Journal of Instruction-Level Parallelism 7 (2005) 1-37 Submitted 02/2005; published 07/2005

A Limit Study on the Potential of Compression for Improving Memory System Performance, Power Consumption, and Cost

Nihar R. Mahapatra NRM@EGR.MSU.EDU
Department of Electrical and Computer Engineering,
Michigan State University, East Lansing, MI 48824, U.S.A.

Jiangjiang Liu LIUJJ@CS.LAMAR.EDU
Department of Computer Science,
Lamar University, Beaumont, TX 77710, U.S.A.

Krishnan Sundaresan SUNDARE2@EGR.MSU.EDU
Department of Electrical and Computer Engineering,
Michigan State University, East Lansing, MI 48824, U.S.A.

Srinivas Dangeti SRINIVAS.DANGETI@SUN.COM
Balakrishna V. Venkatrao BALAKRISHNA.VENKATRAO@SUN.COM
Scalable Systems Group,
Sun Microsystems, Inc., Sunnyvale, CA 94085, U.S.A.

Abstract

Continuing exponential growth in processor performance, combined with technology, architecture, and application trends, place enormous demands on the memory system to allow information storage and exchange at a high-enough performance (i.e., to provide low latency and high bandwidth access to large amounts of information), at low power, and cost-effectively. This paper comprehensively analyzes the redundancy in the information (addresses, instructions, and data) stored and exchanged between the processor and the memory system and evaluates the potential of compression in improving performance, power consumption, and cost of the memory system. Traces obtained with Sun Microsystems' Shade simulator simulating SPARC executables of eight integer and seven floating-point programs in the SPEC CPU2000 benchmark suite and five programs from the MediaBench suite, and analyzed using Markov entropy models, existing compression schemes, and CACTI 3.0 and SimplePower timing, power, and area models yield impressive results.

1. Introduction

Performance, power consumption, and cost are probably the three most important parameters that drive computer system design today ranging from digital signal processors (DSPs), application-specific integrated processors (ASIPs), and field programmable gate arrays (FPGAs) to general-purpose processors and multiprocessors. While their relative importance varies in these systems, all three parameters are recognized as important. Thus, while performance is most important in high-end multiprocessors, performance/cost drives the general-purpose processor market, and power consumption plays a more significant role in embedded and wireless applications.

All computer systems have three main subsystems: the *computation system* or the processor core, the *memory system*, and the *I/O system* (comprising secondary storage I/O and network I/O). The memory system has two main types of components: *storage components* (including registers, one or more levels of caches, main memory) for storing information (primarily instructions and data) and *communication components* (comprising I/O buffers, I/O pads, and pins on the processor and memory chips, and on- and off-chip control, address, instruction, and data buses) for communicating information (primarily addresses, instructions, and data) between the computation system and storage components and between the storage components themselves.

1.1 Motivation

Increasing levels of device integration and continuing rise in clock frequency and die area have resulted in an exponential trend for raw computation system performance enhancement. Architectural advancements to exploit this raw performance potential have been made in the form of increasing levels of bit-level (4-bit, 8-bit, 16-bit, 32-bit, 64-bit), instruction-level (deeper pipelines, out-of-order, wide-issue superscalar and multiscalar), thread-level (simultaneous multithreading), and processor-level (chip multiprocessor) parallelism [1]. Thus, there may be multiple processors on a chip, each of which may execute multiple threads simultaneously, and each thread may be executed by a deeply pipelined, superscalar core clocked at a high frequency. Due to such dramatic increases in computation system performance, there is an enormous pressure on the memory system to store increasing amounts of information (instructions and data) and communicate this information (addresses, instructions, and data) at a high enough bandwidth and low enough latency to avoid performance bottlenecks.

To address the above problem, designers have continued to increase the number of I/O buffers, pads, and pins, widths of buses, number of registers, number and sizes of caches, and the size of main memory, in addition to improving their design. However, since interconnect size does not scale as well as on-chip logic size, on- and off-chip buses, especially the latter, have relatively higher capacitances and delays compared to on-chip logic. Further, there are more stringent constraints on the clock speed at which external pins can be driven compared to on-chip circuitry. Finally, DRAM bandwidth and latency are improving at a slower rate compared to processor performance. All of this contributes to a growing computation-memory system performance gap [2].

As noted above, storage components have increased in number and size in order to reduce performance bottlenecks and hence are occupying larger and larger areas on chip. Due to current technology scaling trends, communication components also occupy a greater fraction of the chip area because interconnect size scales relatively poorly compared to logic (transistor) size. Moreover, in interconnects, not only individual wire capacitances contribute to power consumption, but more so do inter-wire capacitances between adjacent bus lines due to tighter spacing between lines [3]. Consequently, increasingly more fraction of the system power consumption and cost is due to the memory system compared to the computation system [4]. Thus, the memory system is becoming an increasing bottleneck as designers strive towards higher performance, cost-effective, and power-efficient system designs.

1.2 Scope and Contributions of this Work

Information redundancy—in the form of highly sequential address streams, repeated instruction sequences in both program code and dynamic instruction streams, and highly predictable data values when programs frequently loop through data arrays—can be exploited to reduce the processor-memory bottleneck. By compressing information that is stored or transmitted in the memory system, potentially higher performance (improvements in bandwidth and latency of communication components and improvement in capacity of storage components), lower power consumption, and cost benefits can be obtained; we refer to architectures supporting such compression as *compressed memory system* (CMS) architectures. This paper evaluates different CMS architectures in terms of improvements that they can provide. We consider all primary types of information (namely, addresses, instructions, and data) and all important storage and communication components at all levels of the memory system hierarchy where such information is stored or communicated. For addresses, we consider the tag fields of instruction and data caches and instruction and data address buses. For instructions, we consider the data fields of instruction caches, main memory executable code, and instruction buses. For data, we consider integer and floating-point register files, data fields of data caches, and data buses.

We use Sun Microsystems' Shade simulator [5] to collect traces for the various storage and communication components. Our simulated processor-memory system consists of a superscalar processor having a memory hierarchy with split instruction and data caches at the first level (closest to the processor), a unified cache at the second level, and a main memory. We collected register- and cache-data traces, and address, instruction, and data bus traces by running the simulator on SPARC-V9 executables of eight integer and seven floating-point programs from the SPEC CPU2000 benchmark suite and five programs from the MediaBench suite. Analysis of these traces using Markov entropy models, existing compression schemes, and CACTI 3.0 [6] and SimplePower [7] timing, power, and area models shows excellent potential for compression in both storage and communication components at all levels of the memory systems.

The organization of the remainder of the paper is as follows. Sec. 2 discusses CMS architectures in detail. Sec. 3 provides an overview of previous work related to cache, memory, and bus compression. Sec. 4 describes the simulation environment, analysis tools, and methods we used in our study. Sec. 5 presents detailed results from our analysis. Finally, we conclude in Sec. 6.

2. Compressed Memory System Architectures

In this section, we discuss the opportunities for compression present in the memory system, a useful way of classifying CMS architectures, and finally the benefits of CMS architectures and the challenges to be overcome.

2.1 Opportunities for Compression

Compression of some source information consisting of a sequence of symbols is possible when those symbols occur with non-uniform frequencies or likelihoods either in the source as a whole or in any given portion thereof. This allows for the encoding of the more frequent or

likely symbols with shorter codewords compared to the less frequent or likely ones, resulting in an overall compression of the source. The three primary types of information that are stored and communicated by the storage and communication components of the memory system, respectively, are addresses, instructions, and data. All three of these inherently possess significant amounts of redundancy as we explain next.

2.1.1 Address redundancy

Addresses are of two types: instruction addresses and data addresses. Both exhibit spatial and temporal locality, meaning that the next instruction or data address to be issued by the processor is not random, but likely spatially and/or temporally close to recently issued addresses. Instruction addresses issued by the processor to the L1 cache are typically sequential, except when branches or jumps occur, and even then, the target addresses are not typically very far away from the last address. This is the reason why many instruction sets use PC-relative addressing with shorter-than-full-word-size offsets for branch and jump instructions. Addresses issued by the L1 cache to the L2 cache correspond to misses in the former and are more unpredictable compared to those issued by the processor to L1. Similarly, addresses issued by higher levels (away from the processor) of the memory system become increasingly unpredictable and hence more information-rich. Still, these addresses do exhibit temporal and spatial locality, although to lesser extents. Data addresses issued by the processor are also known to exhibit temporal and spatial locality because of scanning of data arrays in loops, although to a lesser extent than instruction addresses. Like instruction addresses, redundancies in data addresses are expected to decrease at higher levels of the memory hierarchy.

As far as storage components are concerned, address information is primarily stored in the tag fields of caches, the TLB, and page tables (and some registers, such as the PC and the memory address register, but this is not much). Since tag fields store a portion of the address (a portion of the instruction address in the case of instruction caches and a portion of the data address in the case of data caches), they are expected to exhibit redundancy as discussed above for addresses. Specifically, the tag fields correspond to blocks that have been recently accessed and as such they should be temporally and spatially close. Note that since the tag field is normally derived from the high-order portion of the address, it is expected to possess a higher amount of redundancy than whole addresses, since the high-order end of the address is where more redundancy lies due to the spatial proximity of addresses issued. Similarly, the TLB and page tables which store address information (virtual and physical page numbers) will have redundancies.

2.1.2 Instruction redundancy

Since instructions fetched correspond to instruction addresses issued by the processor, instructions exhibit the same temporal and spatial locality as instruction addresses. Further, not all instructions, instruction sequences, opcodes, register operands, and immediate constants are present equally frequently. Repetitions of instruction sequences, opcodes, registers, and immediate constants, and correlation between opcodes and registers and between opcodes and immediate constants can be exploited. The reasons for the presence of such redundancies are that all programs have certain basic characteristics, e.g., they have pro-

cedures and procedure calls, they have branches every few instructions (typically every six instructions), they use loops and if-then-else clauses, etc. Moreover, compilers used to generate object code do so based on a set of templates, which naturally lead to redundancies. As discussed for addresses earlier, instruction traffic at higher levels of the memory hierarchy are likely to exhibit less temporal and spatial locality. However, since at higher levels, the instruction traffic consists of larger blocks, more redundancy is present within blocks. Similarly, in storage components, there is redundancy in the instructions stored in main memory and instruction caches.

2.1.3 Data redundancy

Data fetched by the processor also exhibits temporal and spatial locality, although to a lesser extent than instructions. However, there is extra redundancy present in the values of data communicated by data buses and stored in registers, data caches, and main memory. For any given type of data (character, integer, floating-point, etc.), not all values are equally likely. For instance, many programs do not tend to use the entire range of integer values possible, but rather the values used tend to be concentrated around certain values, especially, zero. For such small magnitude two's complement numbers, most high order bits of the data word are likely to be either all zero (positive) or all one (negative) due to sign extension.

2.2 Classification of CMS Architectures

A CMS architecture will be effective only if it is adapted to the characteristics of the source information it seeks to compress. Hence the degree of specialization of a compression scheme is an important parameter that determines its effectiveness. In general, a compression scheme is designed to compress some new raw information based upon symbol statistics or frequencies drawn from some known or typical data set. Depending upon how specialized this data set is, five classes of CMS architectures, from the most specialized to the least specialized, can be identified as described below. Note that in all cases, symbol statistics are drawn from the same type of information (address, instruction, data) as the type of information being compressed.

Block-specific architecture: In this case, symbol statistics used to compress a block of information (e.g., a block in any cache or main memory or a word on a bus) are drawn from the same block. Such a compression scheme utilizes the most specialized information for compression, but it is likely to have the most complexity.

Memory-component-specific architecture: When in a CMS architecture symbol statistics are drawn from the typical data set of a memory component and are used to compress each block of that component, it is referred to as memory-component-specific. For example, symbol statistics may be drawn from all the instruction addresses typically transmitted over the L1-L2 instruction address bus and then used to compress each instruction address transmitted over that bus.

Application-program-specific architecture: In this case, symbol statistics used for compression of information in a memory component are drawn from the typical data sets found in a given application program in all memory components that store or communicate information of the same type.

Application-class-specific architecture: In contrast to the previous case, here symbol statistics are drawn from application programs that belong to the same class (e.g., integer-computation-intensive applications or floating-point-computation-intensive applications), rather than from one particular application program.

General architecture: In this case, symbol statistics used for compressing information in a memory component are drawn from a broad range of applications meant to be executed on a system and from all memory components that store and communicate the same type of information. Here the compression scheme utilizes the most general type of statistical information and is expected to provide some reasonable compression across a range of applications.

It is possible to use different degrees of specialized statistical information to perform compression in different parts of the memory system. Thus, for example, while application-class-specific compression may be better for instruction stream compression, memory-component-specific schemes may yield best results for address bus compression. Also, the compression scheme can be static or dynamic, i.e., the statistical information used for compression can be predetermined and fixed or it may change dynamically.

2.3 Benefits of CMS Architectures

Depending upon the state of the technology at the time of implementation and application requirements, it may not be possible to use compression to advantage in all areas of the memory system, although substantial direct or indirect improvements can be expected in most areas of the system. As an example, using compression in on-chip or off-chip buses can have multiple ramifications. The effective bandwidth of the system will increase as more number of bits can be transmitted using the same number of bus lines. If the emphasis is on reducing power, it may be possible to reduce the number of bus lines while maintaining the same effective bandwidth, and this would result in power savings because fewer bits need to be transmitted and because significant amount of power is consumed in the metal lines of the chip. Similarly, a decrease in the number of bus lines will reduce the die area and hence cost could go down significantly because cost varies as the fourth or higher power of die area [2]. Application of compression in other areas like caches, registers, and main memory have obvious benefits like increasing the effective storage capacity using the same number of transistors or lowering power consumption and cost by using smaller number of transistors that provide the same effective storage capacity.

Compression can also be used possibly to improve cache latency by, for example, storing a portion of the information in cache in compressed form. Using the same number of transistors, this modified cache will have more effective capacity and hence less effective miss rate than a regular, fully-uncompressed cache. The latency of the uncompressed portion of this modified cache will be comparable or better (due to its smaller size) relative to the regular cache. Also, the miss rate of the former will be only slightly worse than the latter for larger cache sizes. This is because, for larger caches, miss rate reduces very slowly as cache size increases. The latency of the compressed portion of the cache will be more than the regular cache, but it will be less than that of the next higher level of the memory hierarchy. As a result, if there is a miss in the uncompressed portion of the cache, the compressed portion

can be checked and if the required information is present, a slower access to the next higher level of the memory hierarchy can be avoided.

2.4 Feasibility and Challenges

As a downside, any implementation of compression in the memory system will have overheads in extra area, latency, and power consumption due to the compression/decompression logic. However, since the size, speed, and power consumption of logic (which will be used to do compression/decompression) scale better than those of interconnect (which will be used to communicate the information), these overheads will continue to decrease over time. Also, the (area, latency, and power) overheads that can be tolerated for compression/decompression vary from one part of the memory system to another and from application to application. For example, more compression/decompression latency overhead can be tolerated at higher levels of cache and main memory than at lower levels. Similarly, less latency overhead can be tolerated in higher performance systems than in non-performance-critical systems. Depending upon the state of the technology, the location in the memory system where compression is to be applied, and the application system requirements, the compression scheme can be more aggressive (better compression, but more compression/decompression overheads) or less aggressive (moderate compression, but less compression/decompression overheads), i.e., the compression scheme, and hence its overheads, can be suitably regulated. For example, we have shown that dynamic cache-based address bus traffic compression schemes like dynamic base register caching [8, 9] and bus-expander [10], described later in Sec. 3.2, need only very small overheads—few hundred bits of cache and typically only a fraction of one cycle access latency for these small compression caches—to compress addresses [11]. Such specific estimation of the overheads of compression and decompression is possible only with respect to a particular compression scheme and architecture. Since we deal with a variety of memory system components for which such accurate overhead analysis will be too time consuming, in this paper we focus on the limits to which compression can be potentially exploited using Markov entropy models, some representative existing compression schemes, and accurate cache and bus timing, power, and area models.

3. Related Work

Previous work in memory system compression has been done both in analyzing compressibility and in the development of specific compression schemes for the memory system. These include schemes for address, instruction, and data bus compression, program code compression and compressed instruction set design for embedded systems, and main memory and cache compression. Related work in traffic optimization for low power using bus encoding has also been reported. We briefly review previous research in these areas next.

3.1 Previous Analysis

In previous analytical research focusing on finding the potential for compression, separate studies by Hammerstorm and Davidson [12] and Becker et al. [13] used entropy measures to evaluate the compressibility of addresses in microprocessors. Wang and Quong analyzed

the potential of instruction compression [14]. They evaluated the effect of instruction compression on the average memory access time for various types of memory systems. Later, compressibility of program code in different architectures on various operating systems was investigated by Kozuch and Wolfe [15]. The potential of main memory compression was studied by Kjelso et al. [16]. We presented a brief analytical study of compression focusing on overall benefits for the memory system in [17] and a broader study in [18]. Apart from analytical studies of compressibility of memory system components, specific compression schemes have also been proposed for various memory system components. We briefly review them next.

3.2 Address, Instruction, and Data Compression

Park and Farrens presented a *dynamic base register caching* (DBRC) scheme for compressing off-chip, processor-memory addresses in [8, 9]. In this scheme, the original address is split into a higher order and a lower order component and the former is stored in a cache of base registers. When a new address results in a base register cache hit, the index to the base-register cache is transmitted on the bus along with the uncompressed lower order part of the original address, thus resulting in compression. They found that by using a 16-bit bus for a 32-bit microprocessor and the DBRC scheme resulted in only a miss rate of 2% for the base register cache and most of the time memory addresses could be transmitted using a 16-bit bus, thus achieving almost a 50% reduction in the number of pins. Citron and Rudolph proposed a similar scheme, called bus-expander (BE), for address, instruction and data bus compression [10]. They reported hit rates of up to 95% for their compression caches [10]. Both these schemes focused on reducing costs and improving pin bandwidth for off-chip accesses. Recent work by Citron studied the feasibility of using bus compression to reduce the growing gate delay versus interconnect delay gap for long on-chip wires [19]. The effectiveness of a BE-like bus compression scheme to reduce the switching activity (power consumption) in off-chip data buses was studied by Basu et al. [20]. A more detailed analysis of the effect of compression on bus power consumption and a comparison of DBRC and BE for on- and off-chip address buses was presented by us in [11]. Also, recently Kant and Iyer analyzed the performance and power benefits of using dynamic, cache-based compressed address and data transfer mechanisms for server interconnects [21].

3.3 Code Memory Compression

Code memory compression schemes compress the text segment of an executable program to reduce code size and thus save power and cost. Code memory compression schemes can be divided into three categories. The first category, called *code compaction* schemes, use compiler optimizations during embedded code generation to minimize sizes of parts of code that are used frequently (e.g., by creating procedures). These are purely software techniques and require no hardware support during run-time. Various code compaction schemes have been reported in the literature [22, 23, 24, 25, 26]. The second category, called *code compression* schemes, refers to techniques that minimize the size of the executable code and require decompression to be done before the compressed code can be executed. Among popular code compression schemes are compressed code RISC processor (CCRP) [27], call-dictionary compression [28], software-managed dictionary compression [29], semi-

adaptive Markov compression (SAMC) and semi-adaptive dictionary compression (SADC) [30, 31], and IBM's CodePack for PowerPC cores [32, 33]. Our previous work provides a side-by-side comparison of the effectiveness of several popular code compression schemes on a standard platform and set of benchmarks [34]. Code compression has also been proposed for VLIW architectures [35, 36] and has been recently adopted in commercial VLIW processors [37]. Simple instruction encoding schemes have also been proposed for low-cost, low-energy embedded processors [38, 39, 40]. The third category of code memory compression schemes is called *compressed instruction sets*; these are supported in popular RISC cores like ARM and MIPS [41, 42].

3.4 Cache and Main Memory Compression

Memory is an important resource for both embedded and general purpose processors and hence several memory compression techniques have been investigated. IBM's Memory eXpansion Technology (MXT) [43] enables the microprocessor to interface with compressed memory (C-RAM) [44] and provides fast hardware compression and decompression to enable access to the memory without significant increase in latency. Selective cache compression techniques [45], frequent value data caches [46], dynamic zero compression in data caches [47], adaptive cache compression [48], and indirect-indexed caches for cache compression [49] are some of the cache compression techniques that have been proposed for cache performance and/or power improvements.

3.5 Bus Encoding

Bus encoding is an area of research that has major implications for low power design of microprocessor systems. Encoding, although closely related to compression, is directed at minimizing unwanted signal transitions in the information stream to reduce bus switching energies during transfer rather than compressing the information itself. Various bus encoding schemes for off-chip address buses like Gray code [50], bus-invert code [51], asymptotic-zero (T0) code [52], and working-zone code [53] have been proposed and some of them have been applied to data buses too [54]. Cheng and Pedram presented a good survey of many bus encoding techniques in [55]. Most bus encoding schemes involve the use of a redundant line that indicates if the current value on the bus is an encoded value or not. Some modified address bus encoding schemes that do not require any redundant lines have been suggested in [56]. More recently, bus encoding schemes have been proposed for on-chip buses taking into account the effect of inter-wire capacitances that are especially important in deep submicron designs [3, 57]. Apart from energy reductions, encoding schemes that reduce bus delay and inter-wire cross talk have also been proposed [58, 59].

3.6 Relationship of Our Work to Previous Research

To our knowledge, this paper's comprehensive analysis of the potential of compression when applied to all parts of the memory system in the context of real-world benchmark programs and using extensive simulations is the first of its kind. The purpose of this paper is not to present specific compression schemes—which will be the subject of our future research—but to estimate the extent of compression possible in various memory system components.

Towards this end, we employ analysis methods and compression tools (such as Markov models, SAMC, Gzip) to estimate the extent of compression possible and the improvements in performance, power consumption, and cost that can be obtained. We present results for all parts of the memory system using realistic timing, power, and area models (CACTI 3.0 [6] and SimplePower [7]). We also present results related to: (1) the compressibility of original, exclusive-OR (XOR), and offset traces of instruction and data addresses; (2) the effect of compression on cache access time, power consumption, and area; (3) the relationship between compression ratio and bit fields and bit-field groupings; (4) the effect of application class, degree of specialization, encoding and multiplexing, analysis tool, static vs. adaptive compression, and multithreading; and (5) the relationship between information content, compression ratio, and power consumption, among others.

4. Simulation Methodology

In this section, we first discuss the target system and the parts of the memory system where we analyze the potential of compression. This is followed by a description of the simulation environment and the tools and methods used in our analysis.

4.1 Target System and Simulation Environment

Our target system has a memory hierarchy consisting of 32 integer and 32 floating-point registers, split instruction and data caches at the first level, a unified cache at the second level, and a paged main memory. The first level caches are write-through, 16KB each, 4-way set associative, and have a block size of 32 bytes. The second level cache is write-back, 256KB, 4-way set associative, and has a block size of 64 bytes. The default cache sizes we use may seem conservative in comparison to many modern systems but, as we will see later in Sec. 5.3.2, larger cache sizes generally improve compressibility. For our target memory system configuration, we used a modified version of the *cachesim5* cache analyzer in SHADE5 [5] running on a SPARC-V9 platform to collect the run-time traffic (addresses, instructions, and data) for benchmark programs. *Cachesim5* simulates cache operation by using address information and hence can be easily modified to collect address bus traces. But we also needed to collect instruction and data block traces for our analysis. To facilitate this, we augmented *cachesim5* by creating an interface to map addresses to the appropriate location in memory where the instruction and data blocks are located. This way, we were able to collect the actual address, instruction, and data traffic between processor, caches, and memory for our analysis.

We used benchmarks from the SPEC CPU2000 suite [60]. To capture the characteristics of both integer and floating-point programs, we chose eight integer and seven floating-point benchmarks randomly out of the 26 in the suite; we used only a subset of benchmarks because, otherwise, simulation time would have been prohibitive–as it is, we used a shared Linux cluster to get our results. For some experiments, especially when studying the effect of workloads, we additionally used five benchmarks from the MediaBench suite [61]. We used the -O2 optimization flag, which does basic local and global optimization to compile these benchmarks. All executables were statically-linked, in which the procedures and libraries are linked with the main program during compilation itself. We ran the benchmark programs using reference input sets provided with the SPEC2000 suite and to limit the execution

times of our simulations we used a methodology similar to the one described by Skadron, et al. [62]. Their research shows that accurate simulation results can be obtained by avoiding unrepresentative behavior at the beginning of a benchmark program's execution and by using a single, short simulation window of 50 million instructions. In our experiments, we simulate (but do not collect results for) instructions before the representative segment (warm-up window) and use a sampling window of 50 million instructions to collect our results. The sizes of the warm-up windows are also different for different SPEC programs [62]. The complete list of benchmarks we used and the warm-up window for each, given in parentheses, is as follows: (1) SPECint benchmarks—*gcc* (221M), *gzip* (2576M), *vortex* (2451M), *parser* (500M), *crafty* (500M), *twolf* (500M), *mcf* (500M), and *vpr* (500M); (2) SPECfp benchmarks (500M for each)—*applu*, *swim*, *wupwise*, *lucas*, *art*, *ammp*, and *equake*; and (3) MediaBench—*jpeg, adpcm, gsm, ghostscript*, and *rasta*. For MediaBench programs, we used input sets provided on the MediaBench Website [61] and collected results for complete execution of the benchmark.

4.2 Trace Collection

For communication components, traces were collected by saving each new value transmitted on a bus (connected between two storage components or between the storage component and the processor) and its corresponding timestamp in a file. Thus, we assume that bus lines are held at previously transmitted values when the bus is idle.

For storage components, the following methodology was adopted to collect dynamic traces and to ensure that the analysis done reflects average compressibility of the component. In instruction caches, a block may be loaded into and be replaced from a cache multiple times during the sampling window of the simulation. A load and the subsequent replacement of a block correspond to a time period during which it is resident in the cache; this is known as *cache residence time* (CRT) of the block. Since the time instant of a load that occurs before the sampling window and that of a replacement that occurs after the sampling window are not known, we ignore these time periods to avoid errors and consider only load-replacements that occur during the sampling window. In a data cache, a data block in cache during the sampling window can take on one or more values because of writes to it. Therefore, for data caches, we consider all data block values (instead of data blocks) that occur and get replaced during the sampling window.

Our trace files were created as follows. During the simulation, we keep a record of the block address and CRT of each block that is loaded and replaced during the sampling window. After simulation, we sort the blocks in decreasing order of CRTs and sum the CRTs of all blocks to get the total CRT (TCRT). Then, starting from the first block, we select blocks in the sorted list, in order, until the total residency time of selected blocks becomes equal to 80% TCRT. Then we write, in random order, the actual contents of these selected blocks a number of times, which is in proportion to each block's CRT, into a file to obtain the trace for our experiment. We use a random order to write the blocks to avoid any optimistic first-order compression ratios that may be obtained if the blocks were written in the order of their sorted residency times. For most of our cache compression analysis experiments, we used both 80% and 90% TCRT traces and averaged the results obtained from the two, instead of using a 100% TCRT trace because, the number of times

each blocks needs to be written into the trace will be extremely large for some blocks and this may result in a very large trace file. However, we used the 100% TCRT trace in a few experiments where it was possible to do so. To analyze tag information stored in cache, we used the higher order portion of corresponding instruction and data block addresses (since tags are obtained from this portion) to create cache tag traces.

Adopting a similar methodology as above for register compression analysis, we considered the residency times of only those values that are loaded and replaced in a register during the simulation window. Note that by considering the residency times of blocks as above, both in the case of cache and register, the trace file we created reflects the average contents of the cache/register. Hence the compression ratios obtained would be those expected from a compression scheme that chooses encodings based on average symbol statistics, rather than one where the choice changes dynamically as cache/register contents change. Therefore, the compression ratios we report in our studies are, in this sense, not optimistic.

4.3 Trace Analysis

We analyze the potential for compression of a particular trace by measuring the following two parameters. First, *compression ratio*, R, for any compression scheme is defined as the ratio of the size of compressed information to the size of the raw uncompressed information. We used various entropy measures and some available compression schemes to estimate the information content or compression ratio possible for our traces. Second, *transition ratio*, T, for the compressed information is defined as the ratio of the number of transitions that occur when the compressed information is transmitted on a bus to the number of transitions that occur when the original uncompressed information is transmitted on the same bus.

4.3.1 Compression ratios from entropy calculations

The entropy of a source denotes the average number of bits required to encode each symbol present in the source. Thus, the lower the entropy value, the more compressible the source. Entropy values can be computed for a source based upon various models—zero-information, zeroth-order Markov, first order Markov, etc. Compression ratios based on these models provide a theoretical lower bound for a particular trace. We describe these entropy models and how we computed compression ratios from entropy values next.

Zero-information entropy: Given a source with symbol set $s_1, s_2, \ldots, s_N$, the compressibility of a symbol in zero-information entropy is determined by its presence or absence in the trace, irrespective of the number of times the symbol occurs in the trace. Thus, if there are M unique symbols that actually occur in a trace out of N total unique symbols that could occur, where $M \leq N$, the zero-information entropy for the trace is $H = \log_2 M$, i.e., every one of the M symbols that actually occurs is represented by a unique $\log_2 M$ bit pattern.

Zeroth-order Markov entropy: Given that the source data has symbol set $s_1, s_2, \ldots, s_N$ and each symbol s_i occurs with probability $p(s_i)$, the entropy for the symbol is $-\log_2 p(s_i)$. The zeroth-order Markov entropy of the source data is given by the following relation: $H_0 = -\Sigma_{\forall i}[p(s_i) \cdot \log_2(p(s_i))]$. Whereas zero-information entropy reflects only the occurrence/non-occurrence of symbols, zeroth-order Markov entropy reflects in addition the frequencies of occurrence of symbols.

First-order Markov entropy: In first-order Markov entropy, we consider the occurrence of a symbol s_i, the probability $p(s_i)$ of that symbol's occurrence, and the probability $p(s_j|s_i)$ that the symbol is preceded by another symbol s_j. The first-order Markov entropy of a source is given by: $H_1 = -\Sigma_{\forall i}\left[p(s_i) \cdot \Sigma_{\forall j}[p(s_j|s_i) \cdot \log_2(p(s_j|s_i))]\right]$. This means that in a sequence of symbols if the current symbol is s_j and the next symbol is s_i, this next symbol s_i can be represented using $-\log_2 p(s_j|s_i)$ bits.

The symbols that we consider while measuring the entropy of any trace (address, instruction, data) correspond to aligned words in the trace, i.e., 32-bit words for addresses and instructions and 64-bit words for data. In our compression analysis study, we use only the low-order 32 bits of the actual 64-bit address in order to keep simulation times reasonable. Doing so results in a pessimistic estimate of the actual address compression potential since the high-order address bits have large amounts of redundancy due to the spatial locality characteristics of addresses. Using the entropy values measured, the corresponding compression ratio can be computed by taking the ratio of entropy times the number of symbols (words) to the number of symbols (words) times the size of a symbol (32 for addresses and instructions and 64 for data) in the original raw trace. Thus, for example, the *average zeroth-order Markov compression ratio* over n benchmarks is:

$$R_{H_0} = \frac{\sum_{i=1}^{n} H_0 \text{ of trace}_i}{n \times \text{ Original wordsize}}.$$

R_H and R_{H_1} are defined similarly.

4.3.2 Compression ratios from practical schemes

Some specific schemes to compress address, instruction, and data have also been proposed recently. We used some of these schemes to measure compression ratios and obtain an estimate of efficiency obtainable with practical schemes.

Instruction and data block compression scheme: Semi-adaptive Markov compression (SAMC), a compression algorithm based on arithmetic coding combined with a precalculated Markov model, was proposed by Lekatasas and Wolf for code compression [63]. We used the SAMC executable, obtained from the authors, to compress instruction and data blocks with the following parameters: block size equal to L1 or L2 cache block size depending on the memory level where the compression is applied, Markov model of depth 32 and width 256, and bits-per-probability of 4. The *average SAMC compression ratio* over n benchmark traces was calculated as follows:

$$R_{SAMC} = \frac{\sum_{i=1}^{n} \text{Size of compressed instruction or data trace}_i}{\sum_{i=1}^{n} \text{Size of original trace}_i}.$$

A point to note is that the SAMC algorithm is a block-based compression algorithm and hence average compression ratio for an individual block of that size is reported as the output.

Address compression scheme: Two techniques, dynamic base register caching (DBRC) and bus-expander (BE), have been proposed to compress addresses that are transmitted on buses [8, 10]. Both schemes use a small fully associative cache at the sending end for compressing addresses and decompress them using registers at the receiving end. In our analysis, we use BE to compress address streams. The *average address compression ratio*

over n benchmark traces is defined as follows:

$$R_{BE} = \frac{\sum_{i=1}^{n} \text{Size of compressed address trace}_i}{\sum_{i=1}^{n} \text{Size of original trace}_i}.$$

Data compression scheme: Gzip is a widely used GNU utility for compression in UNIX systems. It uses Lempel-Ziv (LZ77) dictionary compression algorithm which replaces strings of characters with single codes. Gzip does not do any analysis of the information source. Instead, it just adds every new string of characters it sees to a table of strings. Compression occurs when a single code is output instead of a string of characters. Since Gzip uses an algorithm based on bytes, good compression ratio is achieved on text files. We used Gzip on address, instruction, and data streams to provide an idea of compression achieved using a widely-used text compression utility. The *average Gzip compression ratio* over n benchmark traces is defined as follows:

$$R_{Gzip} = \frac{\sum_{i=1}^{n} \text{Size of compressed trace}_i}{\sum_{i=1}^{n} \text{Size of original trace}_i}.$$

4.3.3 Transition ratio

For CMOS technology, power consumption on a bus line is directly related to the switching activity on it as bits are transmitted one after another over it. We use a methodology similar to the one used in SimplePower [64] to calculate the switching activity of a given bus when information is transmitted across it. They calculate the average probability of a transition for each bit of the bus and find the total average probability across all bits, which is a measure of the per-input switching activity of the bus in bits [7]. Thus, the ratio of bus power consumption for two traces using the SimplePower model is equal to the ratio of the number of transitions for those two traces. We define *average transition ratio* over n benchmarks for compressed traces as follows:

$$T_C = \frac{\sum_{i=1}^{n} \text{No. of transitions in compressed trace}_i}{\sum_{i=1}^{n} \text{No. of transitions in original trace}_i}.$$

When estimating T_C, we used BE as the compression scheme for address traces and SAMC for instruction and data traces.

5. Results and Discussions

For communication components, we performed experiments on traces of address, instruction, and data traffic between the processor and memory for all three levels (processor-L1 cache, L1 cache-L2 cache, and L2 cache-main memory) for each benchmark and calculated the zero-information and zeroth- and first-order Markov entropies, and SAMC compression ratio in each case and, in some cases, we also calculated the Gzip compression ratio. We investigated the compression potential of storage components other than registers by calculating zero-information and zeroth- and first-order Markov entropy values, R_{SAMC}, and R_{Gzip}. For main memory, we calculated these values for the text segment of the statically-linked executable code. For registers, we performed only zeroth-order Markov analysis. The reason we did not do a first-order Markov analysis for registers is because a compression scheme that exploits

first-order behavior will need to represent the current value in a register in a manner that depends upon the previous value. Since a register has only one word, storing the previous and current values, even in compressed form, is unlikely to yield much compression. Moreover, if register compression is attempted, the compression scheme needs to be simple enough not to affect access latency by more than a little.

To keep the number of simulations reasonable and at the same time be able to study a number of parameter variations, we consider certain default settings as follows. We consider the default architecture to be memory-component specific as described earlier in Sec. 2.2. Also, in the default case, for our communication component analysis experiments, we consider demultiplexed buses, in which case there are separate buses for instruction address, data address, instruction, and data. In some cases, we consider a multiplexed bus, with one 'address' bus carrying both instruction and data addresses and one 'data' bus carrying both instructions and data. Also, the default memory level for which we report most of our results is between L1 and L2 caches. The default word size considered as a symbol size in Markov entropy calculations is 32 bits for address and instruction, 64 bits for data, and 20 bits for tag field (see Sec. 4.3.1 for an explanation regarding why we use 32-bit instead of the actual 64-bit address). For entropy analysis, in most cases, first-order Markov provides the best results and the performance of zeroth-order Markov is also better than zero-information. We present these three entropy results in most of our plots. In the experiments that we describe next, we summarize results in plots by averaging over all 15 (8 INT and 7 FP) benchmarks or by showing averages for INT, FP, and MediaBench programs separately for specific components. We calculate the average compression ratios as mentioned earlier in Sec. 4.3.2.

5.1 Overall Memory System Analysis

We investigated how compression ratio and power consumption vary across memory system components, namely, registers, caches, main memory, address bus, instruction bus, and data bus. The compression ratio is indicative of the extent to which performance enhancement or cost savings can be realized. Fig. 1 presents an overview of our analysis. We observe that communication components are in general more compressible than storage components (considering H_1 values which provide the best lower bound for entropy). Among storage components, we observe that the ordering from the most to the least compressible is L1 I-cache data field, L1 I-cache tag field, main memory, and registers. This is to be expected since instructions that are stored in the data fields of I-cache and tag field that corresponds to the high-order portion of the instruction address carry significantly higher amounts of redundancy than main memory or registers.

Among communication components, the ordering, from the most to the least compressible (again considering H_1 values), is instruction bus, data bus, and address bus. A possible explanation for the higher redundancy in the data bus compared to address bus is that a lot of the data blocks transmitted may contain small magnitude numbers that have lots of either 0 or 1 bits. Further, it is observed that the volume of data read traffic (data blocks sent from L2 to L1) is far greater than the write traffic (data blocks sent from L1 to L2), which means that the same blocks may appear in the data bus traffic often without any changes, and this also increases the redundancy. This also explains why data traffic shows

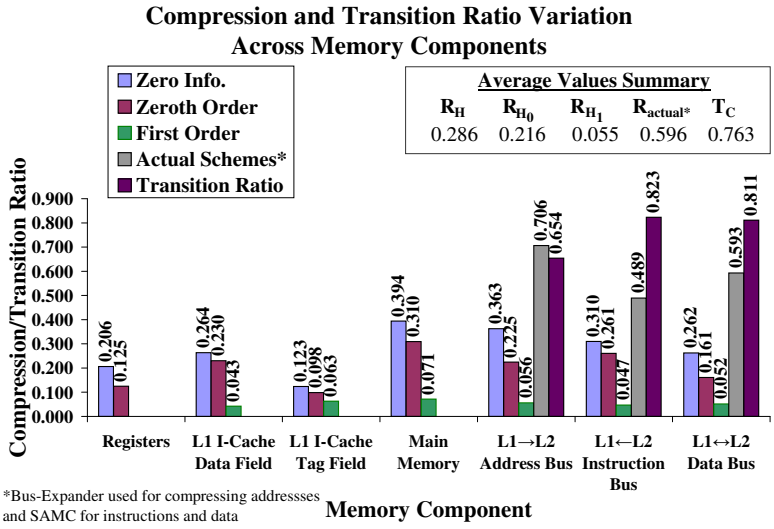


Figure 1: **Overall Memory System Analysis:** Compression ratio variation across memory system components. Communication components are in general more compressible than storage components when first-order entropies are considered.

the best compressibility in zero information and zeroth-order analysis. We also observe that the ordering of the communication components in terms of power savings after compression (from most to least savings) is as follows: address bus, data bus, and instruction bus.

5.2 Register Compression Analysis

For register compression, we performed zeroth-order Markov analysis over all 32 integer registers and 32 single-precision floating-point registers in our target architecture. In SPARC-V9, all integer registers are 64 bits each and the single-precision floating-point registers are 32 bits each [65]. The floating-point register file (FPRF) uses *aliasing*, i.e., some register names overlap. For example, in the 32 single-precision register set, the lower half of the 32 double-precision register set, and the lower half of the 16 quad-precision register set overlay each other. Considering the total number of registers in our analysis and keeping track of all values stored in them for large samples (50 million instructions) would have been computationally intractable. Hence, we study only instructions that manipulate registers in the single-precision FPRF.

Fig. 2 shows the zero-information and zeroth-order compression ratios for each register in the integer and floating-point register files. Considering average values, we find that floating-point registers are more compressible than integer registers. The following observations can be made from the plots.

Integer register compression: The average zeroth-order integer register compression ratio across all 32 registers, excluding register r0, is 0.169. We observe that integer registers r1-r7, r14, r15, r30, and r31 show potential for more compression than the rest. This can be attributed to the register windowing employed in the SPARC register architecture: r1-r7 correspond to the most often used 'global' set of registers that are more likely to be used by a program to store data; hence they show higher compression potential. Registers r14, r15,

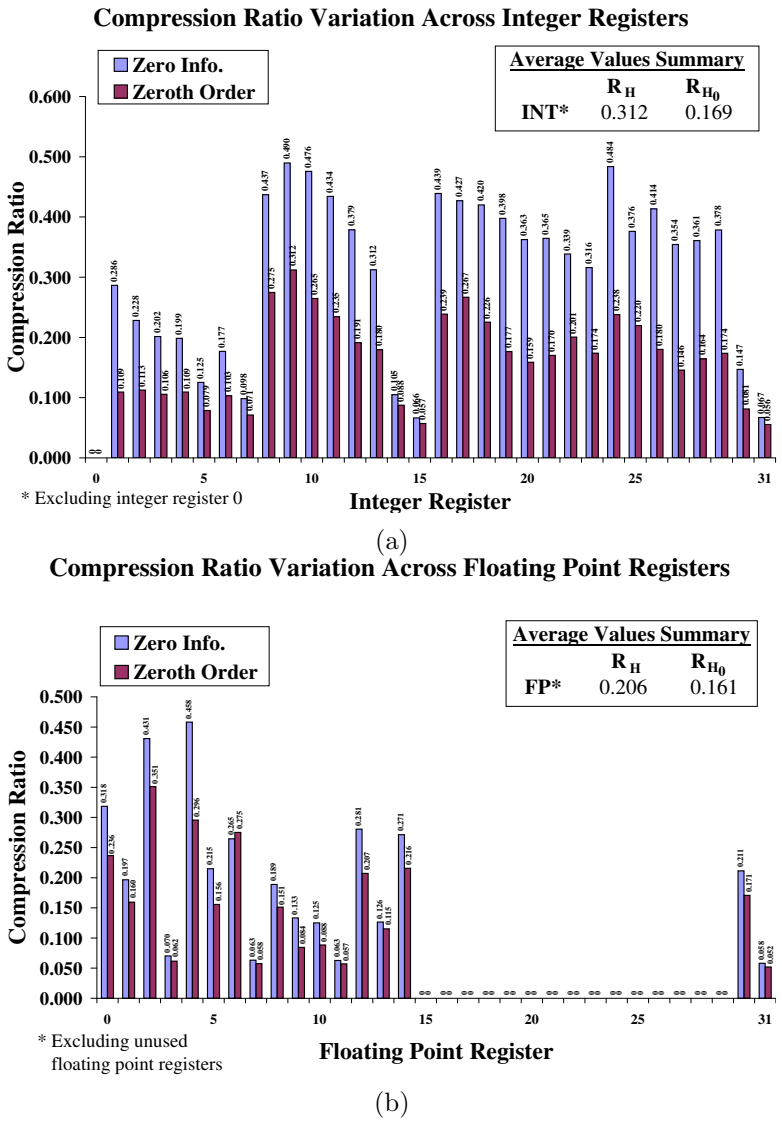


Figure 2: **Compression Potential of Storage Components – Register Compression Analysis :** (a) Average register compression analysis for 32 integer registers. (b) Average register compression analysis for 32 single-precision floating-point registers.

r30, and r31 have dedicated use as stack, frame, temporary, and return-address registers, respectively, and are also likely to be used more frequently than others. But, it may be argued that many integer registers can potentially contain pointer values[1] (32-bit addresses of other locations where data is actually stored) that can take large values and hence may be poorly compressible. But, there is indeed a lot of redundancy present in pointers because they point to roughly a similar region in memory (since they are dynamically allocated).

1. Pointers in SPARC-V9 are 32 bits. A simple C program using the sizeof(void *) functions will reveal this.

Hence many of their high-order bits will be the same resulting in higher redundancy and potential for compression.

Floating-point register compression: The average zeroth-order floating-point register compression ratio we observe for the 32 single-precision registers in SPARC-V9 is 0.161 (excluding registers f16-f29 that were all unused). Note that, as opposed to integer registers, a symbol size of 32 bits was used here to calculate entropy because only single-precision operands were considered. The substantial underutilization of the register set—13 out of 32 were not used by the benchmarks at all—can be explained by the fact that these may have been used as double or quad-precision registers which were not considered in our analysis.

In summary, our results show that although there is good amount of variation in compression ratio across registers, no register (INT or FP) has an average H_0 compression ratio exceeding about 0.35, which implies registers can, on average, be compressed to about one-third of their original size using a very good zeroth-order compression scheme.

5.3 Cache Compression Analysis Across Different Memory Levels

In this subsection, we analyze the compressibility of L1 and L2 caches. First, we explore the potential for instruction cache and data cache compression in separate experiments. Then, we investigate the effect of change in cache parameters (cache size, block size, and associativity) on compression. Finally, we estimate the benefits of cache compression in terms of improvement in cache access times, reduction in power consumption, and reduction in area.

5.3.1 Instruction and data cache compression

Fig. 3 shows results for compression ratios calculated using zero-information, zeroth-order, and first order Markov entropies for instruction and data caches. To limit the running times and memory required for this analysis, we used a smaller sample size of 20M committed instructions to collect a 100% TCRT trace. The methodology for cache trace collection was explained earlier in Sec.4.2.

Comparing instruction and data caches, we observe that data caches are more compressible. One reason for this could be the presence of data blocks with uninitialized values (mostly zeros) that add to redundancy. Comparing between L1 and L2 caches, it would be expected that L1 cache will be more compressible, if both L1 and L2 blocks are dynamically compressed with the same scheme, due to the following reason. L1 cache contains a more frequent symbol set (of instructions or data) and the L2 cache, in addition to storing the contents of L1, also contains additional symbols (instructions or data) that are relatively infrequent. This is observed to hold in the case of instruction cache, but for data caches we observe that L2 is more compressible than L1, albeit slightly (by about 3% or less). One possible explanation is that, since data is more dynamic in nature compared to instructions, blocks in L1 cache tend to be replaced more frequently. This tendency may have been aggravated by a small L1 data cache size (16KB). Both these factors result in a more dynamic mix of data in the L1 cache trace making it less compressible. As we will see later in Sec. 5.3.2, increasing cache size from 16KB to 32KB could have resulted in better compression for L1 D-cache. In contrast, due to the larger size of the L2 cache (256KB), data blocks tend to stay longer and thus the L2 data cache trace is more compressible.

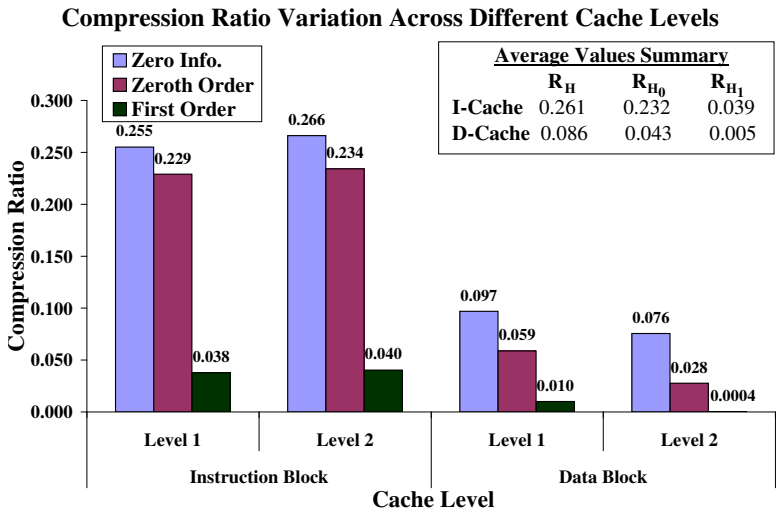


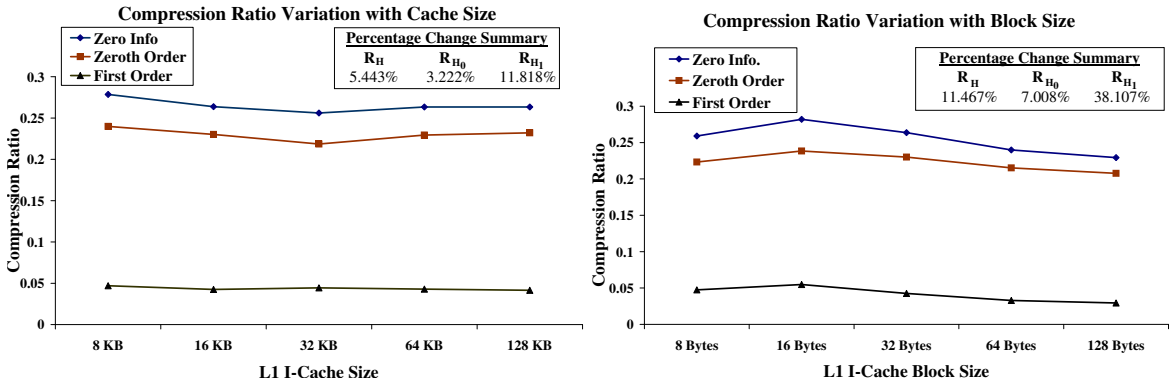
Figure 3: **Compression Potential of Storage Components – Cache Compression Analysis:** Average instruction and data cache compression analysis for L1 and L2 caches.

On average, for instruction caches, we observed a zeroth-order Markov compression ratio of about 0.23 and a first-order Markov compression ratio of about 0.04. This means that, theoretically, we could reduce instruction cache sizes by about 4 to 25 times by applying cache compression methods or store that much more information in the same area.

5.3.2 Compression ratio and cache parameters

We also investigated the sensitivity of cache compressibility to cache parameters, namely, cache size, block size, and degree of associativity and its relationship to access time, power consumption, and area. All experiments in this set were done on L1 instruction cache resident blocks using 80% and 90% TCRT traces; results are reported as the average of the two. From Fig. 4(a), we find that the compression potential of cache first increases and then decreases with increasing cache size. For the range that we studied, cache compression potential is maximum for a 32KB cache. A larger cache has more relatively infrequently occurring blocks than a smaller one, and that explains its lower compressibility. However, even for large caches, the compression ratio is very good.

In general, compression ratio improves when we increase block size as shown in Fig. 4(b). This is because a larger block has more spatially close instructions than a smaller one, and so, for the same cache size, increasing block size increases the number of instructions that are related to each other, and a smaller block size leads to more block boundaries where interruptions in related instructions occur. We also performed experiments to test the impact of varying cache set associativity on compression and we found that it has negligible impact on compression ratio.



(a) (b)

Figure 4: **Compression Potential of Caches:** (a) **Cache Compression and Cache Size:** With increasing cache size, compression ratio first improves and then deteriorates somewhat. (b) **Cache Compression and Block Size:** With increasing block size, compression ratio generally improves. Cache associativity has negligible impact on compression ratio.

Cache Type	Compression Method	Access Time		Total Energy		Tag Area		Data Area	
		(ns)	(% redn.)	(nJ)	(% redn.)	(cm^2)	(% redn.)	(cm^2)	(% redn.)
L1	Uncompressed	1.27	–	1.68	–	0.0011	–	0.0116	–
L2	Uncompressed	1.73	–	3.06	–	0.0051	–	0.1291	–
L1 I-cache	Zeroth-order	1.23	3.30	1.58	6.18	0.0006	43.75	0.0063	45.57
L1 D-cache†	Zeroth-order	0.75	40.96	0.57	68.85	0.0002	79.46	0.0017	84.61
L1 I-cache†	First-order	0.75	40.96	0.57	68.85	0.0002	79.46	0.0017	84.61
L1 D-cache†	First-order	0.73	42.15	0.57	66.10	0.0002	82.14	0.0016	86.24
L2	Zeroth-order	1.30	25.06	1.88	38.55	0.0011	77.24	0.0254	80.26
L2	First-order	1.23	28.60	1.72	43.67	0.0002	95.53	0.0017	98.61

Table 1: **Access Time, Energy Consumption, and Area of Caches:** Cache parameters obtained using the CACTI 3.0 model. Entries marked with a † use a direct-mapped organization for the compressed cache.

5.3.3 Cache compression and cache access time, energy consumption, and area

To estimate the effect of compression on other parameters like access time, power consumption, and area of the tag and data arrays, we used the CACTI 3.0 model for a 0.18 micron SRAM cache implementation [6]. Table 1 gives values of these parameters for L1 and L2 caches. Here, we compare a normal uncompressed cache with a smaller (by compression ratio) compressed cache having the same effective storage capacity. Both caches have similar parameters, such as block size and set associativity, but the compressed cache has fewer blocks (compression ratio times the number of blocks in the corresponding normal uncompressed cache). In some cases, however, the size of the compressed cache was too small (due to the compression ratio being very small) to use a set-associative mapping in CACTI 3.0. In those cases, we used a direct-mapped cache implementation. We observe that with tag

and data field compression in the compressed cache, access times can be reduced by about 41% (29%) and power consumption by about 66% (44%) on the average for L1 (L2) levels w.r.t. normal uncompressed caches with the same effective capacity.

5.4 Compression and Transition Ratios Across Individual Buses

5.4.1 Zeroth order and first-order redundancies in all buses

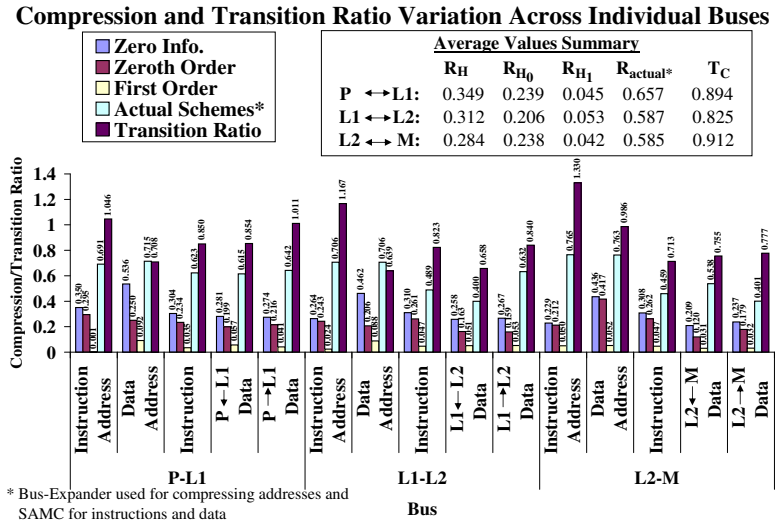
Fig. 5(a) shows compression and transition ratio results for demultiplexed buses at all three levels. We observe that the R_H and R_{H_0} values are similar across all levels. Based on R_{H_1} values, instruction address is most compressible and data address least, except for L2-M, where data is most compressible.

5.4.2 Original, XOR, and offset address trace compression

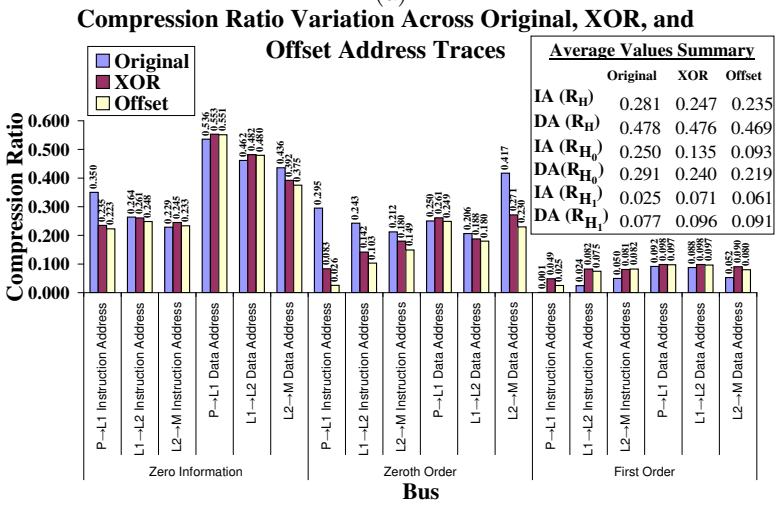
Since instruction and data addresses are known to exhibit spatial redundancy to different degrees, it would be expected that the XOR of consecutive addresses will have a lot of zeros (especially at the high-order bit positions) and that the offset values for consecutive addresses will have small magnitudes. Note that computing bitwise XOR of two n-bit addresses requires constant time and little hardware and offsets can be computed in $O(\log N)$ time using a carry-lookahead tree adder. However, XOR traces have a power disadvantage. Every bit transition in the original trace will cause two bit transitions in the XOR trace, except when consecutive transitions occur in the original trace (not likely), in which case there will not be any transition in the XOR trace. To study the compressibility of original, XOR, and offset address traces, we evaluated their zero-information, zeroth-order, and first-order Markov compression ratios; these are shown in Fig. 5(b). Since instruction addresses occur at some very frequent offsets (typically an instruction word), the zero-information and zeroth-order Markov compression ratios for instruction address offset traces is the best and even the XOR trace has better compressibility than the original trace. However, when considering first-order Markov compression, the original trace provides the best compression and the offset trace the worst. This is expected since, given an offset, the next offset value can vary depending upon the instructions being executed at the time. However, given an instruction address, the next instruction address can be easily predicted. In the case of data addresses, XOR and offset traces do not necessarily give better compression ratios due to more variation in data addresses issued.

5.5 Compression Ratio and Bit Fields

In this experiment, we consider eight consecutive bit fields (from high to low order: F7, F6, ..., F1, F0) corresponding to each nibble for instruction and data addresses. For 64-bit data, we consider four consecutive bit fields (F3, F2, F1, and F0) corresponding to each half-word (16 bits). For 20-bit I-cache tag, we consider five fields each a nibble wide. For 32-bit instructions, we consider six fields (F5, F4, F3, F2, F1, F0 of widths 2, 5, 6, 5, 9, and 5 bits, respectively) based on the field boundaries of the most common instruction format (J-Format) in SPARC-V9 architectures. In the experiments under this subsection, the symbol size for compression corresponds to the above-mentioned bit field sizes. We generated individual bit-field traces for data addresses and instruction addresses at the



(a)

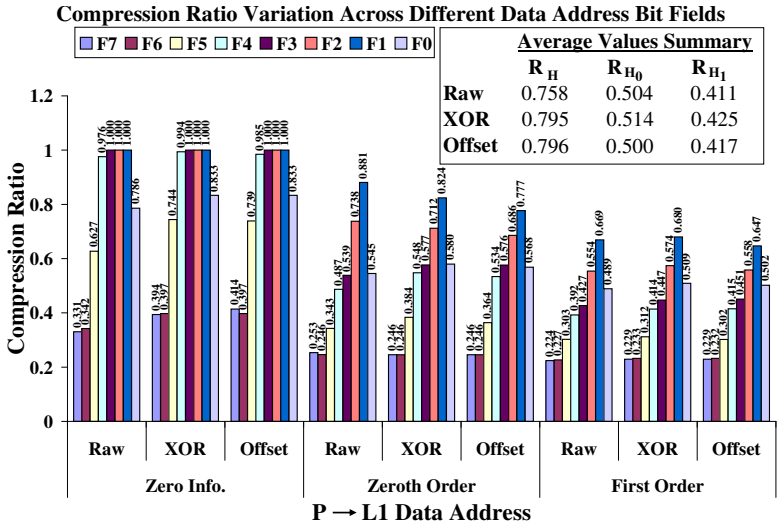


(b)

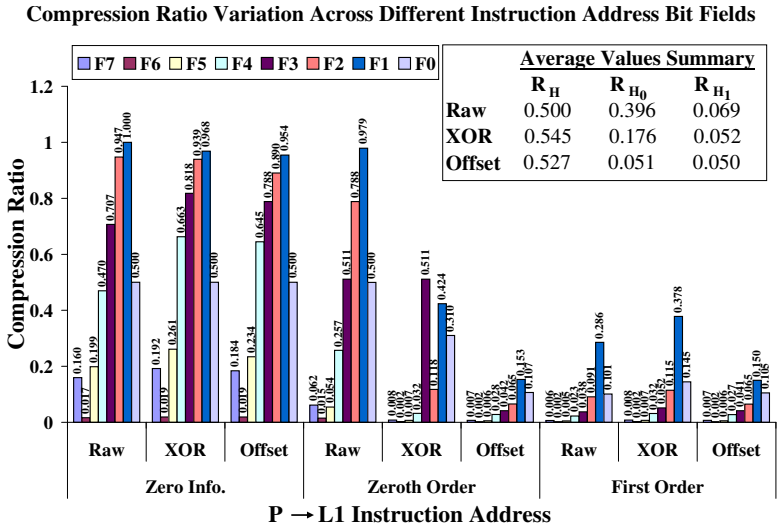
Figure 5: **Compression Potential of Communication Components:** (a) Zero-information and zeroth- and first-order compression ratios for various buses at different levels of the memory system hierarchy. (b) Compression ratios for original, XOR, and offset address traces for various address buses.

P→L1 level, instructions and data at the L1→L2 level, and tag field of L1 I-cache and then analyzed each trace by doing zeroth and first-order Markov analysis. We also considered three different representations for each bit-field stream in addresses: original (raw), XOR-encoded, and offset-encoded. The motivation for studying these address representations was described earlier in Sec. 5.4.2.

From the results shown in Figs. 6 and 7, we observe that compression ratio varies across bit-fields and the variation differs for each type of traffic. In general, across all types of information, we observe that compressibility improves from low to high order bit fields, except in the case of instruction bus traffic. Comparing data addresses and instruction



(a)



(b)

Figure 6: **Compression Ratio and Bit Fields—Data and Instruction Addresses:** (a) Variation of compression ratio across data address bit-fields. (b) Variation of compression ratio across instruction address bit-fields. In both (a) and (b), higher order bit fields show best compression.

addresses (Figs. 6(a) and (b)), we observe the following. First, instruction addresses are more compressible than data addresses. Second, zeroth- and first-order compression of bit-fields yield more returns in instruction addresses than in data addresses. Third, offsets and XORs of instruction addresses are more compressible with higher-order compression schemes.

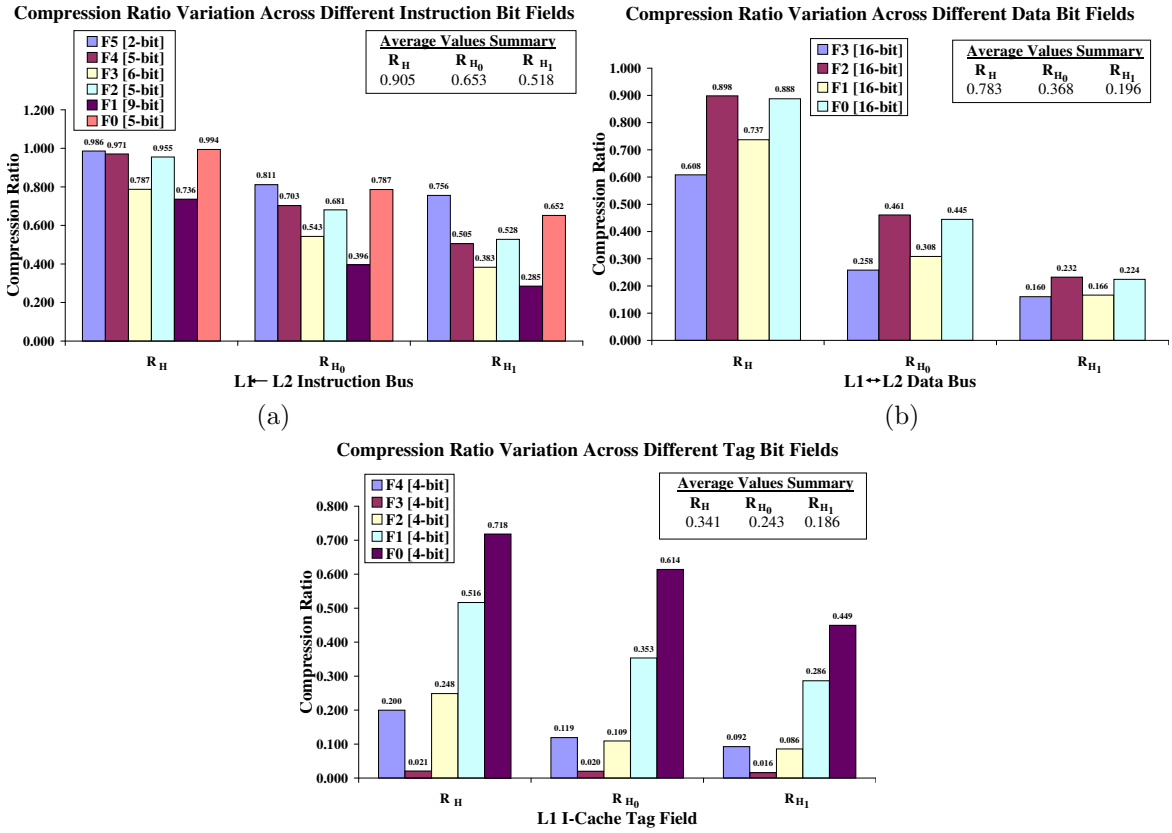
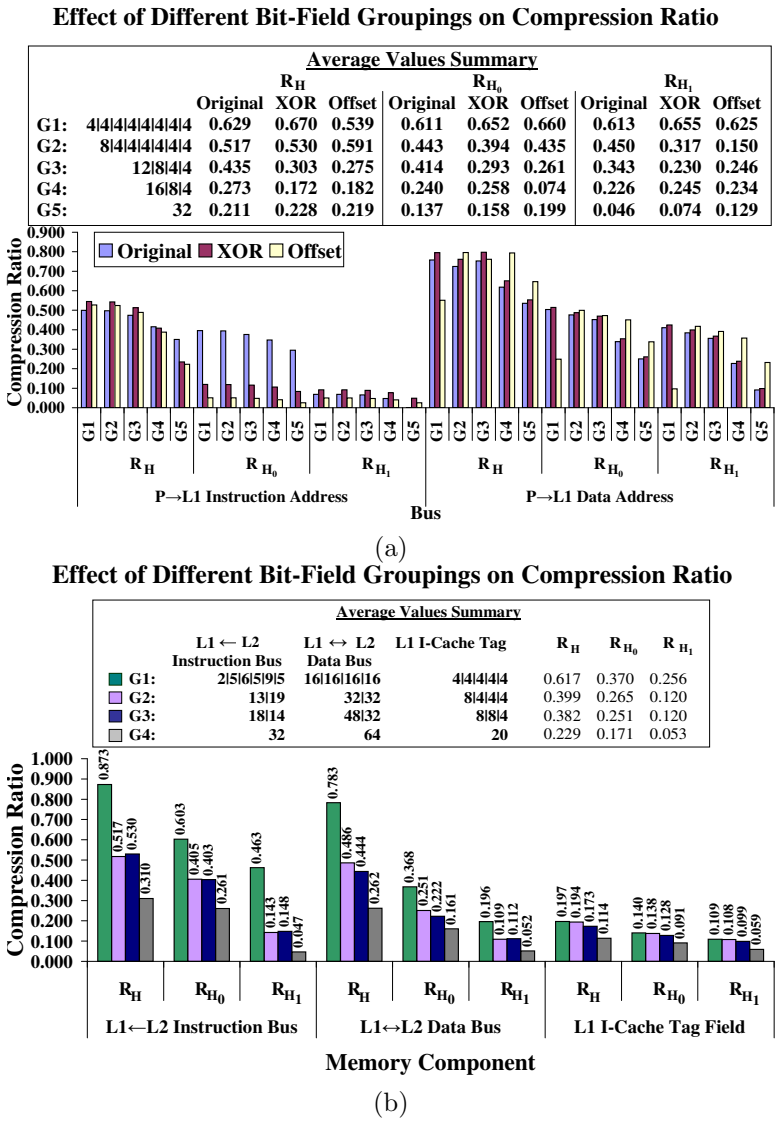


Figure 7: **Compression Ratio and Bit Fields—Instruction, Data, and Tag:** (a) Variation of compression ratio across instruction bit-fields. (b) Variation of compression ratio across data bit-fields. (c) Variation of compression ratio across tag bit-fields.

5.6 Compression Ratio and Bit-Field Groupings

In the previous subsection, we investigated the compressibility of individual bit fields in a word. In this subsection, we evaluate the compressibility of an entire word based on different groupings of bit fields. For this analysis, we considered five bit-field groupings for addresses that are indicated in the top right corner of Fig. 8(a): Group-1 (G1) consists of 8 nibbles with each compressed separately, Group-2 (G2) consists of a most significant byte followed by 6 nibbles, Group-3 (G3) comprises a most significant part of 12 bits followed by a byte and then two nibbles, Group-4 (G4) consists of a most significant half-word, a byte, and then a nibble, and finally Group-5 (G5) considers the whole word as a symbol. In a similar vein, the bit-field groupings that we considered for instruction, data, and cache tag fields are shown in Fig. 8(b). The entropy value for the entire word is equal to the sum of the entropies for the individual bit-fields.

For addresses only, we considered original, XOR-encoded, and offset-encoded values for compression separately. We observe the following from the results shown in Fig. 8. In general, for any type of information, the more the number of bits in the higher order field, the better the overall compression ratio. When we consider the whole word as a



(a)

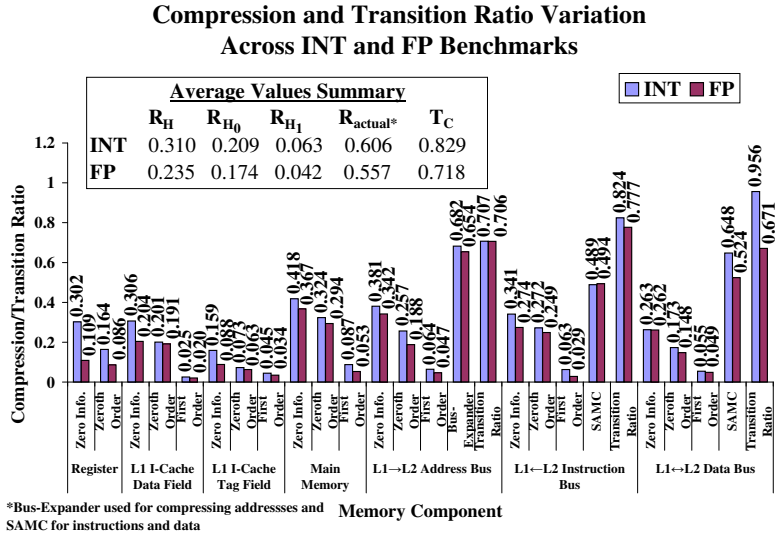
(b)

Figure 8: **Compression Ratio and Bit-Field Groupings:** Variation of compression ratio across different bit-field groupings. (a) Address buses. (b) Instruction and data buses and cache tag fields.

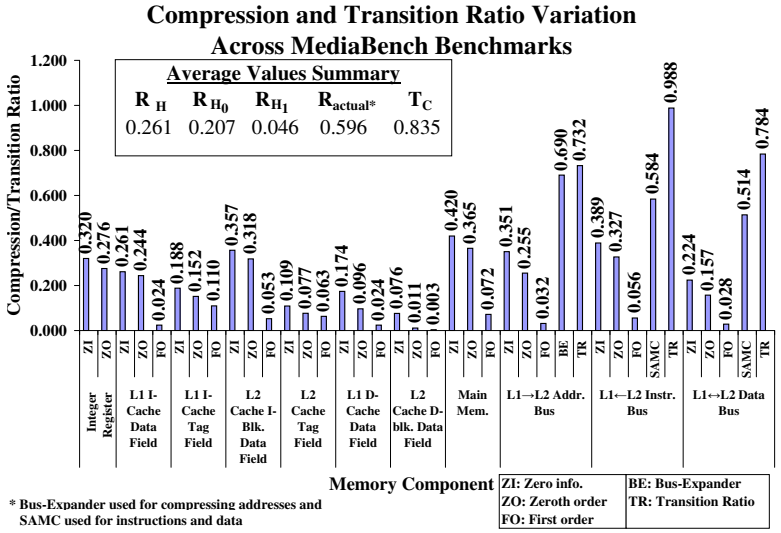
symbol (G5 for addresses and G4 for others), the best compression ratio is obtained. In the case of instruction addresses, we find that XOR-encoded and offset-encoded values, in most cases, perform worse than original values for zero-info and first-order compression. However, for zeroth-order compression, these perform substantially better than original values. This is because the same XOR or offset values repeat for different combinations of original addresses, thus resulting in higher zeroth-order compression.

5.7 Compression Ratio and Power Savings for Different Workloads

Results for experiments reported in previous subsections were averaged over all benchmarks. In this experiment, we compare the compression potential and power savings due to compression of different workloads: integer, floating-point, and embedded. The results of this experiment are shown in Figs. 9(a) and (b) for SPEC CPU2000 and MediaBench programs, respectively.



(a)



(b)

Figure 9: **Application Class Analysis:** Compression ratio and power savings variation across different application classes (a) Desktop/workstation class workloads (SPEC CPU2000 INT and FP programs). (b) Embedded workloads (MediaBench programs).

The following observations can be made for desktop/workstation class workloads represented by the SPEC CPU2000 benchmark programs. As seen earlier in Sec. 5.2, for this

type of workload, data in floating-point registers are more compressible than data in integer registers. For program instructions (stored in I-cache data field and main memory and transmitted on instruction bus) and addresses (in I-cache tag field and instruction address bus), we observe that the information for the FP application class is more compressible than the INT application class. We also see that the FP data sent over the data bus is more compressible than the INT data sent over the same bus. This may be because the FP data blocks sent from L2 to L1 (in the event of an L1 D-cache miss) may contain many unused FP words that are set to zero giving rise to redundancy of information. We also observe that for communication components, FP programs give better power savings than INT programs. For embedded workloads, represented by MediaBench programs, compressibilities are intermediate between integer and floating-point programs.

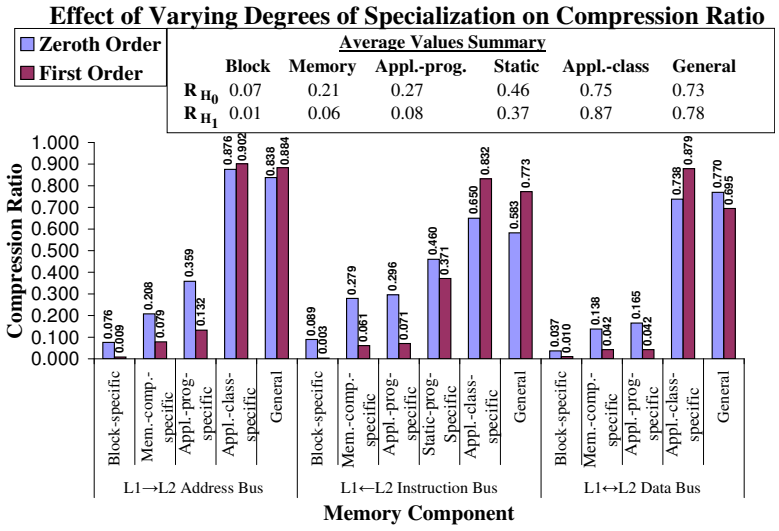
5.8 Compression Ratio and Degree of Specialization

In this experiment, we investigate how varying degrees of specialization of the compression scheme affect compression ratio. We set up five different types of specialization as mentioned in Sec. 2.2. In the *benchmark-specific* architecture, the compression scheme is specific to each benchmark, but same for all blocks and memory components. For this, symbol statistics used for compression of any trace are determined by analyzing symbols from all memory components. In the *application-class-specific* case, symbol statistics for various components for a subset of benchmarks, the training benchmarks, for each application class (INT or FP) are determined and then these statistics are used to compress components for the remaining test-benchmarks in the same application class. To limit the simulation time and memory required for this study, we limited the sample size used for trace collection to 10M instructions. Here, we show separate results for INT and FP.

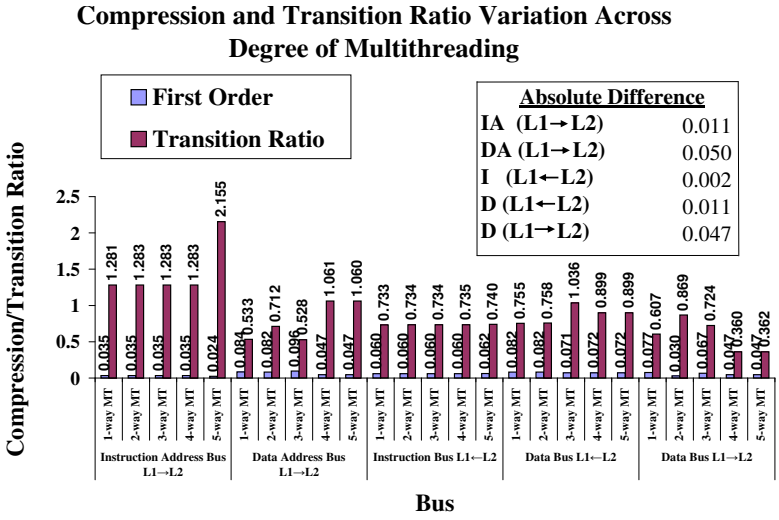
We observe from results in Fig. 10 that with the degree of specialization decreasing, the compression ratio deteriorates. But we observe that compressibility with a general compression architecture is slightly better than an application-class-specific architecture although the former is less specialized than the latter. The general case that we considered here is very similar to the application-specific-class and the only difference is that it draws statistics from all application classes combined. Since the number of distinct application classes considered in our analysis is only two (INT and FP classes–MediaBench programs can be considered to be in the INT class), the general case does not result in worse compression than the application-specific class. For the first four cases, first-order Markov performs better than zeroth-order Markov. But in the application-class-specific case, it is the opposite. This is because for symbols that occur in both test benchmarks and training benchmarks, symbols are compressed according to statistics in training benchmarks in zeroth-order Markov, but if their preceding symbols do not occur in training benchmarks, the symbol is left uncompressed in first-order Markov and this results in worse compression for first-order Markov.

5.9 Compression Ratio and Multithreaded Execution

In a multithreaded system, if a shared (address, instruction, or data) bus is used across different threads, compression and transition ratios may be different compared to single-threaded systems. We simulated the effect of k-way multithreading by merging address, instruction, or data traces from k different benchmarks and creating a single trace (address,



(a)



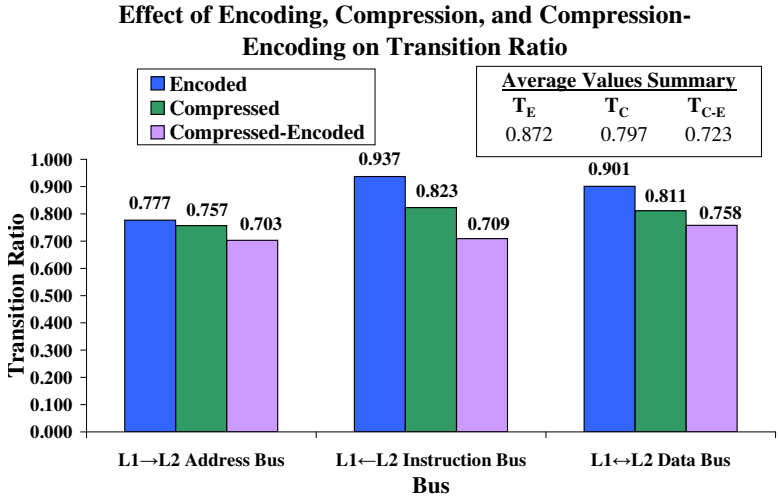
(b)

Figure 10: **Degree of Specialization and Degree of Multithreading Analysis:** (a) Compression ratio variation with degree of specialization. (b) Compression ratio variation with the degree of multithreading.

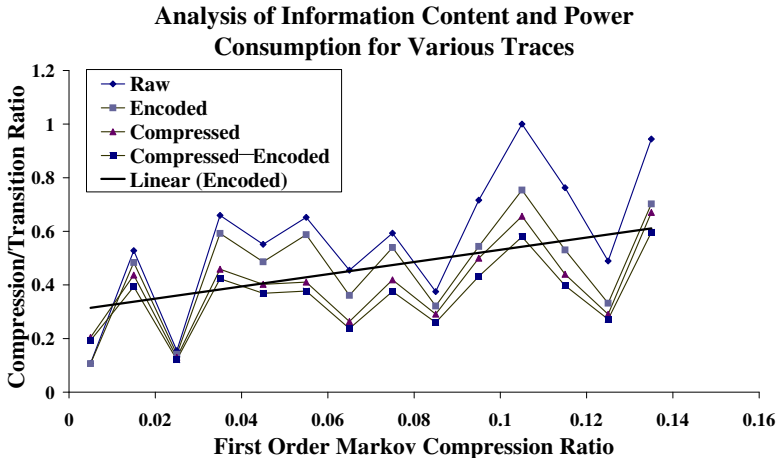
instruction, or data) by ordering the references according to their timestamps. We report results for first-order compression, which is the best as observed from earlier experiments, and transition ratio in Fig. 10(b) for the multithreaded trace. With multithreading, we expect that, because of intermingling of traffic from different threads, more transitions will occur. The results shown in Fig. 10(b) suggest that this is somewhat true, although, transitions often do not increase by much when the degree of multithreading is increased from one to five. Multithreading also does not seem to have a perceptible impact on first-order compression ratios.

5.10 Power Savings Due to Compression, Encoding, and Both Combined

Some experiments above demonstrated that power savings can be achieved with compression alone. We wanted to investigate if bus encoding, compression, or both applied together decrease power consumption further. So, we conducted experiments for the three cases and the results are shown in Fig. 11(a). We found that by using compression and encoding together, we could achieve the best power savings. In fact, on the average, compared to the reduction in transitions due to encoding alone, compression reduces transitions by further 7.5% and compression followed by encoding reduces transitions by further 15%. Thus, a scheme that combines both compression and encoding can provide the best benefits in terms of energy efficiency.



(a)



(b)

Figure 11: **Communication Component Analysis Considering Bus Encoding and Compression:** (a) The extent of power saving due to encoding, compression, and compression and encoding combined. Compression followed by encoding shows best results. (b) The effect of information content of a trace on its power consumption.

In another experiment, we investigated the effect of information content on the power consumption of a particular trace when it is transmitted on a bus. To study this, we grouped all bus traces that we used (address, instruction, and data traces) according to their first-order compression ratio (information content). We used first-order compression ratio since it has the lowest value for all traces and hence it represents the lower bound for compression. Traces with compression ratios in the range (0, 0.1] were placed in one group, those in (0.1, 0.2] in another, and so on until the last group which had traces with compression ratios in the range (0.9, 1.0]. After grouping the traces, we calculated the average number of transitions for each group (total number of transitions in all traces in a group divided by the number of traces in the group) for the original, compressed, encoded, and compressed-encoded versions of the traces in the group. Then, we normalized this number using the trace with the highest number of transitions in each group. We also calculated the mean of the compression ratios of traces in each group. Finally, we plotted the normalized average transitions for each group against the mean compression ratio; the plot is shown in Fig. 11(b). It shows that, for a given trace, the number of transitions increase with information content, although, for a given information content (compression ratio), the compressed-encoded and compressed traces cause fewer transitions.

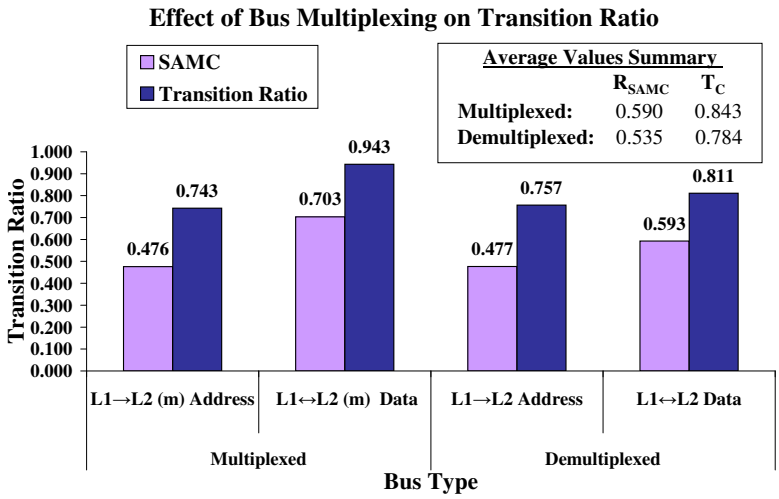
5.11 Other Issues

5.11.1 Power Savings and Bus Multiplexing

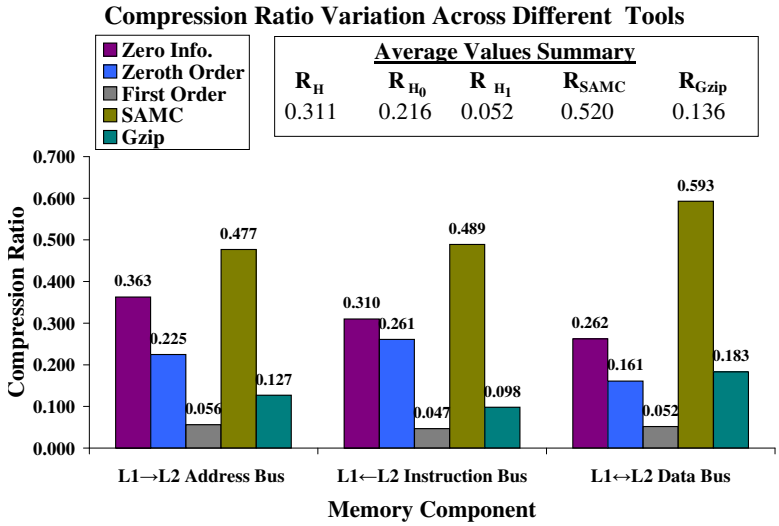
The default bus in our experiments was the demultiplexed bus, and so we also wanted to know how multiplexing affects power consumption. As mentioned earlier, a multiplexed address bus means that both instruction and data addresses are carried on the same bus. Similarly, a multiplexed data bus means that both instructions and data are carried on the same bus. We compared multiplexed and demultiplexed address and data buses and obtained results as shown in Fig. 12(a). While multiplexing an address bus slightly improves both the address compression ratio and power savings, it degrades both in a data bus by a non-negligible amount. This shows that there is sufficient redundancy in multiplexed address streams whereas it is not true for combined data/instruction streams. For data/instruction buses, the degree of specialization of the compression scheme on demultiplexed bus is higher than multiplexed bus. On demultiplexed bus, compression is specific to each trace (instruction, data from L1 to L2, data from L2 to L1, etc.), but on the multiplexed bus, the compression scheme is used for all content on the bus consisting of instruction and data traffic (both directions). This also accounts for lesser compression and power savings on the multiplexed data/instruction bus. Thus, in spite of multiplexed traffic on address buses, benefits can be obtained but the same is not true for data buses.

5.11.2 Compression Ratio and Analysis Tool

SAMC, an arithmetic compression scheme, does not approach the entropy bound, but provides a decent compression ratio of 0.48–0.59 as shown in Fig. 12(b). Among available compression tools, SAMC performs much worse than the commonly used text compression utility *Gzip*, that uses dictionary compression methods. It is also noticeable that there is a wide gap (almost an order of magnitude) between the theoretically achievable compres-



(a)



(b)

Figure 12: **Other Issues:** (a) Compression and transition ratio variation with multiplexed traffic. (b) Compression ratio variation across different compression measures and tools.

sion bound (zeroth and first-order entropies) and that achieved by existing compression techniques such as SAMC or Gzip.

6. Conclusion

In this paper, we presented a comprehensive analysis of all three primary types of information (addresses, instructions, and data) stored and transmitted by the storage and communication components, respectively, at various levels of the memory system hierarchy. The analysis was done in terms of the compression ratio possible, which in turn reflects the amount of performance (storage capacity and bandwidth) and to some extent cost improvements attainable using compression. Our analysis was done on programs from the SPEC

CPU2000 integer and floating-point and MediaBench suites. We have shown that a substantial amount of information redundancy exists in every component of the memory system, such as registers, tag and data fields of caches, main memory (storage components) and also in address, instruction, and data buses (communication components). We should note here that our results represent theoretical limits on compression possible and that practical schemes will only achieve a fraction of these limits. However, as noted earlier in Sec. 5.11.2, the compressibility achieved by current schemes is an order of magnitude or more away from these limits.

Some important results from our analysis are as follows. We observed that information stored in the memory system can be compressed to at least 39% of its original size with ideal zero-information compression schemes and to about 31% with ideal zeroth-order compression schemes. Information transmitted in the memory system through buses was found to be more compressible on the average. We found that by compressing tag and data fields, cache access times can be reduced by about 41% (29%) and power consumption by about 66% (44%) on the average for L1 (L2) levels w.r.t. normal uncompressed caches with the same effective capacity. Also, both tag and data areas of caches can be substantially reduced by compression. Other conclusions from our analysis are as follows: (1) Among storage components, data caches were more compressible compared to instruction caches, and cache size and block size affected compression ratios; (2) among communication components, the level of the memory hierarchy where the component is present, different bit fields, and bit-field groupings play a part in determining the amount of compression that is possible; and (3) compression ratio also depends on the degree of specialization of the compression scheme. We also studied the compressibility of original, XOR, and offset instruction and data address traces, the effect of application class, encoding and multiplexing, analysis tool, static vs. adaptive/dynamic compression, multithreading, and the relationship between information content, compression ratio, and power consumption.

Acknowledgements

We thank the Center for Computational Research (CCR) at the University at Buffalo, The State University of New York for providing us access to their high-performance computers. We also thank the anonymous reviewers for their comments.

References

[1] D. Culler, J. Singh, and A. Gupta, *Parallel Computer Architecture*. Morgan Kaufmann Publishers Inc., 1999.

[2] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach, Third edition*. Morgan Kaufmann Publishers Inc., 2003.

[3] P. Sotiriadis and A. Chandrakasan, "Low Power Bus Coding Techniques Considering Inter-Wire Capacitances," in *Proceedings of Custom Integrated Circuits Conference*, pp. 414–419, May 2000.

[4] W. Wolf, *Computers as Components: Principles of Embedded Computing System Design*. Morgan Kaufmann Publishers Inc., 2001.

[5] B. Cmelik and D. Keppel, "SHADE: A Fast Instruction-Set Simulator for Execution Profiling," *ACM SIGMETRICS Performance Evaluation Review*, vol. 22, pp. 128–137, May 1994.

[6] P. Shivakumar and N. Jouppi, "CACTI 3.0: An Integrated Cache Cycle Timing, Power, and Area Model," Tech. Rep. WRL Research Report 2001/2, Compaq Western Research Laboratory, Aug. 2001.

[7] Y. Zhang, R. Y. Chen, W. Ye, and M. Irwin, "System Level Interconnect Power Modeling," in *IEEE International ASIC/SoC Conference*, pp. 289–293, Sept. 1998.

[8] A. Park and M. Farrens, "Address Compression through Base Register Caching," in *Proceedings of the Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 193–199, Nov. 1990.

[9] M. Farrens and A. Park, "Dynamic Base Register Caching: A Technique for Reducing Address Bus Width," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 128–137, May 1991.

[10] D. Citron and L. Rudolph, "Creating a Wider Bus using Caching Techniques," in *Proceedings of International Symposium on High Performance Computer Architecture*, pp. 90–99, Jan. 1995.

[11] J. Liu, N. Mahapatra, and K. Sundaresan, "Dynamic Address Compression Schemes: A Performance, Energy, and Cost Study," in *Proceedings of IEEE International Conference on Computer Design*, pp. 458–464, Oct. 2004.

[12] D. Hammerstrom and E. Davidson, "Information Content of CPU Memory Referencing Behavior," in *Proceedings of the 4th Annual Symposium on Computer Architecture*, pp. 184–192, ACM Press, 1977.

[13] J. Becker, A. Park, and M. Farrens, "An Analysis of the Information Content of Address Reference Streams," in *Proceedings of the International Conference on Microarchitecture*, pp. 19–24, Nov. 1991.

[14] J. Wang and R. Quong, "The Feasibility of Using Compression to Increase Memory System Performance," in *Proceedings of International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems*, pp. 107–113, Jan. 1994.

[15] M. Kozuch and A. Wolfe, "Compression of Embedded System Programs," in *Proceedings of International Conference on Computer Design*, pp. 270–277, Oct. 1994.

[16] M. Kjelso, M. Gooch, and S. Jones, "Empirical Study of Memory-Data: Characteristics and Compressibility," *IEE Proceedings on Computers and Digital Techniques*, vol. 145, pp. 63–67, Jan. 1998.

[17] J. Liu, N. Mahapatra, K. Sundaresan, S. Dangeti, and B. Venkatrao, "Memory System Compression and Its Benefits," in *Proceedings of the 15th Annual IEEE International ASIC/SOC Conference*, pp. 41–45, Sept. 2002.

[18] N. Mahapatra, J. Liu, K. Sundaresan, S. Dangeti, and B. Venkatrao, "The Potential of Compression to Improve Memory System Performance, Power Consumption, and Cost," in *Proceedings of IEEE Performance, Computing and Communications Conference*, pp. 343–350, Apr. 2003.

[19] D. Citron, "Exploiting Low Entropy to Reduce Wire Delay," *Computer Architecture Letters*, vol. 3, Jan. 2004.

[20] K. Basu, A. Choudhary, J. Pisharath, and M. Kandemir, "Power Protocol: Reducing Power Dissipation on Off-Chip Data Buses," in *Proceedings of the Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 19–24, Nov. 2002.

[21] K. Kant and R. Iyer, "Design and Performance of Compressed Interconnects for High Performance Servers," in *Proceedings of International Conference on Computer Design*, pp. 164–169, Oct. 2003.

[22] C. Fraser, E. Myers, and A. Wendt, "Analyzing and Compressing Assembly Code," *SIGPLAN Notices*, vol. 19, pp. 117–121, June 1984.

[23] J. Ernst, W. Evans, C. Fraser, S. Lucco, and T. Proebsting, "Code Compression," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 358–365, June 1997.

[24] D. Kirovski, J. Kin, and W. Mangione-Smith, "Procedure Based Program Compression," in *Proceedings of the Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 204–213, Dec. 1997.

[25] K. Cooper and N. McIntosh, "Enhanced Code Compression for Embedded RISC Processors," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 139–149, May 1999.

[26] S. Debray, W. Evans, R. Muth, and B. de Sutter, "Compiler Techniques for Code Compaction," *Transactions on Programming Languages and Systems*, vol. 22, pp. 378–415, Mar. 2000.

[27] A. Wolfe and A. Channin, "Executing Compressed Programs on an Embedded RISC Architecture," in *Proceedings of the Annual Symposium on Computer Architecture*, pp. 81–91, Dec. 1992.

[28] S. Liao, S. Devadas, and K. Keutzer, "Code Density Optimization for Embedded DSP Processors Using Data Compression Techniques," in *Proceedings of Conference on Advanced Research in VLSI*, pp. 393–399, Mar. 1995.

[29] C. Lefurgy and T. Mudge, "Code Compression for DSP," Tech. Rep. CSE-TR-380-98, EECS Department, University of Michigan, Ann Arbor, MI, 1998.

[30] H. Lekatsas and W. Wolf, "Random Access Decompression using Binary Arithmetic Coding," in *Prodeedings of Data Compression Conference*, pp. 306–315, Mar. 1999.

[31] H. Lekatsas, J. Henkel, and W. Wolf, "Code Compression for Low Power Embedded System Design," in *Proceedings of Annual ACM/IEEE Design Automation Conference*, pp. 294–299, June 2000.

[32] T. Kemp, R. Montoye, J. Harper, J. Palmer, and D. Auerbach, "A Decompression Core for PowerPC," *IBM Journal of Research and Development*, vol. 42, pp. 807–811, Nov. 1998.

[33] M. Game and A. Booker, "Codepack™: Code Compression for PowerPC Processors." `http://www-3.ibm.com/chips`, May 2000.

[34] K. Sundaresan and N. Mahapatra, "Code Compression Techniques for Embedded Systems and Their Effectiveness," in *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, pp. 262–263, Feb. 2003.

[35] T. M. Conte, S. Banerjia, S. Y. Larin, K. N. Menezes, and S. W. Sathaye, "Instruction Fetch Mechanisms for VLIW Architectures with Compressed Encodings," in *Proceedings of the Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 201–211, Dec. 1996.

[36] Y. Xie, W. Wolf, and H. Lekatsas, "Code Compression for VLIW Using Variable-to-Fixed Coding," in *Proceedings of the International Symposium on System Synthesis*, pp. 138–143, Oct. 2002.

[37] J. Hoogerbrugge, L. Augusteijn, J. Trum, and R. van de Wiel, "A Code Compression System Based on Pipelined Interpreters," *Software Practice and Experience*, vol. 29, no. 11, pp. 1005–1023, 1999.

[38] Y. Yoshida, B. Y. Song, H. Okuhata, T. Onoye, and I. Shirakawa, "An Object Code Compression Approach to Embedded Processors," in *Proceedings of International Symposium on Low Power Electronics and Design*, pp. 265–268, Aug. 1997.

[39] L. Benini, G. D. Micheli, E. Macii, and M. Poncino, "Selective Instruction Compression for Memory Energy Reduction in Embedded Systems," in *Proceedings of International Symposium on Low Power Electronics and Design*, pp. 206–211, Aug. 1999.

[40] I. Kadayif and M. Kandemir, "Instruction Compression and Encoding for Low-Power Systems," in *Proceedings of the IEEE International ASIC/SOC Conference (ASIC/SOC'02)*, pp. 301–305, Sept. 2002.

[41] Advanced RISC Machines Ltd (ARM), *An Introduction to Thumb*, Mar. 1995. `http://www.arm.com`.

[42] K. Kissell, "MIPS16: High-density MIPS for the Embedded Market." `http://www.mips.com`, 1997.

[43] R. B. Tremaine, P. Franaszek, J. Robinson, C. Schulz, T. Smith, M. Wazlowski, and P. Bland, "IBM Memory eXpansion Technology (MXT)," *IBM Journal of Research and Development*, vol. 45, pp. 271–285, Mar. 2001.

[44] P. Franaszek and J. Robinson, "Design and Analysis of Internal Organizations for Compressed Random Access Memories," Tech. Rep. IBM Research Report RC 21146(94535)20OCT98, IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY, Oct. 1998.

[45] J.-S. Lee, W.-K. Hong, and S.-D. Kim, "Design and Evaluation of a Selective Compressed Memory System," in *Proceedings of International Conference on Computer Design*, pp. 184–191, Oct. 1999.

[46] J. Yang, Y. Zhang, and R. Gupta, "Frequent Value Compression in Data Caches," in *Proceedings of the Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 258–265, Nov. 2000.

[47] L. Villa, M. Yang, and K. Asanovic, "Dynamic Zero Compression for Cache Energy Reduction," in *Proceedings of the Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 214–220, Dec. 2000.

[48] A. Alameldeen and D. Wood, "Adaptive Cache Compression for High-Performance Processors," in *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA'04)*, pp. 212–222, IEEE Computer Society, 2004.

[49] E. Hallnor and S. Reinhardt, "A Unified Compressed Memory Hierarchy," in *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA-11)*, pp. 201–212, IEEE Computer Society, 2005.

[50] C. Su, C. Tsui, and A. Despain, "Low Power Architecture Design and Compilation Techniques for High-Performance Processors," Tech. Rep. ACAL-TR-94-01, Advanced Computer Architecture Laboratory, University of Southern California, 1994.

[51] M. Stan and W. Burleson, "Bus-Invert Coding for Low-Power I/O," *IEEE Transactions on VLSI Systems*, vol. 3, pp. 49–58, Mar. 1995.

[52] L. Benini, G. D. Micheli, E. Macii, D. Sciuto, and C. Silvano, "Asymptotic Zero-Transition Activity Encoding for Address Busses in Low-Power Microprocessor-Based Systems," in *Proceedings of Great Lakes Symposium on VLSI*, pp. 77–82, Mar. 1997.

[53] E. Musoll, T. Lang, and J. Cortadella, "Working-Zone Encoding for Reducing the Energy in Microprocessor Address Buses," *IEEE Transactions on VLSI Systems*, vol. 6, pp. 568–572, Dec. 1998.

[54] T. Lang, E. Musoll, and J. Cortadella, "Extension of the Working-Zone Encoding Method to Reduce the Energy on the Microprocessor Data Bus," in *Proceedings of International Conference on Computer Design*, pp. 414–419, Oct. 1998.

[55] W.-C. Cheng and M. Pedram, "Memory Bus Encoding for Low-Power: A Tutorial," in *Proceedings of International Symposium on Quality of Electronics Design*, pp. 199–204, Mar. 2001.

[56] Y. Aghaghiri, F. Fallah, and M. Pedram, "Irredundant Address Bus Encoding for Low Power," in *Proceedings of International Symposium on Low Power Electronics and Design*, pp. 322–327, ACM Press, NY, USA, Aug. 2001.

[57] J. Henkel and H. Lekatsas, "A^2BC: Adaptive Address Bus Coding for Low-Power Deep Sub-Micron Designs," in *Proceedings of Annual ACM/IEEE Design Automation Conference*, pp. 744–749, June 2001.

[58] P. Sotiriadis and A. Chandrakasan, "Reducing Bus Delay in Sub-Micron Technology Using Coding," in *Proceedings of Asia and South Pacific Design Automation Conference*, pp. 109–114, Jan. 2001.

[59] B. Victor and K. Keutzer, "Bus Encoding to Prevent Crosstalk Delay," in *Proceedings of IEEE International Conference on Computer-Aided Design*, pp. 57–63, Nov. 2001.

[60] SPEC, "SPEC CPU2000 Benchmark Suite Ver1.2." `http://www.specbench.org/cpu2000`, 2000.

[61] C. Lee, M. Potkonjak, and W. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems," in *Proceedings of the Annual Symposium on Computer Architecture*, pp. 330–335, June 1997.

[62] K. Skadron, P. Ahuja, M. Martonosi, and D. Clark, "Selecting a Single, Representative Sample for Accurate Simulation of SPECint Benchmarks," *IEEE Transactions on Computers*, vol. 48, pp. 1260–1281, Nov. 1999.

[63] H. Lekatsas and W. Wolf, "SAMC: A Code Compression Algorithm for Embedded Processors," *IEEE Transactions on Computer-aided Design*, vol. 18, pp. 1689–1701, Dec. 1999.

[64] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. Irwin, "The Design and Use of Simplepower: A Cycle-Accurate Energy Estimation Tool," in *Proceedings of Annual ACM/IEEE Design Automation Conference*, pp. 340–345, June 2000.

[65] D. Weaver and T. Germond, eds., *The SPARC Architecture Manual, Version 9*. Prentice Hall, 2000.