Residue Cache: a Low-Energy Low-Area L2 Cache Architecture via Compression and Partial Hits

Soontae Kim
KAIST
335 Gwahangno Yuseong-gu
Daejeon Korea
kims@kaist.ac.kr

Jesung Kim
LG Electronics
Gasan-dong Geumchun-gu
Seoul Korea
jesung.kim@lge.com

Jongmin Lee
KAIST
335 Gwahangno Yuseong-gu
Daejeon Korea
square55@kaist.ac.kr

Seokin Hong
KAIST
335 Gwahangno Yuseong-gu
Daejeon Korea
seokin@kaist.ac.kr

ABSTRACT

L2 cache memories are being adopted in the embedded systems for high performance, which, however, increases energy consumption due to their large sizes. We propose a low-energy low-area L2 cache architecture, which performs as well as the conventional L2 cache architecture with 53% less area and around 40% less energy consumption. This architecture consists of an L2 cache and a small cache called ***residue cache***. L2 and residue cache lines are half sized of the conventional L2 cache lines. Well compressed conventional L2 cache lines are stored only in the L2 cache while other poorly compressed lines are stored in both the L2 and residue caches. Although many conventional L2 cache lines are not fully captured by the residue cache, most accesses to them do not incur misses because not all their words are needed immediately, which are termed as ***partial hits*** in this paper. The residue cache architecture consumes much lower energy and area than conventional L2 cache architectures, and can be combined synergistically with other schemes such as the line distillation and ZCA. The residue cache architecture is also shown to perform well on a 4-way superscalar processor typically used in high performance systems.

Categories and Subject Descriptors

B.3.2 [**Design Styles**]: [Cache Memories]

General Terms

Performance, Design, Experimentation

Keywords

Energy, Area, L2 cache

MICRO'11 December 3-7, 2011, Porto Alegre, Brazil

1. INTRODUCTION

The embedded/portable computing systems are becoming more powerful to increase performance and to run a variety of programs; typical processors used in smartphones, tablet PCs, and smart TVs employ dual-issue superscalar processors and L2 caches as shown in Table 1. Dual-issue superscalar processors are cost-effective in improving performance and the L2 caches are effective in hiding long memory access latencies [22]. However, the L2 caches are typically of large sizes, consume a large portion of the chip areas, and, consequently, incur large energy consumption. Table 1 summarizes popular processors used in high performance embedded/portable computing systems in terms of issue rates, power consumption, L1 caches, and L2 cache configurations. They popularly employ 128KB to 512KB L2 caches with 8-way associativity. Since these systems are typically operated by battery, low energy consumption is critical for their long lifetimes.

Large L2 caches are beneficial for performance while small L2 caches are preferred considering their large power consumption. However, small L2 caches might aggravate overall performance and energy consumption due to increased execution times; leakage energy consumption is typically proportional to execution time. Therefore, a one better way to reduce overall energy consumption is to maintain performance with a small L2 cache, which incurs no more leakage in the datapath and reduces energy consumption in the L2 cache. Small L2 caches also reduce the chip area, which decreases chip production cost or allows more logics to be fabricated on the same chip. Previously proposed cache compression schemes make it possible to improve performance with small L2 caches [21, 7, 26]. However, they increase L2 cache energy consumption because of added extra bits, complex compression/decompression, and more cache accesses, as will be shown in Section 4.

As an attempt to reduce L2 cache energy consumption and area while keeping same performance, we propose a low-energy low-area L2 cache architecture. It consists of an L2 cache and a small cache called "***Residue Cache***". The residue cache architecture introduces three techniques. First, small memory values are compressed by eliminating unnecessary upper bytes of all zeros or ones and are stored in L2 cache lines, which are half sized of the conventional

Table 1: Processor Configurations used in High Performance Embedded/Portable systems

Processor	Typical service	L1 caches	L2 caches	Issue rate	Power
Intel Atom [23]	Netbook	32KB(I) 24KB(D)	512KB /core 8-way	2	2W (45nm)
ARM Cortex A8 [9]	Smartphone	32KB 4-way	64KB-2MB 8-way	2	0.3W (65nm)
ARM Cortex A9 [9]	Tablet PC	16-64KB 4-way	128-512KB 8-way	2	0.25W (40nm)
MIPS32 74K[2]	Smart TV	32KB 4-way	Configurable	2	0.54W (65nm)

Table 2: Access latency, dynamic per-access energy, leakage power, and area for different L2 cache sizes at a 45nm technology node obtained by CACTI 6.5 simulations

L2 cache size	Latency	Per-access energy	Leakage power	Area
128KB	0.9008	0.119nJ	198mW	0.573mm^2
256KB	1.1252	0.211nJ	464mW	1.476mm^2
512KB	1.3560	0.365nJ	1000mW	3.148mm^2

L2 cache lines. As many memory values are typically small in programs [10], there is a good chance to compress conventional L2 cache lines to smaller lines. Second, if the compressed conventional L2 cache lines are larger than 32B, their excess data words are captured by the residue cache. Because the residue cache is small, not all those excess data words can be captured by the residue cache. These compressed L2 cache lines with excess words not captured by the residue cache are called ***partial lines***. Third, since only a particular word incurs a cache miss in the L1 cache, and, consequently, in the L2 cache, if the particular word is present in a partial L2 cache line, this access does not incur an actual L2 cache miss; full L2 cache lines can be built after missing L2 cache line words are fetched from the main memory. This type of L2 cache hits is called ***partial hits***.

We performed experiments to evaluate our proposed residue cache architecture using the SPEC2000 benchmarks on a dual-issue superscalar processor modeled using the Simplescalar toolset [11]. Experimental results show that a large percentage of memory values are small and 40% of conventional L2 cache lines accessed can be compressed into half size lines. The residue cache architecture increases execution times by at most 0.5%, on the average, compared to the conventional double size L2 cache. L2 cache leakage energy consumption is reduced by 43% and total L2 cache energy consumption decreases by around 40%. The residue cache architecture is shown to be robust across different L2 cache sizes from 128KB to 256KB and 512KB. Floating-point benchmarks show little performance difference between 128KB and 256KB L2 caches. Experiments with SPEC2006 benchmarks on a 4-way superscalar processor show similar results to those with SPEC2000 benchmarks.

This paper has the following four contributions. First, our proposed residue cache architecture always shows comparable performance to double sized conventional L2 cache architecture. Second, the residue cache architecture is implementable with low overhead due to its simple compression/decompression mechanism. Third, the residue cache architecture is shown to be superior to previously proposed cache compression schemes in terms of energy consumption, area, and latency. Fourth, we proposed new concepts called partial lines and partial hits useful in reducing L2 cache area and miss rates.

This paper is organized as follows. Section 2 shows our motivation and Section 3 depicts our proposed residue cache architecture. Implementation and associated overheads are discussed in Section 4 and Section 5 presents experimental results. Related work is discussed in Section 6 and we conclude this paper in Section 7.

2. MOTIVATION

To quantify power consumption of various L2 cache sizes, we simulated CACTI 6.5 [1] at a 45nm technology node. L2 cache leakage power consumption is very large as shown in Table 2. To reduce the L2 cache leakage power consumption, high threshold voltage transistors can be used for SRAM cells, which, however, will increase the latency of L2 cache access and, consequently, might increase execution time and overall system leakage energy consumption. In addition, due to increasing importance of process variations, a large number of SRAM cells become faulty due to increased threshold voltage, which will decrease the yield of chips [18]. Therefore, adopting high threshold voltage is a partial solution to L2 cache leakage power reduction. Generally, leakage power consumption is proportional to the number of transistors. Thus, reducing L2 cache size will naturally reduce L2 cache leakage power. However, smaller L2 caches might increase execution time and, consequently, total system leakage energy consumption. Dynamic energy consumption is also proportional to L2 cache size as shown in Table 2. Reduced L2 cache area can be utilized to reduce production cost by making more chips using the same wafer size, to reduce L2 cache access latency since smaller caches are generally faster, and/or to improve performance/throughput by incorporating more logics.

To design the L2 cache with less number of transistors, small memory values prevalent in programs can be exploited [10]. Figure 1 shows the distribution of sizes of L2 cache lines replaced from the L2 cache when small memory values are compressed. Our small memory value compression will be explained in detail in Section 3. As can be observed, average 47% of replaced L2 cache lines can be confined in 32B capacity when 64B L2 cache lines are assumed. These compressed 32B cache lines can be exploited to design a smaller L2 cache than the conventional L2 cache without noticeable performance hit in three ways. First, these well compressed conventional L2 cache lines are placed in an L2 cache with 32B lines. Second, other poorly compressed lines are placed both in the L2 cache and a separate small cache with 32B lines. Third, remaining lines are placed only in the L2 cache in a compressed form. Although not all words of the remaining lines are captured by the L2 cache, it is likely that most of them can be captured in the L2 cache. These partial lines can service many L2 cache accesses as will be shown in Section 5.

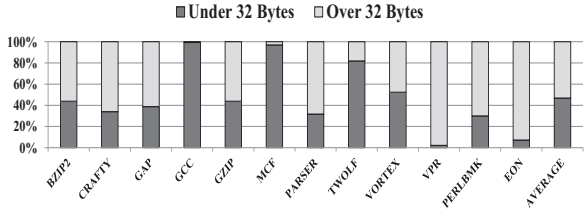

Figure 1: Size distributions of compressed L2 cache lines when replaced to the memory.

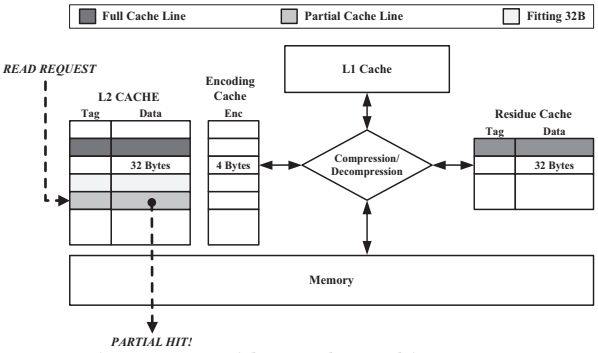

Figure 2: Residue cache architecture.

3. RESIDUE CACHE ARCHITECTURE

3.1 Overall Architecture

To exploit prevalent small memory values, the residue cache architecture is organized as shown in Figure 2. In our explanation of the residue cache architecture, the conventional L2 cache is direct-mapped for simplicity of explanation and has 64B cache lines. The L2 cache in the residue cache architecture is organized to have the same number of sets as that of the conventional L2 cache but its line size is 32B. Thus, the L2 cache size is half of the conventional L2 cache size. To accommodate other 32B data not captured by the L2 cache, a small cache called residue cache is introduced. The residue cache also has 32B cache lines and much smaller number of sets than that of the L2 cache. Consequently, many L2 cache lines are partial; L2 cache lines are full when they have the corresponding lines in the residue cache.

The residue cache has both tag and data parts as the conventional cache. The data part stores the conventional L2 cache line words not captured by the L2 cache. For tag matching, conventional caches use upper bits of memory addresses while the residue cache uses L2 cache locations. The residue cache tag bits are composed of the most significant L2 cache index bits, of which width is calculated by $\log_2$ (L2 cache size/residue cache size). This tagging scheme reduces required tag bits. On an L2 cache miss, missed data fetched from the main memory are placed both on the L2 cache and the residue cache. Replaced L2 cache lines are checked in the residue cache and matching residue cache lines are replaced together.

Because residue cache size is at most a quarter of the L2 cache size, three quarters of L2 cache lines are partial, which will decrease effective L2 cache size to a quarter of the conventional L2 cache in the worst case. In order to increase the number of full L2 cache lines, the residue cache architecture exploits small memory values prevalent in programs. Since upper bits of small memory values are all zeros or ones, they can be compressed to reduce required storage. Compressed conventional L2 cache line words are stored in the L2 cache lines and their excess words that cannot be captured by the L2 cache are stored in the residue cache. Compression information is stored in the Encoding cache to be used later for decompression. Compression occurs when data are fetched from the memory and replaced from the L1 cache to the L2 cache. Decompression occurs when data are written back to the memory on L2 cache replacements, and when data are read from the L2 cache on L1 cache misses. When a large number of words in conventional L2 cache lines are small, they can be compressed into 32B and do not need residue cache lines. This will give more opportunities for other conventional L2 cache lines not well compressed to be stored in the residue cache. Consequently, small memory value compression will increase the effective capacity of the residue cache architecture.

The residue cache architecture has three types of cache lines. Full cache lines span both the L2 cache and the residue cache when they cannot be compressed into 32B. Well compressed conventional 64B lines can be confined in 32B L2 cache lines, which are indicated as fitting 32B lines in Figure 2. When residue cache lines belonging to full cache lines are replaced to the main memory, the full cache lines are demoted to partial cache lines. The partial cache lines are promoted back to full cache lines when they are accessed as will be explained later.

When writeback data from the L1 data cache is larger than 32B and the corresponding data stored in the L2 cache is smaller than or equal to 32B, we allocate the excess data beyond 32B in the residue cache. In this case, a replacement from the residue cache is required. With write through L1 caches, more data will be transferred to the L2 cache. The increased traffic does not increase power and latency if writeback does not increase data size as discussed just above. From our experiments, this case happens rarely. Thus, we assume that this overhead will be small.

3.2 Compressing and Decompressing Small Memory Values

Compression occurs when data are written into the L2 cache, which happens when new data are brought from the main memory and when data are replaced from the L1 cache. Note that compression is not on the critical memory access path because uncompressed data can be forwarded to the datapath directly. Decompression occurs in two situations. First case is when a memory word access incurs a cache miss in the L1 cache but it hits in the L2 cache. Second case happens when data are replaced from the L2 cache to the main memory. The first case is on the critical memory access path but the second case is not.

Figure 3 shows the small memory value compression and decompression mechanism for 16B input data for simplicity. The 16B input data consists of four 4B words. The compressor has three components: Small Value Detectors (SVD), Shifters, and Shift Amount Generator. SVDs check each 4B input data whether it is small or not. Small values are defined as those values of which upper 2B or entire 4B are all zeros or ones. Detection results are fed into the Shift Amount Generator to generate control signals to be used by the shifters, and encoding bits to be stored in the encoding cache. The shifters generate compressed words. They shift

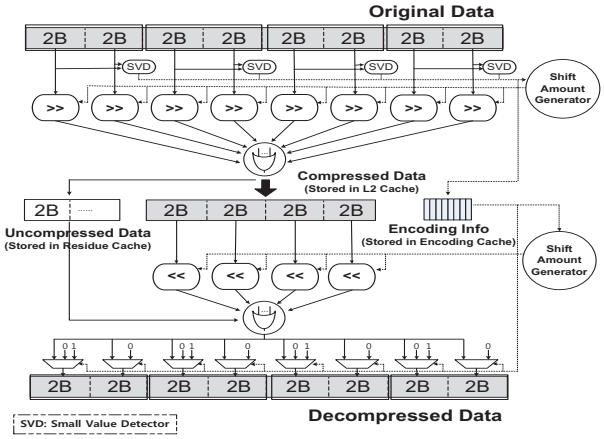

Figure 3: L2 cache line compression/decompression.

each 4B input data by a unit of 2B and the shifted values are concatenated to form a compressed line. When all input words are zeros or ones, they are compressed into zero byte in the best case. In the worst case, all words are never compressed and require all 16B. If the size of a compressed line is larger than 8B, the excess data are stored without compression in the residue cache.

Decompression is performed using the information from the encoding cache. First, the Shift Amount Generator generates appropriate control signals, which are fed into the shifters. Second, the shifters enlarge the compressed data by locating each 2B of compressed data into their original position. The enlarged data are concatenated with the uncompressed data from the residue cache. Finally, empty 2Bs generated during shifting are filled with ones or zeros. Since compression and decompression processes are symmetric to each other, the Shift Amount Generator and the shifters are shared to reduce area and energy costs.

3.3 Partial Cache Hits

Due to the presence of partial cache lines, the residue cache architecture introduces a new type of cache hits called partial cache hits. There are two ways to interpret partial hits. Partial hits can be regarded as actual L2 cache misses since not all words belonging to a L2 cache line are found. However, this approach will increase L2 cache miss rates. Second way is to regard partial hits as actual hits because only a particular word causes a miss in the L1 cache and, consequently, in the L2 cache. Although not all cache line words are available, if the requested word is present in the L2 cache, then this access is regarded as a hit. Since the residue cache is small and there are many partial L2 cache lines, the second approach will show lower miss rates than the first approach.

After we adopt the second approach for handling partial hits, there is another decision issue. We can make the partial cache lines to be full or just let them be partial cache lines. Since spatial locality is general in programs, it will be more beneficial to promote partial cache lines to full cache lines. On next accesses, they can be found in the L2 cache. To support promotion to full cache lines, we extend the conventional MSHR (Miss Status Handling Register) [25] as shown in Figure 4. A partial cache line field is added to each entry of MSHR. When a partial hit occurs in the L2 cache, the

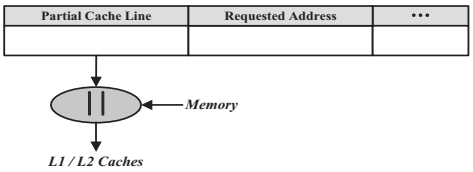

Figure 4: Extended MSHR.

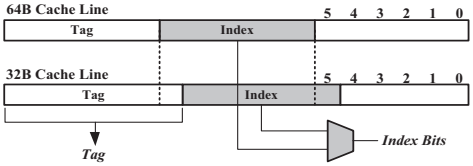

Figure 5: Extended MSHR.

corresponding partial line words are copied into the matching entry of MSHR. When their missing words arrive from the main memory, they are concatenated with the partial line words in MSHR and are sent to the L1 and L2 caches.

3.4 Programmable L2 Cache Line Sizes

Since the residue cache architecture is largely based on small memory values, it will be beneficial for the residue cache architecture to function like the conventional L2 cache when programs do not have many small memory values. For this, the residue cache is disabled, 32B lines are used instead of 64B lines, and memory values will be not compressed. Programs can choose their L2 cache type, the conventional L2 cache or the residue cache architecture, based on program's memory value profiling or on user's choice. For this purpose, the residue cache architecture introduces a one-bit register called "L2-cache-type". If programs want to use the residue cache architecture, the register is set to one. Otherwise, it is set to zero, indicating the conventional L2 cache type. This register is part of an architectural state and stored together with other state on context switches in a multi-tasking environment.

To support both the conventional L2 cache and the residue cache architecture, two issues must be addressed. First, L2 cache index bits will be different as shown in Figure 5. The residue cache architecture uses 64B cache lines while the conventional L2 cache uses 32B cache lines. Thus, index bits start from 6th bit for 64B cache lines while they begin from 5th bit for 32B cache lines. A mux is needed to select between the two index bits. Second issue is tag bits. Because two L2 cache types have the same number of index bits but the index bits start from and end at one bit different address bits, the conventional L2 cache with 32B cache lines needs an another tag bit as shown in Figure 5. To resolve the second issue, the residue cache architecture stores one more tag bit regardless of L2 cache type.

4. IMPLEMENTATION AND OVERHEADS

We selected four competitive schemes for comparison with the residue cache scheme. They are Frequent Value Compression (FVC) [35], Adaptive Cache Compression (ACC) [7, 6], Line Distillation (LD) [31], and Zero Content-Augmented cache (ZCA) [13]. FVC compresses two cache lines mapped to a same set into a single cache line when both of them can be compressed into half size. It was applied to direct-mapped L1 caches as shown in Figure 6(a). ACC generalizes

FVC to set-associative L2 caches as shown in Figure 6(b), where four cache lines mapped to a same set are compressed and stored into two cache lines. LD packs (compresses) "used words" from different cache lines mapped to a same set into a single cache line as shown in Figure 6(c). In fact, FVC, ACC, and LD share the same spirits in their approaches; increase associativity of individual sets dynamically either by exploiting frequent value compression or by packing used words from different lines in a single cache line. This is apparent because the three schemes all add more tag ways as shown in Figure 6. In contrast, Residue Cache (RC) scheme shown in Figure 6(d) takes a very different approach. It does not increase associativity of individual sets but reduces L2 cache line size to 32B. If necessary, individual lines are expanded to 64B using the residue cache. The ZCA scheme shown in Figure 6(e) increases effective cache capacity not by maintaining actual data but by keeping information of null blocks in a ZC cache. Null data is detected by a Zero Detector (ZD) and encoded in ZC$. Both the main and ZC caches are read in parallel not to increase cache access latency. It works only when all words of memory blocks are null or zero, so is more limited than compression-based schemes.

To evaluate and compare our proposed residue cache architecture with other competitive schemes in terms of dynamic energy, leakage power, area, and latency, we simulated CACTI 6.5 [1] to model cache structures and Synopsys Design Compiler to synthesize compression and decompression circuits, at a 45 technology node. A conventional L2 cache is organized as an 8-way 256KB cache with 64B lines. Adaptive cache compression scheme has a 128 KB L2 cache with 8-way tag arrays, 4-way data arrays and 64B lines. Line distill scheme also has a 128 KB L2 cache with 22-way tag arrays, 8-way data arrays and 64B lines. In ZCA scheme, the L2 cache is organized as a 128KB 8-way cache with 64B lines and ZC cache has a 4-way 16KB cache with 4B lines. All these organizations are selected based on their published work. The residue cache architecture has four components. Its L2 cache is a 128KB 8-way cache with 32B lines and the encoding cache is an 8-way 16KB cache with 4B lines. Its residue cache has 8-way 8KB size and 32B lines. The conventional cache, residue cache, Line distillation, and ZCA schemes all access the tag array first and then a particular data way of the L2 cache as in ARM Cortex A8 and Alpha 21164 processors [9, 14]. We used "sequential" mode of CACTI for conventional L2 and residue caches in which tag arrays and a matching data array are accessed sequentially similar to ARM Cortex A8, and "parallel" mode for adaptive cache compression scheme because it requires accessing all cache ways. This results in different area numbers by CACTI.

Table 3 summarizes our evaluation results for the five competing L2 cache architectures. FVC is excluded in the table because ACC is a generalized version of FVC. We incorporated extra bits required to implement each of those architectures as indicated in their work. Adaptive cache compression and line distillation schemes require extra bits in the tag arrays and ZCA scheme needs an extra ZC cache. The residue cache and adaptive cache compression schemes require compression and decompression circuits. Our compression/decompression is cache line-based while adaptive compression scheme is based on a set; it works on all the cache lines in a set requiring several cycles and large hardware for compression/decompression. More logics such as a parallel-prefix adder and compaction hardware are required. Detailed implementation for these logics is not shown in [7, 6]. These overheads are indicated as α in Table 3. Adaptive compression scheme needs to read all cache lines in a set, and write compressed and compacted cache lines, increasing L2 cache access energy consumption as shown in Table 3. We expect that compacting cache lines in a set is as complex as compression because it has to work on all cache lines in a set. Line distillation scheme installs used words replaced from LOC (line-organized cache) in WOC (word-organized cache), which results in reading and writing a WOC line, and consequent additional energy consumption indicated as R in Table 3.

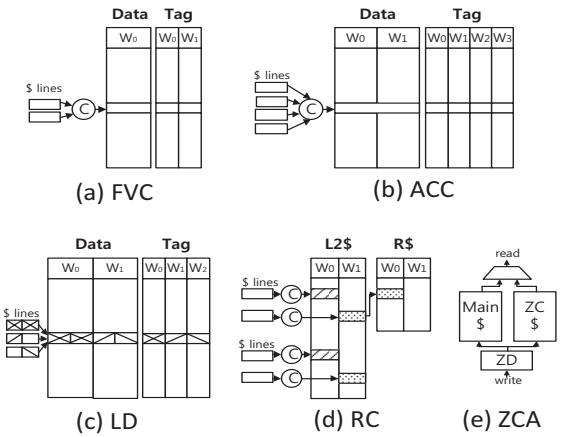

Figure 6: Competitive schemes. In FVC, two cache lines are compressed and stored into a single cache line while ACC does the same for four cache lines. In LD, two used words indicated by / and \are packed into a W1 line in the data array. In RC, the first and third cache lines are compressed and stored into 32B L2 cache lines while the second cache line spans both the L2 cache and R$. The fourth line is a partial cache line. ZCA stores null data information in ZC$ to increase effective cache capacity. C indicates compression. A 2-way cache is assumed for ACC, LD and RC schemes for simplicity.

From Table 3, we can observe that the residue cache, Line distill, and ZCA schemes show similar overheads. Work in [31] demonstrated that compression-based schemes can be used together with the Line distill scheme for synergy. ZCA scheme is also orthogonal to the residue cache scheme. This is because line distillation and ZCA cache schemes capture used words replaced from LOC or null data replaced from the L2 cache. The residue cache architecture is superior to the adaptive scheme for dynamic energy consumption, area, and latency. We expect that leakage power consumption is not much different between the two schemes considering compaction-related overhead indicated by α. Low dynamic energy consumption in the residue cache scheme is thanks to one L2 data way access while the adaptive scheme needs to read and write all cache lines, which also requires more hardware logics increasing energy consumption and area. Finally, the residue cache architecture consumes much less dynamic energy, leakage power and area than the conventional L2 cache architecture.

Table 3: Overheads of the four competing schemes at a 45 technology node. Numbers inside ( ) indicate cache sizes in KB. E$, R$, and ZCA$ indicate Encoding cache, Residue cache, and ZCA cache, respectively. α indicates compaction overhead. c and d stand for compression and decompression. R is the energy consumption on LOC replacement for Line distillation scheme. * indicates projected values based on a 90nm technology [12]

Parameters (size in KB)	Conv. L2$ (256)	Residue cache					Adaptive compression			Line distill	ZCA		
		L2$ (128)	E$ (16)	R$ (8)	c/d	Sum	L2$ (128)	c/d*	Sum		L2$ (128)	ZCA$ (16)	Sum
Per-access energy (pJ)	211	96	11	14	12/4	133/ 125	580(c) 290(d)	26/11 $+\alpha$	606(c) 301(d) $+\alpha$	127+ 254(R)	119	21	140
Leakage (mW)	464	214	29.4	15.4	5.6	264	229	23+α	252 $+\alpha$	245	198	32	230
Area (mm^2)	1.476	0.582	0.056	0.045	0.008	0.691	0.897	0.033 $+\alpha$	0.93 $+\alpha$	0.687	0.573	0.091	0.664
Latency (ps or cycle)	1125	971	309	418	1 cycle	971	969	13/8 cycle	13/8 cycle $+\alpha$	979	901	530	901

Inclusion between the L1 and L2 caches may not be enforced for partial L2 cache lines when corresponding L1 cache lines exist. However, up-to-date missing L2 cache line data is stored in the main memory. Multiprocessor coherency is maintained as follows. Partial hits are allowed for remote-core requests; Partial lines are transferred to the remote-cores, which determine if the accesses are partial hits. In case of a partial hit, missing line data will be retrieved from the main memory by the remote-cores. Thus, remote partial hits are handled like local partial hits.

5. EXPERIMENTAL RESULTS

5.1 Experimental Setup

Our baseline system is similar to ARM Cortex A8 processor. It has a dual-issue superscalar core and 2-level cache hierarchy. The L1 caches have 4-way 32KB size and the L2 cache has 8-way 128KB size with 32B lines. The residue cache is a 8-way 32KB cache with 32B lines. The Encoding cache is organized as a 16KB cache with 4B lines. The baseline system parameters are given in Table 4. We use a Simplescalar tooset to model our baseline system [11]. The SPEC2000 benchmarks [3] are used to evaluate performance. Alpha binaries and reference inputs are used for simulation. For simulation, 10B instructions are fast forwarded and then 1B instructions are executed. Compression/decompression takes an extra one cycle based on the results in the previous section. Lean configuration is used for the following experiments.

Table 4: Baseline system configuration

Parameter	Lean configuration	Fat configuration
Issue rate	2-way superscalar	4-way superscalar
Functional units	2 Int ALUs,1 Mult, 1 Div	4 Int ALUs, 1 Mult, 1 Div, 1 FP adder
Branch prediction	512-entry BTB, 4K-entry 2-level predictor	4K-entry BTB, 4K-entry 2-level predictor
L1 caches	4-way 32KB I/D caches, 2-cycle latency	4-way 32KB I/D caches, 2-cycle latency
L2 caches	8-way 128KB 32B line L2 cache, 8-way 32KB 32B line R-cache, 16KB E-cache, 9-cycle latency	8-way 1MB 32B line L2 cache, 8-way 256KB 32B line R-cache, 128KB E-cache, 15-cycle latency
Memory	80 cycles	300 cycles

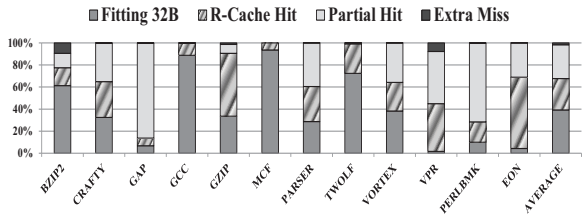

Figure 7: Distribution of L2 cache hit accesses (256KB).

5.2 Performance Results

To observe the behavior of the residue cache architecture, we classified L2 cache hit accesses into four categories: fitting 32B, R-cache hit, partial hit, and extra miss as shown in Figure 7. Fitting 32B is an access to a cache line compressed completely into 32B. R-cache hit is an access to a cache line residing both in the L2 cache and the residue cache. Partial hit occurs when an accessed line is partial but a requested word hits in the L2 cache. Finally, extra miss stands for an L2 cache hit but residue cache miss, which actually increases L2 cache miss rates. The bzip2, gcc, mcf, and twolf benchmarks show large percentages of cache lines fitting 32B indicating many memory values are small. The gap and perlbmk benchmarks include large percentages of partial hits. This is because these benchmarks show small percentages of cache lines fitting 32B but most cache lines are compressed well; the 32B L2 cache lines can capture larger than 32B uncompressed data, which increases the probability of a particular word to hit in the L2 cache. The gzip and eon benchmarks show high R-cache hit rates because they have small working sets so that most actively accessed cache lines are stored both in the L2 cache and the residue cache. The bzip2 and vpr benchmarks incur extra misses increasing L2 cache miss rates as can be observed in Figure 7. Overall, three portions except extra miss contribute L2 cache accesses roughly equally.

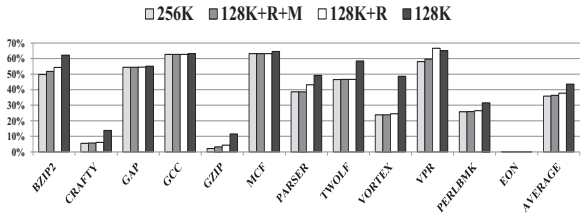

Figure 8: L2 cache local miss rates. 128K+R+M and 128+R indicate our residue cache with and without extended MSHR, respectively.

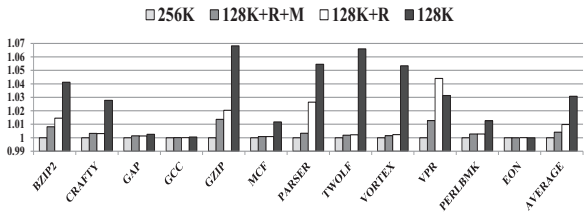

Figure 9: Execution times normalized to a conventional 256KB L2 cache.

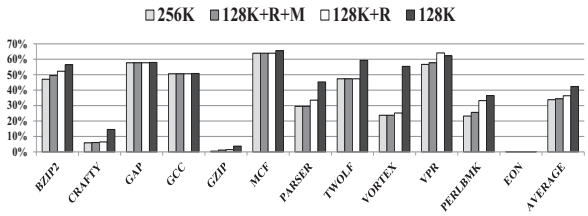

Figure 10: L2 cache local miss rates for 40B fast forward and 1B simulation.

In the following discussions, 256K and 128K configurations stand for conventional L2 caches with 256KB and 128KB capacities with 64B lines, respectively. 128K+R+M and 128K+R configurations indicate our residue cache architectures with a 128KB L2 cache and MSHR support, and with only a 128KB L2 cache, respectively. Figure 8 shows L2 cache local miss rates. 128K+R+M shows lower miss rates than 128K and slightly higher miss rates than 256K. 128K+R also shows better results than 128K except for the vpr benchmark, which shows near 50% partial hit rate as shown in Figure 7 increasing miss rate without extended MSHR support. Overall, our residue cache architecture increases L2 cache miss rates only by 0.4% and 1.8%, with and without extended MSHR, respectively. 128K incurs 7.7% increase in miss rate. The results for execution times show similar behaviors to those for miss rates in Figure 9. The gzip, parser, twolf and vortex benchmarks show large performance differences between 256K and 128K. Our residue cache works well for these benchmarks. 128K+R+M shows only average 0.4% increase in execution times for all the benchmarks, compared to 256K.

From Figure 7, average 30% of L2 cache accesses are partial hits. Promoting partial lines to full lines requires accesses to DRAM. At most 32B are read from DRAM, not full 64B. Thus, 30%*0.5=15% memory read traffic increase is expected in the worst case. From Figure 8, 128KB L2 cache shows 10% higher miss rate than 256KB L2 cache, which will increase memory read traffic and write back traffic by 10% each. Thus, we expect residue cache scheme will

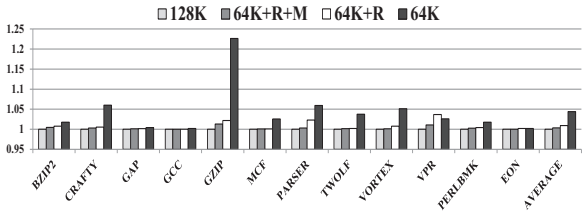

Figure 11: Execution times normalized to a conventional 128KB L2 cache.

show little benefit in reducing DRAM energy consumption. As a trade-off, residue cache without MSHR support will remove the extra DRAM accesses.

5.3 Sensitivity Results

To evaluate robustness of the residue cache architecture, we performed several sensitivity experiments. First, we evaluated the residue cache at different simulation point since programs may show different behaviors at different phases. For this, 40B instructions are fast forwarded and then next 1B instructions are simulated. Figure 10 shows L2 cache miss rates. The bzip2, gcc, gzip, and parser benchmarks show lower miss rates while the gap benchmark show higher miss rate than in Figure 8. Overall behaviors are similar to Figure 8. Only the perlbmk benchmark shows a little bit worse behavior with the residue cache architecture. 128K+R+M and 128K+R show only 0.5% and 2.4% increases, on average, in L2 cache miss rates. Miss rate difference between 256K and 128K is 8.4%, which is 7.7% in Figure 8 indicating 128K+R+M performs better at this simulation point. Execution time results show similar behaviors as in Figure 9. Average execution time increase is only 0.5% over 256K.

Second sensitivity experiments are performed by decreasing conventional L2 cache size by half and by increasing it double times. Figure 11 shows execution times normalized to a 128KB conventional L2 cache. Overall behavior is similar to Figure 9. The bzip2 benchmark shows a better result with 64K+R+M than with 128K+R+M in Figure 9. Execution time increase for 64K+R+M, on average, is less than 0.4% over 128K while the performance difference between 128K and 64K is 4.4%, which is larger than 3.1% between 256K and 128K. This indicates that the residuce cache architecture works better with a smaller cache. Especailly, the gzip benchmark benefits most from a smaller L2 cache. Figure 12 shows execution time results normalized to a 512KB conventional L2 cache. Similar results are observed. The gzip benchmark negatively benefits while the twolf benchmark positively benefits from this larger L2 cache. From these sensitivity experiments by varying an L2 cache size, we show that the residue cache architecture never degrades performance; it consistently shows around 0.4% execution time increase over a conventional L2 cache with roughly double capacity.

Third sensitivity experiments are performed by reducing residue cache size to 16KB and 8KB. The 16KB residue cache results are omitted for space limitation but show similar results to 8KB results. Figure 13 shows L2 cache miss rates for an 8KB residue cache. Even with a quarter size residue cache, 128K+1/4R+M increases miss rates by only 0.6% while 128K+1/4R incurs 4.3% increase over 256K. This demonstrates that extended MSHR is more helpful for an

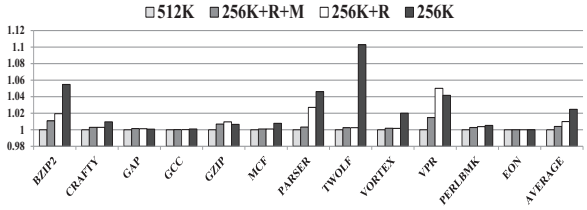

Figure 12: Execution times normalized to a conventional 512KB L2 cache.

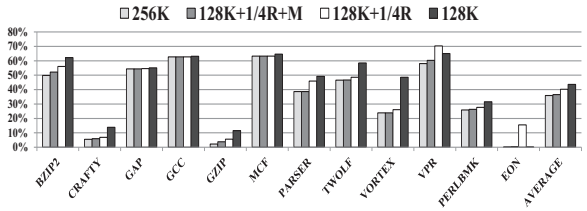

Figure 13: L2 cache local miss rates for an 8KB residue cache.

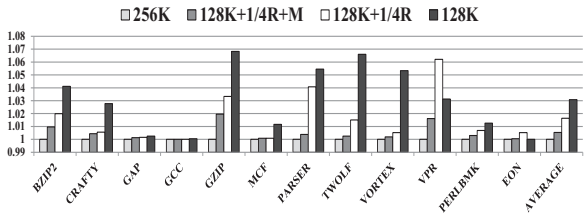

Figure 14: Execution times normalized to a conventional 256KB L2 cache for an 8KB residue cache.

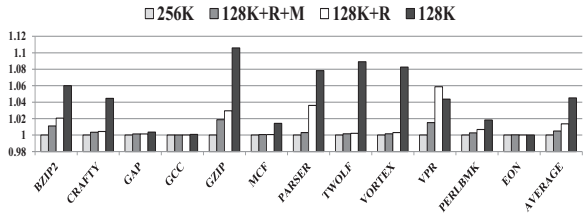

Figure 15: Execution times normalized to a conventional 256KB L2 cache for 120-cycle memory latency.

8KB residue cache. Figure 14 shows execution time results for an 8KB residue cache. Average execution time increase is only 0.5% over 256K.

Final sensitivity experiments are performed by increasing memory access latency to 120 cycles to reflect increasing clock frequency of the embedded/portable processors. Execution time increase with 128K+R+M is still 0.5% over 256K in Figure 15. The execution time difference between 256K and 128K is 4.5%, which is 3.1% for 80-cycle memory access latency.

From the previous sensitivity experiments, the residue cache architecture is shown to be robust across various L2 cache sizes, simulation point, residue cache sizes, and main memory latencies. We also conducted experiments with the MiBench benchmarks [19]. However, because they are small, there is no noticeable performance difference among 128KB, 256KB, and 512KB L2 cache sizes.

5.4 Results for Floating-point Benchmarks

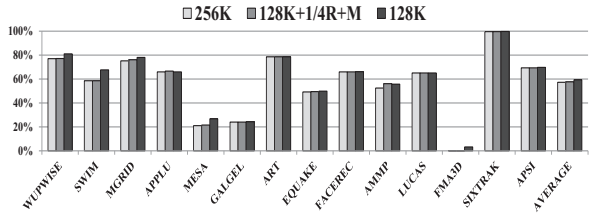

Figure 16: L2 cache local miss rates for floating-point benchmarks.

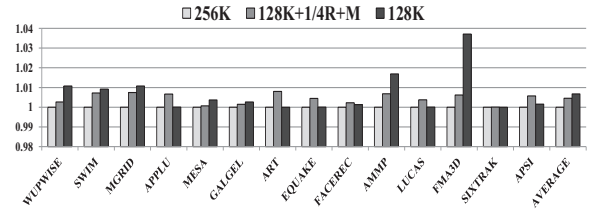

Figure 17: Execution times normalized to a conventional 256KB L2 cache for floating-point benchmarks.

We also performed experiments with SPEC2000 floating-point benchmarks. Figure 16 and Figure 17 show L2 cache local miss rates in percentage and execution time increases normalized to a conventional 256KB cache, respectively. Performance gaps between 256KB and 128KB caches are small. Residue cache scheme works well for those benchmarks for which there are some gaps in miss rates between the two cache sizes. The other benchmarks, such as applu, art, equake, facerec, lucas and apsi benchmarks, show a little bit worse performance due to an extra cycle for compression/decompression. Overall, residue cache scheme shows performance ranging between conventional 128KB and 256KB L2 cache sizes.

5.5 Results for SPEC2006 Benchmarks

To observe the applicability of the residue cache scheme to high-performance systems, we performed experiments with SPEC2006 benchmarks on a 4-way superscalar processor typically used in those systems. We use the fat configuration given in Table 4. To reflect high clock frequencies of those systems, main memory access latency is assumed to be 300 cycles. We show results for eight integer and nine floating-point benchmarks that are compiled and simulated successfully.

Figure 18 shows distribution of L2 cache hit accesses. R-cache hit portion increases compared to Figure 7 for SPEC2000 benchmarks. This indicates that the residue cache performs well for these benchmarks. Six benchmarks (bzip2, hmmer, omnetpp, xalan, namd, soplex) show noticeable differences in L2 cache miss rates between 2MB and 1MB sizes in Figure 19. In terms of execution times, three benchmarks such as omnetpp, xalan, and zeusmp show noticeable performance differences in Figure 20. 1M+1/4R+M shows comparable performance to 2M for the three benchmarks. For some other benchmarks, execution times increase slightly because the residue cache scheme requires an extra cycle for compression/decompression. Overall, the residue cache scheme does not increase L2 cache miss rates and increases execution times by less than 0.5% on average for integer benchmarks. In contrast, the conventional 1MB L2 cache

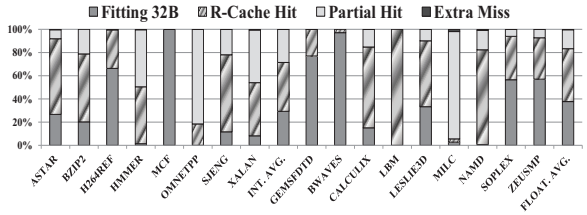

Figure 18: Distribution of L2 cache hit accesses for SPEC2006 benchmarks.

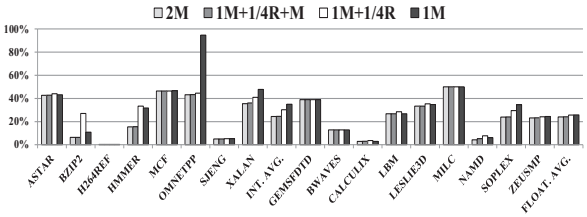

Figure 19: L2 cache local miss rates in percentage.

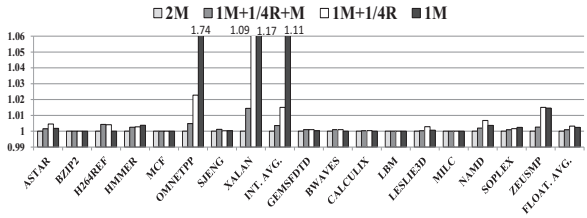

Figure 20: Execution times normalized to a 2MB conventional L2 cache.

incurs execution time increase by average 11% for the same benchmarks. SPEC2006 floating-point benchmarks show a similar behavior to SPEC2000 floating-point benchmarks. This demonstrates that our residue cache scheme is effective in improving performance of high-performance systems as well.

6. REALTED WORK

We classify related work into five categories by their schemes. First, compression-based cache and memory schemes are proposed. Hallnor et al. propose indirect index cache scheme for decoupled cache access [20]. It uses hashing to access tag portion and then an index read from the tag portion to access a particular data block. They also propose a unified compressed memory hierarchy where the last level cache and main memory are both compressed so that there is no compression/decompression overhead between the two levels [21]. This scheme assumes a 3-level cache hierarchy and needs to access the L3 cache multiple times. To remove performance decrease due to large decompression latency, adaptive cache compression (ACC) scheme is proposed [7, 6]. It requires complex compaction and needs to read all cache ways because its compression works on a set, increasing area and energy consumption. Selective compressed memory system scheme compresses two adjacent cache lines into a single line, which requires to access two cache lines on decompression similar to adaptive cache compression scheme [26]. A similar scheme is proposed in [5] with focus on implementation feasibility. Compression/decompression is applied to the main memory in [33, 4, 15]. They also exploit frequent and/or small memory values but their mechanisms are quite different mainly because of locating memory blocks and compression/decompression latency.

Second, filtering-based schemes are proposed to increase effective cache capacity. Pujara et al. investigate utilization of L1 cache and found that cache utilization rate is low [29]. They predict "to be referenced words" to eliminate unnecessary words. Line distillation (LD) scheme is proposed to increase L2 cache capacity by storing only referenced words together in particular ways of the L2 cache [31]. This scheme requires extra bits in tag arrays and needs to access WOC (word-organized cache) double times (a read and a write) on LOC (line-organized cache) replacements, increasing energy consumption. A similar scheme to line distillation is proposed by Pujara et al [30]. They predict to be used words and store them in a single block together. This scheme is applied to the L1 data cache.

Third, value-based schemes are proposed. Dynamic zero compression scheme is proposed to reduce L1 cache dynamic energy consumption [32]. It adds a zero indication bit to each cache byte so that zero byte read and write are disabled, and only the zero indication bit is read and written, reducing cache access energy consumption. Frequent value compression (FVC) is proposed for direct-mapped L1 data caches by Yang et al [35]. By compressing frequent memory values, two cache lines can be stored in a single cache line space increasing effective cache capacity and hit rates. They also propose energy-efficient frequent L1 data cache design in [34]. This cache design accesses frequent values in the first cycle while non-frequent values are accessed in the second cycle. Recently, zero-content augmented cache scheme (ZCA) is proposed to exploit frequent null (zero) values [13]. ZCA stores information about null data values in a separate cache, which can capture memory accesses to null data increasing hit rates. Restrictive compression scheme is proposed not to increase L1 cache access latency by compressing a cache line of which all words are narrow and by storing the upper half words of normal size words in a separate cache [28]. Its compression is based on a word, not on a line or set as in other compression schemes.

Fourth, there have been many techniques for reducing L2 cache energy consumption [17, 8, 27, 24, 16]. They try to reduce dynamic and/or leakage energy consumption by accessing or activating a small portion of L2 caches. In contrast, the residue cache architecture reduces required L2 cache capacity but maintains the same performance level. Therefore, those techniques can be used with the residue cache architecture to further reduce L2 cache energy consumption.

The residue cache architecture is more efficient in reducing area and energy consumptions than above compression-based schemes thanks to simple compression/decompression, and introduction of partial lines and partial hits. Also, it is orthogonal to other types of schemes such as filtering-based schemes, and energy reduction schemes discussed above.

7. CONCLUSIONS

Low energy and low area are critical issues in the embedded/portable systems typically operated by battery and with limited cost. L2 caches are being adopted for these systems for high performance but they increase energy consumption and area. We proposed a low-energy low-area L2 cache architecture called residue cache. The residue cache architecture compresses conventional L2 cache lines and stores them in half sized L2 cache lines. Remaining

words not confined in the L2 cache lines are stored in a separate residue cache. Cache misses are minimized by utilizing partial cache hits. The residue cache architecture reduces dynamic energy consumption by employing half sized L2 cache lines and a simple small value compression and decompression mechanism. Leakage power and area are reduced due to a small L2 cache and a small residue cache. The residue cache architecture decreases dynamic energy consumption by 40%, leakage power by 43%, and area by 53% with at most 0.5% performance degradation, compared to the conventional L2 cache architecture. We demonstrated that the residue cache architecture works well for SPEC2000 benchmarks running on a 2-way superscalar processor and for SPEC2006 benchmarks running on a 4-way superscalar processor. This architecture is also shown to be robust across various configurations. Therefore, we argue that the residue cache architecture is general enough to be adopted by high-performance as well as embedded systems.

8. ACKNOWLEDGMENTS

Jesung kim was with KAIST during performing this work. This research was supported by the National Research Foundation of Korea (NRF) grants funded by the Ministry of Education, Science and Technology (2011-0005378, 2011-0018245).

9. REFERENCES

[1] Cacti 6.5. http://www.hpl.hp.com/research/cacti/.
[2] MIPS32 74K. http://www.mips.com/products/cores/32-64-bit-cores/mips32-74k/.
[3] Spec2000 benchmarks. http://www.specbench.org/osg/cpu2000.
[4] B. Abali, H. Franke, X. Shen, D. E. Poff, and T. B. Smith. Performance of hardware compressed main memory. In *HPCA*, 2001.
[5] A.-R. Adl-Tabatabai, A. M. Ghuloum, and S. O. Kanaujia. Compression in cache design. In *ICS*, pages 190–201, 2007.
[6] A. R. Alameldeen and D. A.Wood. Frequent pattern compression: A significance based compression scheme for L2 caches. Technical report 1500, University of Wisconsin, Madison, Apr. 2004.
[7] A. R. Alameldeen and D. A. Wood. Adaptive cache compression for high-performance processors. In *ISCA*, pages 212–223, 2004.
[8] D. H. Albonesi. Selective cache ways: On-demand cache resource allocation. *J. Instruction-Level Parallelism*, 2, 2000.
[9] ARM. Cortex-a processors. http://www.arm.com/products/processors/cortex-a/.
[10] D. Brooks and M. Martonosi. Dynamically exploiting narrow width operands to improve processor power and performance. In *HPCA*, pages 13–22, 1999.
[11] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-1997-1342, University of Wisconsin, Madison, June 1997.
[12] X. Chen, L. Yang, R. P. Dick, L. Shang, and H. Lekatsas. C-pack: A high-performance microprocessor cache compression algorithm. *IEEE Trans. VLSI Syst*, 18(8):1196–1208, 2010.
[13] J. Dusser, T. Piquet, and A. Seznec. Zero-content augmented caches. In *ICS*, pages 46–55, Yorktown Heights, NY, USA, June 2009. ACM Press.
[14] J. H. Edmondson. Internal organization of the Alpha 21164, a 300-MHz 64-bit quad-issue CMOS RISC microprocessor. *Digital Technical Journal of Digital Equipment Corporation*, 7(1):119–135, Winter 1995.
[15] Ekman and Stenstrom. A robust main-memory compression scheme. *CANEWS: ACM SIGARCH Computer Architecture News*, 33, 2005.
[16] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy caches: Techniques for reducing leakage power. In *ISCA Computer Architecture News (CAN)*, Anchorage, AK, 2002.
[17] K. Ghose and M. B. Kamble. Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line segmentation. In *ISPLED*, pages 70–75, 1999.
[18] M. Goudarzi and T. Ishihara. SRAM leakage reduction by row/column redundancy under random within-die delay variation. *IEEE Trans. VLSI Syst*, 18(12):1660–1671, 2010.
[19] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization*, Dec. 2001.
[20] E. G. Hallnor and S. K. Reinhardt. A fully associative software-managed cache design. In *ISCA*, pages 107–116, 2000.
[21] E. G. Hallnor and S. K. Reinhardt. A unified compressed memory hierarchy. In *HPCA*, pages 201–212. IEEE Computer Society, 2005.
[22] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, 4th ed.* Morgan Kaufmann Publishers Inc., Sept. 2006.
[23] Intel. Intel atom processor. http://www.intel.com/technology/atom.
[24] S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: exploiting generational behavior to reduce cache leakage power. In *ISCA*, pages 240–251, 2001.
[25] D. Kroft. Retrospective: Lockup-free instruction fetch/prefetch cache organization. In *25 Years ISCA: Retrospectives and Reprints*, pages 20–21, 1998.
[26] J.-S. Lee, W.-K. Hong, and S.-D. Kim. An on-chip cache compression technique to reduce decompression overhead and design complexity. *Journal of Systems Architecture*, 46(15):1365–1382, 2000.
[27] G. Memik, G. Reinman, and W. H. Mangione-Smith. Just say no: Benefits of early cache miss determinatio. In *HPCA*, pages 307–316, 2003.
[28] P. Pujara and A. Aggarwal. Restrictive compression techniques to increase level 1 cache capacity. In *ICCD*, pages 327–333. IEEE Computer Society, 2005.
[29] P. Pujara and A. Aggarwal. Increasing the cache efficiency by eliminating noise. In *HPCA*, pages 145–154. IEEE Computer Society, 2006.
[30] P. Pujara and A. Aggarwal. Increasing cache capacity through word filtering. In B. J. Smith, editor, *ICS*, pages 222–231, Seattle, Washington, USA, June 2007.
[31] M. K. Qureshi, M. A. Suleman, and Y. N. Patt. Line distillation: Increasing cache capacity by filtering unused words in cache lines. In *HPCA*, pages 250–259. IEEE Computer Society, 2007.
[32] L. Villa, M. Zhang, and K. Asanovic. Dynamic zero compression for cache energy reduction. In *MICRO*, pages 214–220, 2000.
[33] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis. The case for compressed caching in virtual memory systems. In *Proceedings of the USENIX 1999 Annual Technical Conference*, pages 101–116, 1999.
[34] J. Yang and R. Gupta. Energy efficient frequent value data cache design. In *MICRO*, pages 197–207, 2002.
[35] J. Yang, Y. Zhang, and R. Gupta. Frequent value compression in data caches. In *MICRO*, pages 258–265, 2000.