LEMap: Controlling Leakage in Large Chip-multiprocessor Caches via Profile-guided Virtual Address Translation

Jugash Chandarlapati and Mainak Chaudhuri
Department of Computer Science and Engineering
Indian Institute of Technology
Kanpur 208016
INDIA
jugash.ch@gmail.com, mainakc@cse.iitk.ac.in

Abstract

The emerging trend of larger number of cores or processors on a single chip in the server, desktop, and mobile notebook platforms necessarily demands larger amount of on-chip last level cache. However, larger caches threaten to dramatically increase the leakage power as the industry moves into deeper sub-micron technology. In this paper, with the aim of reducing leakage energy we introduce LEMap (Low Energy Map), a novel virtual address translation scheme to control the set of physical pages mapped to each bank of a large multi-banked non-uniform access L2 cache shared across all the cores. Combination of profiling, a simple off-line clustering algorithm, and a new flavor of Irix-style application-directed page placement system call maps the virtual pages that are accessed in the L2 cache roughly together onto the same region of the cache. Thus LEMap makes the access windows of the pages mapped to a region roughly identical and increases the average idle time of a region. As a result, powering down a region after the last access to the clusters of the corresponding virtual pages saves a much bigger amount of L2 cache energy compared to a usual virtual address translation scheme that is oblivious to access patterns. Our execution-driven simulation of an eight-core chip-multiprocessor with a 16 MB shared L2 cache using a 65 nm process on eight shared memory parallel applications drawn from SPLASH-2, SPEC OMP, and DIS suites shows that LEMap, on average, saves 7% of total energy, 50% of L2 cache energy, and 52% of L2 cache power while suffering from a 3% loss in performance compared to a baseline system that employs drowsy cells as well as region power-down without access clustering.

1. Introduction

Over the last decade the DRAM latency has emerged as the biggest bottleneck to the evolution of high-end computers and has severely hampered the performance growth in the desktop, server, and mobile notebook computers. To mitigate this high off-chip data access latency, the microprocessor industry has incorporated large on-chip caches [20, 25]. With the increasing number of cores on the same die, the size of on-chip cache is expected to increase further in future. However, large on-chip caches threaten to dramatically increase the leakage energy as the industry moves further into deep sub-micron processes. The left bar of each group in Figure 1 shows the energy dissipated by a 16 MB shared L2 cache designed with 65 nm nodes in a simulated eight-core chip-multiprocessor as a fraction of the total energy for eight explicitly parallel shared memory applications chosen from SPLASH-2, SPEC OMP, and DIS (Data Intensive Stressmark) suites. The last group of bars shows the harmonic mean. On average, 37% of total energy is dissipated in the L2 cache 97% of which (not shown) comes from leakage for these applications when no leakage optimization technique such as drowsy cells [7], sleep transistors, or power gating is applied.

On the other hand, the right bar in each group of Figure 1 shows the average dead time of a page (4 KB in size) residing in the L2 cache as a fraction of the total execution time. The dead time of a page is calculated by excluding the cycles starting from the first L2 cache access to any block in this page from any core till the last L2 cache access to this page. For six applications the average dead time of a page is more than 15% of the total execution time. On average, a page remains powered on unnecessarily in the L2 cache for 10% of the total execution time. Past proposals have explored various leakage reduction techniques to exploit this dead time at cache block granularity. However, as the caches grow bigger, the book-keeping overhead per cache block becomes prohibitive. The page-level dead time statistic shows that there is considerable potential of managing leakage at a much higher grain than cache blocks. Also, page-level techniques allow smart involvement of the operating system in leakage management.

In addition to the page dead time, a second important property of a program is that a set of pages are usually accessed together over a time window. These pages form the working set of the program over that time window. Following these two observations, we present LEMap (Low Energy Map), a novel profile-guided virtual address translation technique to control leakage in large chip-multiprocessor caches. The central idea is to assign physical addresses to the pages that are accessed together such that all of them get

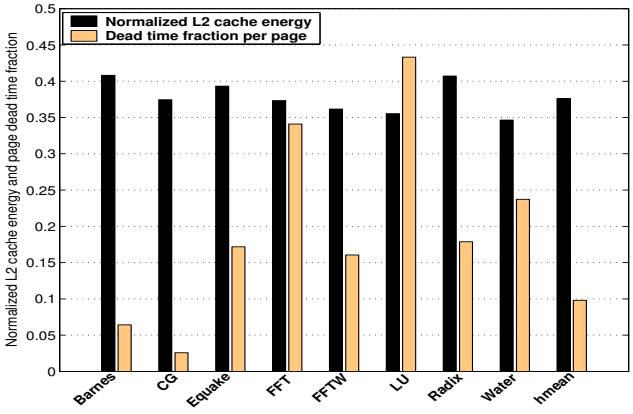

Figure 1. The left bar shows the L2 cache energy as a fraction of the total energy without drowsy cells. The right bar shows the average dead time of a page as a fraction of the total execution time.

clustered into one small region of the L2 cache as opposed to getting scattered all over the cache. The average dead time of each of these regions improves dramatically due to access clustering allowing us to apply drowsy and power-down modes more efficiently to each of these regions.

We collect the page access timestamps in the L2 cache via a profile run and cluster the pages that are accessed together. The number of pages in a cluster depends on the grain of the region to which we would like to apply leakage control. Next, we assign physical addresses to the pages in a cluster through a new flavor of Irix-style application-directed page placement system calls such that all these pages get mapped to a region in the L2 cache. Finally, we insert a special system call in the application to power down a region of the L2 cache after the last access to the cluster(s) mapped to that region. In the background, we keep the drowsy mode enabled in each of the regions so that during the idle cycles of a region, it can be put to a data preserving low voltage supply according to the drowsy protocol. We summarize the major contributions of this paper below.

- For the first time, this paper presents a virtual address translation scheme for region-based leakage control in chip-multiprocessor caches. We present a new flavor of application-directed page placement system call to realize our proposal (Section 2).
- Our execution-driven simulation results (Sections 3 and 4) show that LEMap, on average, saves 7% of total energy, 50% of L2 cache energy, and 52% of L2 cache power while suffering from a 3% loss in performance for eight shared memory parallel applications chosen from SPLASH-2, SPEC OMP, and DIS suites running on an eight-core chip-multiprocessor with a 16 MB shared L2 cache at 65 nm compared to a baseline that employs drowsy as well as power-down modes but does not take advantage of access clustering.

2. Virtual Address Translation Schemes

Virtual memory management layer of an operating system assigns a physical address to a virtual address when the corresponding virtual page is brought into main memory from the next level of storage, which is usually a disk. At a very high level, this involves hashing the virtual page number into a color bin and picking a free physical page frame from that bin. In a non-uniform memory access (NUMA) kernel, a node number is also decided following a round-robin, first-touch, or some other policy. The selected physical page frame resides in the main memory of this node, which is called the home node of the page. In this paper, we focus only on single-node systems. The physical address assigned to a virtual address naturally decides its location within the caches. Most importantly, in a multi-banked L2 cache it decides the bank number where this address resides [4]. In general, the bank number can be formed out of any part of the cache index bits. For example, in a block-interleaved scheme, the lowest $\log(n)$ bits of the cache index are used to decide the bank number in an n-way banked cache. Similarly, in a page-interleaved scheme, the lowest $\log(n)$ bits of the physical page number are used to decide the bank number. In this paper, we use the upper $\log(n)$ bits of the cache index to decide the bank number. This banking scheme allows a large chunk of contiguous data to be mapped onto a single bank and increases the idle time of all the banks on average because part of one single bank is often accessed for a long time due to spatial locality. As a result, with standard leakage optimization techniques such as drowsy cells [7], applied at a 128 KB grain to amortize the book-keeping cost, our baseline L2 cache architecture enjoys very low leakage energy dissipation (roughly 1 W/MB) and offers a challenging starting point for further energy optimization. In the following, we discuss the LEMap proposal in detail including the necessary application and operating system modifications to take advantage of LEMap.

2.1. Basic Principle of LEMap

LEMap aims at increasing the dead time of regions of a large L2 cache so that these regions can be powered down after the last access without fearing any data loss. In this paper, we statically fix the region size to be 128 KB. Each region is a contiguous portion of area within each bank and will be referred to as a subbank. As an example, consider a 16-way set associative 16-way banked 16 MB L2 cache. Thus, each bank is 1 MB in size, 16-way set associative, and has eight 128 KB subbanks. Assuming a 128-byte block size and a 4 KB page size, a 40-bit physical address in our system is broken down into several L2 cache components according to Figure 2. Notice that the subbank number could be chosen from any part of the cache index within a bank. However, since we would like to control the placement of pages in the L2 cache, we do not want to have pages split across subbanks. Therefore, we need to pick the subbank number from the part of the physical page number that overlaps with the cache index within a bank. We choose the bits right after the bank number so that a contiguous range of 64 address sets (lower six bits from cache index) forms one subbank (see Figure 2).

LEMap exploits two fundamental properties of programs. First, as already mentioned in Section 1, a page is dead during a reasonable portion of the application execution time. Second, a group of pages is accessed together in

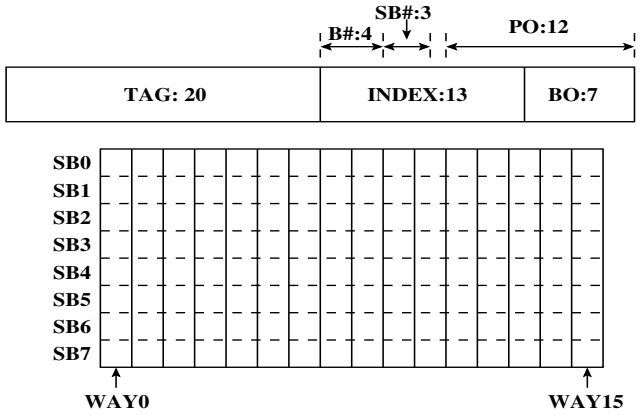

Figure 2. Address bits for accessing an example 16 MB L2 cache. Organization of one bank is shown below. BO is block offset, PO is page offset, B# is bank number, and SB# is subbank number within a bank.

time and this group forms the working set of the application at the given point in time. Therefore, if we can map a group of pages that are accessed together onto a subbank, we can turn off the entire subbank after the last access to the group. This technique leads to an increase in the idle time of each subbank, since the virtual address to physical address mapping is now aware of the temporal behavior of the accesses. In the following, we discuss the entire process in detail.

2.2. Access Clustering

We collect the timestamp, source core id, and the virtual page number of each L2 cache access during a profile run. We use this profile data to build clusters each spanning at most 128 KB data or equivalently 32 pages that are accessed roughly at the same time. Our clustering algorithm mimics a hierarchical agglomerative procedure. Initially, each distinct page belongs to a singleton cluster. We define the birth time b_k of a cluster k to be the timestamp of the earliest access to any page belonging to that cluster. Similarly, we define the death time d_k of a cluster k to be the timestamp of the last access to any page in that cluster. Clearly, the difference between the death time and the birth time of a cluster k can be defined as the life time l_k of the cluster. At every step of the clustering algorithm, for every pair of clusters i and j, we compute $B_{ij} = |b_i - b_j|$ and $D_{ij} = |d_i - d_j|$. We pick the cluster pair (i, j) such that $B_{ij} + D_{ij}$ is minimum and $C_i + C_j \leq 32$ where C_k is the number of pages in cluster k. We combine these two clusters into a bigger cluster. The birth and death times of the new cluster become $\min(b_i, b_j)$ and $\max(d_i, d_j)$, respectively. This completes one step of the algorithm. The algorithm terminates when no more agglomeration is possible while maintaining the size of each cluster within a bound of 32 pages. Therefore, at the end of the clustering phase we get a number of clusters each with at most 32 pages such that the pages belonging to a cluster have similar active and idle phases. The next step is to map these clusters of virtual pages to the physical subbanks of the L2 cache so that each subbank will have pages that are accessed roughly together.

2.3. Cluster to Subbank Mapping

We number the subbanks such that we visit all the subbanks within a bank first and then move on to the next bank. A simple algorithm to map the virtual pages in clusters to the physical subbanks would first rank the clusters by increasing value of birth time and then assign the clusters from this ranked list to the subbanks in round-robin order. Thus the first cluster gets mapped to subbank zero of bank zero, the next one on subbank one of bank zero, and so on. A problem arises when all the subbanks are exhausted. In this case we need to wrap back to subbank zero of bank zero and start overbooking the subbanks. However, since the clusters are ranked by birth time, the temporal overlap in life time of two clusters mapped to the same subbank is reduced. Such a scheme reduces the possibility of conflict misses in a subbank, although increases the life time of the subbank. However, we found that an L2 cache miss due to subbank conflict is much more expensive in terms of energy compared to the lost opportunity of leakage saving.

An improved mapping algorithm would try to exploit the proximity information between the cores and the banks in a non-uniform cache architecture (NUCA), as the one we are considering in this paper. Suppose we number the pages in a cluster k from zero to $C_k - 1$ where C_k is the number of pages in the cluster k. Further, suppose that a page p is accessed N_{pq} times by core q. Finally, let the access latency between core q and bank r be L_{qr}. Then we can define the quantity $A_{kr} = \sum_{q=0}^{M-1} \sum_{p=0}^{C_k-1} N_{pq} L_{qr}$ as the total number of cycles spent by all cores (M in number) accessing the pages in cluster k if this cluster is mapped onto a subbank in bank r. Note that accessing any region within a bank requires the same amount of time. In other words, the quantity L_{qr} is the worst case access latency between core q and any part of bank r. Therefore, we would like to map cluster k to a subbank in bank r such that A_{kr} is minimized. Our proximity-aware subbank mapping algorithm prepares a ranked list R_k of banks for each cluster k such that $A_{kR_k[0]} \leq A_{kR_k[1]} \leq \ldots \leq A_{kR_k[n-1]}$ for n banks. Before mapping any cluster, it prepares a bitvector of length S where S is the number of subbanks. Initially, the bitvector is cleared. Whenever a cluster is mapped to a subbank, the corresponding bit position is set. The mapping algorithm picks clusters according to the birth time rank. When considering cluster k for mapping, the algorithm selects a bank r from R_k with as low an A_{kr} value as possible such that there is a subbank in bank r with the corresponding bit position reset in the bitvector. When all the bit positions in the bitvector get set meaning that all the subbanks are populated in this round, the bitvector is cleared and the algorithm continues until all the clusters are mapped. In the next section we discuss how this mapping process is integrated into the application source.

2.4. Application and OS Modifications

The starting addresses of the virtual pages belonging to each cluster are first translated to the corresponding pointers to source-level data structures. For example, if a one-dimensional integer array A starts at virtual address 0x1000

and spans up to virtual address 0x2fff, a virtual page number of 0x2 i.e. a virtual page address of 0x2000 (assuming 4 KB pages) would get translated to &A[1024] in C syntax. To carry out this translation, we collect the virtual page numbers belonging to each data structure of the application under consideration and store this information into a hash table during the profile run. Later we pick a virtual page at a time from a cluster, look up the hash table, and translate the starting address of the virtual page into a pointer. At the end of this phase, each cluster is composed of at most 32 pointers to various (possibly identical) parts of data structures of the application.

The next phase involves passing this cluster composition to the virtual memory manager of the operating system (OS) and map all the virtual pages in a cluster to a physical subbank of the L2 cache. Usual application-directed page placement system call, as available in NUMA Irix, allows an application to specify a range of contiguous virtual addresses (in terms of a starting address e.g. &A[0] and an ending address e.g. &A[1023]) and a node number. The system call handler maps this range of virtual addresses to physical page frames on the specified node. However, LEMap requires mapping a list of discrete pages as opposed to a large contiguous range of virtual addresses. It is possible to use the conventional page placement system call, one for each page, but the system call overhead will outweigh all the benefits of the technique. Instead, we introduce a new flavor of page mapping system call where the application passes a list of virtual pages to be mapped to the OS. For each cluster k, the application prepares a buffer with the following composition. The first element of the buffer is the number of pages in the cluster i.e. C_k. The next C_k elements are the virtual page addresses. If proximity-aware mapping is enabled, the next element of the buffer is the number of banks in the L2 cache and the next several elements contain the ranked list R_k of the banks. This ranked list is prepared offline and stored with the profile data to minimize run time overhead. Finally, the application passes the new system call number and the starting pointer of the buffer as the two arguments to the new system call.

The virtual memory manager in the OS needs to maintain a list of physical page frame numbers that map to each subbank, in addition to the usual color bins. Thus if there are S subbanks in the entire L2 cache, the OS needs to maintain, for each color bin, S different free lists of physical page numbers corresponding to the subbanks. These lists can be prepared as a part of the boot sequence. The OS system call handler for the new page placement call first copies the entire buffer contents from the application area to the kernel area. Next it follows the subbank mapping algorithm discussed in the last section. Once it decides a subbank number s for a cluster k, it picks the first C_k free physical page frame numbers from the corresponding list and maps the virtual pages in the cluster to these physical pages. Note that these one-time system calls do not affect the overall execution time much. Also, we did not observe any performance loss due to code size inflation resulting from the insertion of these system calls (one per cluster).

The virtual memory manager also remembers the composition of each cluster along with the mapped subbank number in a table. This information is needed when a page is swapped back into physical memory. At this time the OS finds out which cluster the virtual page being swapped in belongs to and maps it onto the assigned physical subbank by either picking a free physical page from that bin or replacing a physical page belonging to that bin.

Finally, the application needs to instruct the L2 cache controller to power down a subbank after the last access to the clusters mapped to that subbank. This is achieved by introducing another new system call. This system call is inserted after all accesses to a subbank are done. The system call passes the subbank number to be powered down. Note that if a cluster of pages contains at least one page with unordered shared accesses from a set of threads, a last touch counter needs to be maintained. The last access to such a cluster from a thread atomically increments the counter corresponding to the accessed cluster and checks if the counter value has reached the expected number of sharers, in which case it invokes the system call. In this paper, we detect the last accesses to clusters of pages from each thread by manual inspection and leave the exploration of the necessary data-flow analyses to future research.

2.5. Hardware Support

Our baseline L2 cache architecture implements the conventional drowsy cells which switch a subbank to a low voltage supply while retaining the data, if the subbank is not accessed for a threshold number of cycles. This threshold has to be high enough to be able to accommodate all the accesses to a subbank from multiple cores in a time window. In our simulations this threshold is 1000 L2 cache clock cycles. Since all our power management is done at subbank level, each subbank should be designed in isolation so that while switching the power supply of a subbank or gating off the power supply from a subbank, activities in the other subbanks are not affected. On top of this, LEMap requires couple more supports. First, it should be able to gate the power supply off from a subbank when instructed by the application. Note that since this command comes from the application, we can safely power down the subbank without writing the dirty blocks back to main memory. We can be sure that the data in the subbank will not be needed in the future. Second, LEMap enables a different power-down mode right after an application starts running. In this mode if a subbank is not accessed for a threshold number of cycles (we use 50000 L2 cache clock cycles in our simulations), it is powered down. However, this mode is disabled as soon as the first access to a subbank arrives from one of the cores marking the birth time of the first cluster mapped to that subbank. The subbank remains powered up, occasionally venturing into the low power drowsy mode, until it is powered down by the application. We do not power down a subbank during the life time of the clusters mapped onto it even if it has a significantly large number of idle cycles, since a high amount of energy is dissipated while writing back the dirty blocks to main memory and subsequently bringing them into the L2 cache, if needed in future.

3. Simulation Environment

We simulate a MIPS ISA-based eight-core chip-multiprocessor using a detailed in-house execution-driven

simulator. A high-level floorplan is shown in Figure 3. Each core has its private first level instruction and data caches. The shared L2 cache is organized into 16 banks and the bank controllers are connected to the private L1 caches through a crossbar. Each L1 cache block has four coherence states, namely, M, E, S, and I. The L1 caches are kept coherent through a directory-based write-invalidate bitvector protocol. A directory entry is maintained per L2 cache block and is kept with its tag. When an L1 cache request is forwarded to its corresponding L2 cache bank, the directory entry is looked up along with the tag and the necessary coherence actions are taken.

We size the L2 cache such that the entire chip area budget does not exceed 550 mm^2 with 65 nm process. First, we conservatively estimate the out-of-order multiple issue core size to be 4 mm×4 mm [16]. Next, we estimate that within a crossbar area of 20 mm×5 mm, we can fit 128 bidirectional 128-bit data busses, 40-bit address busses, and 9-bit control busses (6 bits for L2 cache request/response opcode type and 3 bits for source/destination core id) assuming the M4 layer wiring pitch of 280 nm [25]. Note that the crossbar connects 8 cores to 16 banks thereby requiring 128 busses in each direction. We design each wire in the crossbar with optimally placed repeaters [1] and compute the latency of each core-bank pair interconnect assuming the mid-level metal capacitance and resistance presented in [1]. The major portion of the remaining area of the chip is devoted to the L2 cache. We assume that the basic memory cell size of the L2 cache is 0.624μm^2 [25]. From CACTI [10] we estimate that the area devoted to the bit cells in a 1 MB bank is about 45% of the total area of the bank when equipped with one read port and one write port, and providing a 128-bit output. Therefore, we estimate the total area of a 1 MB bank to be 0.624μm^2×($2^{23} + 31 \times 2^{13}$)/0.45 or 12 mm^2. Note that each block has 31 bits of tag and state (20 bits of tag, two state bits to encode MESI, eight bits of sharer vector, and one MRU bit). We use the aspect ratio of a 1 MB bank from CACTI to finally compute the height and width of a bank. Including the L2 cache bank controllers and the four on-die integrated memory controllers (each shared across four L2 cache banks), we estimate the chip size to be roughly 20 mm×27 mm. The other salient features of our simulated system are shown in Table 1. All the cache latencies are determined with CACTI. We assume serial tag and data access for L2 cache banks.

Our dynamic power model is significantly influenced by Wattch [2], but improved considerably on various fronts. First, we include the power models of register free lists, issue queue payload RAMs, store forwarding logic, miss handling table (MHT), and various off-core components such as all the buffers between the crossbar and the L1 caches, the two-phase round-robin crossbar arbiter, the repeaters in the crossbar, all the buffers associated with the L2 cache bank controllers, the virtual queues in the L1-L2 interface to avoid coherence deadlocks, and various buffers, queues, and arbiters in the memory controller. Second, we use simple shift register logic instead of elaborate decoders to drive the wordlines of FIFO and LIFO RAMs. Finally, we have modified CACTI to compute the subarrays in the tag and data RAMs so that the energy-delay-squared is optimized. Our peak core dynamic power has been verified to be within 3% of the published results of the Alpha 21264 [8]. In this paper all the simulations are done with aggressive clock gating enabled.

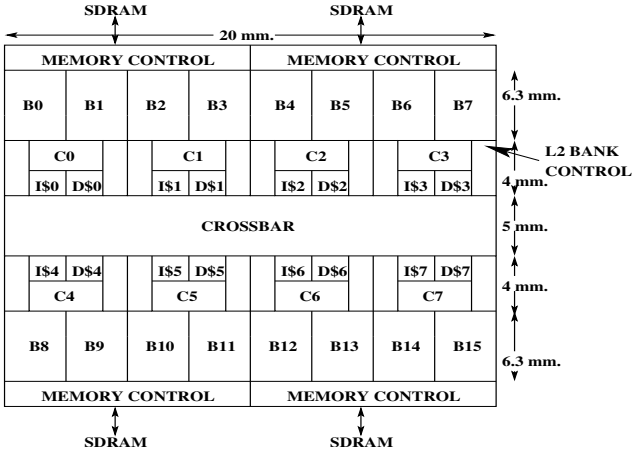

Figure 3. The simulated chip floorplan (not drawn to scale). Cn is the n^{th} core, Bn is the n^{th} L2 cache bank, I$n is the instruction cache of the n^{th} core, and D$n is the data cache of the n^{th} core.

Table 1. Simulated System

Parameter	Value
Number of cores	8 clocked at 3.2 GHz
Process/V_{dd}/V_t	65 nm/1.1 V/0.18 V
Pipe stages	18
Front-end/Commit width	4/8
BTB	256 sets, 4-way
Branch predictor	Tournament (Alpha 21264)
RAS	32 entries
Br. mispred. penalty	14 cycles (minimum)
Active list	128 entries
Branch stack	32 entries
Integer/FP Register	160/160
Integer/FP/LS queue	32/32/64 entries
ALU/FPU	8 (two for addr. calc.)/3
ITLB, DTLB	64/fully assoc./NMRU
Page size	4 KB
Private L1 Icache	32 KB/64B/4-way/LRU
Private L1 Dcache	32 KB/32B/4-way/LRU
Shared L2 cache	16 MB/128B/16-way/NMRU
L1 MSHR	16+1 for retiring stores
L2 MSHR	16 per bank
Store buffer	32
L1 cache hit	3 cycles
L2 bank tag latency	7 cycles
L2 bank data latency	3 cycles
Memory cntr. freq.	1.6 GHz
SDRAM access time	70 ns (row buffer miss) 30 ns (row buffer hit)
SDRAM bandwidth	6.4 GB/s per controller

Our leakage power model is divided into two parts, namely, subthreshold leakage model and gate leakage

model. These models are developed based on the techniques proposed in [3] and [23], respectively. However, we have improved these models wherever possible by cross-validating the leakage current values with HSPICE simulations. Our SRAM schematic model includes the cell, the sense amplifiers, the sense isolation circuitry, the write circuitry, and the precharge circuitry [19]. We appropriately extrapolate these leakage components from smaller SRAMs to derive the leakage power of bigger SRAMs. After enabling drowsy mode with a 0.3 V low power supply at a 128 KB subbank grain, our SRAM leakage comes down to roughly 1 W/MB which is usually considered the industry thumb-rule today. Since we never put the L2 tags in drowsy mode, the wakeup latency of the L2 data subbank (all 16 ways) from drowsy mode is hidden under tag access latency. Recall that the L2 cache exercises a serial tag/data access.

Finally, our approximate DRAM energy model is developed based on Micron technical notes [12, 21]. The model uses published figures for a highly loaded DDR2-400 512Mb x4 chip. The DRAM power supply is assumed to be 1.8 V. Since this chip runs at 200 MHz and we model a 400 MHz DRAM in our simulator, we scale up the frequency-dependent power components by a factor of two. Overall, our modeled power per DRAM access turns out to be 4.3 W when all the 16 participating chips on the target 1 GB DIMM are accounted for. We multiply this power by the average DRAM access latency to compute the DRAM energy per access. In this paper, we do not charge any power when a DRAM module is idle and assume that aggressive low power modes are enabled [5, 6, 17]. We also model the DRAM channel energy by assuming a channel capacitance of 30 pF [27], a supply voltage of 1.8 V, and a width of 64 bits. Note that 16 channel transactions are needed to transfer a 128-byte cache block.

We use eight explicitly parallel shared memory applications drawn from SPLASH-2, SPEC OMP, and DIS (Data Intensive Stressmark) suites. The applications along with their problem sizes are shown in Table 2. The applications are chosen to represent a good mix of compute-intensive and communication-intensive workloads. Problem sizes are chosen to get good parallel efficiency on an eight-core chip with shared L2 cache. We had to increase the ARCHduration value in the MinneSPEC input file of Equake to 0.5 to get good speedup. Both FFT and FFTW are tiled for good TLB and cache performance. We run all the applications from beginning to completion.

Table 2. Applications

Name	Input	Source
Barnes	16384 bodies	SPLASH-2
CG	8192×8192 matrix, 256K non-zeros, 50 iterations	DIS
Equake	MinneSPEC, ARCHduration 0.5	SPEC OMP
FFT	256K complex doubles	SPLASH-2
FFTW	1024×16×16 complex doubles	FFTW
LU	768×768 matrix, 16×16 tile	SPLASH-2
Radix	2M keys, radix 32	SPLASH-2
Water	1024 molecules	SPLASH-2

4. Simulation Results

In this section, we evaluate our proposal by presenting the L2 cache power, L2 cache energy, total energy (includes core, clock, L2 cache, DRAM, and various system buffers), and parallel execution time. We compare five points in the design space: baseline with no leakage power optimization (Baseline), baseline with drowsy mode enabled (Drowsy), baseline with drowsy mode and subbank power-down enabled (Drowsy+PD), access clustering with drowsy mode and subbank power-down enabled (Cluster+Drowsy+PD), and proximity-aware access clustering with drowsy mode and subbank power-down enabled (ProxCl.+Drowsy+PD). The last two design points employ our virtual address translation schemes and we would be interested to see how these two design points fare when compared to the third design point. Figures 4, 5, 6, and 7 show these results normalized to Baseline.

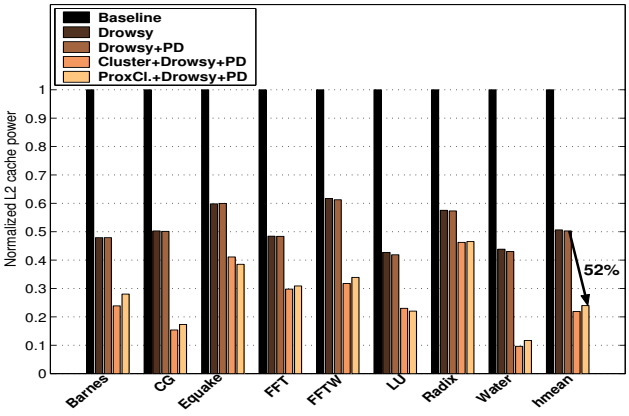

Figure 4. Power consumed by the L2 cache.

Figure 4 shows the results for L2 cache power. On average (the last group shows the harmonic mean), drowsy mode saves 50% of L2 cache power while our proposal saves another 55% of the remaining L2 cache power. Interestingly, drowsy mode with subbank power-down does not help at all when not employing access clustering. This brings out the importance of access clustering for effective application of power-down mode. In fact, we found that even the drowsy mode becomes more effective with access clustering because now each subbank shows very regular active and idle phase patterns. Although proximity-aware access clustering burns a little extra L2 cache power, on average, it saves 52% and 75% of L2 cache power compared to Drowsy+PD and Baseline, respectively. Figure 5 shows how this power saving gets translated to L2 cache energy saving. Here also our proximity-aware access clustering proposal continues to save 50% of L2 cache energy compared to Drowsy+PD.

Figure 6 shows that in terms of total energy the system with proximity-aware access clustering emerges the best design out of the five designs we consider. On average, it saves 7% and 23% of total energy compared to Drowsy+PD and Baseline, respectively. Only FFTW and Radix are the two applications that dissipate more energy with access clustering compared to Drowsy+PD. This is explained by the execution time chart shown in Figure 7. In Figure 7 we observe

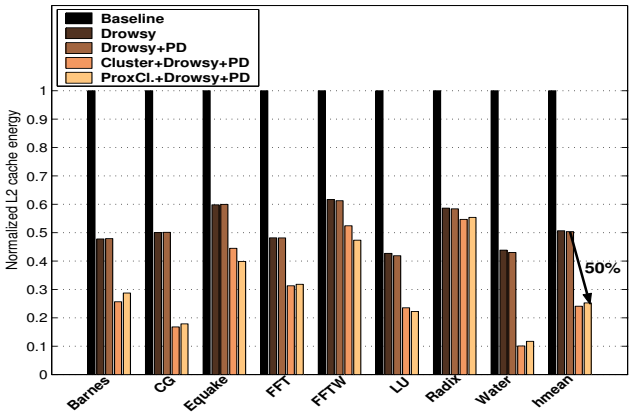

Figure 5. Energy dissipated by the L2 cache.

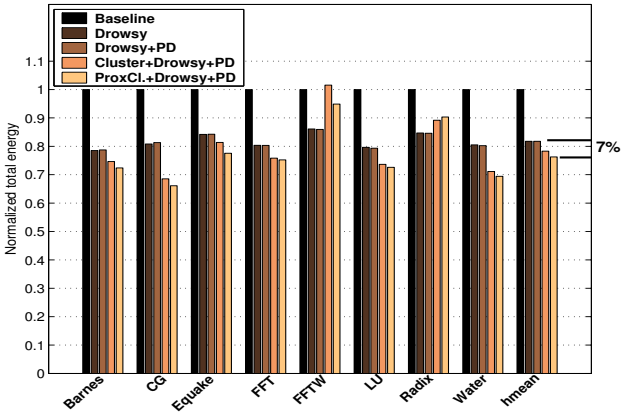

Figure 6. Total dissipated energy.

that across the board, access clustering without proximity awareness hurts performance due to extra subbank conflicts. However, proximity-aware access clustering compensates this loss by making on-chip accesses faster and brings down the loss in performance to an average of 3%. FFTW and Radix are the ones suffering from the largest performance degradation: 10% and 5% compared to Baseline. Overall, our profile-guided virtual address translation scheme offers excellent power and energy savings while nominally inflating the execution time. Introducing proximity awareness in a NUCA helps improve the performance of seven out of the eight applications that we consider.

5. Related Work

Power and energy optimization in caches has been explored extensively over the past years. In the following we briefly discuss some of the studies most relevant to our work on leakage optimization in caches. Due to shortage of space we do not discuss any dynamic power optimization techniques. The leakage optimization techniques can be largely divided into three categories, namely, circuit techniques, combination of circuit and architectural techniques, and compiler-assisted techniques. A comprehensive study of various components of leakage current in CMOS transistors and the circuit techniques for leakage optimization

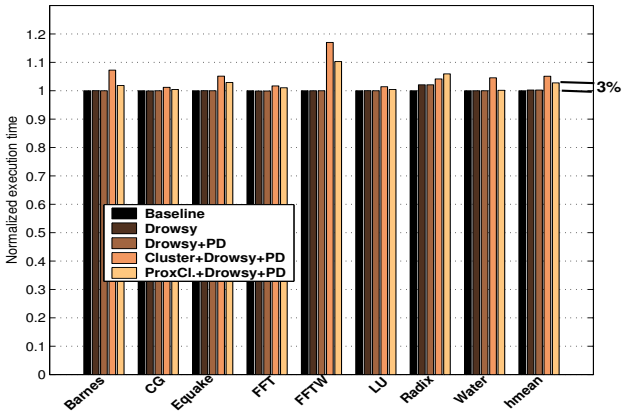

Figure 7. Parallel execution time.

can be found in [24]. These techniques include channel design, self-reverse biasing (commonly known as stacking), multi-threshold CMOS (MTCMOS) design, variable threshold CMOS (VTCMOS) design, dynamic threshold CMOS (DTCMOS) design, and supply voltage scaling. Reduction of leakage power by employing the circuit technique of leakage-biased bitlines in instruction caches and register files has been explored in [9].

Drowsy caches [7, 14] put the inactive blocks in a low supply voltage mode while retaining the data. Cache decay technique [13] invalidates and turns off the cache blocks that are unlikely to be used in future. Our proposal of LEMap uses drowsy cells as the baseline mechanism and differs significantly from the cache decay technique to take care of the access pattern in a chip-multiprocessor.

Dynamically resizable instruction cache (DRI i-cache) and gated-V_{dd} have been introduced in [22, 26] as ways to reduce leakage in instruction caches. DRI i-cache monitors the miss rate in the instruction cache and resizes the cache by adjusting the index range. Gated-V_{dd} is a circuit technique that employs stacking effect to reduce leakage current in the unused cache portions.

A software-configurable cache is presented in [27] where the associativity, total size, and block size of the cache can be configured. Various algorithms to manage the leakage energy in a multi-level cache hierarchy in the presence of prefetching are explored in [18]. Dividing the instruction cache into small "subcaches" for energy reduction and carrying out a dynamic page remapping onto subcaches via a second level of remapping and page migration for enhancing locality have been explored in [15]. Interaction between DRAM power saving modes and random and first-touch page allocation policies of OS has been explored in [17].

Compiler-directed leakage optimization strategies for instruction and data caches are presented in [28] and [29], respectively. The only studies that explore compiler-directed access clustering in single-threaded array-based programs are [5, 6]. Both the proposals aim at reducing DRAM energy by mapping variables accessed together onto the same memory module. However, the first proposal assumes that the declaration order of the variables is the same as the mapping order in the physical memory, which is often not true. The second proposal combines multiple arrays that are accessed together into a new single array so that all the accesses in a time window can be concentrated onto a few

memory modules while the other modules can be put in a low energy mode. Such a transformation may be easy for single-threaded programs, but is often very tricky for a shared memory parallel program, even for array accesses.

6. Summary and Future Directions

We have presented a virtual address translation scheme to control leakage in large multi-banked non-uniform access chip-multiprocessor caches. Our scheme maps subsets of pages that show similar access behavior onto fixed size cache regions which we call subbanks. This allows us to power down a subbank after the last access to it. A baseline mapping scheme that is oblivious to access patterns cannot apply power down modes effectively because accesses to a subbank normally get scattered throughout the entire execution. Access clustering also helps improve the effectiveness of the subbank-grain drowsy mode because each subbank now shows regular idle and active phase patterns. Further, introducing the notion of bank-to-core proximity while composing the subbanks improves performance. We evaluate our proposal through profile analysis and find that it saves 7% of total energy, 50% of L2 cache energy, and 52% of L2 cache power while suffering a 3% performance loss compared to an otherwise identical system that is oblivious to access patterns. The natural next step is to explore how the source modifications can be automated with dataflow analyses and compiler transformations. We would also like to explore prefetching to bridge the performance gap.

Acknowledgments

This work is supported by an Intel graduate fellowship and an IBM faculty award. We thank Vijay Degalahal of Intel for helping us with HSPICE and Arkaprava Basu for initial help with the clustering algorithms.

References

[1] V. Agarwal et al. Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 248–259, June 2000.

[2] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-level Power Analysis and Optimizations. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 83–94, June 2000.

[3] X. Chen and L-S. Peh. Leakage Power Modeling and Optimization in Interconnection Networks. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 90–95, August 2003.

[4] S. Cho and L. Jin. Managing Distributed , Shared L2 Caches through OS-level Page Allocation. In *Proceedings of the 39th International Symposium on Microarchitecture*, pages 455–468, December 2006.

[5] V. Delaluz et al. DRAM Energy Management Using Software and Hardware Directed Power Mode Control. In *Proceedings of the 7th International Conference on High-Performance Computer Architecture*, pages 159–170, January 2001.

[6] V. Delaluz et al. Compiler-directed Array Interleaving for Reducing Energy in Multi-bank Memories. In *Proceedings of the 15th International Conference on VLSI Design*, pages 288–293, January 2002.

[7] K. Flautner et al. Drowsy Caches: Simple Techniques for Reducing Leakage Power. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 148–157, May 2002.

[8] M. K. Gowan, L. L. Biro, and D. B. Jackson. Power Considerations in the Design of the Alpha 21264 Microprocessor. In *Proceedings of the 35th Design Automation Conference*, pages 726–731, June 1998.

[9] S. Heo et al. Dynamic Fine-grain Leakage Reduction using Leakage-biased Bitlines. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 137–147, May 2002.

[10] HP Labs. CACTI 4.2. Available at http://www.hpl.hp.com/personal/Norman_Jouppi/cacti4.html.

[11] J. S. Hu et al. Exploiting Program Hotspots and Code Sequentiality for Instruction Cache Leakage Management. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 402–407, August 2003.

[12] J. Janzen. Calculating Memory System Power for DDR SDRAM. In *Micron Designline*, **10**(2):1–12, Second quarter, 2001.

[13] S. Kaxiras, Z. Hu, and M. Martonosi. Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power. In *Proceedings of the 28th International Symposium on Computer Architecture*, pages 240–251, June/July 2001.

[14] N. S. Kim et al. Single-V_{DD} and Single-V_T Super-Drowsy Techniques for Low-Leakage High-Performance Instruction Caches. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 54–57, August 2004.

[15] S. Kim et al. Partitioned Instruction Cache Architecture for Energy Efficiency. In *ACM Transactions on Embedded Computing Systems*, **2**(2):163–185, May 2003.

[16] R. Kumar, V. V. Zyuban, and D. M. Tullsen. Interconnections in Multi-core Architectures: Understanding Mechanisms, Overheads, and Scaling. In *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 408–419, June 2005.

[17] A. R. Lebeck et al. Power Aware Page Allocation. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 105–116, November 2000.

[18] L. Li et al. Managing Leakage Energy in Cache Hierarchies. In *Journal of Instruction-level Parallelism*, vol. 5, pages 1–24, April 2003.

[19] M. Mamidipaka and N. Dutt. eCACTI: An Enhanced Power Estimation Model for On-chip Caches. *CECS Technical Report 04-28*, University of California, Irvine, September 2004.

[20] C. McNairy and R. Bhatia. Montecito: A Dual-Core, Dual-Thread Itanium Processor. In *IEEE Micro*, **25**(2): 10–20, March-April 2005.

[21] Micron Technology Inc. DDR2 Offers New Features and Functionality. *Micron Technical Note TN-47-02*.

[22] M. Powell et al. Gated-V_{dd}: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 90–95, July 2000.

[23] R. M. Rao et al. Efficient Techniques for Gate Leakage Estimation. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 100–103, August 2003.

[24] K. Roy, S. Mukhopadhyay, and H. Mahmoodi-Meimand. Leakage Current Mechanisms and Leakage Reduction Techniques in Deep-Submicrometer CMOS Circuits. In *Proceedings of the IEEE*, **91**(2):305–327, February 2003.

[25] S. Rusu et al. A Dual-core Multi-threaded Xeon Processor with 16 MB L3 Cache. In *Proceedings of the International Solid State Circuits Conference*, February 2006.

[26] S-H. Yang et al. An Integrated Circuit/Architecture Approach to Reducing Leakage in Deep-Submicron High-Performance I-Caches. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, pages 147–158, January 2001.

[27] C. Zhang, F. Vahid, and W. Najjar. A Highly Configurable Cache for Low Energy Embedded Systems. In *ACM Transactions on Embedded Computer Systems*, **4**(2):363–387, May 2005.

[28] W. Zhang et al. Compiler-directed Instruction Cache Leakage Optimization. In *Proceedings of the 35th International Symposium on Microarchitecture*, pages 208–218, November 2002.

[29] W. Zhang et al. Reducing Data Cache Leakage Energy Using a Compiler Based Approach. In *ACM Transactions on Embedded Computing Systems*, **4**(3):652–678, August 2005.