data cache for storing streaming data which demonstrate little or no temporal locality. Data cacheability is controlled through control registers. This design is a multi-lateral cache design approach [49][61][93], but it actually increases dynamic power since both mini-cache

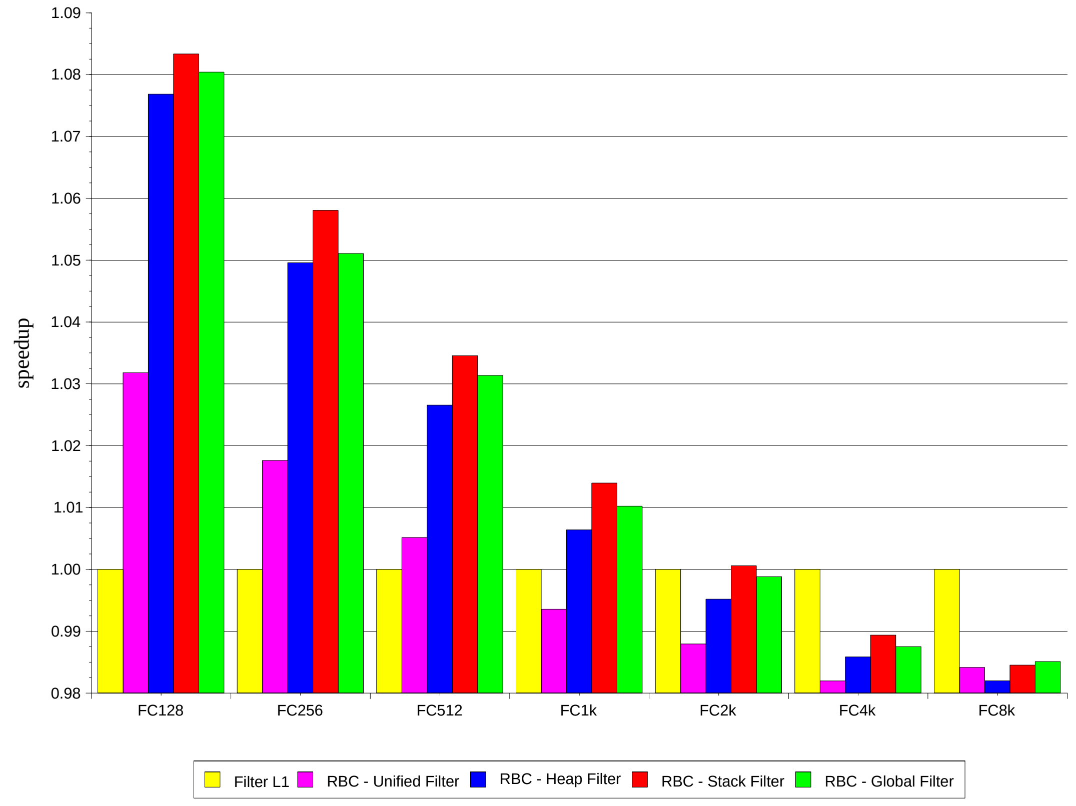


Figure 4.20: Performance Speedup of Dedicated Region Filter Caches

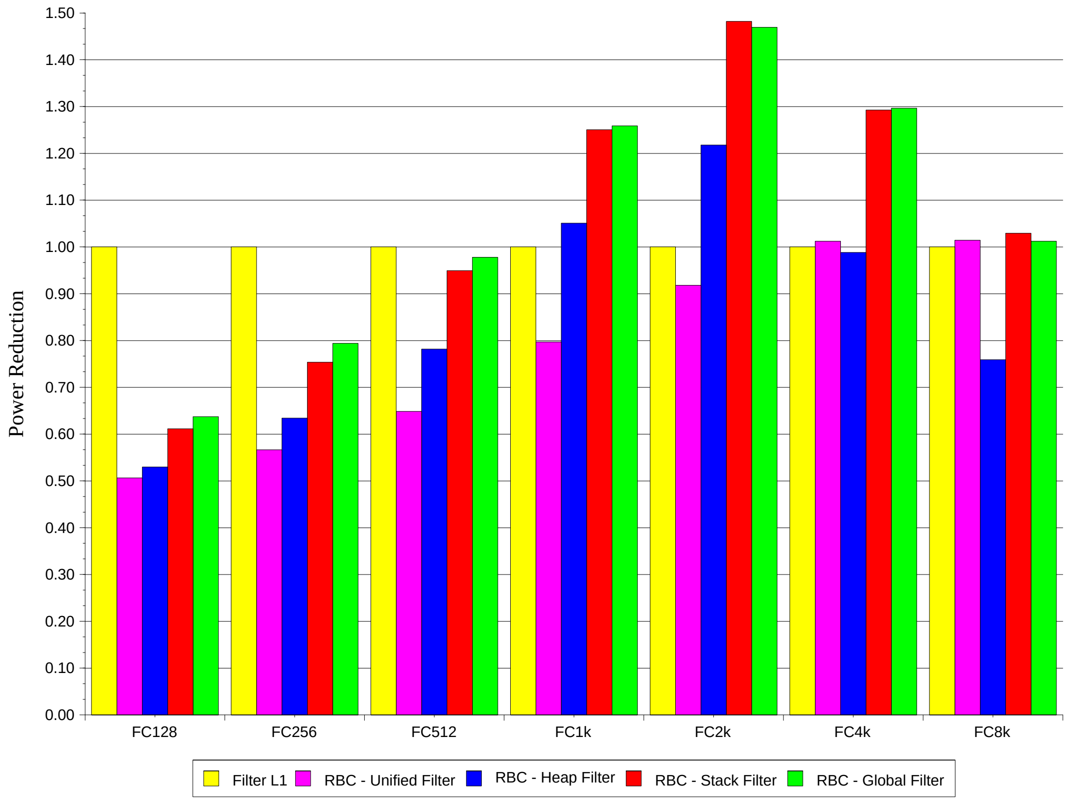


Figure 4.21: Power Comparison of Dedicated Region Filter Caches

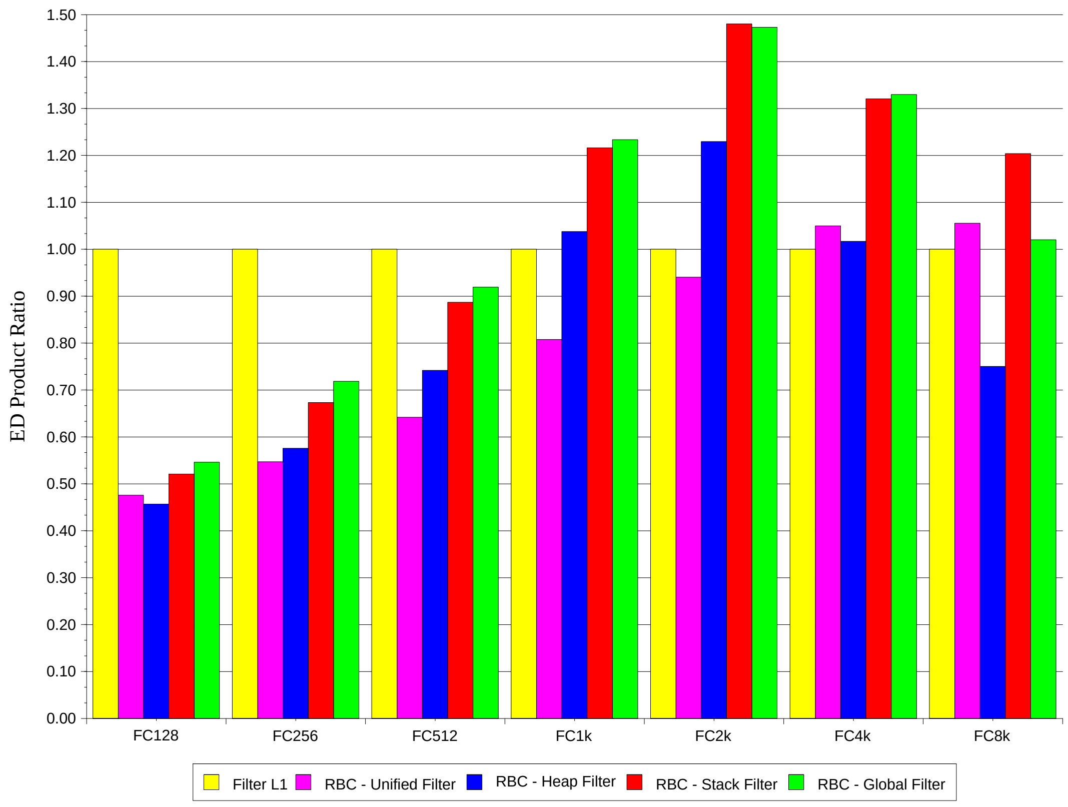


Figure 4.22: Energy-Delay Product Comparison of Dedicated Region Filter Caches

and main cache are probed in parallel. Bellas et al. [13] proposed a dynamic instruction caching scheme to determining what to be cached inside a mini L0-cache as an extension to the idea of instruction filter cache. By restricting the use of the mini-cache to only most frequently executed blocks, the total number of mini-cache accesses is reduced at the same time the mini-cache hits are increased.

Albonesi in [3] proposed a horizontally partitioned cache design that can disable a subset of cache set lines in a set-associative cache via ISA and microarchitectural support when a full cache is not critical to overall performance. This concept is called "Performance On-Demand" coined by George Cai at Intel Corp. The idea is to turn on the die area needed to satisfy the performance goal of a running application. Compiler and profiling tools can be used to determine when and how many set items can be disabled for power-savings.

More recently, Huang at el. [56] combined our proposal with a pseudo set-associative cache design to further reduce power consumption in data caches.

4.7 Chapter Summary

In this chapter, we have proposed a new *Region-based Cachelets* design that can effectively reduce the power dissipation of a data cache organization while retaining the execution performance of a conventional cache design. This is accomplished by partitioning data references based on semantically defined memory regions into distinct caches. Stack references and global static references, which exhibit a high degree of temporal and spatial locality, are routed to specialized (and small) cache structures. Since 70% of the references hit in the stack and global static data region, the overall data cache power consumption can be significantly reduced. Since 4KB stack and global static caches achieve a 99% hit rate, execution performance is not degraded. Additionally, by partitioning the cache into regions, conflicts are eliminated between regions making it more feasible to implement each cache with lower associativity. Building smaller segregated cachelets also provides more flexibility in increasing overall data storage than enlarging a single monolithic cache since cache designs do not need to double in size or increase associativity to grow. Our results show that a region-based cachelets design can reach an average power dissipation reduction of between 50% and 70% compared to more traditional designs.

The power can be further reduced if we apply existing techniques such as filter caches or sub-banking, or if smaller cachelet line sizes are used. A quantitative analysis of the filter cache effect was also presented in this chapter. Using our region-based cachelets scheme with an additional unified filter cache, we found that we can further reduce power consumption by as much as 40% while having very little impact on the overall performance. Furthermore, we analyzed how the region-based cachelets can enhance a memory system design with a filter cache, in terms of EDP efficiency. As we expected, the region-based cachelets design with a 128-byte filter cache can reach approximately 2 times design efficiency in EDP.

The region-based cachelets design also has the potential to reduce power in multi-ported caches for high performance microprocessors. For a multiple issue processor that issues multiple memory instructions in the same cycle, region-based cachelets design offers an alternative that allows building smaller power-economic region caches in lieu of a monolithic (power-hungry) multi-ported cache. Quantitative analysis of this applications is suggested in Chapter 7.

CHAPTER 5

HIGH-PERFORMANCE STACK VALUE FILE

In order to achieve ever higher performance in microprocessors, we continue to see an increase in complexity of the microarchitectural designs. To help manage this complexity and achieve designs that function in more restrictive time constraints, processor architects have relied more heavily on a technique of subdividing general structures into multiple, more specialized structures, which can be implemented more effectively. Examples include predicting indirect branches using a dedicated history buffer [25] and speculatively processing loads with good value locality [80]. As discussed in Section 3.1, memory accesses can be partitioned into address regions including the generated instruction code, literal pool, static data, dynamic stack and heap regions. This partitioning can then be exploited to reorganize the cache structure to improve performance. Separate instruction and data caches are found in almost all processors, and recently we have seen architectures proposed that include stack and non-stack caches [27][28], as well as temporal and non-temporal caches [49][62][93]. Each of these designs uses a conventional cache organization and achieves improved performance by enabling parallel accesses to two cache structures and/or reducing contention in cache line allocation.

In this chapter, we focus on a high-performance implementation by optimizing the performance of the stack memory references. We propose a hardware structure called a Stack Value File (SVF), which is used to exploit the unique characteristics of stack references. The SVF is a non-architected register file containing the data near the top of stack (TOS), which is normally held in memory or the data cache(s). All references to these locations are diverted to the SVF instead of the L1 data cache.

5.1 Stack Reference Characteristics

Stack references account for an appreciable portion of all memory references as discussed in Chapter 3, and their unique characteristics allow them to be handled more effectively than general memory references. In the subsequent discussion, we will demonstrate that stack references are a worthy target for optimization. Since the technique described in this chapter focuses on high-performance processor segment, therefore, the Compaq Alpha architecture and the SPECint2000 benchmark are used in our experiments throughout this chapter.

Memory accesses fall into several different categories, according to the region of memory they access and the access method used. The Compaq Alpha processor allocates a system-defined amount of space allocated to the stack, which grows down towards virtual address 0 from a mid-range address. The *TOS* pointer dynamically maintains the size of the stack, which forms a lower dynamic boundary on address references to the stack. The middle address range beyond the stack, which is allocated during compilation, includes the read-only data region (.rdata), the code region (.text), and the global data region (.data). Memory is dynamically allocated by the program from the heap, which

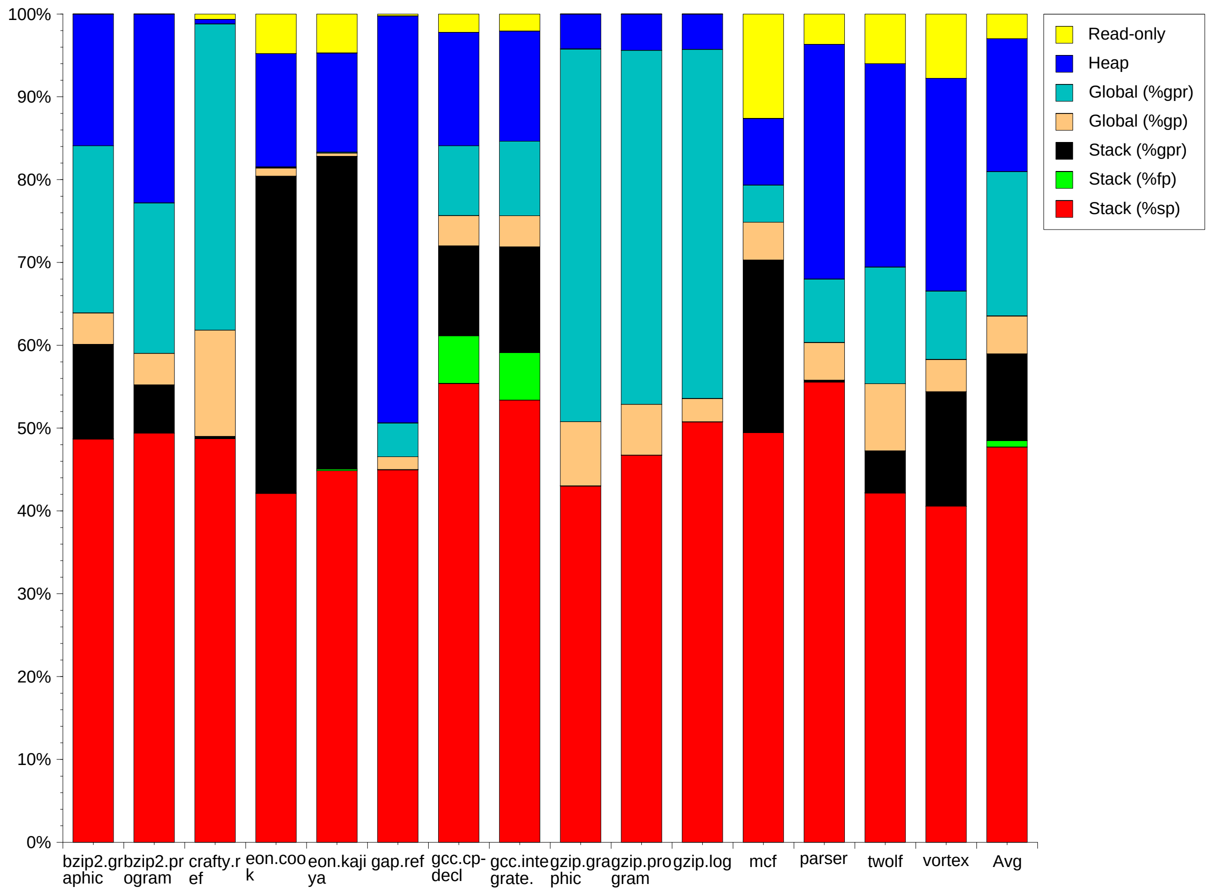


Figure 5.1: Run-time Memory Access Distribution by Access Methods for SPECint2000

grows upward from the top of the middle address range. This partitioning is depicted in Figure 3.1.

For the Compaq Alpha Processor, the stack may be accessed by several means: via the stack pointer (\$*sp*), the frame pointer (\$*fp*), or through general-purpose registers (\$*gpr*).

To understand the breakdown of memory references by regions and access method, in addition to the simple memory region distribution shown in Figure 3.2, a more detailed classification based on the distributions of access methods is shown in Figure 5.1. Each bar in Figure 5.1 is normalized to the total number of memory instructions. For these benchmarks, an average of 42% of the instructions executed access memory. Stack references account for an average of 56% of all memory accesses, while global data references account for only about 21% and a majority of the remaining accesses are to the heap. The \$sp-relative addressing mode is the dominant access method to the stack, accounting for 82% of all accesses to the stack, or 46% of total memory accesses.

Since stack references are so common and \$sp-relative addressing mode is the dominant access method to the stack, a closer examination of the characteristics of \$sp-relative accesses is in order.

\$sp-relative accesses can be easily identified during instruction decode. Thus they can be treated specially by diverting them to different pipelines or functional units. Removing stack references from the stream of references to the L1 cache reduces the demand for L1 cache bandwidth. It also potentially reduces the required size and associativity of a conventional L1 cache. Processing stack references in parallel with conventional L1 cache references increases effective memory bandwidth and allows for the exploitation of more instruction-level parallelism. This has been demonstrated for stack machine architectures in the early CRISP processor [14] and more recently for conventional architectures with a local variable decoupled pipeline [28].

The data above shows that the overwhelming majority of accesses are via $sp-relative addressing except for 252.eon which has a large number of accesses via $gpr. Thus while accesses to locations in a special stack structure using methods other than $sp-relative addressing (e.g. with pointer accesses) must be handled, they can incur the regular cache access latency without causing a significant performance penalty.

Since $sp-relative addressing is simple and fast, the additional pipeline stage often used for complex address calculations can be avoided, enabling a shorter access latency. This early address calculation is easily performed for those references using $sp-relative addressing by using the techniques described in [8] and [12].

Stack adjustments carry with them semantic assumptions regarding liveness that can be exploited to significantly reduce total memory bandwidth. A large fraction of the transactions between a stack structure or first-level cache and the second-level cache or main memory can be eliminated with this additional semantic information. The references that can be eliminated are fetches on write for cache lines that are being written to newly-allocated stack space (i.e., any data that might be fetched is uninitialized), and writebacks of dirty lines that are in the region of memory that has been deallocated from the stack. These references have no semantic impact, and can be eliminated without repercussions for well-behaved programs. A well-behaved program should not access locations below the TOS, where memory contents are undefined by legal programming language semantics.

Stack references can be renamed and treated like registers. With the right design, this reduces store-forwarding costs, as described in Section 5.3. Stack references are easier to rename for two reasons. First, the association of memory references with locations is simple and fast: the least significant bits of the address generated by adding the stack pointer and offset are used to directly index into the SVF. No associative lookups are required. Second, the choice of which references to rename is simple, namely the top *N* locations on the stack. No prediction of locality is required. In addition, the implemented register renaming logic for an out-of-order microarchitecture can be reused for stack reference renaming; thus minimum extra hardware cost is required.

There is clearly a strong potential for performance gain by exploiting these characteristics. The next section details some additional characteristics that suggest the stack value file (SVF) implementation.

5.2 Motivation for Stack Value File Design

A structure for storing stack data could take several forms, among them register windows, a stack cache or a stack value file. Register windows [114] explicitly make one microarchitectural implementation part of the instruction set architecture. This limits flexibility and adds a design constraint for future implementations. A decoupled stack cache [27][28] partitions memory streams by predicting reference regions, and retrieving data either from a stack or non-stack cache according to the prediction. It does not exploit the contiguous nature of stack locations, the relative stability of the top of stack, nor the semantics associated with stack adjustments.

The Stack Value File (SVF) holds the *N* locations closest to the top of stack (TOS) in a structure separate from the first-level cache. Space is allocated for these locations and no others; references to the stack outside this region are directed to a traditional cache or memory. As the TOS is adjusted, locations outside the range of the *N* locations farthermost from the TOS are written (or *dribbled* [101]) to the next level in the memory hierarchy if they are dirty. As will be demonstrated later, although the dribbling traffic is extremely small in most of the cases, it is the dominant element of the memory traffic between the SVF and the next level memory hierarchy (L1 cache) since an SVF of an appropriate size efficiently handles most references itself and also eliminates fetch on write cache line loads.

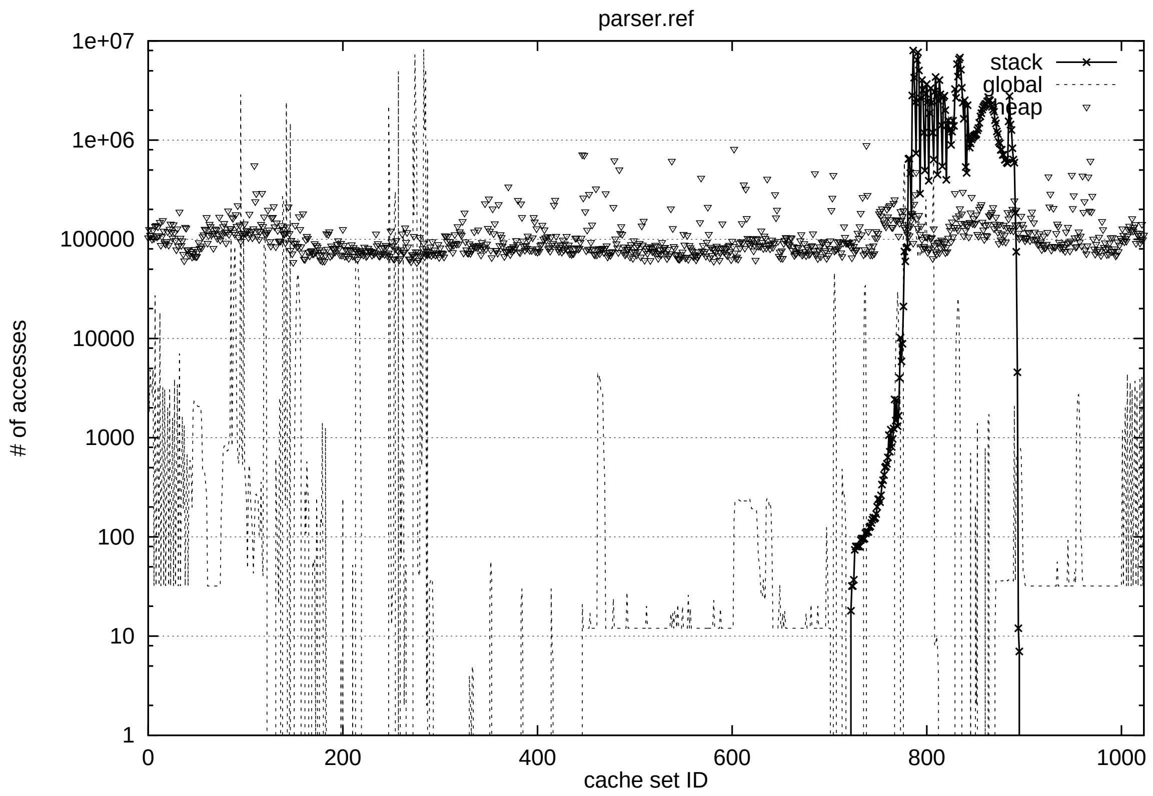


Figure 5.2: Cache Footprint of References by Regions (197.parser)

5.2.1 Contiguity

The first notable feature is that stack memory region is accessed contiguously which is exploited by the SVF scheme. This approach has three advantages relative to a cache. First, stack data can be conveniently stored in a relatively simple, fast structure with a limited die area and less power dissipation. Because it holds all of a contiguous range, it is directly addressable, and can be banked using low-order address bits with good capacity balance. Second, since locations are exclusively contiguous, no tags are required. Figure 5.2 shows the cache footprint of the 197.parser benchmark from the SPECint2000. Given a data cache with 1024 sets (x-axis), the number of references to a particular cache set was accumulated and plotted on the y-axis. The footprint of stack data measured in number of accesses to each cache set is denoted by a cross-marked solid thick line, the footprint of global static data with a dotted line, and the footprint of heap data with scattered triangles. Note that the y-axis was plotted on a log scale. As shown in the figure, the stack data were accessed heavily only between set 782 to set 888 and contain no footprint for almost all the other cache sets. In contrast, the heap data were evenly accessed across the entire 1024 cache sets. The global static data show the most irregular reference footprint. This observation reveals that the stack data show very high reference locality and imply the feature of contiguity in the reference sequences. The footprint plots for the entire SPECint2000 benchmark can be found in Appendix C. Most of them demonstrate similar behavior as the 197.parser benchmark. The only exception is 176.gcc, whose stack footprint is as spread out as the heap footprint. Even so, as we are going to show in our later analysis, the data traffic between SVF and L1 cache are tolerable because many of these data are either dead or not reused when the program returns from function invocations.

The third advantage of the SVF design over a conventional cache is that those locations which are known to be invalid or dead because of a stack adjustment are easily identified. This advantage

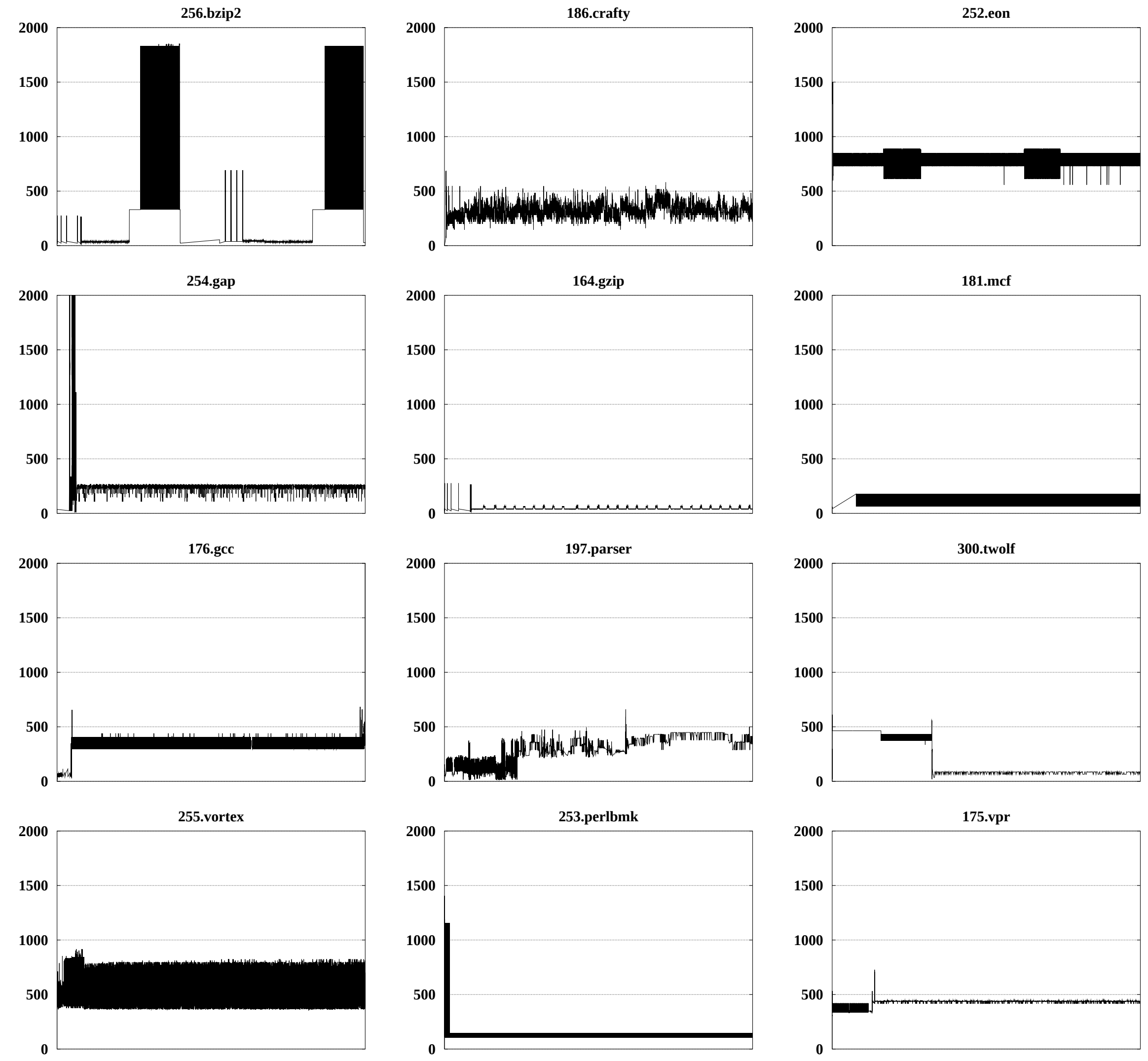


Figure 5.3: Stack Depth Variations

is more fully explained in Section 5.5.3.2.

5.2.2 Stack Depth Variations and Locality

The top of stack is adjusted at least twice for each procedure invocation (function call and return stack adjustments) and perhaps more often. Upon each adjustment, the set of locations stored in the stack or data cache structures can change. Changing this set of locations has three implications. Newly-allocated locations may need to be fetched from memory. Displaced locations may need to be written back. Finally, adjustments may need to be made as to how those locations are referenced. With the SVF design, none of these three implications has much performance impact, as explained here and shown quantitatively in Section 5.5.

As the stack grows, new space is allocated, which is initially invalid. If the size of the SVF is smaller than the size of the stack[1], the valid data closest to the stack base that is displaced from the SVF must be dribbled back to memory. As the stack shrinks, the data past the new TOS are disposed from the SVF and can no longer be accessed, according to program semantics, they are

[1]The size of the stack is the difference between the base of the stack region and the TOS.

dead and need not be written back to memory. Section 5.3.4 has further discussion on this point.

If the size of the SVF is smaller than the size of the stack, the locations that become part of the SVF region as the stack shrinks are initially undefined, and may need to be read from memory (dribbling in) if a load is performed before a store to any of these locations. There are two design alternatives: references to the SVF can stall while this dribbling is done en masse, or that region of the SVF can be marked invalid upon dribbling in, necessitating a fetch on write load only if/when the datum is read before written; again stores would not require a fetch on write load. The second approach is used in our design.

Thus dribbling for dirty writebacks and required reads only occurs if the stack size is larger than the SVF. A burst of traffic only occurs for writing back dirty data. Just as in dirty writeback, dribbling overhead can be hidden in the background as the processor continues to execute. Stanley and Wedig [106] investigated various schemes to hide the dribbling latency, e.g. the Barometer algorithm. Dribbling is only a performance issue if the TOS is changing quickly and the SVF is small relative to the working set.

Data were collected for SPECint2000 benchmarks to show the variations in stack depth over the lifetime of a program. The TOS address, relative to the stack base address, was logged each time the stack pointer is updated. The figures in Figure 5.3 show sampled graphs[2] that map stack depth variation over time. The x-axis is time, starting at the beginning of the program. The y-axis plots the stack depth, starting from zero. The basic data size in the SVF is 64 bits of data, and this is the unit of the y-axis. So 1000 units corresponds to 8KB.

There are two observations to be made from these data. First, an SVF of 1000 units is larger than the maximum stack size for most of the applications. Even though it seems that 256.bzip2 has some variations larger than 1000 units, we show later that these variations incur very little dirty data dribbling. This is because dirty stack data does not tend to be used across function calls. Thus no major amount of memory dribbling will be triggered. Second, the stack depth is quite stable after the initialization phase. This holds for almost all of the SPECint2000 benchmarks. For example in 186.crafty, the most representative active stack region is [200, 600]. Thus most of the stack references fall within a reasonably compact memory space, 400 units (a little under 4KB) in this case. Thus the valid values that may be dribbled in and out of a small SVF will still have good locality in the next level cache. Furthermore, if the working set is small enough and close to the TOS, the working set will stay in the SVF.

Stack references also tend to be reasonably close to the top of the stack. Figure 5.4 shows the cumulative distribution within a function of offsets of references into stack region with the x-axis plotted on a log scale. Across all of the SPECint2000 benchmarks, the average distance from TOS ranges from 2.5 (256.bzip2) to 380 bytes (176.gcc), and over 99% of all references (except for 176.gcc) are within 8KB of the TOS. Thus spatial locality with respect to the TOS is excellent. No references are beyond the top of the stack for these benchmarks. The graphs illustrate that most stack references are in one contiguous space (between 0 and 300 bytes offset from TOS), indicating that there is no need for a mechanism flexible enough to maintain a non-contiguous working set.

The conclusion to be drawn from these data is that an SVF that is 8KB or less will still capture the locality well enough to eliminate most dribbling. The amount of dribbling may be large if the variation in stack depth is large, but when this happens, a decoupled stack cache will also perform poorly if the number of distinct locations accessed per stack adjustment is also large.

The data presented in this subsection and later, in Section 5.5, suggests that potential losses due to managing only spatial locality with respect to the TOS, relative to managing only temporal locality with a temporal cache, are minimal if the SVF is adequately sized.

In summary, the stack reference characteristics that the SVF exploits include the following:

[2]To ensure representativeness, the sampling is based on the changes of the stack pointer rather than on time period.

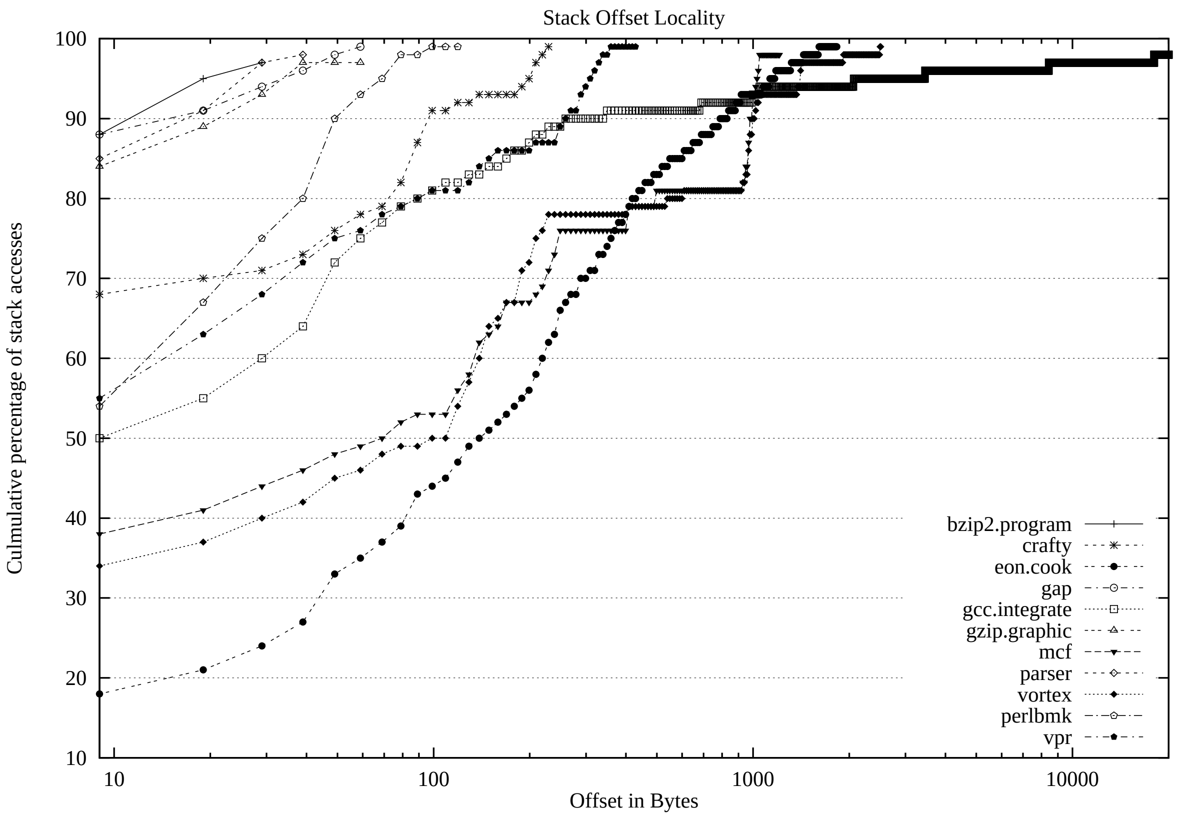


Figure 5.4: Offset Locality within Functions

- Stack references have extremely good temporal and spatial locality. Furthermore, the active portion of the cache is almost[3] always in a single continuous region at the top of the stack. This contiguity means that partitioning a cache into multiple blocks, each with its own tag, is unnecessary; the SVF can be organized as a simple register file structure that is dynamically mapped to the contiguous region at the top of the stack. The access hit rate for this contiguous region will approximate that of a stack cache design of the same size.

- The first reference to each word in a new region of the stack (i.e., a new activation record) is *almost* always a store. This means that any demand-driven line allocation in a conventional cache is likely to bring in many words that will be overwritten on their first reference. This wastes bus bandwidth by bringing in **dead** data values. Our SVF organization eliminates this useless traffic by organizing the data words into a single register file with a valid bit associated with each word. Store misses need not reference memory and the rare instance of a load miss simply brings a single word from higher levels of the memory system (i.e. the L1 cache). This significantly reduces the read traffic incurred by stack reference misses in conventional caches.

- Data from deallocated stack frames do not have to be written back to memory when deallocated. Identifying invalid dirty data in the SVF reduces the writeback traffic incurred by conventional caches.

- Most stack references use an $sp-relative addressing mode. The simple addition/subtraction of a stack displacement can be done early in the pipeline, thereby reducing access latency for stack references. Furthermore, by directly translating the least significant bits of the full

[3]Prefetch instructions can read stack memory before a store occurs.

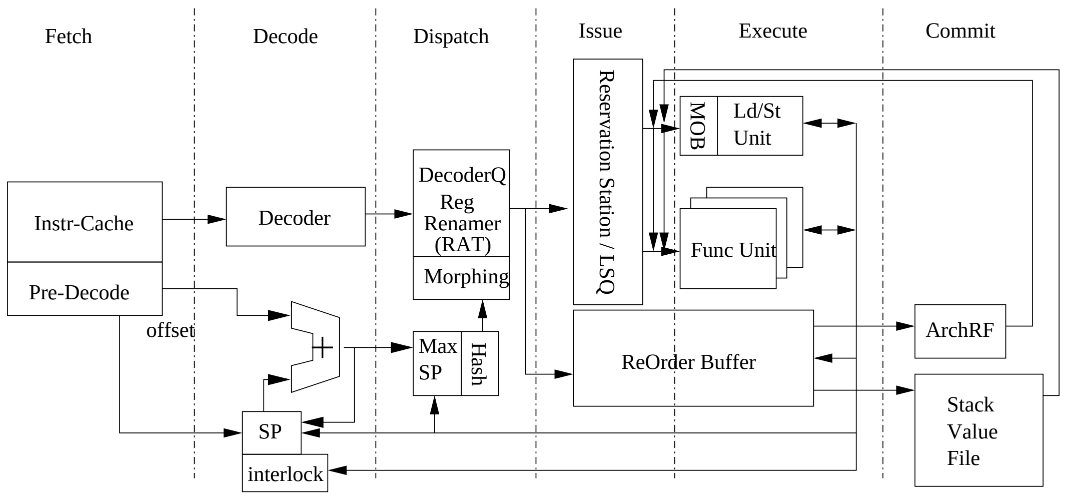


Figure 5.5: Microarchitecture Extension with a Stack Value File Implementation

address into an SVF index, full address calculation and data tag comparison is eliminated. This reduces the circuit design complexity.

In addition to these points, if references to a stack structure become nearly as cheap as registers, then there is less need to have more architectural registers. Instead of imposing the design constraint of a larger number of architectural registers on all future designs, the compiler can allocate the frequently-referenced variables that do not fit in architectural registers near the top of the stack, and implementations may vary with respect to the number of stack locations they treat more efficiently. However, we do not propose a new stack machine architecture here; we in fact propose a new microarchitectural component, specialized for the unique characteristics of stack references on current processor architectures, e.g. the Compaq Alpha processor in this study.

5.3 Stack Value File Design

The stack value file is specifically tailored to optimize stack references. This section provides an overview of its main features. The SVF is a register file large enough to hold those stack locations near the TOS. Arrays with thousands of registers have become reasonable to build today. Because the SVF stores locations that would otherwise be in the first-level cache, and allows that cache to be used more effectively, it is reasonable to borrow some of the first-level cache's area. The SVF can be more area efficient because it needs almost no tag space and can be direct-mapped instead of associative. It may also alleviate the need for dual-porting the first-level cache.

The SVF is architecturally invisible, leaving the designer with the freedom to choose an appropriate level of support for stack references without the constraints of a large architected register file. References to cacheable locations allocated in the address range covered by the SVF are diverted from the first-level cache as described in Section 5.3.1, thus reducing its bandwidth, capacity and associativity demands. \$sp-relative references are recognized early enough to avoid the added latency that general address calculations require. SVF references are renamed like general-purpose registers through the register alias table, further reducing delays and effectively implementing data forwarding. Figure 5.5 shows our re-architected out-of-order pipeline. Other references to locations in the SVF are detected using a bound check. Such references are diverted from the first level cache at a modest performance penalty, as described in Section 5.3.2.

The SVF is a circular buffer, with memory locations mapped to SVF registers according to the lowest-order address bits. Changes to the stack pointer are detected, and lead to data movement to and from the first-level cache as necessary. Status bits are associated with entries in the SVF to identify dirty data that needs to be pushed onto the stack. Valid and dirty bits minimize or delay traffic when a TOS adjustment and an SVF overflow or underflow makes it necessary to dribble out to or in from memory. The following subsections describe these new microarchitectural features in greater depth.

5.3.1 Morphing Stack-Pointer Based References

An extended pre-decode circuit in fetch stage is added to identify stack pointer based memory references and to determine their immediate offset values. A special adder in the decode stage enables fast address calculation using this predecoded information. Prior studies [9][8] have already demonstrated similar viable techniques.

In our design, a pipeline interlock incorporated in the decode stage can stall instruction decoding if the stack pointer update requires registers other than itself, or complex operations, e.g. `add sp, sp, r5`. These are not common, however. Since most of the \$sp updates are simply adjustments with an immediate constant, we can perform these computations in the decode stage early by keeping a speculative \$sp register copy in the decode stage. All the following \$sp-based references fetch \$sp content from this speculative copy to index their SVF register ID. If the branch is correctly speculated, execution continues. However, if the branch is mispredicted, then the speculative \$sp copy will be recovered with the value from the architectural \$sp before the pipeline restarts at the correct branch path. All other \$sp updates require a reference to other general purpose registers (except for zero register \$r31 in Alpha). In those cases, the interlock stalls decoding to prevent following instructions from reading a stale TOS address.

Once the memory address with $\pm$IMM($\$sp$) addressing mode (immediate offset from stack pointer) is computed in the decode stage, the address is checked against the range of stack memory currently held in the SVF. If a hit is detected, the instruction is morphed into a register-move operation and dispatched to the reservation station. The low-order bits of the address are used as the register ID to index into the SVF. These architecturally-transparent register IDs can be considered as an extension of general-purpose register IDs. The hardware register renamer can rename each active SVF register into a corresponding entry in the physical registers of the reorder buffer. After the dispatch stage, all the morphed \$sp load/store instructions will have been mapped into register space. Thus dependencies on these SVF registers are treated just like any regular register dependency.

5.3.2 Stack Memory Reference Disambiguation

Since only memory references indexed by stack pointer are deposited into the SVF, stack data references through other means such as the frame pointer or general purpose registers must be disambiguated and redirected into the SVF for data consistency.

For each stack-pointer based reference morphed into the register move form, two micro-operations (uops) are generated after instruction decoding. One uop is the converted register move, while the other one carrying the early resolved stack address is enqueued into the Load/Store Queue (LSQ). The uop in the LSQ is used for disambiguation purposes before the morphed references are committed to the SVF. If any later load instruction collides with these uops in the LSQ, regular store forwarding will be performed.

All memory instructions that reference the stack memory region through registers other than the stack pointer have their addresses checked against the current stack range in the SVF. The load/store operation is then re-routed to SVF if a match is detected.

There is one particular circumstance in which a simple re-routing operation cannot correctly maintain data dependency. This happens because of the relative timing of when references are

```
i1   lda a1, 16(sp)          i1   lda a1, 16(sp)
i2   bsr ra, func1           i2   bsr ra, func1
i3   lda sp, -16(sp)         i3   lda sp, -16(sp)
i4   stq a1, 8(sp)           i4   stq a1, 8(sp)
i5   ldq t0, 8(sp)           i5   ldq t0, 8(sp)
i6   stq v0, 0(t0)           i6   stq v0, 0(t0)
     ..                           ..
i7   lda sp, 16(sp)          i7   lda sp, 16(sp)
i8   ret                     i8   ret
     ..                           ..
                             i9   lda t0, 16(sp)
i9   ldq v0, 16(sp)          i10  ldq v0, 0(t0)
```

Figure 5.6: Load Squashing Scenario and its Optimization

determined to access the SVF. When a store through a general-purpose register is followed by a colliding load through stack pointer, the load can retrieve a stale value from the SVF, as follows. The store is executed late: it does not access the SVF until the execution stage because it uses a different addressing mode. The load, though it follows the store in program order, can execute before the store, since $sp-relative loads execute early, in the decode stage.

This condition is detected in the LSQ when the store executes. A pipeline squash, similar to the recovery from a memory ordering violation, is invoked to avoid a chain of incorrect data dependent instructions. This problem can be avoided by introducing a redundant dependence that forces the load to be executed late. The $sp-relative load is broken into two instructions. The first, designed to go through the execute stage, performs the address calculation ($sp plus offset). The second, dependent on the first, performs the load itself, at a time which is guaranteed to be after the store.

Nevertheless, this squash can be eliminated by inserting a redundant dependence with one additional instruction before the dependent load. The code on the left-hand side in Figure 5.6 shows the scenario that leads to a load squashing. Instruction `i1` computes the address and assigns it to register `a1`. Then `a1` is stored onto the new stack frame and `v0` is later stored to that address as shown in `i6`. After the function returns and the old stack frame is restored, the load (`i9`) reloads the value through $sp into `v0`. The aliased memory dependency between `i6` and `i9` causes `i9` to be squashed. An optimized version code to prevent the squashing from happening is to insert one additional instruction before the load as shown in the right-hand side of the figure. In this new code, `i9` is inserted as an address calculation instruction. The memory dependency between `i10` and `i6` is now resolved in the LSQ if `i6` is still in-flight without taking on the squashing penalty.

5.3.3 SVF Status Bits

Each SVF register contains two status bits. The dirty bit identifies the subset of all locations between the new and old TOS that need to be dribbled out upon a TOS adjustment to maintain data coherence. The dirty bit is set when its corresponding SVF register is written, and cleared when the data is written back. The valid bit indicates whether locations exposed by a TOS adjustment need to be read. The valid bit is set when data is written in the SVF, either upon a write from the processor core or a fill from memory. It is cleared for locations between the new and old TOS when the TOS is adjusted (both shrinking and growing). If an SVF load accesses an invalid register, a dribble-in is triggered.

These status bits improve performance for the SVF design in several ways. First, the dirty bits avoid writing back clean data. Second, the valid bits avoid a burst of unnecessary reads when the stack shrinks. Locations are read only when needed, like a cache. Third, valid and dirty bits can be used to aid in snooping in a multi-agent (e.g. multiprocessor) system.

The granularity of these status bits is most naturally the smallest data type that is frequently used. For the Alpha architecture, this is 64 bits. If the granularity is larger than this, there may be more traffic to memory, as was just alluded to. Larger granularity can also increase the degree of false sharing, but this is not a significant issue for stack locations.

At first glance a caching scheme may seem more advantageous than a contiguous scheme on context switches, but this is not necessary to be an issue. The use of a stack cache does not necessarily eliminate writebacks of dirty data on a context switch. Any new process is likely to displace much of the data in their stack cache. Valid bits make the SVF perform similar to a stack cache on a context switch in that only dirty words, not entire dirty lines are written back.

5.3.4 Accesses Outside Defined Scope

Eliminating writeback transactions for locations that are past the TOS and presumed dead can boost performance. While accessing locations past the TOS is illegal by semantic convention, assuming that this can never happen is unsafe. So such optimizations can be under the control of a mode bit that is set upon the loading of a well-behaved program.

The convention of not writing beyond the TOS holds for well-behaved high-level language programs, since a later subroutine call could overwrite these data. For those operating systems that do not use a separate system stack, an interrupt service routine could also overwrite these data. However, it is possible to write a source-level or an assembly program which violates this convention. One way to end up with a reference past the TOS is by taking the address of a location on the stack and using it outside the scope of the procedure that allocated space for that location. This should normally be considered a bug; it is dangerous at best. The SVF can access these locations, nonetheless, the contents of these locations are undefined.

5.3.5 Management of Performance Liabilities

The SVF performs well in most cases, as is supported by the data above. There are three cases where the SVF may not perform well, and we propose means for dealing with each of these.

The first case is context switches. This will be addressed in Section 5.5.3.3.

A second case is excessive changes in the TOS. If dribbling becomes excessive, i.e. where the ratio of TOS movement in one direction to the number of references is too high, the SVF can be locked. This can be monitored with fairly simple hardware. Note that with the use of valid and dirty bits the read and writeback traffic is a function of the number of references, and not only the number and distance of adjustments.

A third case is operating systems calls and interrupts serviced on a system stack. For this case, there is a high probability that the SVF entries that are written back will be reloaded. Transitions among different privilege levels are generally detectable in hardware, e.g. by a change between the user and system stack pointers in the Motorola 68k family, or by accesses to the task segment structures in the IA32 family. A change to a higher privilege level can be noted and can cause the SVF to be locked. When reverting to the user privilege level, the SVF can be unlocked.

In general, excessive dribbling is detectable and the SVF can be temporarily disabled, if necessary, without disrupting the rest of the pipeline. Performance then reverts to being no worse than it would be in a conventional microarchitecture.

5.4 Experimental Configuration

The SPECint2000 programs are used in this study. The binaries were compiled using the Compaq Alpha compiler with appropriate optimizations enabled. All the simulation results presented below

Components	4-wide	8-wide	16-wide
Decode width	4	8	16
Issue width	4	8	16
Commit width	4	8	16
IFQ size	16	32	64
RUU size	64	128	256
LSQ size	32	64	128
IL1 cache	8-way 256KB	8-way 256KB	8-way 256KB
DL1 cache	4-way 64KB	4-way 64KB	4-way 64KB
IL1 hit	1 cpu clk	1 cpu clk	1 cpu clk
DL1 hit	3 cpu clks	3 cpu clks	3 cpu clks
Unified L2	4-way 512KB	4-way 512KB	4-way 512KB
L2 hit	16 cpu clks	16 cpu clks	16 cpu clks
Mem latency	60 cpu clks	60 cpu clks	60 cpu clks
CPU-Mem clk ratio	6:1	6:1	6:1
Store forwarding	3 clks	3 clks	3 clks
Int/FP ALU	16	16	16
Int/FP Mult	4	4	4

Table 5.1: Processor Models.

ran up to 600 million instructions. Section 2.2.1 contains detailed description for each benchmark application.

The simulators used in this research were derived from the *SimpleScalar* tool suite version 3.0 [19]. Refer to Section 2.1 for basic information of the SimpleScalar tool set. We re-architected the pipeline structure to incorporate our stack value file design.

The machine models used in our experiments are summarized in Table 5.1. It is worth noting that the store forwarding latency used in all of our experiments is 3 cycles. As a result, the L1 cache hit latency is also 3 cycles[4]. However, the throughput of the cache hit can be a single cycle since L1 cache accesses are fully pipelined.

In order to demonstrate the performance potential of our scheme and to reduce the performance interference from the front-end, we use a fairly large and fast first-level instruction cache as well as a perfect branch predictor, unless elaborated.

5.5 Performance Evaluation and Design Trade-Offs

In this section, we provide data to show how the characteristics of stack references can be effectively exploited with a stack value file to alleviate first-level cache bandwidth, reduce latency for stack memory references, reduce the memory traffic for the memory subsystem, and eventually improve execution performance.

5.5.1 Improving Cache Bandwidth, Latency and ILP

The primary benefits of treating stack references separately from all other memory references are the opportunities the SVF provides for:

- exploiting more instruction-level parallelism with the existing physical registers in the RUU and additional SVF ports for stack references

[4]The store forwarding latency matches our measurement of the actual latency on the Intel Pentium III processor, including cache latency and store forwarding delay in the pipeline.

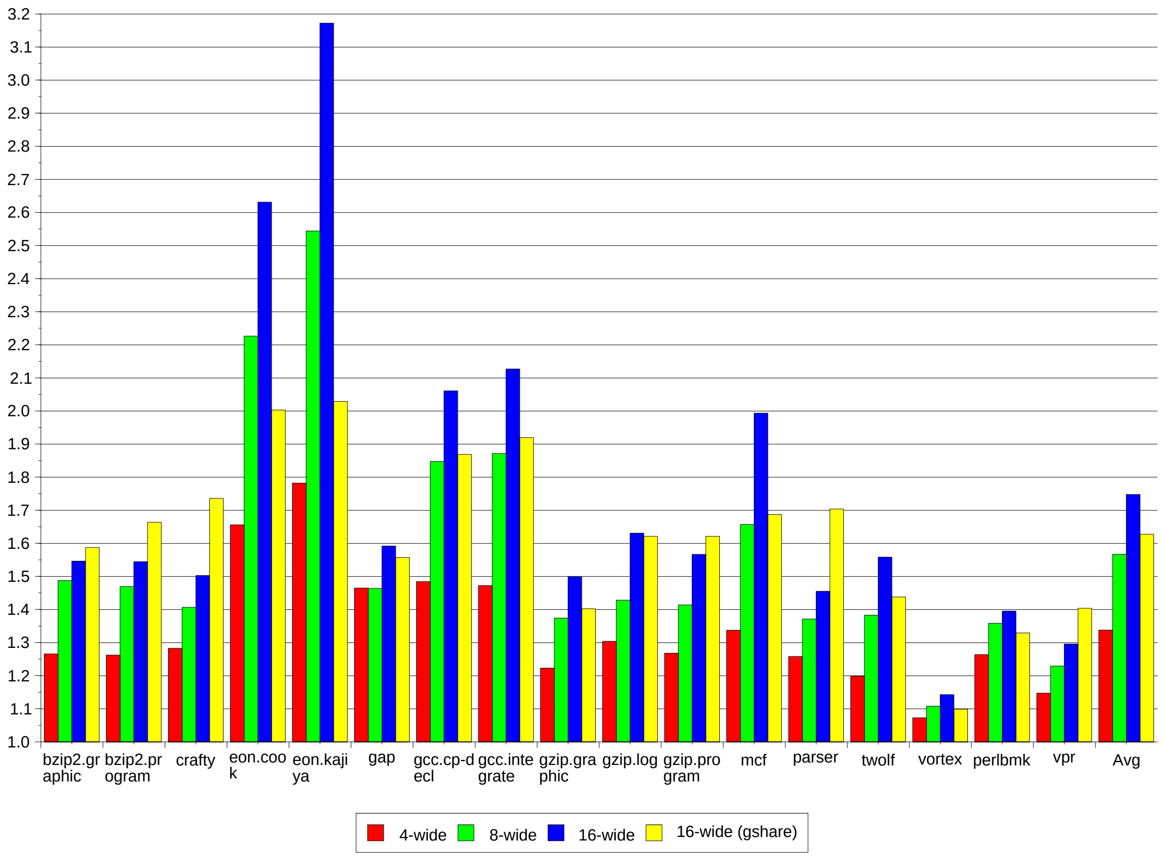


Figure 5.7: Speedup Potentials of Morphing All Stack Accesses to Register Moves (single-ported cache)

- disambiguating stack references through existing register alias table
- eliminating the stack references on the data cache ports

The latency of the stack memory references can be reduced if the SVF entries can be accessed like registers. The gains from these improvements are quantified in Figure 5.7 and Figure 5.8. These figures demonstrate the potential performance gains from implementing an SVF with infinite SVF ports for various generations of processors, assuming all the stack references can be morphed into register-to-SVF moves. For example in Figure 5.8, the first three bars show average speedups of 1.11, 1.19, and 1.31 for 4-wide, 8-wide and 16-wide machines respectively, with a dual-ported first level data cache and a perfect branch predictor.

The 4-, 8- and 16-wide speedups are all relative to a baseline with a perfect branch predictor. The last column shows 16-wide speedups with a gshare branch predictor [81], relative to a baseline with gshare predictor. The average speedup for a configuration with a dual-ported cache is 1.25. Some benchmark cases show a greater speedup with gshare. In these cases, SVF's latency-shortening benefits allow branches to be resolved early, reducing the branch misprediction penalty. However, the more realistic branch prediction of gshare reduces the effective basic block size, reducing the potential stack parallelism and leading to a smaller average gain than for the perfect prediction case.

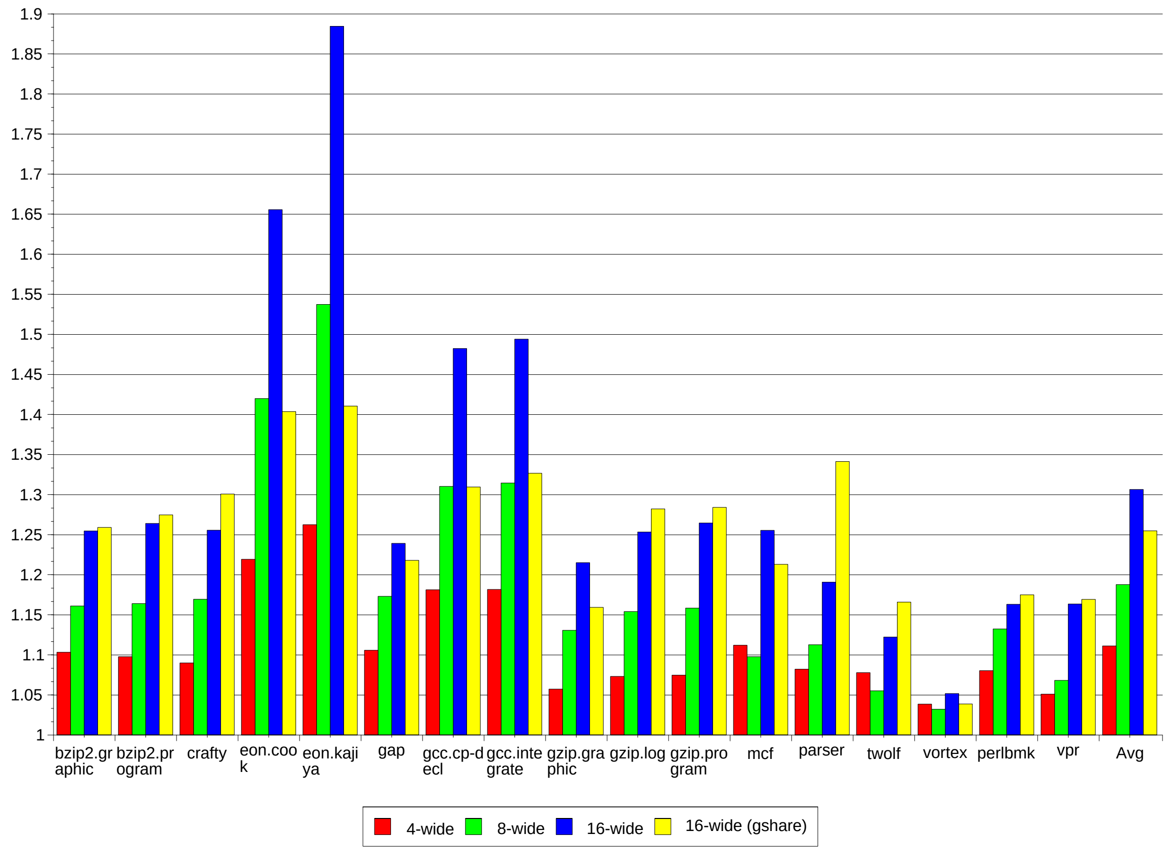


Figure 5.8: Speedup Potentials of Morphing All Stack Accesses to Register Moves (dual-ported cache)

5.5.2 Hierarchical Performance Analysis

To understand the performance gain in a quantitative manner, Figure 5.9 shows hierarchical performance improvements under different constraints for a 16-wide machine. Starting from the baseline machine model described in Table 5.1, we relax the machine constraints for each bar in the figure.

First, the first level data cache size is doubled (from 64KB to 128KB) without increasing the access latency. As shown in Figure 5.9, the speedups from enlarging L1 size for all the SPECint2000 benchmarks are negligible. This leads to the following conjectures. First, improving overall performance for SPECint2000 benchmark by adding more L1 data cache space is less cost-effective. Second, L1 miss latencies are tolerated well in an out-of-order machine.

In the next configuration, we remove address computation instructions for all stack references (denoted as `no_addr_cal_op` in the graph), thereby eliminating their dependencies. This dependency reduction benefits some benchmarks such as 256.bzip2, which improves by 11%. Since our processor model supports out-of-order execution with a 256-entry RUU, the address calculation could be easily hidden by other independent instructions, and the overall speedup is only 3%. This observation concurs with the results reported in [9] where their zero-cycle load technique posted significant gains only for in-order machines.

Most of the performance boost comes from the implementation of the stack value file, posting an incremental improvement of 28%. We show the speedups with a single-ported SVF as well as a dual-ported SVF. A dual-ported SVF on a 16-wide machine (which gives an incremental gain of

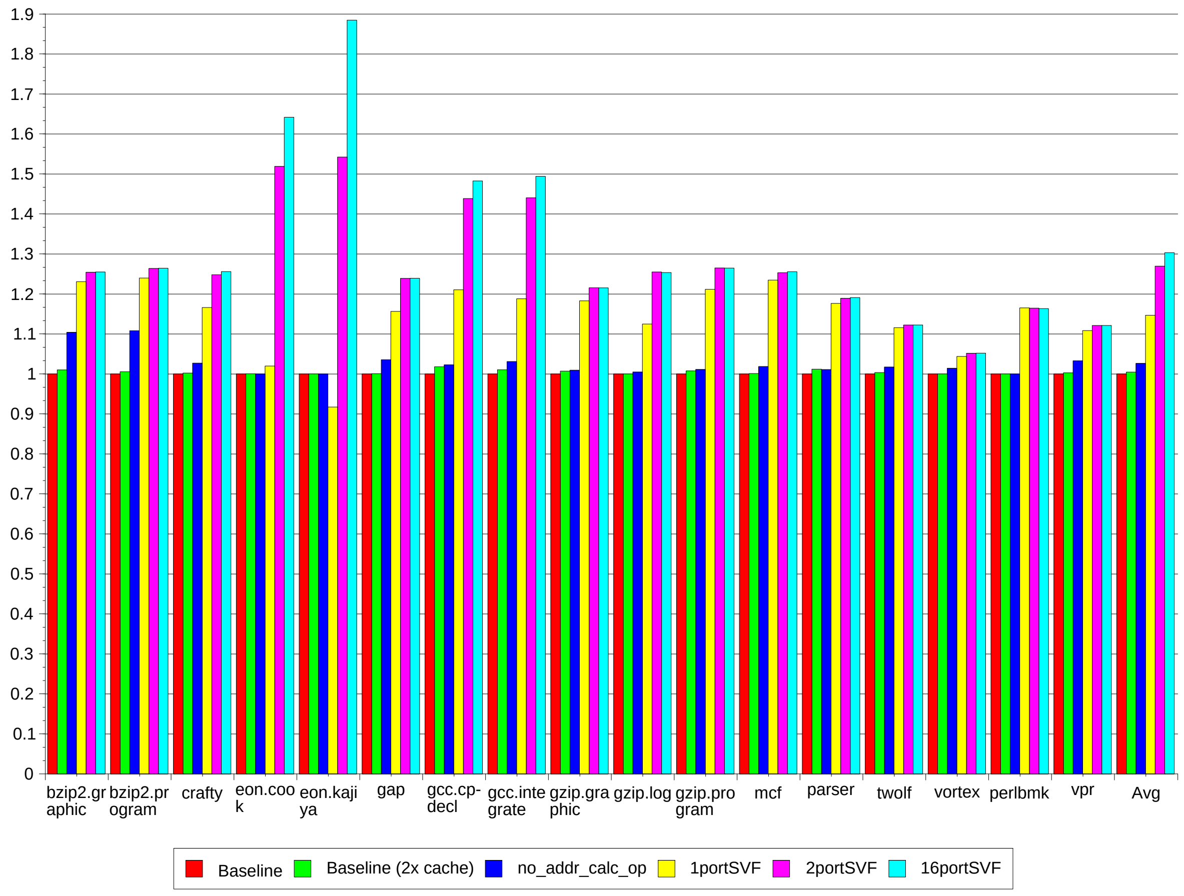


Figure 5.9: Hierarchical Performance Analysis (dual-ported cache)

27%) performs almost on par with a 16-ported SVF for most of the SPECint2000 benchmarks. This suggests that a limited number of ports covers most of the potential gain, except for 252.eon. This benchmark seems to have many more clustered stack references that lie on the critical path of the performance, consequently, a single ported SVF machine actually achieves less performance than the baseline mode. A larger number of SVF ports accommodates this bursty stack reference parallelism.

5.5.3 SVF vs. Stack Cache

5.5.3.1 Performance

A related approach, the decoupled stack cache [28], is compared against our SVF scheme along with the baseline microarchitecture. The stack cache is implemented as a direct-mapped cache and has the same capacity (8KB) as our stack value file (1024 entries × 8 bytes). Figure 5.10 shows the comparison of the SVF, stack cache and baseline approach with different port combinations. The *(R+S)* symbol represents the configuration with *"R"* regular L1 cache ports and *"S"* SVF or stack cache ports. The (4+0) configuration uses a longer data cache hit latency (4 cycles instead of 3 cycles) than (2+S)'s because of the larger number of ports. The performance numbers for the SVF scheme were generated by the complete implementation described in Section 5.3.

The baseline (4+0) might be expected to outperform the (2+2) because the four universal L1 data cache ports in (4+0) can service four concurrent memory references, no matter which memory regions these references are going to, whereas two memory references out of the (2+2) must be from stack, otherwise the ports are left unused. However, in several cases, the SVF scheme outperforms

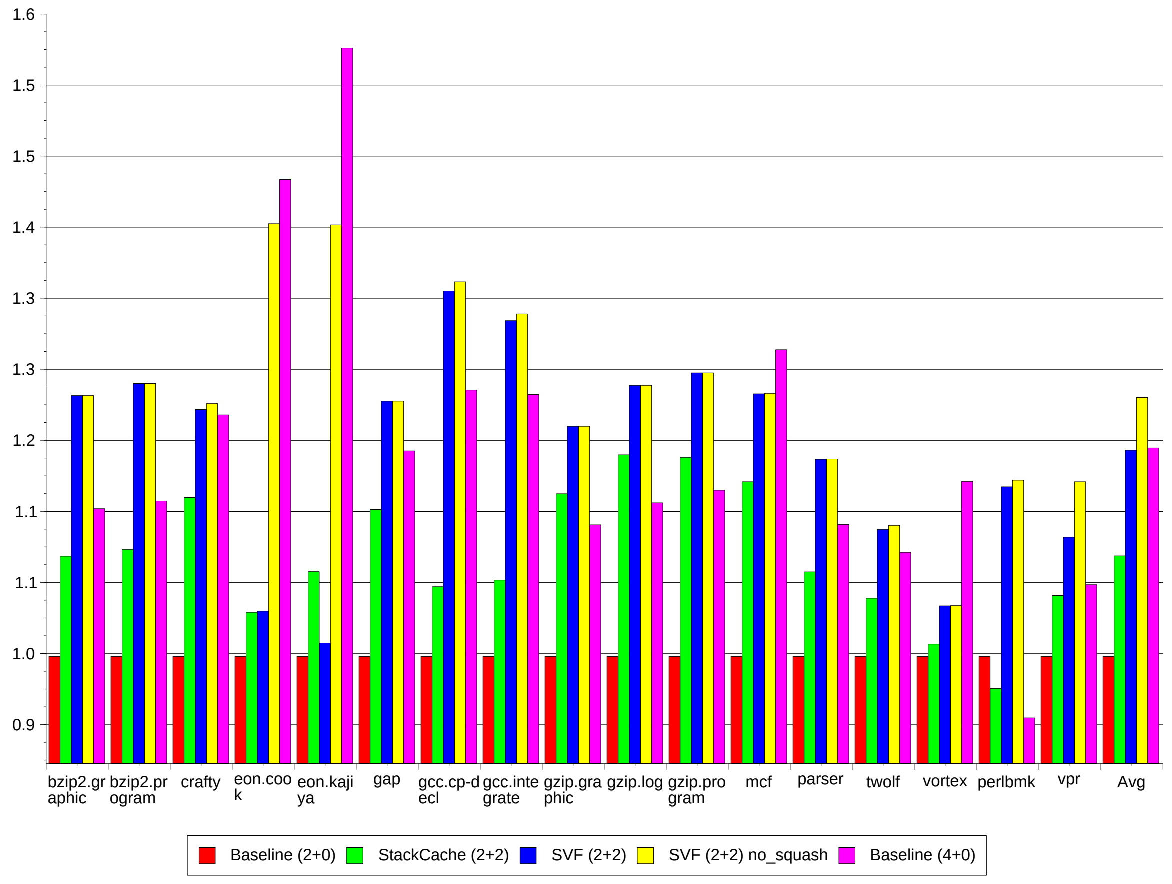


Figure 5.10: Comparison of Different Cache Implementations

the more flexible configuration, yielding a 4% improvement in overall. One reason is the longer latency in (4+0). The other is that input data of instructions on the critical path can be directly read from the physical registers in the RUU, indexed through the register alias table. Even though the store-forwarding mechanism exists in a conventional microarchitecture, yet it will take some cycles (3 in our case) to poll for a hit in the LSQ.

There is one anomaly for 253.perlbmk, where the stack cache (2+2) runs a little bit slower than the baseline (2+0). We found that the stack cache misses dominate the critical path whereas these data fit into L1 better for the baseline architecture.

Building a four-ported full-size L1 cache is more expensive than building an extra dual-ported smaller SVF or stack cache on top of a dual-ported L1. This is due to the expansion of the wordlines and bitlines in the cache structure [44]. In addition to the die area, the SVF can be more energy-efficient than a stack cache because of power savings in the cache tag arrays and the lower memory traffic. Quantifying this impact is suggested in Chapter 7.

Figure 5.10 also shows that the (2+2) SVF implementation outperforms the (2+2) stack cache scheme with one exception, 252.eon. In this benchmark, we found that a large number of load squashes occurs due to stores through \$gpr followed by loads through \$sp where these references map to the same stack addresses. As discussed in Section 5.3.2, these squashing activities can be eliminated using a different code generator tailored to the SVF implementation. By applying this optimization, represented by the *no_squash* bars, we can greatly improve the performance of the 252.eon, making it outperform the stack cache scheme by over 30%. Without the no_squash code optimization, the SVF outperforms the stack cache by roughly 14%, and with the no_squash feature

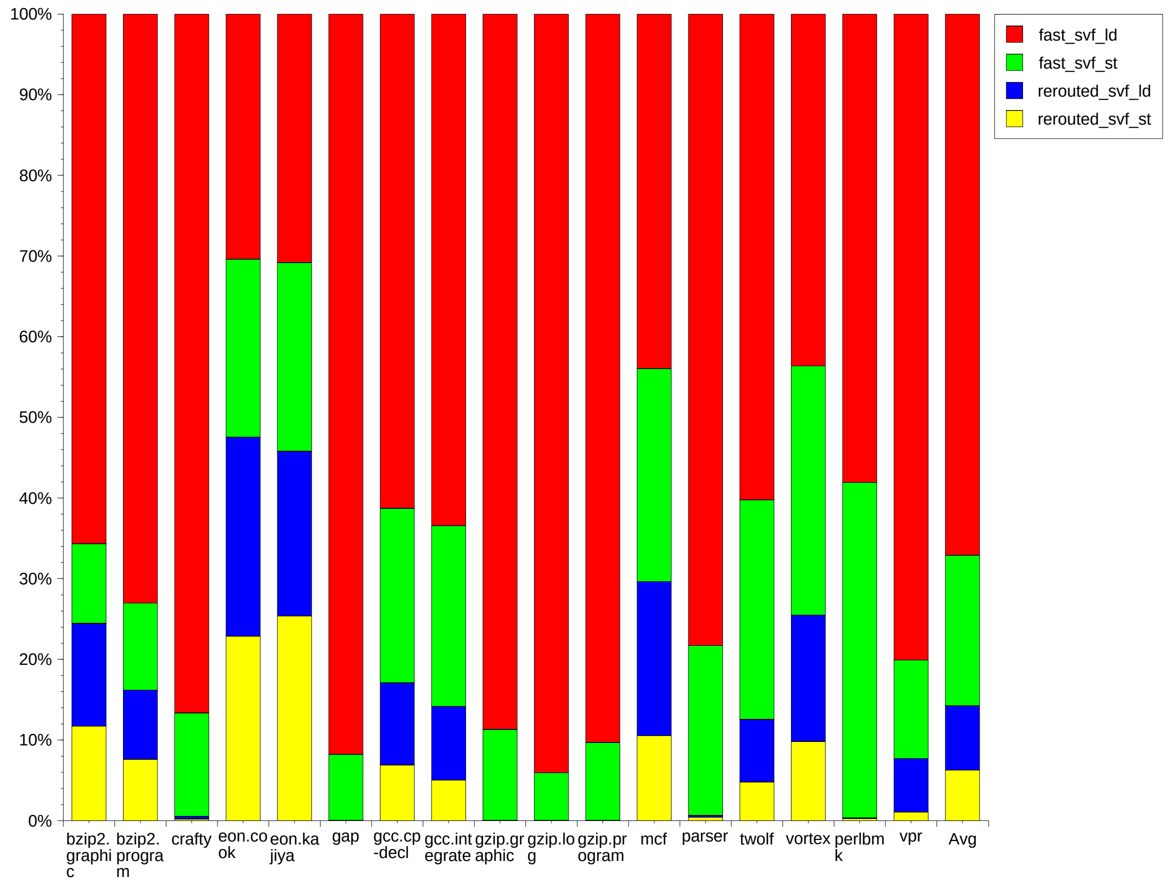


Figure 5.11: Breakdown of SVF Reference Types

the average is 9%.

Figure 5.11 shows the breakdown of the SVF references. The fast SVF loads and stores are the references that were directly morphed in the front-end pipeline. The re-routed SVF references are SVF references but through registers other than $sp and got rerouted after their addresses are calculated. In average, around 86% of stack references can be directly morphed into register moves in the front-end, while 14% of them are re-routed into the SVF.

A stack cache has a potential advantage in that it may be able to capture a larger working set (i.e., locations farther from TOS) than the SVF if spatial locality is poor. Since the SVF exploits temporal locality only when frequently accessed data are near the top of the stack while the stack cache can exploit temporal locality across the entire stack. However, Figure 5.3 and Figure 5.4 show that this does not happen for SPECint2000.

5.5.3.2 Memory Traffic

The main performance difference between the SVF and a stack cache arises from the exploitation of the semantic information inherent in adjustments to the stack pointer. Because the region of memory contained in the SVF is guaranteed to be contiguous, some assumptions can be made that cannot be made for a stack cache:

1. Allocations: A new allocation made as the stack grows downward for a SVF implies that the data must be invalid. No such assumption can be made for a stack cache, since the data may have already been written and replaced. Thus a stack cache must read the rest of the line

size	*2KB*			
	Quad-Words In		Quad-Words Out	
Benchmark	Stack $	SVF	Stack $	SVF
bzip2.graphic	1350700	28159	493604	70971
bzip2.program	770608	19515	289096	64253
crafty.ref	45811024	45	6634372	435
eon.cook	85730448	63	37119416	131136
eon.kajiya	33019440	71	23431164	3375088
gap.ref	193148	4575	175880	5056
gcc.cp-decl	17523908	3833	11472364	1162725
gcc.integrate	30826224	5195	20421952	1275410
gzip.graphic	296	7	176	0
gzip.log	296	7	176	0
gzip.program	324	8	200	0
mcf.inp	220	6	40	0
parser.ref	76980	105	76108	2661
twolf.ref	2989324	13	2762292	7644
vortex.ref	988	7	712	240
perlbmk.scrabbl	99116	2242	89508	57524
vpr.ref	1432	17	1104	140

Table 5.2: Memory Traffic for Stack Cache and SVF schemes (2KB)

size	*4KB*			
	Quad-Words In		Quad-Words Out	
Benchmark	Stack $	SVF	Stack $	SVF
bzip2.graphic	452848	28138	101124	96358
bzip2.program	434992	19492	109684	90758
crafty.ref	572	45	72	0
eon.cook	677452	56	16180	172
eon.kajiya	12512884	63	11544940	172
gap.ref	130148	4453	125564	5984
gcc.cp-decl	13323984	974	8756656	1282023
gcc.integrate	27329584	3706	18086172	2138591
gzip.graphic	144	7	0	0
gzip.log	144	0	0	0
gzip.program	148	8	0	0
mcf.inp	204	6	0	0
parser.ref	592	0	80	0
twolf.ref	652	13	188	0
vortex.ref	488	7	44	0
perlbmk.scrabbl	89948	2243	81112	57580
vpr.ref	652	17	152	0

Table 5.3: Memory Traffic for Stack Cache and SVF schemes (4KB)

size	*8KB*			
	Quad-Words In		Quad-Words Out	
Benchmark	Stack $	SVF	Stack $	SVF
bzip2.graphic	744	6	0	501
bzip2.program	756	6	0	474
crafty.ref	556	45	0	0
eon.cook	1568	56	532	38
eon.kajiya	905544	63	446440	38
gap.ref	118232	454	116344	0
gcc.cp-decl	6783056	1016	5274116	1053545
gcc.integrate	22657988	2739	15272932	2671678
gzip.graphic	144	7	0	0
gzip.log	144	7	0	0
gzip.program	148	8	0	0
mcf.inp	204	6	0	0
parser.ref	592	105	0	0
twolf.ref	564	13	0	0
vortex.ref	480	7	0	0
perlbmk.scrabbl	79312	14	78812	0
vpr.ref	644	17	0	0

Table 5.4: Memory Traffic for Stack Cache and SVF schemes (8KB)

before data can be written.

2. Dirty Replacements: When locations are replaced as the stack shrinks for the SVF, they are semantically guaranteed to be dead, and need not be written back. No such assumption can be made for a stack cache, and the line must be written back.

Table 5.2, Table 5.3, and Table 5.4 summarize the in (read) and out (write) memory traffic incurred for our SVF design and for a decoupled stack cache design, for different SVF and cache sizes. The stack cache's memory traffic corresponds to the 3C misses (compulsory, capacity, and conflict misses) [52], along with dirty writebacks, which generate traffic between the stack cache and the L2. The SVF's dribbling traffic to the L1 only occurs on demand, for dirty and live data. For instance in 256.bzip2 with a 2KB stack cache, about 1.35 million quad-words were allocated into the stack cache and 0.49 million quad-words were evicted due to dirty replacements. In contrast, the SVF dribbled in only 28,159 quad-words and dribbled out 70,791 quad-words. In most of the scenarios, the SVF dramatically reduces traffic by orders of magnitude. As aforementioned, the traffic is reduced because the SVF transfers dirty data in a finer granularity and requires no demand load on write misses. In addition, the SVF does not write the deallocated stack frame out to memory as data on deallocated stack frame are semantically dead. There are some SVF cases that generate dribbling-in traffic with no dribbling-out writes beforehand because the Compaq Alpha compiler code generator generates data prefetching instruction automatically, e.g. `ldq_u r31, 0(sp)`.[5] These instructions bring cache lines in by loading undefined values into the read-only register. Therefore, data could be dribbled in even though they were never defined in the code.

5.5.3.3 Context Switches

Upon a context switch, there is likely to be an increase in memory traffic, for either a stack cache or the SVF because both have to write back dirty data.

[5]Note that the r31 in Alpha ISA is defined as a `zero` register.

Benchmark	Stack Cache	Stack Value File
256.bzip2	83	33
186.crafty	1040	201
252.eon	7053	740
254.gap	830	64
176.gcc	4235	188
164.gzip	3	1
181.mcf	477	153
197.parser	1253	66
300.twolf	727	248
255.vortex	7	2
253.perlbmk	571	134
175.vpr	800	87

Table 5.5: Memory Traffic on Context Switches

On the context switch both SVF and a stack cache contain data for the current process. Thus the new process will replace the current process data in the stack cache or stack value file with its new data. Since both stack cache and stack value file are small, a large percentage of their locations will likely be replaced soon after the context switch. This would cause then significant writeback traffic soon after the context switched occurred.

However overall SVF to L1 traffic on the context switch is lower than stack cache to L2 traffic. This occurs because the SVF invalidates deallocated stack frames, so they are never written back to L1 cache. In addition, the SVF maintains dirty bits on a per word basis while the stack cache uses per block dirty bits. Since almost all words are accessed first by a store instruction, many words are dirty and therefore almost all cache lines are dirty. Unaccessed words need not be stored by the SVF whereas they must be stored as a member of a dirty block in the stack cache. Table 5.5 quantifies the traffic for both stack cache and stack value file in bytes averaged over the total number of context switches with a context switch period of 400000 instructions. For instance, writeback traffic for the stack cache in the case of 252.eon is 7K per context switch on average, which is about 10 times more than in the case of a stack value file. Table 5.5 illustrates that writeback traffic for the stack value file is 3 to 20 times smaller then the writeback traffic for the stack cache.

5.5.4 Interlock Stall Overheads

As described in Section 5.3.1, our SVF design employs an interlock mechanism to avoid the inconsistency issue of the stack pointer value when it is being updated through complex operations or using registers other than $sp. This overhead is quantified in our simulations and it is observed that the impact is fairly insignificant. For the configuration with one cache port and one SVF port, this overhead is almost 0% for most of the SPECint2000 benchmark except for 176.gcc. The overheads are 1.3% and 0.9% for the training input (cp-decl.i) and the reference input (integrate.i), respectively.

5.5.5 Memory and SVF Ports

The speedups measured for the complete SVF implementation versus a baseline machine are illustrated in Figure 5.12. The execution speedup measured for both single-ported and dual-ported data caches are shown. The average performance improvement for adding a single-ported SVF to a single-ported data cache is 50%. Although most current processors incorporate dual-ported

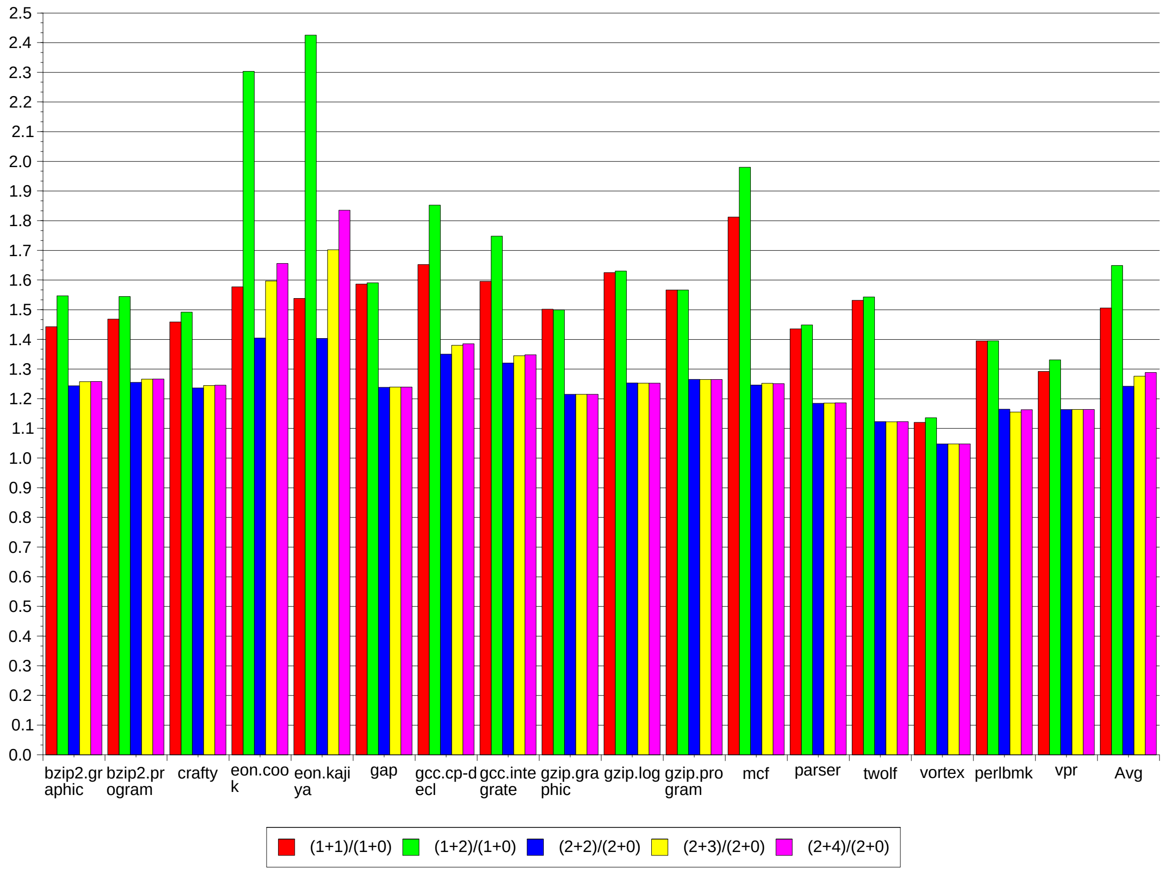


Figure 5.12: Performance Improvements over Baseline Microarchitecture

data cache designs, however, from a low-cost/low-power processor or embedded processor design perspective, it is appealing to build a large single-ported data cache with a small double-ported stack value file to reduce both cost and power dissipation, and in the meantime also improve overall performance.

When the SVF is dual ported, improvement climbs to 65%. For most of the benchmarks, performance is saturated when two SVF ports are supported, except for 252.eon which continues to improve its performance as the number of SVF ports are increased. Improvements are lower for cache designs supporting dual-ported first level data caches since port contention is reduced; however, adding an SVF will still yield significant additional improvement. For a reasonable configuration with a dual-ported SVF added to a dual-ported data cache, performance improves by average of 24% over a conventional microarchitecture, the largest performance improvement is 84% for 252.eon.

5.6 Related Work

Techniques for fast procedure calls [70] were broadly studied in the 1980's when CISC machines dominated all market segments. The overheads of saving and restoring the register file associated with each procedure call were rather significant [43]. Many prior commercial and research microprocessors had tried to address this issue at extra hardware cost.

The HP3000 Series II [17], a stack-oriented architecture designed by Hewlett-Packard in the late 70's, used a 4-entry top-of-stack (TOS) cache as an extension to stack memory. Data on the TOS are passed onto the TOS registers for processing. A push or pull is performed whenever an overflow or underflow occurs. Since this machine does not have a data cache, the stack cache functions as a tiny direct-mapped cache with FIFO replacement policy for stack references. Data in this study showed that a 4-entry stack cache was more than sufficient for applications of the day.

Following the RISC philosophy, the CRISP [14][15][41] and Hobbit [6] processors, developed at Bell Labs adopted a complete memory-to-memory instruction set architecture with few addressing modes to avoid the overheads of procedure calls. The design offloads the burden of register allocation on the top of the stack from the compiler to the hardware by incorporating a small stack cache (32 entries). The processors index the stack cache on-the-fly using the low-order bits of the referenced address. The stack cache is the processors' only data cache.

Register windows [7][83][85][114] or the register stack engine (RSE) [33] are used in some of today's high-performance microprocessors to eliminate procedure call and return overheads. Extra instructions may be needed, e.g. *save* and *restore* in SPARC-V9 or *alloc* in IA-64. This general approach is part of the architecture, not just the implementation.

There have been several proposals for early address resolution to improve memory instruction latencies. These techniques enable our SVF design by providing a mechanism for early address resolution. Austin, Pnevmatikatos and Sohi [8] introduced a fast address resolution scheme by predicting effective addresses early in the pipeline. They found that with simple compiler and linker support, the prediction accuracy ranges from 62 to 99%. In [9], Austin and Sohi proposed and evaluated pipeline designs to support zero-cycle loads. Although the speedups are encouraging for in-order processors, the speedups for latency-tolerant out-of-order processors are generally less than 10%. In [12], Intel researchers proposed a technique dubbed *register tracking* for early memory address resolution for operations of the form reg$\pm$imm in the front-end pipeline. They demonstrated that this technique reduces load-to-use latencies to the data cache by experimenting a deep pipeline that contains 8 stages between decode and execution.

The number of cache ports becomes more crucial as the processor's issue width gets wider with more aggressive and accurate multiple branch prediction mechanisms. In more recent work [28], Cho, Yew and Lee proposed a data-decoupled architecture that partitions memory reference streams into two different substreams and feeds them through decoupled memory pipelines for execution. In this scheme, they studied the performance impact of decoupling local variables allocated on the run-time stack. They concluded that a small 2KB local variable cache (LVC) achieves a 99% hit rate for most of the SPEC95 benchmark programs and as a result, leaves more headroom for increasing data cache bandwidth. In their follow-up work [27], they introduced the notion of access region locality and proposed an access region prediction table (ARPT) in the fetch stage to predict which region an instruction is referencing. In the decode stage, the memory operation is directed into the predicted region pipeline for future processing.

Tyson and Austin [111] devised a mechanism that performs memory renaming dynamically to reduce memory traffic. A memory dependency predictor is used to predict the relationship of a producer (stores) and a consumer (loads). The predictor uses this information to index a non-architected value file for loads. They employ a confidence mechanism to control the prediction. Their simulation shows 16% performance improvement on average.

Rivers et al. [94] identified limitations of existing multi-ported cache designs and proposed a Locality-Based Interleaved Cache (LBIC) that multi-ports a line buffer instead of the entire cache bank to exploit the spatial locality of a cache line while reducing the cost of building a true multi-ported cache.

5.7 Chapter Summary

In this work, we perform a detailed analysis of stack reference behavior identifying several unique characteristics regarding how the stack is accessed. These characteristics led us to propose a new microarchitectural enhancement, the stack value file (SVF), designed to optimize the stack references induced by high-level language conventions.

The contributions of this research are threefold:

1. We identified several characteristics of stack references that differ from general data references. These include: a single contiguous access region (eliminating the need for tags), a much higher percentage of first reference store operations (making per word valid bits attractive), frame deallocations invalidate dirty data above the new TOS (making writebacks unnecessary), and most references use a single \$sp-relative address mode (making fast address calculation feasible).

2. We proposed a new microarchitectural structure, the SVF, to exploit those characteristics and show how it can be integrated into existing processor pipelines to improve cache access latency and reduce memory traffic requirements.

3. We evaluated our scheme, comparing it to a previous cache-oriented approaches to partitioning stack references. These results show that an SVF can obtain a 24% average performance improvement for conventional microarchitectures, while significantly reducing memory overhead traffic over data-decoupled stack/non-stack caches.

Furthermore, our microarchitecture design transforms stack pointer-based memory accesses into register-to-register moves. This increases exploitable instruction-level parallelism by adding ports, off-loading bandwidth from the first-level data cache, and reducing the latency of the access. For a 16-wide machine, this increases performance for an SVF of infinite size and ports by an average of 31% for the SPECint2000 benchmarks.

Overall, these performance results make the stack value file an attractive design option, boosting performance without significant increases in area or complexity. The die area allocated to the SVF can be reallocated from space that otherwise would have gone to a larger first-level cache. The SVF is direct-mapped, can be single-ported, and can easily be banked. It uses no tag area like its cache counterpart.

The additional complexity for the SVF is quite limited. \$sp-relative stack references are identified easily; thus their special handling does not add much complexity. References to the stack without an \$sp-relative addressing mode are infrequent, so added recovery cost is reasonably amortized. Cache tag space is eliminated in preference to a per word valid bit, resulting in little or no additional data storage overhead relative to a stack cache implementation. For more deeply pipelined processors, the value of early address computation is increased and our technique should deliver increasing performance gains.

CHAPTER 6

EFFICIENT BANDWIDTH UTILIZATION USING AN EAGER WRITEBACK CACHE

Caches are very effective at reducing memory bus traffic by intercepting and handling most of the read and write requests generated by the processor. Support for writes (stores) tends to be simple – on a store the data item is either written into both the cache and through the cache hierarchy to the memory (referred to as a *write-through* policy), or it is written into the cache exclusively and the data item is written out to memory only when the cache line is evicted (known as a *writeback* policy.)

Caches employing a write-through policy generate memory traffic every time a store occurs in the program. Since it would largely defeat the purpose of having a cache if the processor had to block on each store until the write completed, write-through caches use a structure known as a *store buffer* or *write buffer*[103] to buffer writes to memory. Whenever a write occurs, the data item is written into both the cache and this structure, allowing the processor to continue executing without blocking (until the store buffer becomes full). The store buffer will send its contents to memory as soon as the bus is idle.

Writeback caches, on the other hand, generate memory traffic much less frequently. When a store occurs in a writeback cache the data value is written into the corresponding line in the cache, which is then marked *dirty*. Writes to memory occur only when a line marked *dirty* is evicted from the cache (usually due to a cache miss) in order to make room for the incoming data item.

Whenever there are many consecutive misses (caused by context switches, or working set changes, or by algorithms in certain graphics applications, for example) the writeback cache can find itself blocked waiting for a dirty line to be written to memory. This is the same problem faced by the write-through cache, and can be dealt with in much the same manner by adding a writeback buffer. However, there are certain classes of programs which suffer from memory delay penalties that even a large writeback buffer cannot eliminate. For example, many newer applications (e.g. 3D graphics or multimedia) have enormous incoming data streams. In these programs, the stream of incoming data items can cause many conflict cache misses and trigger the eviction of many dirty lines. This dirty writeback traffic must compete for available memory bandwidth with the arriving data, and often impedes the delivery of the data to the cache. For programs where overall performance is bound by memory bandwidth, this competition for bandwidth can have a substantial negative impact.

In this chapter we propose a modification of the writeback policy which spreads out memory activity by selectively writing some dirty lines to memory whenever the bus is free, instead of waiting until those lines in the cache are replaced. This early writing of dirty lines to the memory system reduces the potential impact of bursty reference streams, and can effectively re-distribute and balance the memory bandwidth and improve system performance.

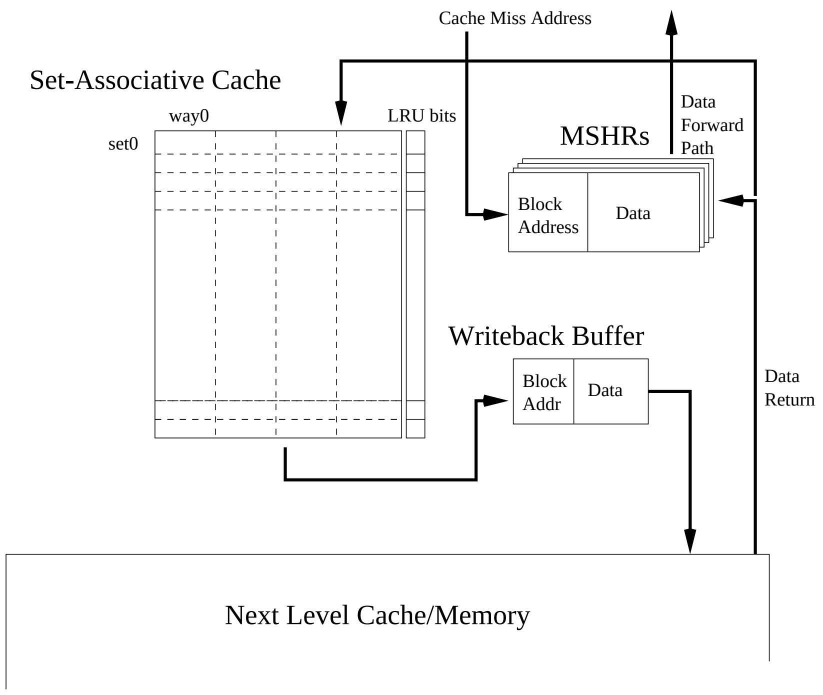


Figure 6.1: Architectural Block Diagram of non-blocking caches.

6.1 Caches and Memory Subsystem

Caches that employ a writeback policy reduce memory traffic by delaying the transfer of data to memory as long as possible. Most modern microprocessors using a writeback cache policy incorporate a writeback (or cast-out) buffer, which is used as temporary storage space for holding dirty cache lines while the data request that caused the eviction is serviced. Upon eviction, a dirty cache line is deposited into the writeback buffer, which usually has the highest bus scheduling priority among all types of non-read bus transactions. Once the writeback buffer fills up, subsequent dirty line replacements cannot take place. As a result, their corresponding data demand fetch operations cannot be committed into the cache, and the processor pipeline stalls waiting for the dependent data.

It is possible to alleviate this problem somewhat by extending cache hardware. *Non-blocking caches* which use a set of *miss status holding registers* (MSHRs) to manage several outstanding cache misses have been proposed by Kroft [67]. When a cache miss occurs in a non-blocking cache, it is allocated an empty MSHR entry. Once the MSHR entry is allocated, processor execution can continue. If none of the MSHRs are available (i.e. a structural hazard [52] exists due to resource conflicts), the processor has to block until an MSHR entry becomes free.

By adding data fields to the MSHRs, it would be possible to use them to temporarily store returning cache lines. This would allow fetched data to be immediately forwarded to the appropriate destination registers, and help overcome the situation where the cache cannot be written to because the writeback buffer is full. However, this scenario delays MSHR deallocation and can lead to processor stalls on a cache miss because of there being no free MSHRs. Figure 6.1 illustrates a non-blocking cache organization.

In addition, in a modern computer system memory bandwidth is not exclusively dedicated to the host processor. There are often multiple agents on the bus (such as graphics accelerators or multiple processors) issuing requests to memory over a short period of time. A typical system architecture

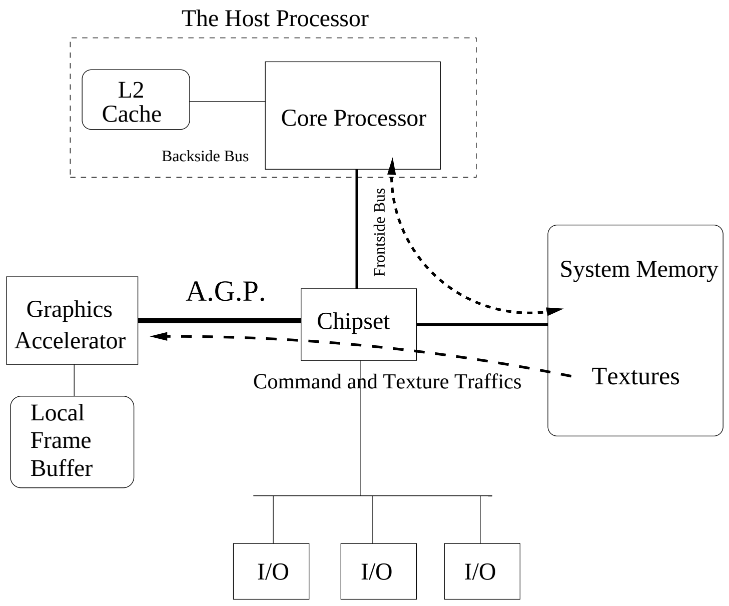


Figure 6.2: A PC system architecture with AGP.

of a contemporary PC system is illustrated in Figure 6.2.

In a contemporary PC platform with an Accelerated Graphics Port (AGP) interface [29] running a graphics-centric application, for example, the graphics accelerator shares system memory bandwidth with the host processor by constantly retrieving graphics commands and texture maps from the system memory. As shown in Figure 6.2, the same system memory serves as the instruction and data repository for both the processor and graphics accelerator.

In a common 3D graphics application, for instance, the processor reads instructions and triangle vertices, processes and then stores them with rendering state commands back into AGP memory space. The graphics accelerator then reads these commands out of AGP memory for rasterization. In addition to the command traffic, the graphics accelerator also reads a large amount of texture data (which constitutes the major portion of AGP traffic on the bus). These textures are mapped onto polygon surfaces to increase the visual realism of computer-generated images. In the future, with richer content 3D graphics applications or graphics accelerators with enhanced quality features such as bi-linear/tri-linear interpolation, AGP command and data bandwidth demands for graphics accelerators will undoubtedly be even greater than they are now.

Current cache designs have difficulty in efficiently managing the flow of data in and out of the cache hierarchy in these data intensive applications. Buffering techniques, including write buffers and MSHRs can help, but do not alleviate the problems of clustering bus traffic caused by writeback data. In the next section we introduce a new technique designed to distribute the writes of dirty blocks to times when the bus is idle.

6.2 Eager Writeback

6.2.1 Overview

To address the performance drawbacks of a conventional writeback policy, we are proposing a new technique called *Eager Writeback*. The fundamental idea behind Eager Writeback is to write dirty cache lines to the next level of the memory hierarchy and clear their dirty bits earlier than in a conventional writeback cache design, in order to better distribute bandwidth utilization and alleviate memory bus congestion. If dirty cache lines are written to memory when the bus is less congested, then there will be fewer dirty lines that require eviction during peak memory activity.

In essence, we are speculating that certain dirty lines will not be written into again before eviction and thus there is no need to wait until eviction time to perform the cache line write. An Eager Writeback will never impact the correctness of the architectural state even if the operation that triggers it was wrongly speculated - if our speculation is incorrect and we write too often, we approach the limiting case of write-through cache behavior. If we do not speculate often enough, we approach writeback cache behavior. However, in neither case do we violate any correctness constraints. In the worst case, incorrect speculation may lead to excessive memory traffic, by consistently writing and cleaning lines in the cache that are then quickly marked dirty again.

In order to select the best "trigger" to cause an eager writeback, we examined the probability of rewriting a dirty line in a set-associative cache when it was in a given state (somewhere between MRU and LRU) for the SPEC95 benchmarks and four applications from the X benchmark suite.

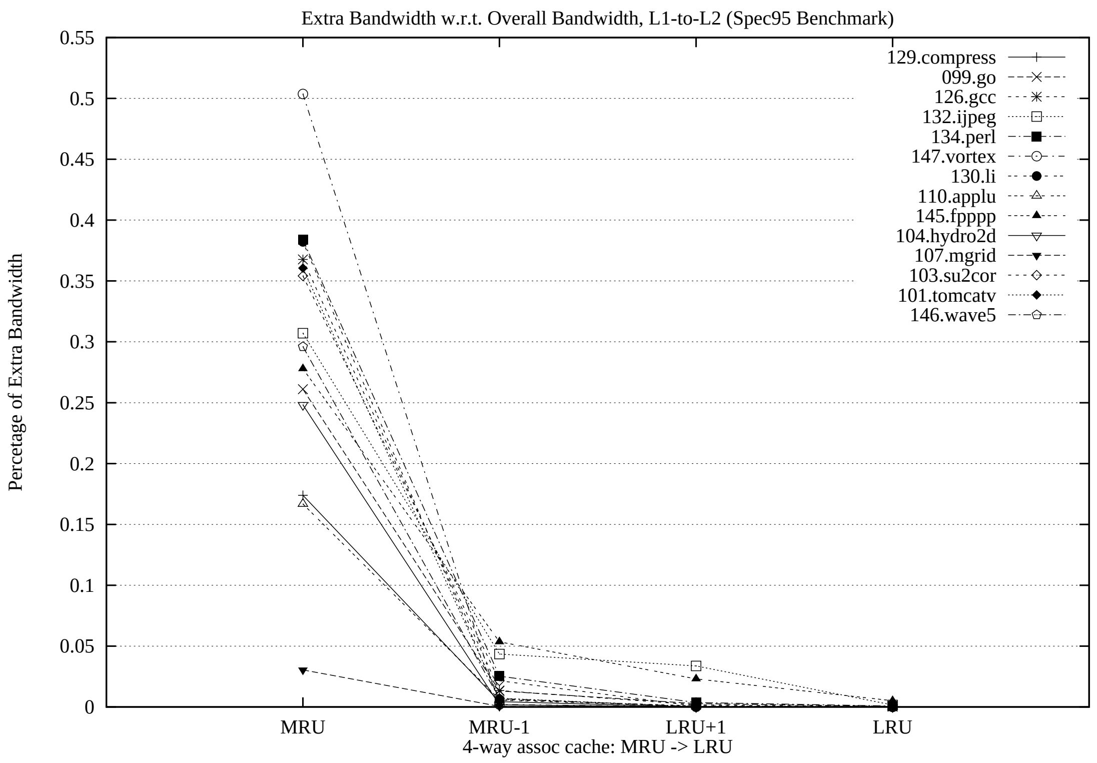
As discussed in Section 3.4, our results indicate that cache lines that have been marked dirty and reach the LRU (Least Recently Used) state in a 4-way set-associative cache are rarely written to again before they are evicted. To recapitulate the data, Figure 3.13 and Figure 3.14 show the probability that a line that was marked dirty is written to again as it moves from the MRU (Most Recently Used) state to the LRU state for both L1 and L2 caches. The graph on the top in Figure 3.13, for example, shows that in the L1 cache the average probability of a dirty line in the LRU state being re-written is 0.15, while the similar probability for a dirty line in the MRU state is 0.95. The probabilities of re-dirtying lines in the LRU state are even lower in the L2 cache - in fact, close to 0 as shown in the graphs on the bottom of Figure 3.13 and Figure 3.14.

These figures indicate there are some programs (such as *fpppp* and *su2cor*) that have a fairly high probability of writing to dirty lines after they have entered the LRU state. In order to further evaluate this, we looked at the ratio of the number of times that a dirty line in the LRU state is written to normalized to the number of times that a dirty line in the MRU state is written to. The results were presented in Figure 3.15 in Section 3.4, which shows that while the probabilities may be high, the actual number of these occurrences is negligible compared to the rewriting that occurs when a line is in other states (MRU, MRU-1, etc.). These trends held across a wide range of cache configurations, and imply that once a line enters the LRU state it becomes a prime candidate for Eager Writeback, since there is a very low occurrence of it being re-written (and thus marked dirty again).

6.2.2 Design Issues in Eager Writeback Caches

There can be many different approaches to deciding when to trigger an Eager Writeback. As shown in the previous section, one obvious candidate is to use the transition of a dirty line into the LRU state as a trigger point for an Eager Writeback. For example, when a cache set is being accessed and its corresponding LRU bit is being updated, the line can be checked to see if it is marked dirty. If it is, then a dirty writeback can be scheduled, and the dirty bit can be reset. Note that the line is not cast out from the cache, thus the line can be found in the cache without performance loss if subsequent loads or stores access it. However, if a store instruction does write into such a line and make it dirty again, the line will simply be written back to memory again in the future and generate more memory traffic. Figure 6.3 shows the extra bandwidth incurred for SPEC95 as a function of which state of the LRU stack is used as the Eager Writeback trigger. It is shown that only about 1% extra memory traffic is incurred if the LRU state is selected as the trigger. In addition, when the MRU state is used as the Eager Writeback trigger, it degenerates to the write-through policy.

If the writeback buffer is full at this point, two approaches can be considered; (a) simply abort the Eager Writeback; the actual dirty writeback will take place later when the line is evicted, or



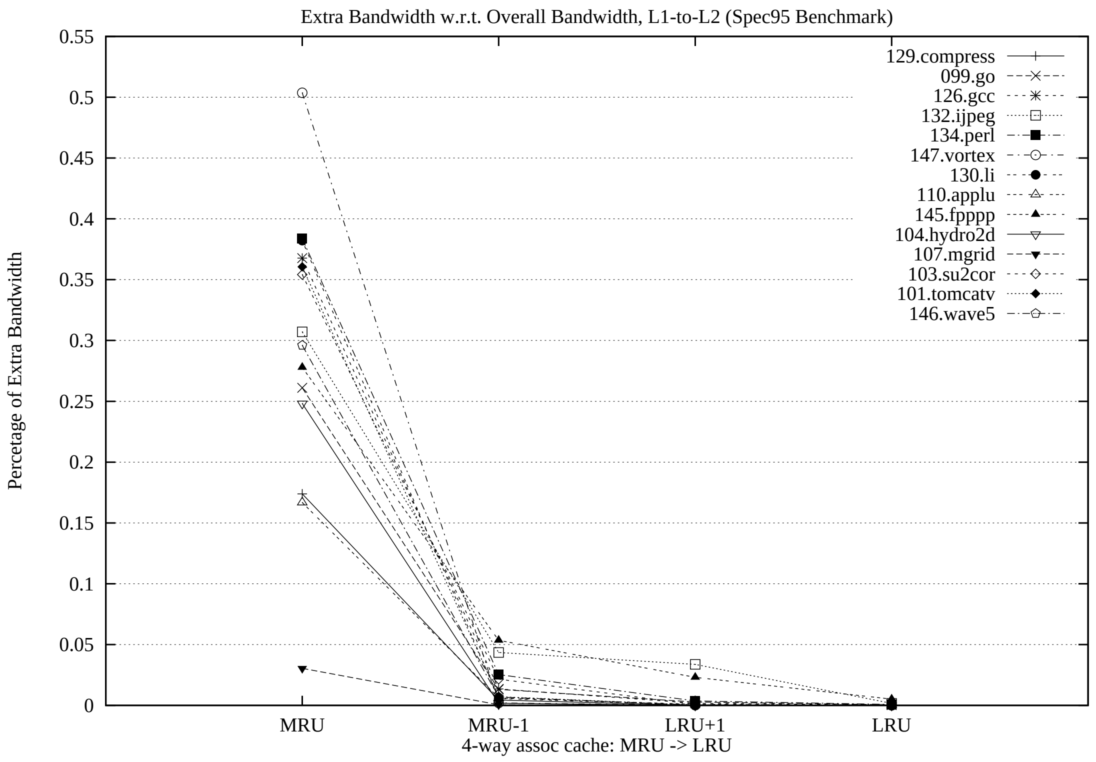


Figure 6.3: Extra traffic by triggering writeback upon different usage states for L1 and L2 caches (SPEC95)

Processor Architectural Parameters	Specifications
Core frequency	1 GHz
1st Level I-Cache	2-way, 256 sets, 32B line
1st Level D-Cache	4-way, 128 sets, 32B line
2nd Level Cache	Unified, 4-way, 4096 sets, 32B line
I- and D-TLBs	2-way, 128 sets, 32B line size
Backside bus	500 MHz (i.e. half-speed L2), 64bit wide
Frontside bus	200 MHz, 64bit wide
Memory model	Rambus DRAM model (1.6GBytes/sec peak)
Branch predictor	2-level *gshare* adaptive, 10-bit history
Instr. fetch/decode/issue/commit width	8 / 8 / 8 / 8
Load/Store Queue size	32
Register update unit size	64
Memory port size	2
INT/FP ALU size	4 / 4
INT/FP MULT/DIV size	1 / 1

Table 6.1: Summary of the Baseline Processor Model.

(b) perform the eager writeback when an entry in the writeback buffer becomes free. This provides the ability to perform eager writeback anytime between when a line is marked LRU and when it is evicted.

To provide this capability using a minimum of hardware, we chose to simulate an *Eager Queue* which holds attempted eager writebacks that were unable to acquire writeback buffer entries. Whenever an entry in the writeback buffer becomes available, the Eager Queue checks the cache set on the top of the queue to see if the dirty bit of the LRU line in the indexed set is set. If it is, the line is moved into the writeback buffer.

An alternate implementation considered during this research was *Autonomous Eager Writeback.* This implementation used a small independent state machine which autonomously polled each cache set in round-robin fashion and checked the dirty bit of its LRU line, initiating eager writeback on these dirty lines when the writeback buffer was not full. Whether eager queues or the autonomous state machine is more feasible is highly dependent on the processor and cache organization. For this study we present results for the more conservative approach which used eager queues.

6.3 Simulation Framework

Our simulation environment was based on the SimpleScalar tool set version 2.0. Refer to Section 2.1 for basic information of the simulator. The microarchitectural parameters used in our baseline processor model are shown in Table 6.1. Table 6.2 lists the latencies of each functional unit modeled in the simulation. A non-blocking cache structure, writeback buffer and eager queue associated with each cache level were added to the simulator for this study. The number of entries in each buffer was re-configurable from 1 to 256, and varied from simulation to simulation.

A pseudo-Rambus DRAM model was used in the external memory system. This single-channel RDRAM with 64 dependent banks can address up to 2GB of system memory. In the model, 32 independent banks can be accessed simultaneously (contiguous banks share the same sense amplifier for driving data out of the RAM cells). Row control packets, column control packets and data packets can be pipelined and use separate busses. RDRAM address re-mapping[35] was modeled to reduce the rate of bank interference. The peak bandwidth that can be reached in our RDRAM model is 1.6GB/sec.

A simplified uncacheable write-combinable memory [32] was implemented for the purpose of correctly simulating our benchmark behavior. Whenever a data write to an uncacheable region results in an L1 cache miss, the write operation will immediately request access to the bus and drive data out to the system memory directly (skipping a next-level cache look-up). Only cache line writes

Processor Architectural Parameters	Cycles in Processor Clock
1st Level I- and D-Cache	3 cycles, pipelineability = every 1 cycle
2nd Level Cache	18 cycles, pipelineability = every 10 cycles
I- and D-TLBs	2 cycles, pipelineability = every 1 cycle
Backside bus arbitration	4 cycles
Frontside bus arbitration	10 cycles
RDRAM Trcd, RAS-to-CAS	20 cycles
RDRAM Tcac, CAS-to-data return	20 cycles
RDRAM Trp, Row Precharge	20 cycles
INT ALU latency/thruput	1 / 1
INT multiplier latency/thruput	3 / 1
INT divider latency/thruput	20 / 19
FP ALU latency/thruput	2 / 1
FP multiplier latency/thruput	4 / 1
FP divider latency/thruput	12 / 12

Table 6.2: Latency Table (in core cycles) of the Baseline Processor.

are modeled — partial cache line updates are treated as full cache line writes in the simulator.

For modeling multiple agents on the memory bus, a memory traffic injector was also implemented. This injector allowed us to imitate the extra bandwidth consumed by other bus agents by configurable periodic injection of data streams onto the memory bus.

In order to evaluate the effectiveness of the Eager Writeback technique, we ran extensive simulations on the SPEC benchmark suite and two kernels representative of graphics applications. By concentrating the analysis on small, representative kernels, we can illustrate the potential benefits of our scheme in far greater detail than can be achieved running an entire application. The first of these kernels is a small 3D geometry processing kernel [91] [121] (*mini-geometry*), which is present in most triangle-based rasterization algorithms. Two different graphics rendering configurations were simulated, one which was very simple (i.e. ambient light with no external light sources), and one which included multiple diffuse light sources. The ambient light configuration reduces the computational requirements of the algorithm in order to maximize frame rate at the expense of picture realism.[1] The multiple light source configuration increases the computational demands, thereby reducing the relative impact of bus utilization as the processor spends more time processing between data element requests.

The second kernel represents a very general streaming data algorithm which processes large data sets. This kernel processes a large array of data (both reading and writing the data in the array), experiencing frequent cache misses as well as dirty writebacks.

The detailed discussion of these benchmarks were described in Section 2.2.

6.4 Simulation Results and Analysis

The simulation results are presented and analyzed in this section. For each kernel studied, we present two different data sets, one with no memory contention from other potential bus agents, and one with artificially injected memory traffic.

6.4.1 Spec95 Benchmarks

Table 6.3 shows the simulation results for the SPEC95 benchmark suite using 3 configurations — Baseline, Eager and Free Writeback. The Baseline case uses a single entry writeback buffer, while Free Writeback models a system in which dirty writebacks do not generate any memory traffic on the bus (thus it serves as an upper bound on performance.)

Looking at the table it is apparent that there is little performance gain possible for the programs

[1]This would be preferred in the DOOM application when processor performance is lacking.

sim cycle	*Baseline*	*Eager*		*Free Writeback*	
benchmark	cycles	cycles	speedup	cycles	speedup
go	4106741898	4106316586	1.000	4105050891	1.000
gcc	1425690611	1423578223	1.001	1419981686	1.004
li	401639628	401635232	1.000	401481752	1.000
ijpeg	2125521487	2123322070	1.001	2117908634	1.004
perl	3705579465	3701065936	1.001	3683430936	1.006
tomcatv	5436594306	5436670500	1.000	5436456381	1.000
su2cor	4625207540	4625248569	1.000	4625117247	1.000
mgrid	2138832527	2132120132	1.003	2061823555	1.037
fpppp	8404705112	8410760399	0.999	8404047239	1.000
wave5	2221747518	2208702430	1.006	2179225372	1.020

Table 6.3: Performance of SPEC95 Benchmarks. (WB buffer = 1, EQ = 4)

	Baseline	*Eager (EQ=0)*		*Eager (EQ=4)*		*Eager (EQ=256)*		*Free Writeback*	
Write Buffer size	cycles	cycles	speedup	cycles	speedup	cycles	speedup	cycles	speedup
1 (No light)	25364637	23876911	1.062	21838002	1.162	21837952	1.162	21798206	1.164
4 (No light)	25320139	21820627	1.160	21820566	1.160	21820566	1.160	21798206	1.162
256 (No light)	25320139	21820566	1.160	21820566	1.160	21820566	1.160	21798206	1.162
1 (3 diff. lites)	30643341	29200004	1.049	27176616	1.128	27176333	1.128	27134147	1.129
4 (3 diff. lites)	30643153	27158044	1.128	27158049	1.128	27158044	1.128	27134147	1.129
256 (3 diff. lites)	30643153	27158044	1.128	27158044	1.128	27158044	1.128	27134147	1.129

Table 6.4: Simulated cycles of 3D Geometry Pipeline.

in this suite, since the difference in the cycle count between the baseline case and the upper bound is negligible. This is not surprising, since it is well-known that the SPEC95 benchmark suite does not exercise the memory system aggressively. The SPEC95 suite is not a good candidate for memory system performance studies primarily due to its small working set size. For the rest of this study we will focus on the benchmarks that more aggressively exercise the memory system, and are arguably more representative of future workloads.

6.4.2 Analysis of 3D Geometry Pipeline

6.4.2.1 Without Injected Memory Traffic

Table 6.4 contains the number of mini-geometry kernel execution cycles for a variety of memory configurations. In this table, each row represents a different combination of writeback Buffer size and lighting conditions, while the columns contain different writeback strategy cycle counts. The first column, *Baseline*, contains the cycle count using a conventional writeback policy. The next 6 columns contain the results for 3 different variations of the *Eager Writeback* scheme and the speed-up of each scheme over the baseline case, with each scheme identified by the size of its Eager Queue (EQ). The simplest design choice is EQ=0, in which Eager Writebacks are dismissed if the writeback buffer is full. The other two cases can queue up attempted eager writebacks within Eager Queues of specified sizes. The rightmost column contains the Free Writeback case, which as stated earlier is the upper bound to available performance.

There are several things of interest to note in this table. Perhaps most significantly, it can be seen that increasing the depth of the writeback buffer has virtually no impact on the performance of the Baseline case. In fact, going from 1 to 256 entries in the writeback buffer only improves performance by 0.17%. This is because a large number of dirty writebacks are competing for bandwidth with the demand fetches, and the bus congestion can not be alleviated by a deeper writeback buffer.

On the other hand, adding Eager Writeback increases the performance of the system by 4.9% to 16.2% (depending on the light sources and the depth of the Eager Queue). For the simplest case of no Eager Queue and a single entry writeback buffer, the speedup ranges from 6.2% (for no light source) to 4.9% (with 3 light sources). This speedup is smaller than for the other cases, because many eager writebacks are dropped due to the lack of space in the writeback buffer. When the

RUU Full cycles	*baseline*	*Eager (EQ=0)*		*Eager (EQ=4)*		*Eager (EQ=256)*		*Free Writeback*	
Write Buffer size	cycles	cycles	improved	cycles	improved	cycles	improved	cycles	improved
1 (No light)	8404023	6678659	20.5%	4452469	47.0%	4452265	47.0%	4409553	47.5%
4 (No light)	8375679	4439397	47.00%	4439226	47.00%	4439226	47.00%	4409553	47.35%
1 (diff. lites)	8045791	6541028	18.7%	4361651	45.8%	4361259	45.8%	4313710	46.4%
4 (3 diff. lites)	8033850	4344799	45.92%	4344670	45.92%	4344653	45.92%	4313710	46.31%

Table 6.5: Resource Hazard Improvement of 3D Geometry Pipeline.

number of writeback buffer entries is increased (or the Eager Queue size is increased), the speedup achieved approaches the upper bound.

The "bandwidth shifting" effect is quite apparent in Figure 6.4 and Figure 6.5. These two figures present the utilization profile of memory bandwidth requested by the processor using the Baseline (Figure 6.4) and Eager Writeback (Figure 6.5) configurations, running the mini-geometry kernel. The y-axis plots the instantaneous bandwidth[2] versus the execution timeline on the x-axis.

The 12 broad spikes that saturate the peak RDRAM bandwidth in Figure 6.5 occur within the driver loop, where command output is being written into the write-combining graphics memory while eager writebacks of dirty lines are concurrently taking place. Since within the driver loop there is still some computation occurring, the bandwidth is not fully utilized, and eager writeback writes can use the available idle slots and maximize bandwidth. Conversely, in the baseline case, the same writebacks occur within the geometry computation loop. Thus these requests compete for the bus with the return of the data requested by vertex loads, and thus slow down the processing. Maximizing the utilization of the bandwidth during the driver loop leads to a lower and sparser average memory bandwidth in Eager Writeback than in the Baseline case outside the driver loop.[3]

The overall performance improvement is obviously gained from the shifting of dirty writeback traffic to where this traffic does not impede the return of any critical data. This can be seen in Figure 6.6, which presents an execution profile of the benchmark. In this figure the sequence of vertex data load requests appears on the y-axis, and the cycle upon which the corresponding data item returns is plotted on the x-axis. As execution begins, the profiles of Baseline and Eager Writeback are completely overlapped, because data is returning at the same time for both schemes. Beginning at around 2.6 million cycles, these two curves start to deviate from one another, and continue to diverge as execution time increases. The speedup due to Eager Writeback as measured is around 16%.

By looking carefully at this figure it is possible to distinguish the geometry computation loop from the device driver loop. The segments with shorter but steeper slopes are where the driver loop is executing. The steeper slope occurs because the requested data, $OutV[]$, was returned faster (since the loop read the output vertices generated in the transformation and lighting stages from the L2 cache directly, rather than from memory).

Table 6.5 shows how Eager Writeback affects the performance bottleneck in the Register Update Unit (RUU) of the processor. The layout of this table is similar to Table 6.4, and contains the number of cycles the processor is stalled due to the RUU being full.

As the table shows, Eager Writeback is able to remove a substantial number of stall cycles due to a full RUU and keep the execution pipeline running smoother. These stalls are reduced because in conventional writeback schemes dirty writebacks are competing with demand fetches for available bandwidth, causing delays in data arrival and in the filling of the reservation stations in the RUU. The eager writebacks shift the dirty writes to an earlier time, freeing up the available bandwidth to handle only data reads and reducing the pressure on the RUU.

[2]This was calculated by sampling the data phase on the memory bus every 2000 core clocks, e.g. if 1600 bytes are seen on the bus in 2000 core cycle period, its instantaneous bandwidth is 800MB/sec for a 1GHz processor.

[3]It should be emphasized that the total traffic required by a system using Eager Writeback is not reduced, rather it is re-distributed by the early eviction of dirty cache lines.

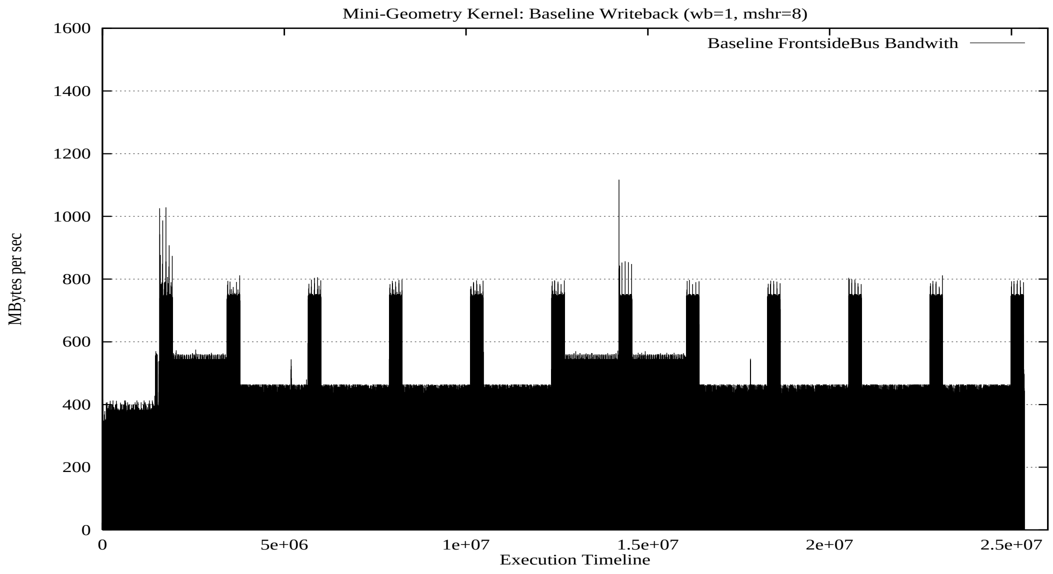


Figure 6.4: Memory Bandwidth Profile by *Baseline Writeback* for 3D Geometry Pipeline (No light)

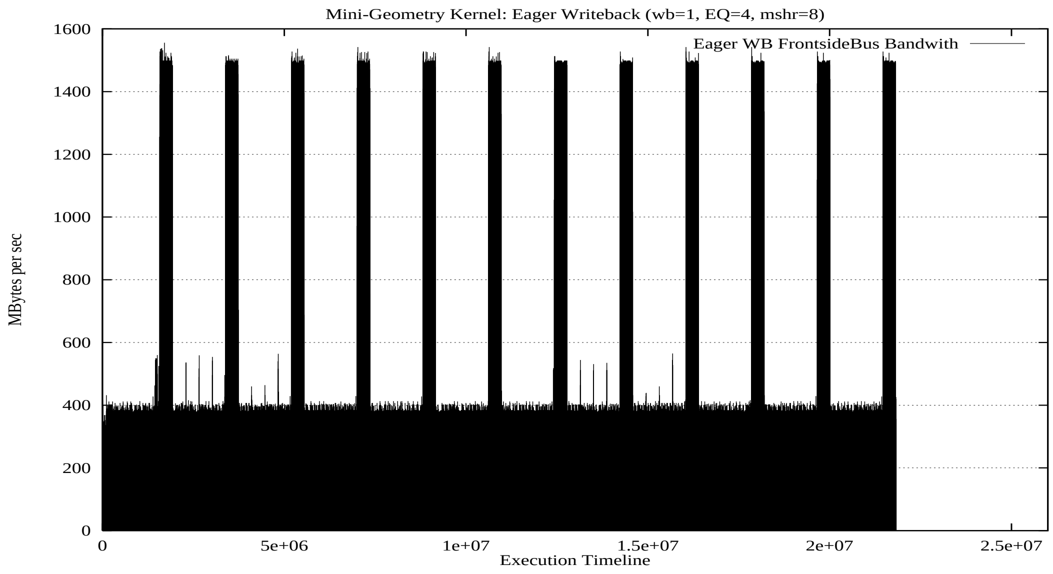


Figure 6.5: Memory Bandwidth Profile by *Eager Writeback* for 3D Geometry Pipeline (No light)

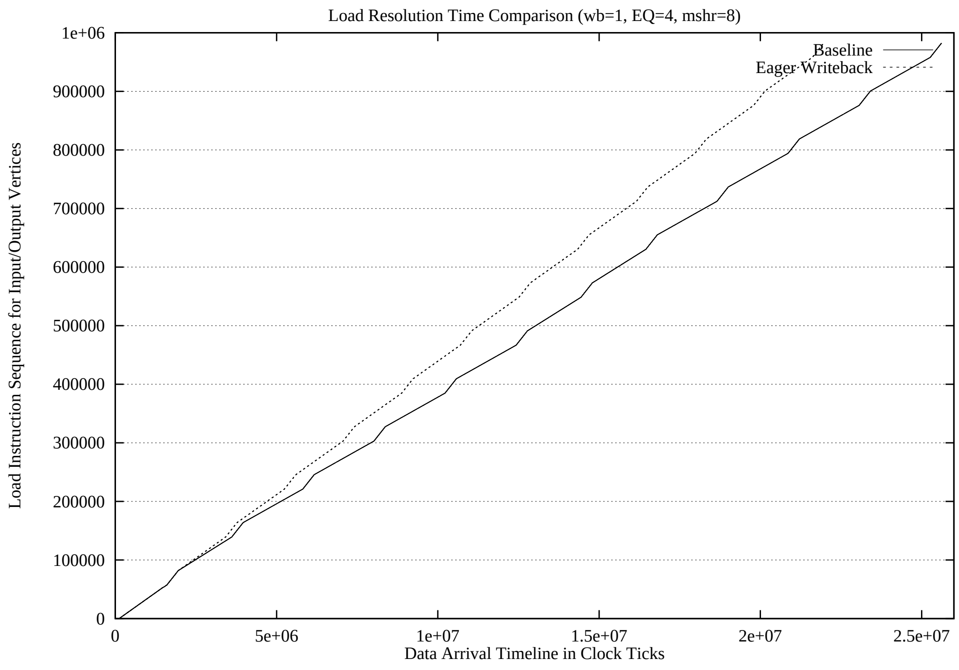


Figure 6.6: Load Response Time for Input Vertex in 3D Geometry Pipeline

bandwidth	*sim cycles*			*RUU Full cycles*		
injection (no light)	Baseline	Eager	speed-up	Baseline	Eager	improved
0 GB/sec	25364637	21838002	1.16	8404023	4452469	47.0%
high 0.4GB/sec	27323771	25434535	1.07	10529817	8448695	19.76%
high 0.8GB/sec	33567580	33775835	0.99	16760998	17024045	-1.6%
high 1.2GB/sec	60699573	59162773	1.03	44206642	42864369	3.0%
low 0.4GB/sec	32539684	28636072	1.14	15604083	11364679	27.2%
low 0.8GB/sec	47365936	42559653	1.11	30356564	25269290	16.8%
low 1.2GB/sec	87400980	83426435	1.05	70248220	66015191	6.0%

Table 6.6: Memory Traffic Injection to 3D Geometry Pipeline. (Eager Queue = 4)

6.4.2.2 With Injected Memory Traffic

In order to evaluate the effectiveness of Eager Writeback in a real system, we implemented a memory traffic injector which we used to model other bus agents requesting the memory bus and consuming memory bandwidth. For this benchmark study, we injected three different external bandwidths onto the bus using two different injection frequencies during the simulations. The external bandwidths chosen were 400MB/sec, 800MB/sec and 1.2GB/sec. For each bandwidth configuration, data was injected at a high frequency and a low frequency. For the high frequency injection, 160, 320, and 480 bytes data were injected every 400 processor clock cycles; for the low frequency injection, 1280, 2560, and 3840 bytes data were injected every 3200 processor clock cycles. Data were injected onto the bus in blocks - for example, in the 800MB high frequency case, every 400 cycles the injector took over the bus and held it until it had completed transferring 320 Bytes of data. The injections are uniformly distributed throughout the simulation.

The results for simulations of the mini-geometry kernel using no light sources are shown in Table 6.6. The top line of the table is the base case with no injected memory traffic, while the other entries are for the different injected bandwidths at the different frequencies. In this table we can see that (as expected) memory traffic injection causes additional stall cycles in the RUU. In addition, as the amount of injected bus traffic increases, the opportunity to do Eager Writeback decreases and the RUU stalls increase dramatically.

	Baseline	*Eager (EQ=0)*		*Eager (EQ=4)*		*Eager (EQ=256)*		*Free Writeback*	
Write Buffer size	cycles	cycles	speedup	cycles	speedup	cycles	speedup	cycles	speedup
WB buf = 1	10230328	9054559	1.130	9053851	1.130	9053851	1.130	9045154	1.131
WB buf = 4	10067331	9052957	1.112	9052957	1.112	9052957	1.112	9045154	1.113

Table 6.7: Simulated cycles of Streaming Kernel.

The table also shows that Eager Writeback provides virtually no speedup when a bandwidth of 0.8GB/sec is injected at the higher frequency, while the same bandwidth injected at a lower frequency allows a speedup of 11%. By examining the dirty writeback bandwidth utilization profile of this scenario (Figure 6.7 and Figure 6.8), one can see that many eager writebacks (i.e. the spikes) are prevented from occurring by the higher frequency injection. The advantages of Eager Writeback are lost and it performs almost on a par with the baseline scenario, due to more frequent bus contention.

6.4.3 Streaming Kernel

The mini-geometry kernel highlighted the problem of implicit dirty writebacks causing loss of performance due to delays in receiving data. Finite memory peak bandwidth is another serious performance issue, which is exposed by the Streaming kernel.

6.4.3.1 Without Injected Memory Traffic

Table 6.7 contains the results of simulation runs of the Streaming kernel, presented in the same format used in Table 6.4. For this benchmark, an eager queue of length 0 (EQ=0) is enough to approximate the optimal case of no dirty writeback traffic at all. Further size increases of the EQ provide only marginal performance gains.

Looking at the memory bandwidth utilization profiles for this kernel (Figure 6.9 and Figure 6.10), we see three spikes that appear repeatedly in both writeback schemes (because the outer loop contains three iterations). The spikes are much wider in the Baseline case, however, indicating the program is spending more execution cycles in these phases. Examining the algorithm, it is clear these spikes are related to the time during the third inner loop where incoming $array_B[]$ data items collide and share memory bandwidth with the induced dirty writebacks of $array_A[]$. Because the finite memory bandwidth (1.6 GB/sec in this study) must be shared between both memory accesses,[4] the rate of demand fetches for $array_B[]$ in the third inner loop is (theoretically) cut in half and thus the overall performance degrades.

Figure 6.9 also shows three bandwidth grooves where memory bus bandwidth has dropped to zero. This corresponds to the second inner loops, where all data references hit in the cache. To take the advantage of this available resource, Eager Writeback fills these bus idle states with early evictions of dirty data cache lines as shown in Figure 6.10. By shifting these bandwidth requests to idle cycles, the memory bandwidth during the course of the third inner loop can be fully dedicated to the demand fetches of $array_B[]$, speeding up the cache fill requests.

As was done for the mini-geometry kernel, we examined how Eager Writeback interacted with internal processor resources when running this benchmark. Table 6.8 shows that the Load/Store Queue is used heavily by this benchmark, and that Eager Writeback can remove more than half of the stalls due to a full Load/Store Queue. As the LSQ is kept less full, instructions are able to leave the IFQ faster and as a result cycles lost due to a full IFQ are reduced substantially.

[4]Read and write turnarounds between demand fetch and dirty writeback streams also prevent peak memory bandwidth from being achieved.

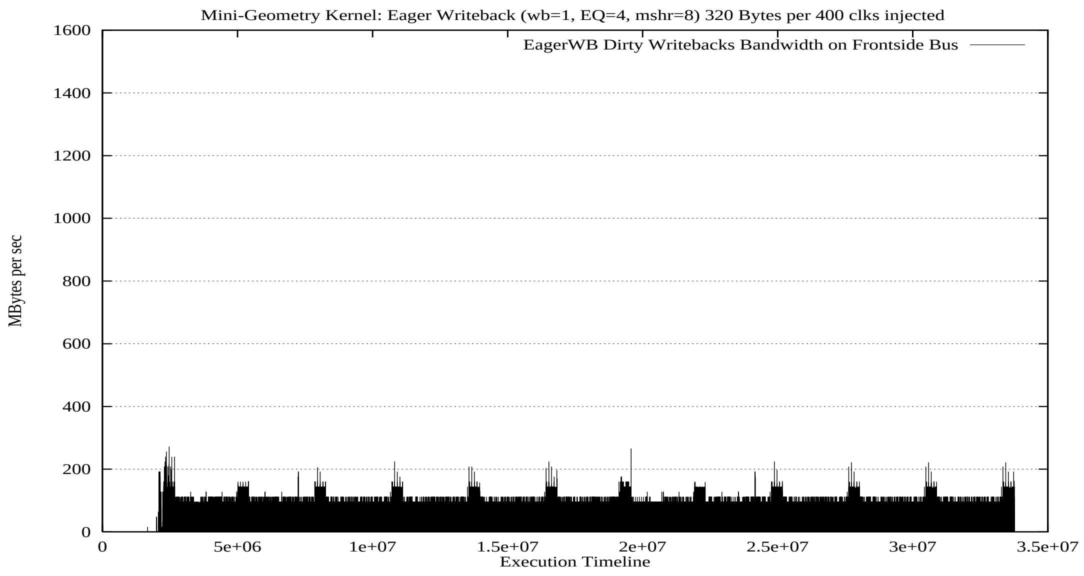


Figure 6.7: Dirty WB L2-to-Mem Bandwidth with 320B/400clks Injection (*Eager*) for Geometry

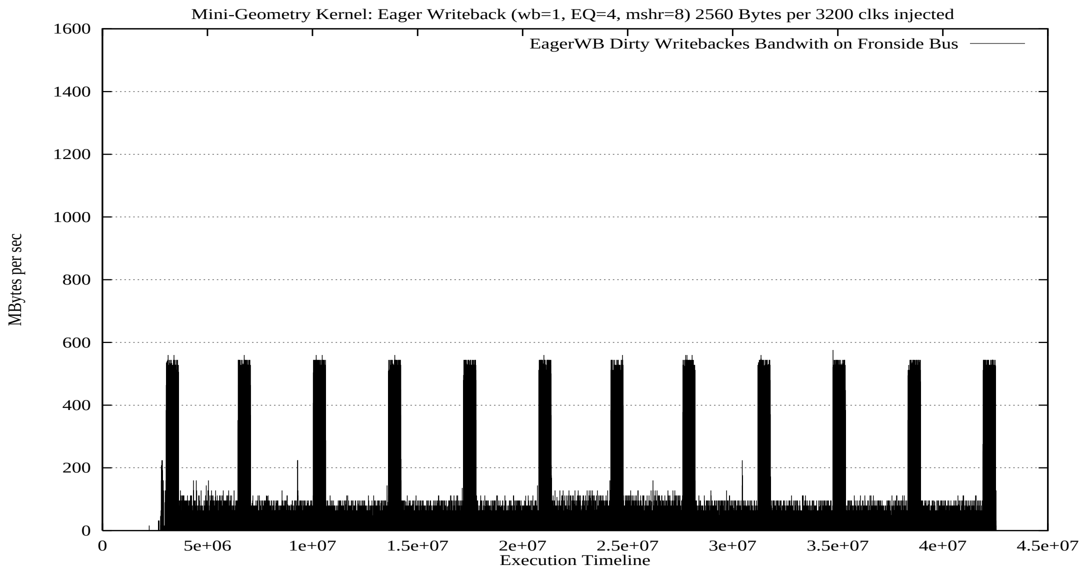


Figure 6.8: Dirty WB L2-to-Mem Bandwidth with 2560B/3200clks Injection (*Eager*) for Geometry

	baseline	*Eager (EQ=0)*		*Eager (EQ=4)*		*Eager (EQ=256)*		*Free Writeback*	
Bottlenecks	cycles	cycles	imprv	cycles	imprv	cycles	imprv	cycles	imprv
IFQ Full	5770175	4594401	20.38%	4594631	20.37%	4594631	20.37%	4587638	20.49%
RUU Full	4274868	4260784	0.33%	4260703	0.33%	4260703	0.33%	4258811	0.38%
LSQ Full	1978596	864867	56.29%	866341	56.21%	866341	56.21%	862880	56.39%

Table 6.8: Resource Constraint Improvement of Streaming Kernel. (Writeback buffer = 1)

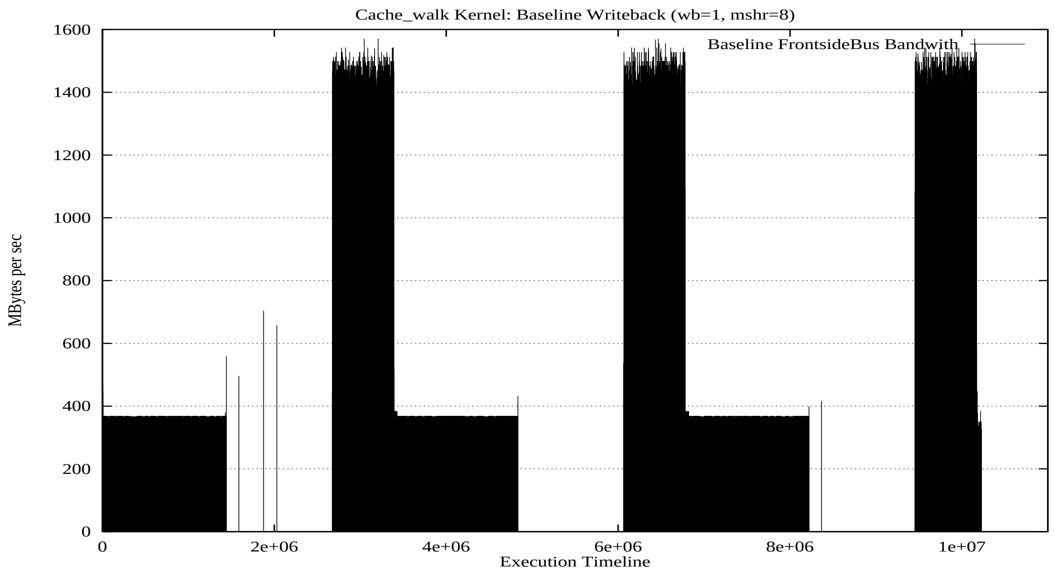


Figure 6.9: Memory Bandwidth Distribution by *Baseline Writeback* for Streaming Kernel

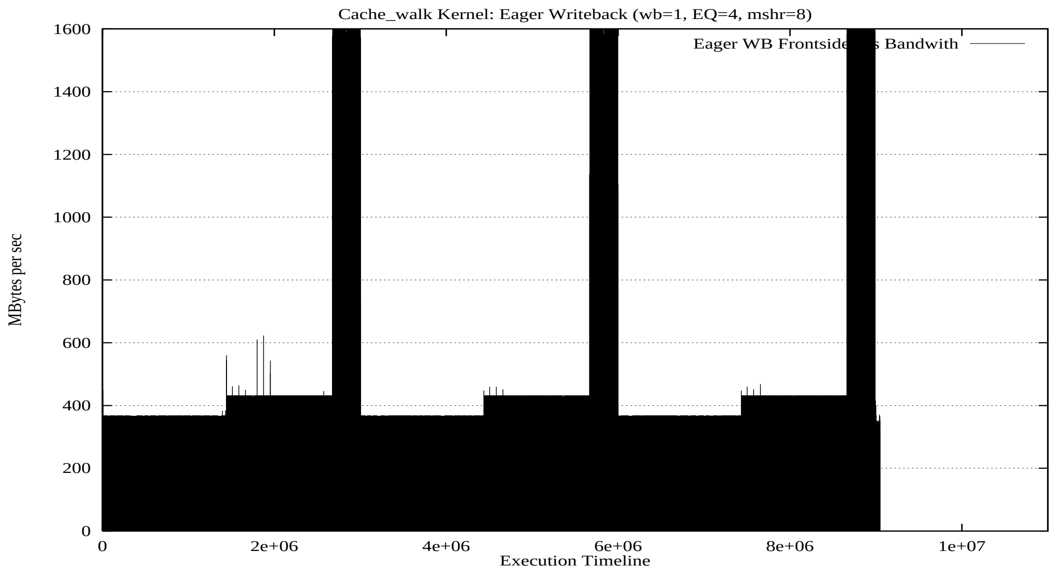


Figure 6.10: Memory Bandwidth Distribution by *Eager Writeback* for Streaming Kernel

bandwidth	*simulated cycles*			*IFQ Full cycles*			*LSQ Full cycles*		
injection	Baseline	Eager	speedup	Baseline	Eager	imprv	Baseline	Eager	imprv
0 MB/sec	10230328	9053851	1.13	5770175	4594631	20.4%	1978596	866341	56.2%
high 0.4GB/sec	11807448	10039848	1.18	7340618	5576536	24.0%	2903145	1205358	58.5%
high 0.8GB/sec	15025957	12389159	1.21	10540877	7908077	25.0%	4428473	1882587	57.5%
high 1.2GB/sec	24250335	21412735	1.13	19717746	16880309	14.4%	8309036	5480188	34.05%
low 0.4GB/sec	12379290	10991058	1.13	7908538	6521201	17.5%	2030932	1417595	30.2%
low 0.8GB/sec	16593748	15115348	1.10	12101456	10622058	12.2%	4264295	2818313	33.9%
low 1.2GB/sec	29048835	27135235	1.07	24495295	22585042	7.8%	8903039	7007451	21.3%

Table 6.9: Memory Traffic Injection to Streaming Kernel. (Eager Queue = 4)

6.4.3.2 With Injected Memory Traffic

We also repeated the experiments involving injecting memory traffic onto the bus for this benchmark program. The results are shown in Table 6.9, and indicate that higher frequency injection seems to have a greater impact on the Baseline case than on the Eager Writeback case. The number of simulated cycles for the Baseline case using high frequency injection increases faster than for the Eager Writeback case, while the increase stays roughly the same for both schemes while injecting lower frequency traffic.

The reason the cycle count climbs faster for the Baseline case than for the Eager Writeback case can be understood by analyzing Figure 6.11. This figure contains an execution profile of the Streaming benchmark and plots the arrival time for each load instruction. From left to right, the four curves represent Eager Writeback with no extra bus injection, Baseline with no extra bus injection, Eager Writeback with higher frequency injection, and Baseline with higher frequency injection. Each curve can be divided into 3 repeated patterns, which bear the following three piecewise line segments: flat (zero increment), steep rise, and slowdown knee. These 3 line segments correspond to the three inner loops in the benchmark.

The first loop contains only data stores, so the load instruction count stays flat as execution time continues. The steep vertical climb corresponds to the second inner loop, which has a high number of cache hits (a large number of loads completing in a short period of time). Finally, the third segment represents the behavior of the third loop, which loads another array that misses in both the L1 and L2 caches.

This third segment, shown as a knee in the curve, is the key to the performance deviation between Baseline and Eager Writeback. Figure 6.12 shows a close-up view of part of Figure 6.11, focusing on the knees of the curves. The slopes ($\tan\theta$) of these knees are the key - the flatter the slope, the longer it will take to complete. Comparing the slope changes between Baseline and Eager Writeback, it is obvious that the slope of the Baseline segment is much shallower than that of the Eager Writeback segment. This means that for the same number of loads in the third loop, the execution time of the Baseline case was more sensitive to and severely delayed by other transactions, which in this case are composed of the dirty writebacks induced by the loads and the periodic injection of memory traffic. For the Eager Writeback case, the dirty writebacks were mostly completed in the second loop, so the slope of the knee is steeper and the third loop can be completed more swiftly than its Baseline counterpart.

Repeating the same experiment using lower frequency injection (as plotted in Figure 6.13 and Figure 6.14) reveals that the slope of the knees of the curve are much more similar to one another. As a result, roughly the same number of penalty cycles were added to both Baseline and Eager Writeback, and the speedups due to Eager Writebacks are smaller in Table 6.9. These results suggest higher frequency memory interference can degrade baseline case performance more in a bandwidth-limited code.

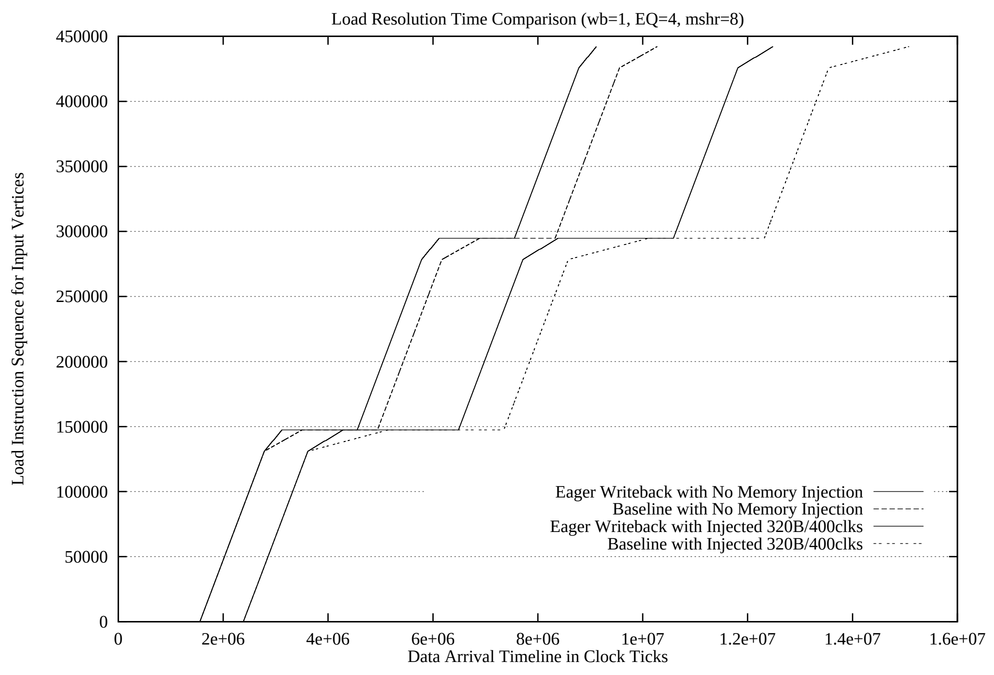


Figure 6.11: Load Response Time for Data Reads in Streaming Kernel (Higher Frequency Injection)

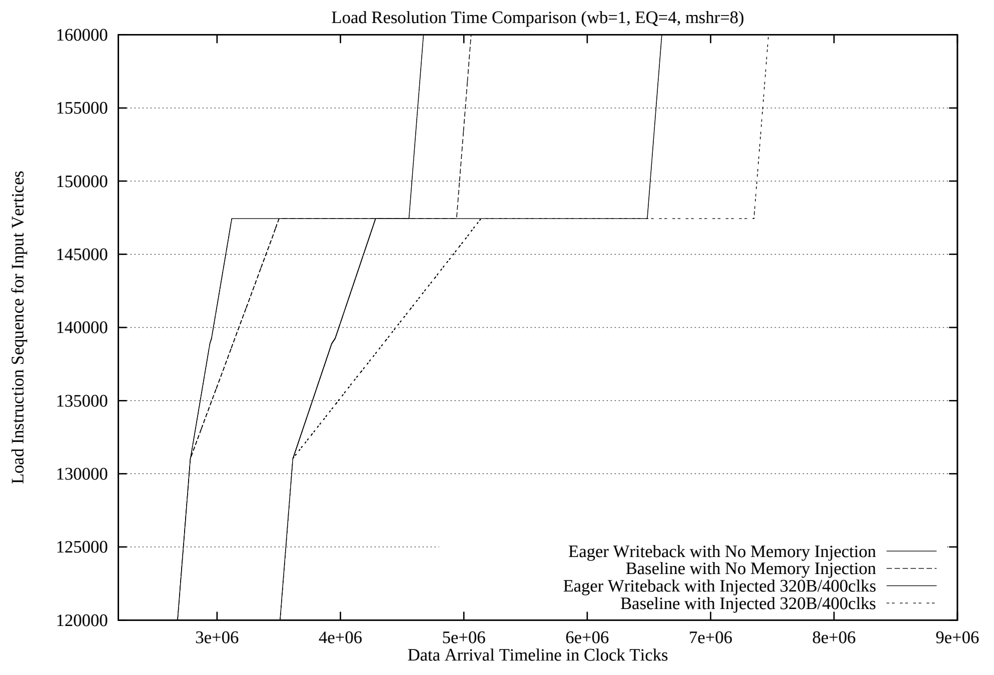


Figure 6.12: Zoom-In of the Load Response Time for Data Reads in Streaming Kernel (Higher Frequency Injection)

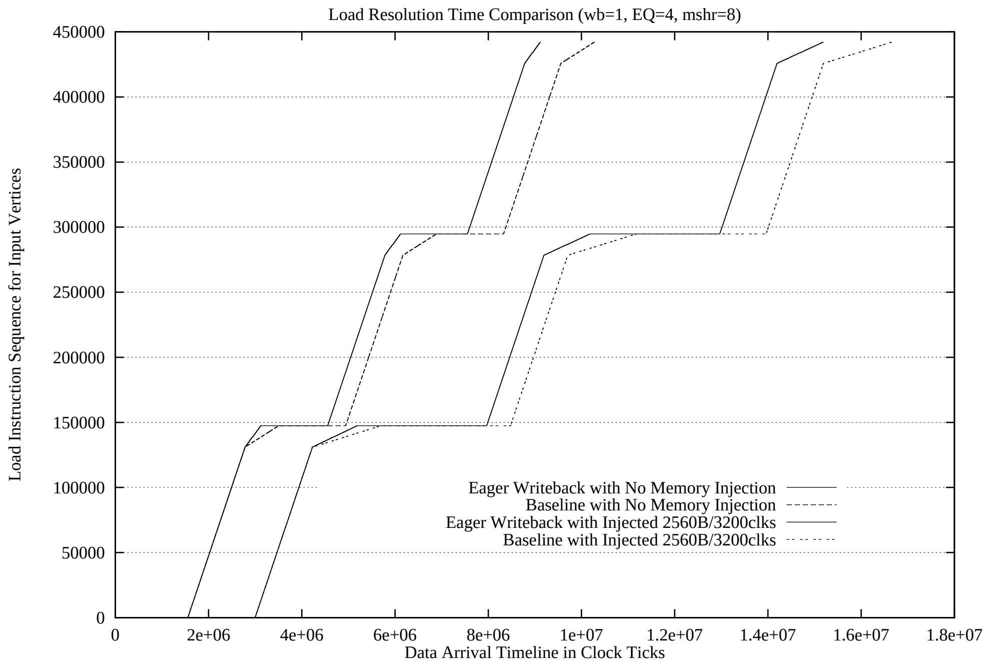


Figure 6.13: Load Response Time for Data Reads in Streaming Kernel (Lower Frequency Injection)

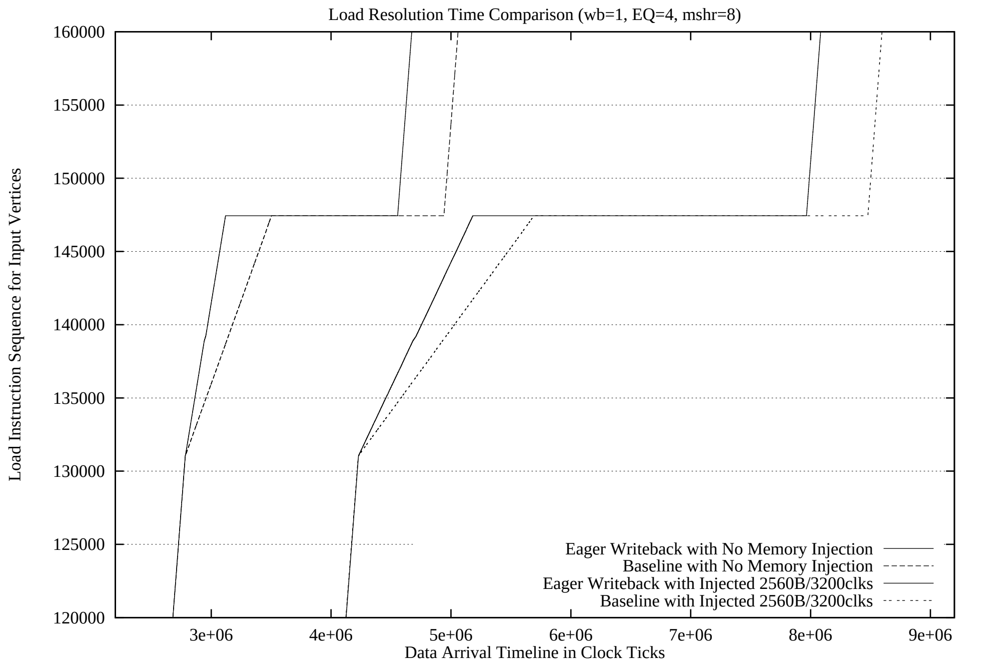


Figure 6.14: Zoom-In of the Load Response Time for Data Reads in Streaming Kernel (Lower Frequency Injection)

6.5 Related Work

Traditionally, the bandwidth issues are often alleviated by providing a higher bandwidth memory system. Today's memory subsystem not only needs to deliver data bandwidth requests by the processors in a timely fashion, but also needs to provide enough bandwidth for a graphics processor as discussed in Section 6.1. Memory bandwidth can be increased by increasing bus frequency, increasing the bus width, or installing multiple indepedent memory agents to handle memory requests. Contemporary memory technologies such as Double Data Rate (DDR) DRAM, SLDRAM [57] or Direct Rambus DRAM [35] provide some solutions to enable a high bandwidth system. For example, a single channel 266MHz DDR-DRAM with 8-byte data bus can provde a maximum 2.1GB/sec bandwidth from main memory.

The Advanced Graphics Port (AGP) [29] initiated for multimedia PC platforms provides a new interface to enhance the data bandwidth delivered to the graphics accelerators, primarily useful for content-rich 3D graphics applications. It relieves the congestion of PCI bus by adding a dedicated high-speed bus between the graphics accelerator and chipset. The latest AGP8x specification is aimed at providing a peak bandidth of 2.1GB/sec.

Self-invalidation techniques are used to eliminate the latency of invalidation and acknowledge messages for a directory-based multiprocessor system. A processor can invalidate its own cache copy before other processors make a conflicting access. The principle of self-invalidation is similar to our eager writeback approach. Lebeck and Wood [71] proposed a hardware approach to perform dynamic self-invalidation. Based on the sharing pattern history, a directory controller can predict a cache line for self-invalidation and send the message to the corresponding processor node. Lai and Falsafi [68] proposed Last-Touch Predictors that identify cache lines in a shared memory system that can be speculatively self-invalidated.

6.6 Chapter Summary

Systems employing writeback caches have to contend with the following two issues: (1) Dirty writebacks contend with demand fetches for bandwidth and can impede the delivery of data, and (2) Finite memory bandwidth shared between demand fetches and implicit dirty writebacks limit the performance of memory bound programs. These performance issues are important to a large and growing class of programs – those that consume large amounts of memory bandwidth and generate many data stores.

In this chapter we have presented Eager Writeback, a new technique for dealing with these issues which can effectively improve overall system performance by filling or shifting the writing of dirty cache lines from on-demand to times when the memory bus is not fully utilized or in idle state. This time-shifting is accomplished by identifying and speculatively writing ("cleaning") dirty lines whenever the bus is free. We have shown that for a wide variety of programs, once a dirty cache line has entered the LRU state it is rarely written to again. We use this fact to identify the lines that should be speculatively written (although this information could be of interest to many other intelligent cache management techniques as well).

We have shown that applying this technique can alleviate bandwidth congestion and improve performance for two benchmarks that are representative of graphics (data intensive) or streaming types of applications. We have shown that when conventional writebacks compete with memory loads and defer the delivery of data, the Eager Writeback technique is able to remove the competition by evicting dirty data earlier. We have also shown that when the available memory bandwidth limits overall performance, Eager Writeback can alleviate this situation by utilizing earlier idle bus cycles. Eager Writeback can be implemented in a number of ways - for example, as an additional programmable memory type on top of the existing memory types provided by a processor to speed up bandwidth-hungry applications, e.g. 3D graphics or content-rich applications.

Further investigation of this Eager Writeback mechanism will include the effects this approach has on other system performance issues. For example, Eager writeback can potentially reduce context switching time overhead by flushing dirty lines in advance of the context switch. In addition, Eager Writeback can push modified data closer to the globally observable memory level earlier to reduce coherence miss latency, and as a result, respond to other processors' requests faster. The same analysis performed in this chapter can similarly be applied to write-update and write-invalidate protocols in a shared memory system to reduce their coherence traffic.

CHAPTER 7

CONCLUSIONS AND FUTURE DIRECTIONS

7.1 Thesis Summary

In this dissertation we focus on data cache architecture optimizations with respect to three major issues in modern processor designs: energy efficiency, performance and memory bandwidth utilization. We analyzed data memory reference streams for a variety of applications and found several ways to exploit their reference characteristics. By categorizing these characteristics, we proposed three novel techniques to address these design issues. The following sections summarize our contributions in this dissertation.

7.1.1 Memory Reference Characterization

First, we characterized data memory reference streams based on virtual memory space partitioning by high-level programming languages. The partitioning segregates memory accesses into semantically distinct regions including stack, global static, heap and read-only data. We also exploited the information content inherent in cache line frame addresses. Based on information theory, we found that the entropy of stack data references is roughly one third of the entropy of heap data references, about half of that of global static data references. In addition, our analysis indicates that processor architects seemingly design ever larger caches in an attempt to accommodate heap data, although it exhibits the most unpredictable and intractable reference behavior.

Then we analyzed cache line write behavior and found that dirty cache lines, once they leave the MRU state, are seldom written again before they are replaced in the caches. This interesting property is utilized by our new Eager Writeback policy which is capable of re-distributing memory traffic, and thereby balances memory bandwidth.

7.1.2 Region-based Cachelets

Using the characteristics of memory references that we identified, we proposed to direct distinct memory reference streams into distinct cache structures, which we called *Region-Based Cachelets.* We allocate an exclusive cache for each semantic region. More than 70% of the memory references in heap and global static regions can be redirected into their individual much smaller cachelet structures, thereby reducing dynamic power dissipation. Due to the high reference locality of stack and global static data, the region-based cachelets design does not degrade performance relative to a conventional cache design, and in some cases it even improves performance by eliminating conflict misses among memory regions. This new partitioned memory organization is suitable for embedded processors in which processor architects can tweak the most energy-delay efficient cache configurations for their target applications.

We also quantitatively investigated the synergistic benefits of combining the region-based cachelets design with a prior proposed filter cache design. We found that the power dissipation can be further reduced by as much as 40% with a minimum impact (less than 8%) on the overall performance.

7.1.3 Stack Value File Design

An in-depth analysis of stack reference characteristics and its stack pointer-relative addressing mode was performed. The stack memory references through this simple addressing mode are identified in the pre-decode stage and fed into an extended unit which computes the offset relative to the top of stack (TOS). Then these memory instructions are morphed into register to register move operations inside the out-of-order execution core. This reduces the load-to-use latency associated with each memory instruction. Instead of committing the computed results into memory, the processor commits them into the *Stack Value File* (SVF), an architectural register file transparent to compilers and programmers. In theory, all the stack addresses can be accessed relative to the TOS pointer through the compiler's help. We found that 86% of the stack references in Compaq Alpha binaries belong to this instruction class.

In this study, we identified several characteristics of stack references that differ from other data references. Typically, stack references exhibit *contiguity* which enables us to eliminate the need for storing every address tag. Therefore, these data can be allocated into a register file like structure rather than a cache for fast accesses. The initial references are store operations (except for prefetch instructions), thus we can save memory traffic by writing directly into the SVF without accessing the cache data. When the activation records are deallocated, dirty data can be discarded as they are no longer valid, thus saving dirty writeback traffic. The morphing scheme also implicitly enables memory renaming for stack references via the existing register renaming circuits. The overall performance can be improved by an average of 24% for conventional microarchitectures with a dual-ported L1 cache, while providing the added benefit of reducing memory overhead traffic.

7.1.4 Eager Writeback Cache

According to the characteristics of cache line writes, a new cache write policy called *Eager Writeback* is presented. Eager Writeback, a modified mechanism to maintain memory consistency whose behavior falls somewhere between a writeback cache and a write-through cache, is able to improve overall system performance by speculatively shifting the writing of dirty cache lines from on-demand to times when the memory bus is less congested. The implementation of Eager Writeback is inexpensive. Basically we trigger dirty writeback upon the point when a dirty cache line enters the Least Recently Used (LRU) state. Once the writeback of the LRU dirty line is acknowledged, the dirty bit of this corresponding line is cleaned.

This new technique avoids conflicts between writeback traffic and demand fetch traffic, thereby reducing the potential delay of demanded data due to the bandwidth limitation. The Eager Writeback is shown to be very effective for 3D geometry pipelines and memory streaming applications. These applications possess a property which brings in a large working set (introduces demand misses) and pollutes these data working set (generates dirty data writebacks). Dirty writebacks usually have a higher priority for accessing the memory bus and can conflict with critical demand misses, thus degrading system performance. The Eager Writeback mechanism, however was shown to be capable of balancing memory bandwidth and alleviating this performance degradation. Eager Writeback can be implemented as an additional programmable memory type on top of the existing memory types provided by the processors.

7.2 Future Directions

7.2.1 Cache Port Design

The concept of Region-based Caching can have more benefits yet to be explored. In this dissertation, we have studied its energy-delay efficiency for in-order single issue embedded processors used in mobile PDA-like devices. It can also be beneficial for high performance superscalar or multi-issue VLIW processors. For these processors, multiple memory operations can be issued simultaneously. The Region-based Cachelets provide an alternative solution for multi-porting the cache structure. In our proposal, a Region-based Cachelets system itself is partitioned into three discrete caches, one for each semantic region. Additionally, multi-porting a monolithic large cache can be expensive. It quadruples the cache die size when the cache size is doubled because the wordlines and bitlines of a cache need to be doubled in order to enable two simultaneous access requests. One can study the combination of memory instructions based on access regions for a multi-ported cache design. Moreover, it can be less costly in terms of die size area to build multi-ported region-based cachelets than a multi-ported monolithic cache.

7.2.2 Energy Consumption in the Stack Value File

We demonstrated the effectiveness of the Stack Value File in increasing memory level parallelism and system performance. One interesting potential gain we did not explore is the energy saving. As we mentioned earlier, the SVF only needs to keep the current TOS base address register instead of a full blown tag array for each SVF entry. Compared to a conventional cache design, the SVF can further reduce power consumption whenever a stack memory reference occurs. To address and quantify the energy saving of an SVF implementation will make the SVF design more appealing.

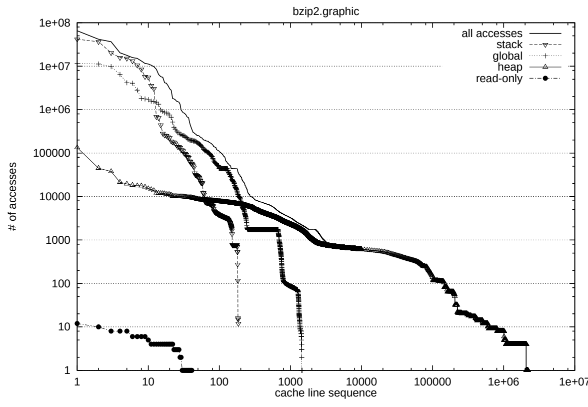
7.2.3 Heap Object Management

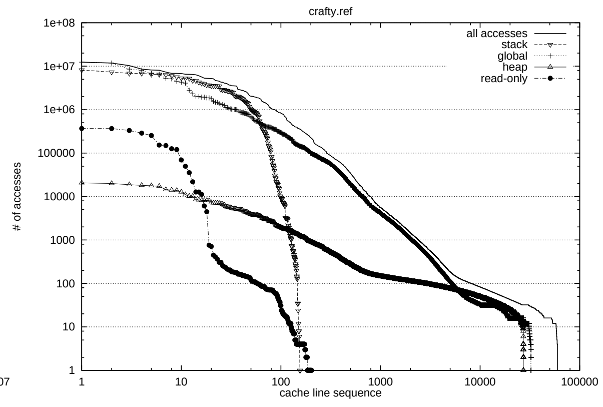
In the characteristics analysis of memory reference, heap data references show the maximum entropy among the memory regions. It is also a known fact that dynamic storage allocation is responsible for poor memory performance. As object-oriented languages, such as C++, Java, or Microsoft's C#, become more popular in application software development, heap objects are allocated much more frequently [21], and effective heap data management becomes imperative for performance improvement. Prior studies [4][5][10] [42] showed heap-allocated data are likely to die young. This property is used as the foundation of the generational garbage collector. Recent compiler research is emphasizing the cache locality for the garbage collection algorithms [116]. Through the synergy of compiler and microarchitecture cooperation, one could devise a more effective heap data management and replacement policy to improve cache locality and efficiency.

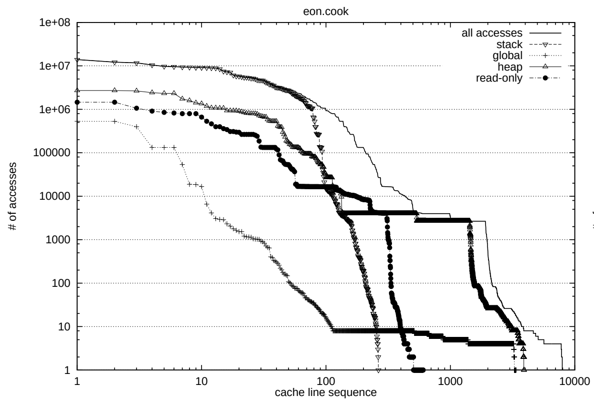
APPENDICES

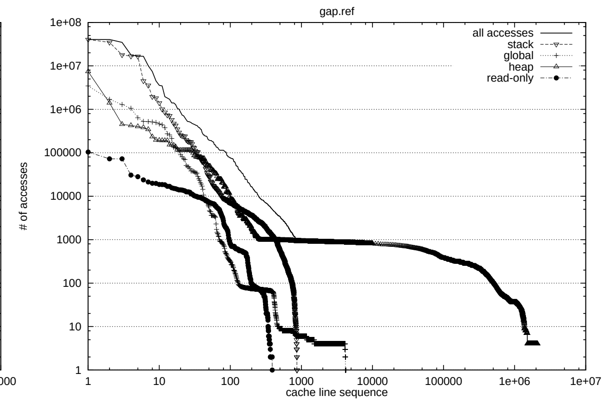
APPENDIX A

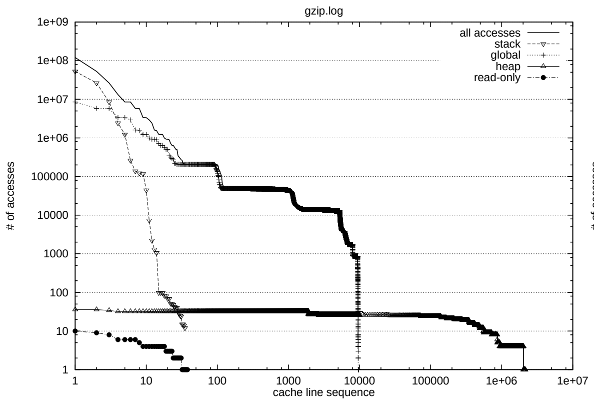
Region Reference Frequency of SPEC2000int

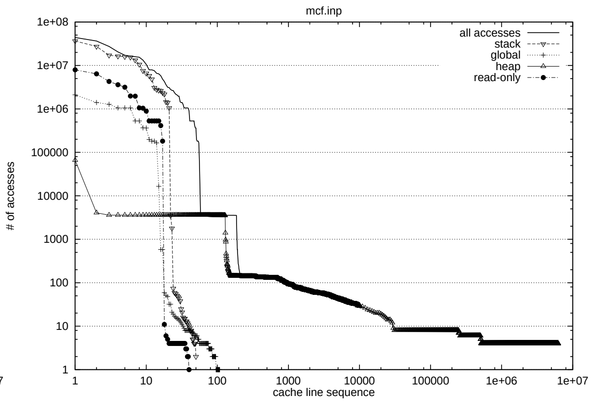


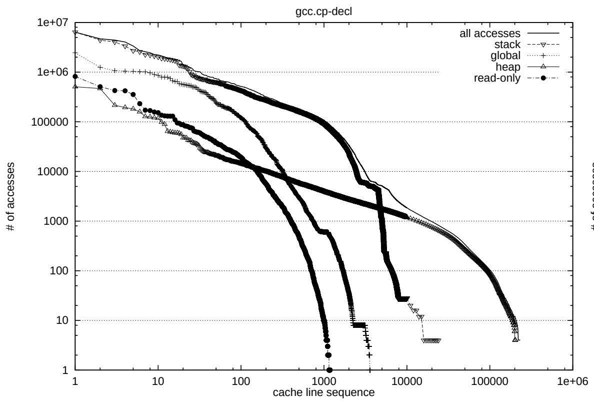


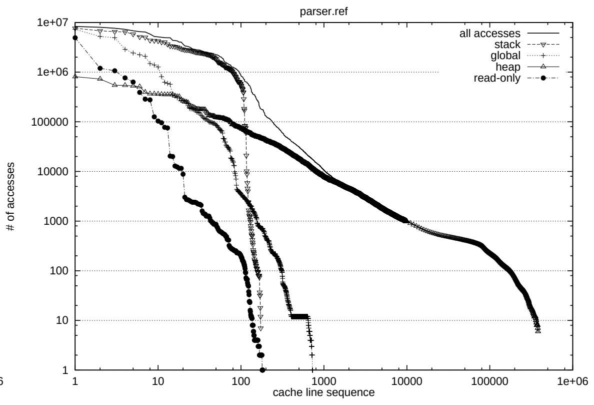


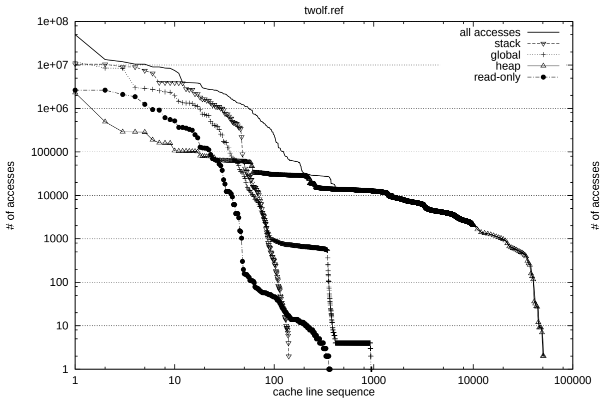


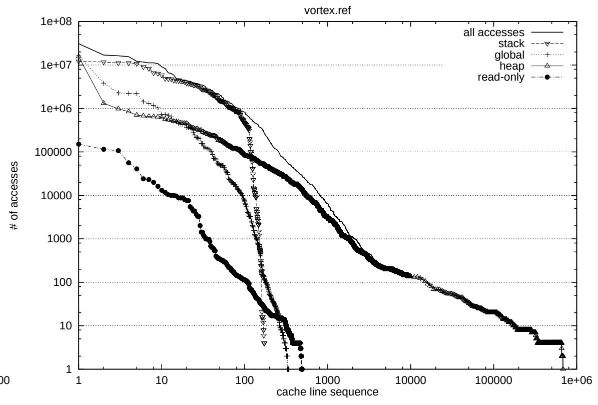

gzip.log
all accesses
stack
global
heap
read-only
# of accesses
cache line sequence

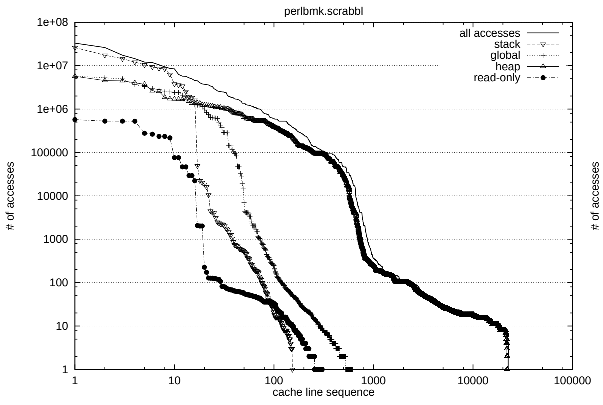

mcf.inp
all accesses
stack
global
heap
read-only
# of accesses
cache line sequence

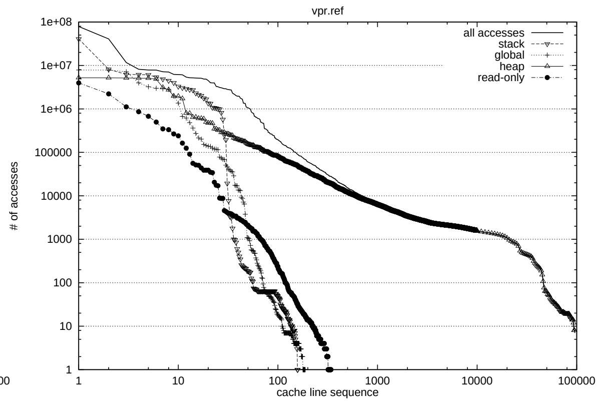

gcc.cp-decl
all accesses
stack
global
heap
read-only
# of accesses
cache line sequence


parser.ref
all accesses
stack
global
heap
read-only
# of accesses
cache line sequence


twolf.ref
all accesses
stack
global
heap
read-only
# of accesses
cache line sequence

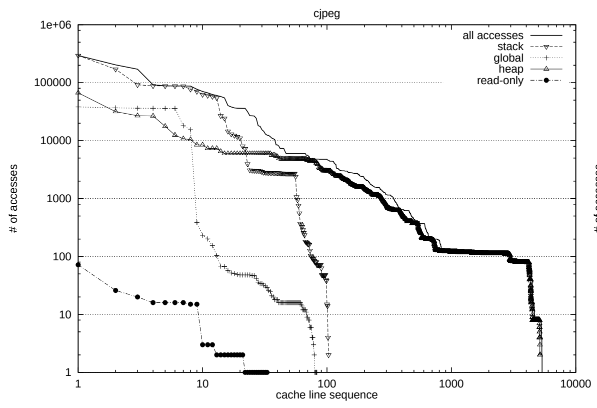

vortex.ref
all accesses
stack
global
heap
read-only
# of accesses
cache line sequence

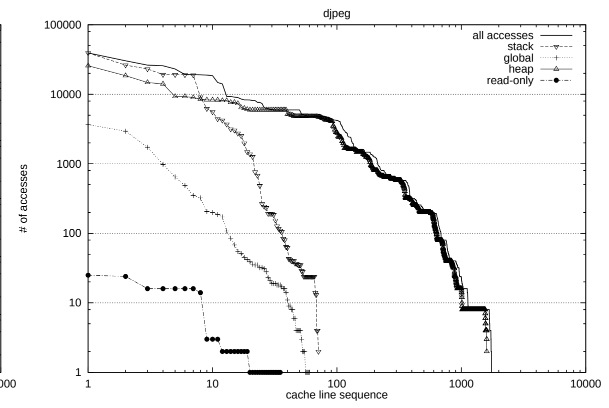

perlbmk.scrabbl
all accesses
stack
global
heap
read-only
# of accesses
cache line sequence

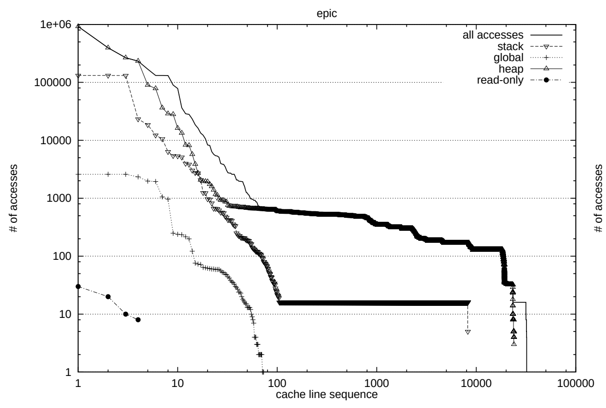

vpr.ref
all accesses
stack
global
heap
read-only
# of accesses
cache line sequence

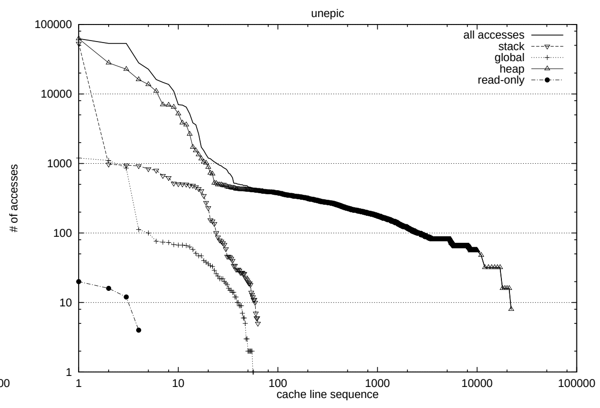
APPENDIX B

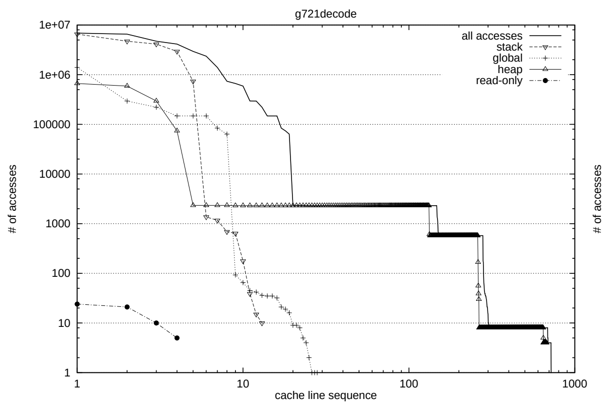
Region Reference Frequency of Mediabench

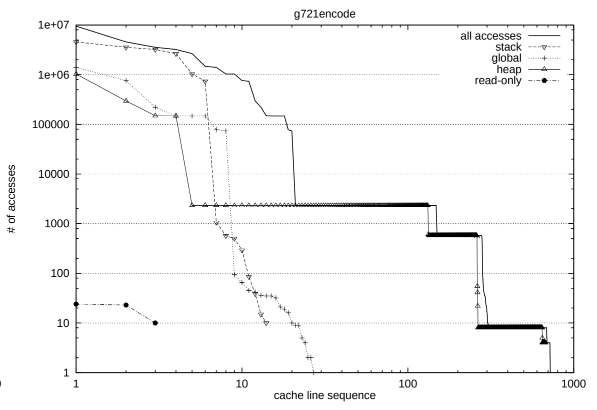


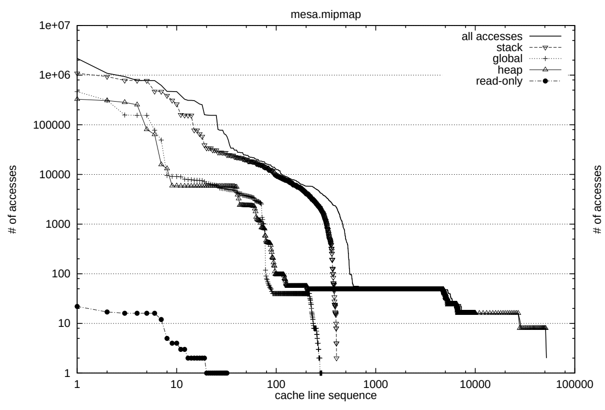


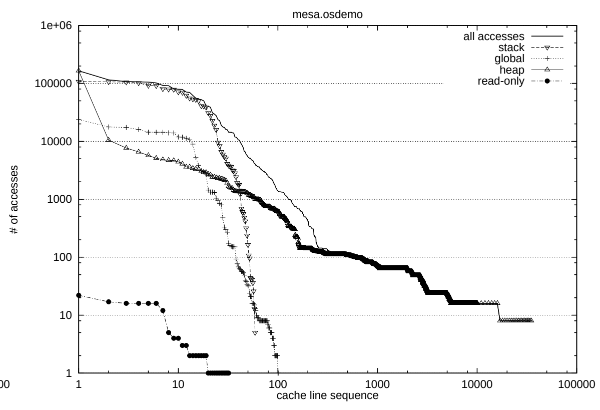


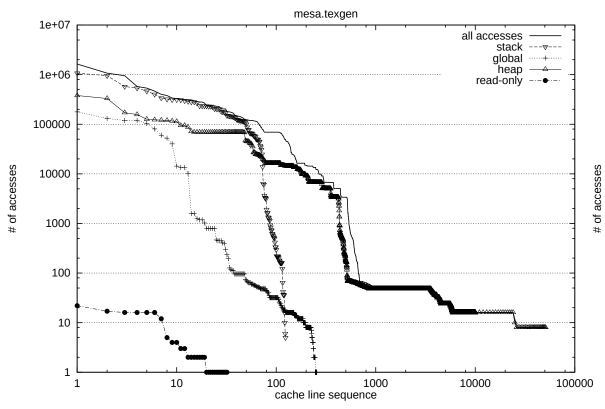


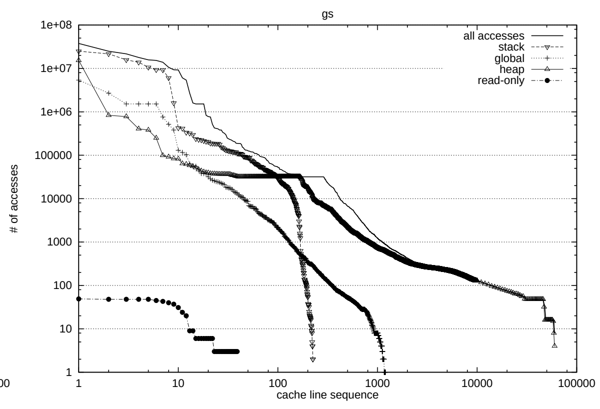

g721decode
all accesses
stack
global
heap
read-only
# of accesses
cache line sequence

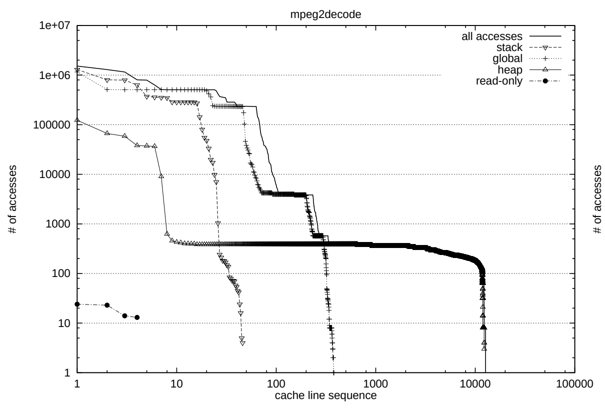

g721encode
all accesses
stack
global
heap
read-only
# of accesses
cache line sequence

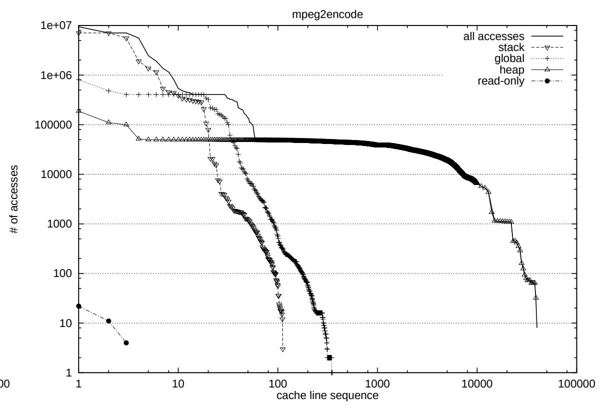

mesa.mipmap
all accesses
stack
global
heap
read-only
# of accesses
cache line sequence

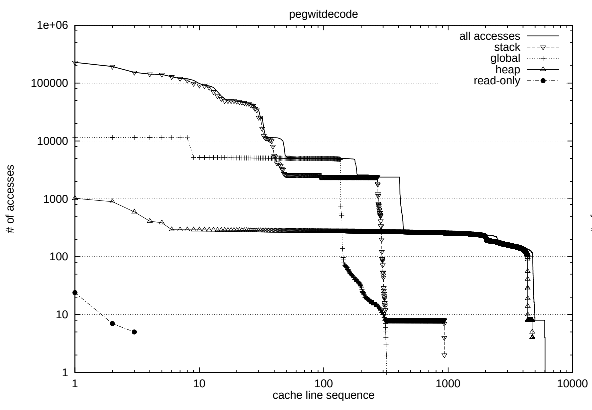

mesa.osdemo
all accesses
stack
global
heap
read-only
# of accesses
cache line sequence

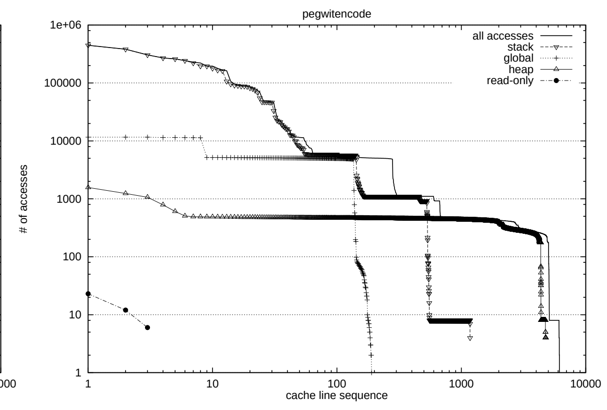

mesa.texgen
all accesses
stack
global
heap
read-only
# of accesses
cache line sequence

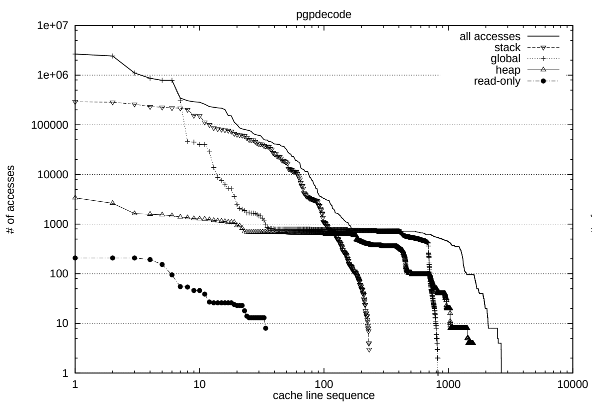

gs
all accesses
stack
global
heap
read-only
# of accesses
cache line sequence

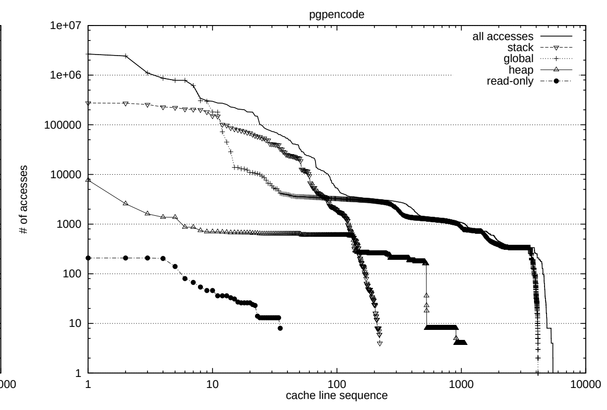

mpeg2decode
all accesses
stack
global
heap
read-only
# of accesses
cache line sequence

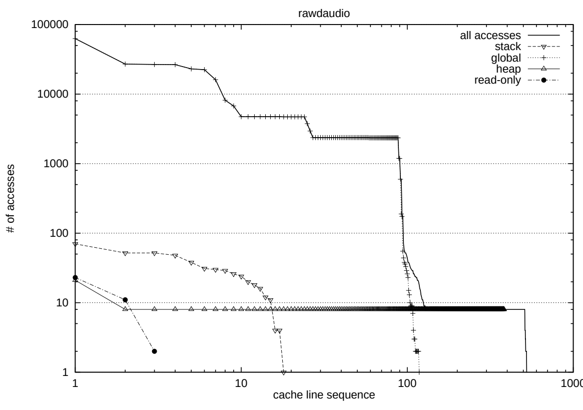

mpeg2encode
all accesses
stack
global
heap
read-only
# of accesses
cache line sequence

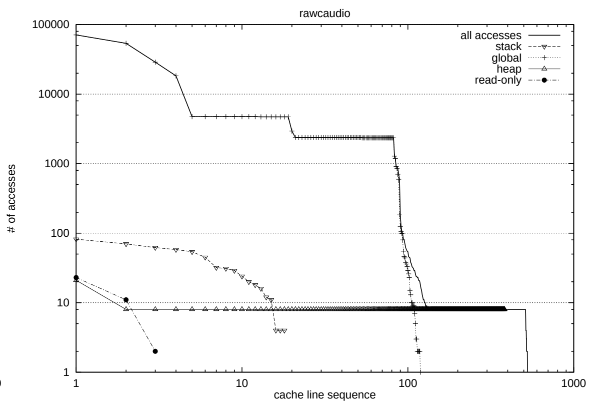

pegwitdecode
all accesses
stack
global
heap
read-only
# of accesses
cache line sequence

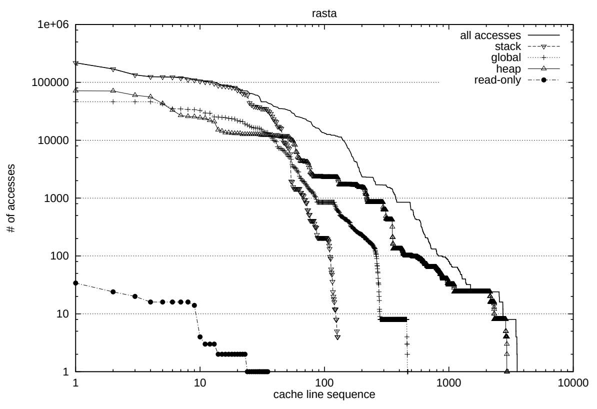

pegwitencode
all accesses
stack
global
heap
read-only
# of accesses
cache line sequence


pgpdecode
all accesses
stack
global
heap
read-only
# of accesses
cache line sequence


pgpencode
all accesses
stack
global
heap
read-only
# of accesses
cache line sequence

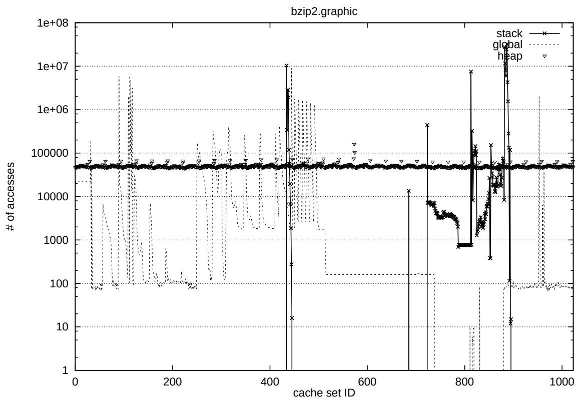

rawdaudio
all accesses
stack
global
heap
read-only
# of accesses
cache line sequence

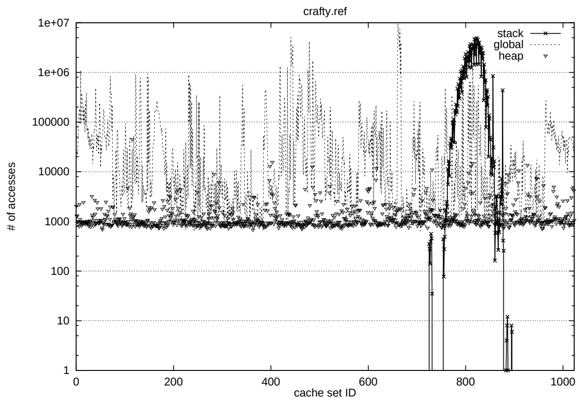

rawcaudio
all accesses
stack
global
heap
read-only
# of accesses
cache line sequence

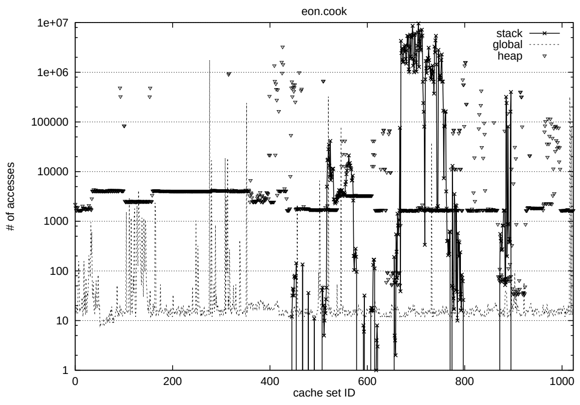

rasta
all accesses
stack
global
heap
read-only
# of accesses
cache line sequence

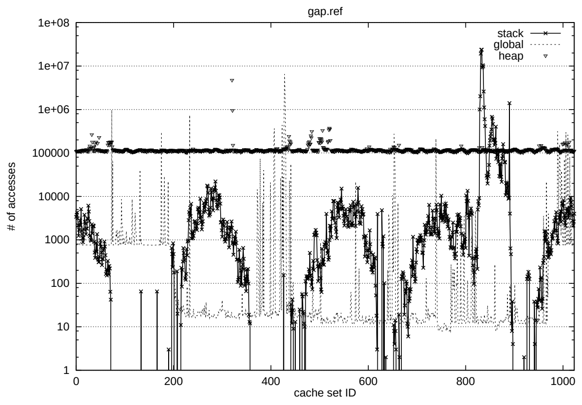
APPENDIX C

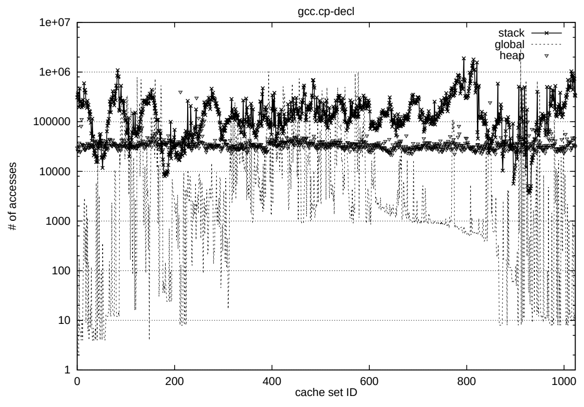
Footprint of Data Cache by Regions

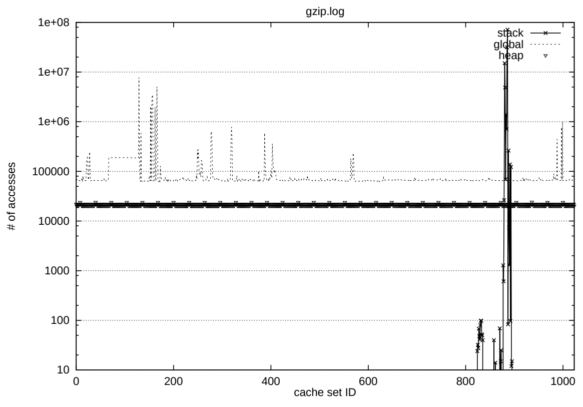


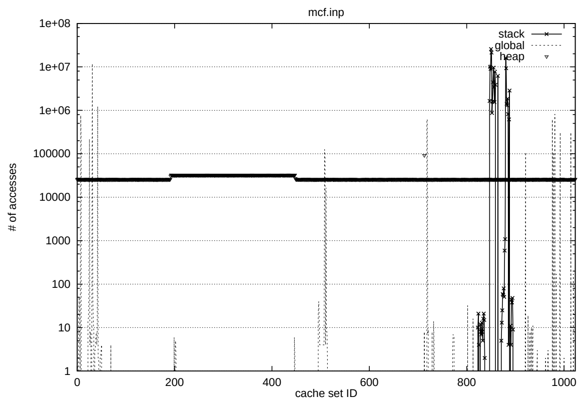


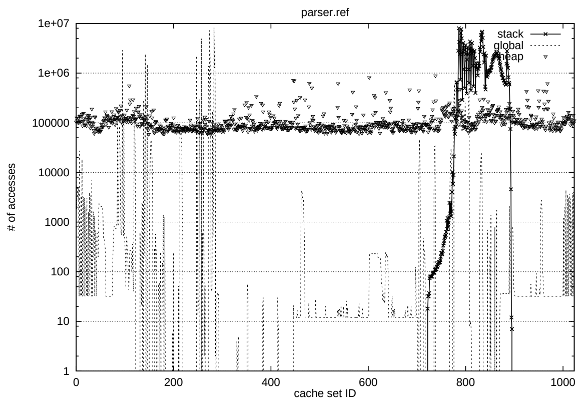


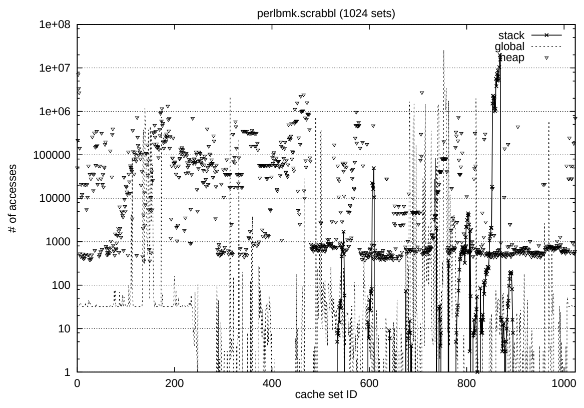


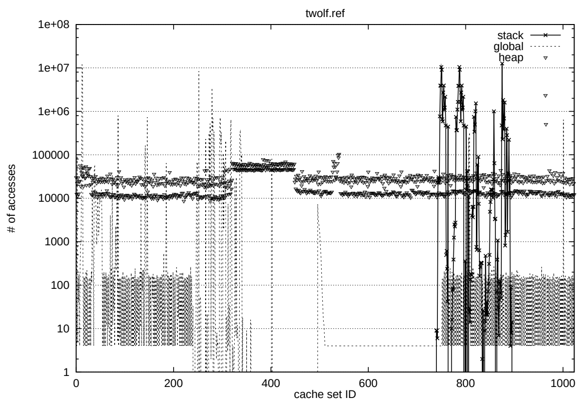

gcc.cp-decl
stack
global
heap
# of accesses
cache set ID


gzip.log
stack
global
heap
# of accesses
cache set ID


mcf.inp
stack
global
heap
# of accesses
cache set ID


parser.ref
stack
global
heap
# of accesses
cache set ID


perlbmk.scrabbl (1024 sets)
stack
global
heap
# of accesses
cache set ID


twolf.ref
stack
global
heap
# of accesses
cache set ID

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Norman Abramson. *Information Theory and Coding.* McGraw-Hill Book Company, 1963.

[2] Alfred V. Aho, Ravi Sethi, and Jeffery D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1988.

[3] David H. Albonesi. Selective Cache Ways: On-Demand Cache Resource Allocation. In *Proceedings of the 32nd International Symposium on Microarchitecture*, 1999.

[4] Andrew W. Appel. Simple Generational Garbage Collection and Fast Allocation. *Software — Practice and Experience*, Vol. 19 No. 2, Feburary 1989.

[5] Andrew W. Appel. *Modern Compiler Implementation in Java.* Cambridge University Press, 1998.

[6] P.V. Argade, S. Aymeloglu, A.D. Berenbaum, M.V. DePaolis, R.T. Franzo, R.D. Freeman, D.A. Inglis, G. Komoriya, H. Lee, T.R. Little, G.A. MacDonald, H.R. McLellan, E.C. Morgan, H.Q. Pham, and G.D. Ronkin. Hobbit: A High-Performance, Low-Power Microprocessor. In *Proceedings of COMPCON Spring*, 1993.

[7] Russel R. Atkinson and Edward M. McCreight. The Dragon Processor. In *Proceedings of the 2nd International Symposium on Architectural Support for Programming Languages and Operating Systems*, 1987.

[8] Todd M. Austin, Dionisios M. Pnevmatikatos, and Guri S. Sohi. Streamlining Data Cache Access with Fast Address Calculation. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995.

[9] Todd M. Austin and Guri S. Sohi. Zero-cycle Loads: Microarchitecture Support for Reducing Load Latency. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, 1995.

[10] David A. Barrett and Benjamin G. Zorn. Using Lifetime Predictors to Improve Memory Allocation Performance. In *Proceedings of the 6th Conference on Programming Language Design and Implementation*, 1993.

[11] Roland Bechade and et al. A 32b 66MHz 1.8W Microprocessor. In *Proceedings of the International Solid-State Circuits Conference*, 1994.

[12] Michael Bekerman, Adi Yoaz, Freddy Gabbay, Stephan Jourdan, Maxim Kalaev, and Ronny Ronen. Early Load Address Resolution Via Register Tracking. *Proceedings of the 27th International Symposium on Computer Architecture*, June 2000.

[13] Nikolaos Bellas, Ibrahim Hajj, and Constantine Polychronopoulos. Using Dynamic Cache Management Techniques to Reduce Energy in a High-Performance Processor. In *Proceedings of the 1999 International Symposium on Low Power Electronics and Design*, 1999.

[14] A. D. Berenbaum, D. R. Ditzel, and H. R. McLellan. An Introduction to the CRISP Architecture. In *Proceedings of the Spring COMPCON*, 1987.

[15] 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, October 1987.

[16] Dileep P. Bhandarkar. *Alpha Implementations and Architecture: Complete Reference and Guide.* Digital Press, 1996.

[17] Russell P. Blake. Exploring a Stack Architecture. *IEEE Computer*, May 1977.

[18] 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*, June 2000.

[19] Doug C. Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report 1342, Computer Science Department, University of Wisconsin-Madison, 1997.

[20] Michael Butler, Tse-Yu Yeh, Yale Patt, Mitch Alsup, Hunter Scales, and Michael Shebanow. Single Instruction Stream Parallelism is Greater than Two. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, 1991.

[21] Brad Calder, Dirk Grunwald, and Benjamin Zorn. Quantifying Behavioral Differences between C and C++ Programs. *Journal of Programming Languages*, Vol. 2, No. 4, 1993.

[22] Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. Cache-Conscious Data Placement. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

[23] David Callahan, Ken Kennedy, and Allan Porterfield. Software Prefetching. *Proceedings of 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[24] Anantha P. Chandrakasan and Robert W. Brodersen. Minimizing Power Consumption in Digital CMOS Circuits. *Proceedings of the IEEE*, Vol. 83 No. 4, April 1995.

[25] Po-Yung Chang, Eric Hao, and Yale Patt. Target Prediction for Indirect Jumps. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[26] Tien-Fu Chen and Jean-Loup Baer. Effective Hardware-Based Data Prefetching for High-Performance Processors. *IEEE Transactions on Computers*, May 1995.

[27] Sangyeun Cho, Pen-Chung Yew, and Gyungho Lee. Access Region Locality for High-Bandwidth Processor Memory System Design. In *Proceedings of the 32nd International Symposium on Microarchitecture*, 1999.

[28] Sangyeun Cho, Pen-Chung Yew, and Gyungho Lee. Decoupling Local Variables Accesses in a Wide-Issue Superscalar Processors. In *Proceedings of the 26th International Symposium on Computer Architecture*, 1999.

[29] Jim Chu and Frank Hady. Maximizing AGP Performance. http://www.agpforum.org/downloads/guide21.pdf, 1998.

[30] Compaq Corporation. Redefining the Pocket PC — Compaq iPAQ H3650 Pocket PC. http://www.compaq.ca/English/channel/mktgmtl/mkmtlpdf/ipaq_pocket_eng.pdf, 2000.

[31] Intel Corporation. Pentium Pro Family Developer's Manual, volume 2: Programmer's Reference Manual. Intel Literature Centers, 1996.

[32] Intel Corporation. Pentium Pro Family Developer's Manual, volume 3: Operating System Writer's Manual. Intel Literature Centers, 1996.

[33] Intel Corporation. IA-64 Application Developer's Architecture Guide. Intel Literature Centers, 1999.

[34] Intel Corporation. Intel Architecture Optimization Reference Manual. http://developer.intel.com/design/pentiumii/manuals/245127.htm, 1999.

[35] Rambus Corporation. Direct Rambus Memory Controller (RMC.dl) Data Sheet. http://www.rambus.com/docs/RMC.d1.0036.00.8.pdf, 1999.

[36] Rambus Corporation. Rambus Technology Overview. http://www.rambus.com/developer /downloads/TechOV.pdf, 1999.

[37] Standard Performance Evaluation Corporation. SPEC CPU95 Benchmarks. http://www.specbench.org/osg/cpu95/, 1995.

[38] Keith Diefendorff. Compaq Chooses SMT for Alpha: Simultaneous Multithreading Exploits Instruction and Thread-Level Parallelism. *Microprocessor Report*, Vol. 13, No. 16, December 1999.

[39] Keith Diefendorff. Power4 Focuses on Memory Bandwidth. *Microprocessor Report*, Vol. 13, No. 13, October 1999.

[40] Keith Diefendorff. PowerPC G4 Gains Velocity. *Microprocessor Report*, Vol. 13, No. 14, October 1999.

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

[42] Amer Diwan, David Tarditi, and Eliot Moss. Memory System Performance of Programs with Intensive Heap Allocation. *ACM Transactions on Computer Systems*, Vol. 13 No. 3, August 1995.

[43] Joel Emer and Doug Clark. A Characterization of Processor Performance in the VAX-11/780. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, 1984.

[44] Keith I. Farkas, Norman P. Jouppi, and Paul Chow. Register File Design Considerations in Dynamically Scheduled Processors. Technical Report 95/10, Compaq Corporation, 1995.

[45] Jerry Frenkil. A Multi-level Approach to Low-poer IC Design. *IEEE Spectrum*, Feburary 1998.

[46] Freddy Gabbay and Avi Mendelson. Using Value Prediction to Increase the Power of Speculative Execution Hardware. *ACM Transactions on Computer Systems*, Vol. 16, No. 3, August 1998.

[47] Kanad Ghose and Milind B. Kamble. Reducing Power in Superscalar Processor Caches using Subbanking, Multiple Line Buffers and Bit-Line Segmentation. In *Proceedings of the 1999 International Symposium on Low Power Electronics and Design*, 1999.

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

[49] Antonio Gonzalez, Carlos Aliagas, and Mateo Valero. A Data Cache With Multiple Caching Strategies Tuned to Different Types of Locality. In *Proceedings of the International Conference on Supercomputing*, 1995.

[50] Linley Gwennap. Merced Shows Innovative Design. *Microprocessor Report*, Vol. 13, No. 13, October 1999.

[51] D. W. Hammerstrom and E. S. Davidson. Information Content of CPU Memory Reference Behavior. In *Proceedings of the 4th Annual International Symposium on Computer Architecture*, 1977.

[52] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kauffmann Publishers, Inc., second edition, 1996.

[53] John L. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millennium. *IEEE Micro*, July 2000.

[54] Glenn Hinton, Dave Sager, Mike Upton, Darrel Boggs, Doug Carmean, Alan Kyker, and Patrice Roussel. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, Q1 2001.

[55] Peter Y.-T. Hsu and Edward S. Davidson. Highly Concurrent Scalar Processing. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, 1986.

[56] Michael Huang, Jose Renau, Seung-Moon Yoo, and Josep Torrellas. Cache Decomposition for Energy-Efficient Processors. In *Proceedings of the 2001 International Symposium on Low Power Electronics and Design*, 2001.

[57] SLDRAM Inc. http://www.sldram.com.

[58] Intel Corporation. *Intel StrongARM SA-1110 Microprocessor Developer's Manual*, June 2000. http://developer.intel.com/design/strong/manuals/278240.htm.

[59] Kiyoo Itoh, Katsuro Sasaki, and Yoshinobu Nakagome. Trends in Low-Power RAM Circuit Technologies. *Proceedings of the IEEE*, Vol. 83. No. 4., April 1995.

[60] Mike Johnson. *Superscalar Microprocessor Design.* Prentice Hall International Ltd., 1991.

[61] Teresa L. Johnson and Wen-Mei W. Hwu. Run-Time Adaptive Cache Hierarchy Management via Reference Analysis. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[62] Teresa L. Johnson, Matthew C. Merten, and Wen-Mei Hwu. Run-Time Spatial Locality Detection and Optimization. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, 1997.

[63] Norman Jouppi and et al. A 300-MHz 115-W 32-b Bipolar ECL Microprocessor. Technical Report 93.8, Compaq Western Research Lab, November 1993.

[64] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture.* Prentice Hall International Ltd., 1992.

[65] Stefanos Kaxiras, Zhigang Hu, and Margaret Martonosi. Cache Decay: Exploiting Generational Behavior to Reduce Leakage Power. In *Proceedings of the 28th International Symposium on Computer Architecture*, 2001.

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

[67] David Kroft. Lockup-Free Instruction Fetch/Prefetch Cache Organization. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, 1981.

[68] An-Chow Lai and Babak Falsafi. Selective, Accurate and Timely Self-Invalidation Using Last-Touch Prediction. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.

[69] Monica Lam and Robert Wilson. Limits of Control-Flow on Parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992.

[70] Butler W. Lampson. Fast Procedure Calls. In *Proceedings of the 1st International Symposium on Architectural Support for Programming Languages and Operating Systems*, 1982.

[71] Alvin R. Lebeck and David A. Wood. Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors. *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995.

[72] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: A Tool for Evaluating Multimedia and Communications Systems. In *Proceedings of the 30th International Symposium on Microarchitecture*, 1997.

[73] Hsien-Hsin Lee, Youfeng Wu, and Gary Tyson. Quantifying Instruction-Level Parallelism Limits on an EPIC Architecture. In *Proceedings of the 2000 IEEE International Symposium on Performance Analysis of Systems and Software*, 2000.

[74] Hsien-Hsin S. Lee, Chris J. Newburn, Mikhail Smelyanskiy, and Gary S. Tyson. Optimizing Memory Subsystem Designs by Exploiting Region Reference Characteristics. *To appear in IEEE Transactions on Computers.*

[75] Hsien-Hsin S. Lee, Mikhail Smelyanskiy, Chris J. Newburn, and Gary S. Tyson. Stack Value File: Custom Microarchitecture for the Stack. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, 2001.

[76] Hsien-Hsin S. Lee and Gary S. Tyson. Region-Based Caching: An Energy-Delay Efficient Memory Architecture for Embedded Processors. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2000.

[77] Hsien-Hsin S. Lee, Gary S. Tyson, and Matthew K. Farrens. Eager Writeback — A Technique for Improving Bandwidth Utilization. In *Proceedings of the 33rd Annual International Symposium on Microarchitecture*, 2000.

[78] Charles Lefurgy, Peter Bird, I-Cheng Chen, and Trevor Mudge. Improving Code Density Using Compression Techniques. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, 1997.

[79] Mikko H. Lipasti and John P. Shen. Exceeding the Data-Flow Limit Via Value Prediction. In *Proceedings of the 29th International Symposium on Microarchitecture*, 1996.

[80] Mikko H. Lipasti, Chris B. Wilkerson, and John P. Shen. Value locality and local value prediction. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[81] Scott McFarling. Combining Branch Predictors. Technical Report TN-36, Compaq Western Research Lab, June 1993.

[82] James Montanaro and et al. A 160-MHz, 32-b, 0.5-W CMOS RISC Microprocessor. *Digital Technical Journal*, Vol. 9 No.1, November 1996.

[83] Glenford Myers, Konrad Lai, Michael Imel, Glenn Hinton, and Robert Riches. Stack Frame Cache on a Microprocessor Chip. US Patent 4,811,208, 1989.

[84] Shien-Tai Pan, Kimming So, and Joseph T. Rahmeh. Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation. *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992.

[85] David A. Patterson. Reduced Instruction Set Computers. *Communications of ACM*, 28(1), January 1985.

[86] Alexander Peleg and Uri Weiser. Dynamic Flow Instruction Cache Memory Organized around Trace Segments Independent of Virtual Address Line. US Patent 5,381,533, 1995.

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

[88] Matthew Postiff, David Greene, Gary Tyson, and Trevor Mudge. The Limits of Instruction Level Parallelism in SPEC95 Applications. In *Proceedings of the 3rd Workshop on Interaction between Compilers and Computer Architectures*, 1998.

[89] Mikd D. Powell, Se-Hyun Yang, Babak Falsafi, Kaushik Roy, and T. N. Vijaykumar. Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories. In *Proceedings of the 2000 International Symposium on Low Power Electronics and Design*, 2000.

[90] Jan M. Rabaey. *Digital Integrated Circuits: A Design Perspective.* Prentice Hall International Ltd., 1997.

[91] Srinivas K. Raman, Vladimir Pentkovski, and Jagannath Keshava. Implementing Streaming SIMD Extensions on the Pentium III Processor. *IEEE Micro*, July-August 2000.

[92] Glenn Reinman and Norman Jouppi. Extnesions to CACTI. http:// research.compaq.com /wrl/people/ jouppi/CACTI.html, 1999.

[93] Jude A. Rivers and Edward S. Davidson. Reducing Conflicts in Direct-mapped Caches with a Temporality-based Design. In *Proceedings of the 1996 International Conference on Parallel Processing*, 1996.

[94] Jude A. Rivers, Gary S. Tyson, Edward S. Davidson, and Todd M. Austin. On High-Bandwidth Data Cache Design for Multi-Issue Processors. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, 1997.

[95] Eric Rotenberg, Steve Bennett, and James E. Smith. Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetchig. In *Proceedings of the 29th International Symposium on Microarchitecture*, 1996.

[96] Vatsa Santhanam, Edward Gornish, and Wei-Chung Hsu. Data Prefetching on the HP PA-8000. *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[97] Simon Segars, Keith Clarke, and Liam Goudge. Embedded Control Problems, Thumb and the ARM7TDMI. *IEEE Micro*, October 1995.

[98] Simplescalar Tool set. X benchmark suite. http://www.cs.wisc.edu/∼austin /simple/xbenchmarks.tar.gz, 1998.

[99] C. E. Shannon. A Mathematical Theory of Communication. *Bell System Tech. J.*, vol. 27, 1948.

[100] Tien-Pao Shih and Edward S. Davidson. Grouping Array Layouts to Reduce Communication and Improve Locality of Parallel Program. In *Proceedings of the 1994 International Conference on Parallel and Distributed Systems*, 1994.

[101] Richard L. Sites. How to Use 1000 Registers. In *Proceedings of the CalTech Conference on VLSI*, 1979.

[102] Richard L. Sites. *Alpha AXP Architecture Reference Manual.* Digital Press, 1995.

[103] Kevin Skadron and Douglas W. Clark. Design Issues and Tradeoffs for Write Buffers. In *Proceedings of the 3th International Symposium on High Performance Computer Architecture*, 1997.

[104] James E. Smith. A Study of Branch Prediction Strategies. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, 1981.

[105] Guri Sohi and Sriram Vajapeyam. Instruction Issue Logic for High-Performance Interruptable Pipelined Processors. *Proceedings of the 14th Annual International Symposium on Computer Architecture*, 1987.

[106] Timothy J. Stanley and Robert G. Wedig. A Performance Analysis of Automatically Managed Top of Stack Buffers. In *Proceedings of the 14th International Symposium on Computer Architecture*, 1987.

[107] Ching-Long Su and Alvin M. Despain. Cache Design Trade-off for Power and Performance Optimization: A Case Study. In *Proceedings of the International Symposium on Low Power Design*, 1995.

[108] Ching-Long Su and Alvin M. Despain. Cache Designs for Energy Efficiency. In *Proceedings of the 28th Hawaii International Conference on System Science*, 1995.

[109] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995.

[110] Jim Turley. MIPS RM7000 Emerges From QED. *Microprocessor Report*, Vol. 12, No. 10, August 1998.

[111] Gary Tyson and Todd Austin. Improving the Accuracy and Performance of Memory Communication Through Renaming. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, 1997.

[112] David W. Wall. Limits of Instruction-Level Parallelism. Technical Report 93.6, Compaq Western Research Lab, 1993.

[113] Alan Watt. *3D Computer Graphics*. Addison-Wesley Publishers, 1993.

[114] David L. Weaver and Tom Germond. *The SPARC Architecture Manual*. SPARC International, Inc., 1994.

[115] Chih-Po Wen. Improving Instruction Supply Efficiency in Superscalar Architectures using Instruction Trace Buffers . *ACM Symposium on Applied Computing*, 1992.

[116] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Caching Considerations for Generational Garbage Collection. *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, 1992.

[117] Steven J.E. Wilton and Norman P. Jouppi. An Enhanced Access and Cycle Time Model for On-Chip Caches. Technical Report 93/5, Compaq Western Research Laboratory, July 1994.

[118] Mason Woo, Jackie Neider, and Tom Davis. *OpenGL Programming Guide*. Addison-Wesley Developers Press, second edition, 1997.

[119] Kenneth C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, April 1996.

[120] Tse-Yu Yeh and Yale N. Patt. Two-Level Adaptive Training Branch Prediction. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, 1991.

[121] Paul Zagacki, Deep Buch, Emile Hsieh, Daniel Melaku, Vladimir Pentkovski, and Hsien-Hsin Lee. Architecture of a 3D Software Stack for Peak Pentium III Processor Performance. *Intel Technology Journal*, Q2 1999.

[122] V. Zhuban and P. Kogge. The Energy Complexity of Register Files. In *Proceedings of the 1998 International Symposium on Low Power Electronics and Design*, 1998.