6-1-2012

Drowsy cache partitioning for reduced static and dynamic energy in the cache hierarchy

Brendan Fitzgerald

Follow this and additional works at: <http://scholarworks.rit.edu/theses>

Recommended Citation

Fitzgerald, Brendan, "Drowsy cache partitioning for reduced static and dynamic energy in the cache hierarchy" (2012). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by the Thesis/Dissertation Collections at RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.

Drowsy Cache Partitioning for Reduced Static and Dynamic Energy in the Cache Hierarchy

by

Brendan Fitzgerald

A Thesis Submitted in Partial fulfillment of the Requirements for the Degree of
Master of Science in Computer Engineering

Supervised by

Assistant Professor, Dr. Sonia López Alarcon
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
June 2012

Approved by:

Dr. Sonia López Alarcon, Assistant Professor
Thesis Advisor, Department of Computer Engineering

Dr. Muhammad Shaaban, Associate Professor
Committee Member, Department of Computer Engineering

Dr. Dhireesha Kudithipudi, Assistant Professor
Committee Member, Department of Computer Engineering

Thesis Release Permission Form

Rochester Institute of Technology
Kate Gleason College of Engineering

Title:

Drowsy Cache Partitioning for Reduced Static and Dynamic Energy in the
Cache Hierarchy

I, Brendan Fitzgerald, hereby grant permission to the Wallace Memorial Library to
reproduce my thesis in whole or part.

Brendan Fitzgerald

Date

Dedication

To my family, through thick and thin, have always supported me, as I worked toward finishing this work and my degree.

Acknowledgments

This thesis work and my degree would not have been possible without the help and support of my advisor, Dr. Sonia López Alarcon. Her guidance has helped me grow as a student and researcher, providing the foundation for the rest of my academic career. Without the feedback from my committee members, Dr. Muhammad Shaaban and Dr. Khireesha Kudithipudi, this document and my degree would not have been finished. I would also like to thank Dr. Julio Sahuquillo Borrás for his invaluable help while living across an ocean. A thanks to Paul Mezzanini in Research Computing, without his resources I would not be able to have the comprehensive results that I have, and who raised the priority of my jobs so I could get results when it mattered. I would also like to thank Dave Warth, my track coach and friend who helped to teach me how to be mentally strong. Finally, I would like to thank the Computer Engineering department for putting up with me for so many years.

Abstract

Drowsy Cache Partitioning for Reduced Static and Dynamic Energy in the Cache Hierarchy

Brendan Fitzgerald

Supervising Professor: Dr. Sonia López Alarcon

Power consumption in computing today has lead the industry towards energy efficient computing. As transistor technology shrinks, new techniques have to be developed to keep leakage current, the dominant portion of overall power consumption, to a minimum. Due to the large amount of transistors devoted to the cache hierarchy, the cache provides an excellent avenue to dramatically reduce power usage. The inherent danger with techniques that save power can negatively effect the primary reason for the inclusion of the cache, performance.

This thesis work proposes a modification to the cache hierarchy that dramatically saves power with only a slight reduction in performance. By taking advantage of the overwhelming preference of memory accesses to the most recently used blocks, these blocks are placed into a small, fast access A partition. The rest of the cache is put into a drowsy mode, a state preserving technique that reduces leakage power within the remaining portion of the cache. This design was implemented within a private, second level cache that achieved an average of almost 20% dynamic energy savings and an average of nearly 45% leakage energy savings. These savings were attained while incurring an average performance penalty of only 2%.

Contents

Dedication	iii
Acknowledgments	iv
Abstract	v
Glossary	xi
1 Introduction	1
2 Background	4
2.0.1 Set Associativity	4
2.1 Previous Work	5
2.1.1 Selective Cache Ways	5
2.1.2 Cache Hierarchy Reconfiguration	7
2.1.3 Accounting Cache	9
2.1.4 Phase-Adaptive Cache	11
2.1.5 Drowsy Cache	12
2.1.6 Temporal Locality for Drowsy Caches	15
2.2 Related Work	15
2.2.1 MorphCache	16
2.2.2 DRG-Cache	17
2.2.3 Smart Cache	17
2.2.4 Adaptive Width Data Cache	18
2.2.5 Reconfiguration Management Algorithm	19
2.2.6 Comparisons	19
3 Drowsy Phase-Adaptive Cache	21
3.1 Accounting Cache Protocol	21
3.2 Phase-Adaptive Cache	25

3.3	Cost functions	26
3.4	Drowsy Cache	27
4	Methodology	29
4.1	SPICE	29
4.2	CACTI	31
4.2.1	Energy & Power	33
4.2.2	Latency	35
4.3	Multi2Sim	36
4.4	Simulation Configuration	39
4.5	Benchmarks	40
5	Results	43
5.1	Experiments	43
5.2	Performance	44
5.2.1	Configuration Time	45
5.2.2	Accesses, Hits, & Misses	47
5.2.3	Speedup & IPC	51
5.3	Energy & Power	53
5.3.1	Dynamic Energy	53
5.3.2	Leakage Energy	55
5.3.3	Total Energy	57
5.3.4	Summary	58
6	Conclusions	59
6.1	Future Work	60
	Bibliography	61
A	CACTI configuration for L2 cache	65
B	SPICE file for SRAM Cell	70

List of Tables

3.1	Possible L2 cache configurations with associated latency	22
4.1	Various L2 banking options with associated pertinent values	32
4.2	Various L3 banking options with associated pertinent values	33
4.3	L2 energy and power numbers	34
4.4	L2 drowsy energy and power numbers	34
4.5	L2 cache configurations with associated latency	35
4.6	Memory Hierarchy Configuration	40
4.7	Processor Configuration	40
4.8	Names and descriptions of SPEC2006 tests used	41
5.1	Simulation configuration names and descriptions	44

List of Figures

2.1	Design of a single way within the Selective Cache Ways [3]	6
2.2	Subarray partitioning of a 512KB structure [6]	8
2.3	Possible configurations for the virtual two-level, physical one-level cache for a single 512KB structure [6].	9
2.4	Possible configurations of a 4-way cache and swapping of cache blocks [12]	10
2.5	GALS clock domains [12]	11
2.6	Drowsy memory circuit [13]	12
2.7	Additional logic required for the implementation of a drowsy cache line [13]	14
2.8	Various topologies of the MorphCache [30]	16
2.9	Selection logic of the Smart Cache architecture [31]	18
3.1	Representation of L2 cache configurations	23
3.2	Example access pattern	23
3.3	Example phase behavior	25
4.1	Representation of 6T SRAM cell and peripheral circuitry used in SPICE simulation	30
4.2	Simulation of 6T SRAM cell with Drowsy voltage of 0.7V	31
4.3	Interpolation and line fitting of dynamic energy for L2	34
4.4	Interpolation and line fitting of leakage power for L2	35
4.5	Interpolation and line fitting of dynamic energy for L2	36
4.6	Interpolation and line fitting of leakage power for L2	36
4.7	Multi2Sim Cache Design	38
5.1	Configuration time for Phase	45
5.2	Configuration time for PhaseED	46
5.3	Configuration time for Drowsy	46
5.4	Configuration time for DrowsyED	47
5.5	Average associativity of the simulations	47
5.6	Average percent of Hits and Misses in the A partition	48

5.7	Percentage of Hits in the A partition	49
5.8	Percentage of Hits in the B partition	49
5.9	Percentage of Misses	50
5.10	Speedup relative to the baseline	51
5.11	IPC of all the simulations	52
5.12	Dynamic energy usage of all the simulations	54
5.13	Dynamic energy savings of the simulations	55
5.14	Leakage energy usage of the simulations	56
5.15	Leakage energy savings of the simulations	56
5.16	Total energy savings of all the simulations	57

Glossary

A

- ACF** Active Cache Footprint, p. 16.
- AWDC** Adaptive Width Data Cache, p. 19.

C

- CMP** Chip multi-processor, p. 36.

D

- DVS** Dynamic voltage scaling, p. 13.

G

- GALS** Local domains run at the same frequency while the domains might not all run the same frequency, p. 11.

I

- IPC** The average number of instructions a processor can execute each cycle, p. 1.
- itrs** International Technology Roadmap for Semiconductors, p. 31.

J

- joules** The amount of energy required to move 1 ampere of current through 1 ohm of resistance, p. 54.

L

- LRU** Replacement policy that discards the cache block with the longest period of inactivity, p. 9.

M

- MOESI** Modified Owned Exclusive Shared Invalid, p. 38.
- MRO** Most Recently used On, p. 15.
- MRU** The dual of LRU, used for tracking state information, p. 9.

N

NUCA Non-uniform cache access, p. 32.

P

PDT The amount of performance loss the system is willing to accept before incurring increased energy usage, p. 6.

PWL Based on starting and ending times and values, uses interpolation to determine the intermediate values, p. 30.

R

RMRO Reused Most Recently used On, p. 15.

S

SMT Simultaneous multi-threading, p. 36.

SPICE Simulation Program with Integrated Circuit Emphasis, p. 29.

SRAM Typically a 6 or 8 transistor cell that retains a bit value as long as there is power, p. 13.

T

TMRO Two Most Recently used On lines, p. 15.

U

UCA Uniform cache access, p. 32.

Chapter 1

Introduction

Power consumption has become a major design concern in modern microprocessors. The age of ever increasing processor speeds, commonly referred as the megahertz race, has been over for some time. The end of the era of consistent speed improvements was directly related to the amount of power used by a processor, as the transistors switched faster, the heat generated and power consumption rose. This has lead to the current industry standard of increasing the core count, while keeping or even reducing the operating frequency of the processor. As manufacturing technology also improved, the ability to increase the number of transistors on a single chip has been exponential for 40 years.

A common side effect to the increasing in transistor count has been the increase in size of cache memory on the processor, both in individual cache capacity and the number of levels within the hierarchy. The increase in size is due to the discrepancy between the performance improvement of processors and memory technology, as processors improve by roughly 60% each year, while memory only improves by about 10% [24]. Reducing this gap is necessary to improving the IPC of the system.

With the cache hierarchy comprising up to 35% of the total number of transistors on a processor [29], the need to combat its power usage is paramount. Previously, various techniques have been explored that alter either the architecture of the cache [2, 15], the

circuit level design [1, 26, 36] or the materials transistors are made with [8].

This thesis work looks at a combination of transistor level and circuit level techniques to reduce power consumption within the cache hierarchy of a processor. Previous work has either looked at only dynamic power [3, 11], only static power [25], or improved performance [19, 20]. The goal is to show the potential for dynamic and static power savings with only modest reduction in IPC using a novel cache design, by combining the accounting cache, phase-adaptive cache, and drowsy cache.

By leveraging the inherent temporal locality of cache, where up to 92% of accesses are made in the most recently used (MRU) cache line and up to 98% of accesses are made in the two MRU cache lines [25], splitting the cache into two partitions: an active A partition and a drowsy B partition. After recording the hits in each MRU state, the phase-adaptive cache can alter the partitions dynamically using cost functions that are based on either the energy usage or the energy-delay product. The total set associativity of the cache level never changes, though the associativity of each partition can change, the two partitions associativity always compliment each other.

Dynamic power is saved by first only accessing the A partition, which is generally only a portion of the cache, with the B partition only being accessed when the data is not located in the A partition. To keep the most recently used cache blocks in the A partition, the data is swapped between partitions when the required block is located in the B partition or found in the next level of cache. While this technique saves dynamic power, it does not save static power, due to the operation of SRAM cells. In this thesis work, dynamic energy consumption is assumed to be averaged over the entire cache, regardless of where the requested cache block may be located.

To save static power, the B partition is kept at a lower operating voltage, which reduces static power consumption and after modifying the cost function of the accounting cache to account for both dynamic energy and leakage energy, the A partition is sized to just one

way, with the B partition sized to the remaining seven ways. Even with this configuration, performance is kept well within acceptable limits. This reduces the need for complex logic to determine the costs for each possible configuration, requiring just a small amount of circuitry to swap cache blocks between partitions.

The rest of this thesis is organized as follows: Chapter 2 discusses the previous work along with other techniques that have been proposed. Chapter 3 explains how the proposed design works. Chapter 4 explains the testing and simulation setup. Chapter 5 gives the results of the work and Chapter 6 provides conclusions and possible future work.

Chapter 2

Background

As smaller and an ever increasing amount of transistors are put in processors, the amount of power and consequently the heat that is dissipated continues to grow out of hand. Current generations of CPUs have up to 35% of the transistors allocated to cache [29]. Because of the importance of the cache hierarchy to the performance of CPUs, considerable amounts of research work has been done in search of techniques that save power while minimizing performance loss. First, a review of set associativity is presented. Next, previous, related work is described to give an outline of what has been inspiration for this thesis work. Finally, related work is described to show other techniques that have been researched.

2.0.1 Set Associativity

Within each level of the cache hierarchy, the degree of associativity becomes a trade-off that computer architects need balance carefully. At one end of the spectrum, the direct-mapped cache means a specific cache block is always placed in the same location based on the index bits. This means that when another cache block has the same index as one that is already sitting in the cache, the first block is evicted. This process can repeat indefinitely depending on the access pattern of the program, decreasing performance. The upside to the direct-mapped cache is the speed and reduced complexity of the design.

At the other end of the spectrum, a fully associative cache means that a cache block can be placed anywhere within the cache level. This provides the best hit rate, as blocks placed in the cache are only evicted when the cache is full of data. The downside to a fully associative cache is that it is very complex, as the entire cache has to be searched to find the requested block.

Between these two policies lie an infinite amount of n-way set associative cache policies. A typical set associative cache is commonly 2-way, 4-way, 8-way, or 16-way set associative. This means that multiple cache blocks sharing the same index can reside in the cache at the same time, increasing the hit ratio at the expense of increased complexity. This thesis work examines an 8-way set associative L2 cache and splits the cache into two partitions. In essence, each partition will have its own associativity, but overall the associativity of the entire cache does not change. For example, when the cache is evenly partitioned, each partition will be a 4-way set associative cache, while collectively the cache is still an 8-way set associative cache.

2.1 Previous Work

2.1.1 Selective Cache Ways

Selective Cache Ways [3] by Albonesi is the starting point for much of this work. Albonesi's work took advantage of the partitioning inherent in the design of cache structures. A trade-off can be made between having a small performance penalty for significant energy savings. For this technique to function, both hardware and software modifications have to be made. The general hardware overview can be seen in Figure 2.1. The tag arrays were left alone as the energy savings was not enough to warrant the increased performance penalty with disabling tag ways corresponding to data ways. The cache controller uses four

Figure 2.1: Design of a single way within the Selective Cache Ways [3]

bits to enable or disable particular data ways within the cache. When a way is disabled, the precharge, sense amplifiers and decoders are disabled, so the particular way effectively dissipated no dynamic power. The cache controller also needs to be modified so when a way is disabled new data cannot be brought into that particular way. Because data is not flushed with this method, the data that is found in a disabled way within DL1 needs to also reside in L2, making it an inclusive cache [5].

To determine the number of ways that are to be disabled, a performance degradation threshold (PDT) is set at either 2%, 4%, or 6%. Should the IPC fall past one of the three thresholds, an additional way will be enabled to increase performance. To control the cache ways, an additional register called the *Cache Way Select Register (CWSR)*, is added

to the cache to signal to the hardware which cache ways are enabled and which cache ways are disabled. The length of the CWSR is n -bits, where n is the number of cache ways in the particular level. As seen in Figure 2.1, the CWSR is only four bits, equal to the number of ways. To control the CWSR, two additional instructions need to be added to the instruction set architecture (ISA), RDCWSR and WRCWSR for reading and writing respectively. Overall, this design provided a 40% reduction in energy dissipation in the cache hierarchy with a modest 2% loss in performance.

2.1.2 Cache Hierarchy Reconfiguration

One downside to this approach is that this is a static method of saving energy, the program needs to be profiled before setting the ways to be enabled or disabled [6]. This never took into account that programs can have periods of low cache activity followed by periods of high activity or even thrashing, requiring the entire cache to be enabled. Balasubramanian *et al.* looked to solve the problems encountered by Albonesi by creating a dynamically configurable cache that behaves as a virtual two-level, physical one-level non-inclusive hierarchy [6]. Starting with an overall cache of 2MB, it was divided into two 1MB banks, with each bank further broken down into 512KB structures. Figure 2.2 shows the organization of a 512KB structure.

Figure 2.3 shows the partitioning of a virtual two-level, physical one-level, non-inclusive cache design. This is used as a replacement for a traditional two-level cache hierarchy. Each block represents a 128 KB cache block, with the top line acting as a direct mapped L1 cache, and the second line acting as a two-way set associative L1 cache. Because the cache can be partitioned in this way, it is possible to reduce the energy consumption while keeping the same size of the cache while having a reduction in the set associativity.

Accessing data is slightly different from normal cache operation. Using Figure 2.3 as

Figure 2.2: Subarray partitioning of a 512KB structure [6]

a reference, on a hit in L1, a single way is accessed. On a miss in a specific L1, all of the tag arrays are read in parallel to speed up the search, should the data be found, the blocks are swapped between the specified L1 way and the way in L2 that the data was found. If the data is not present in the cache, the data in the specified L1 way is moved to L2 and the new data is placed into L1. This ensures the non-inclusive nature of the cache is kept without trashing L1.

The initial size of the cache was set to the smallest size, a direct-mapped 256KB L1, and the initial state was set to unstable. This means that at the end of the current period the cache should be evaluated to see if the cache performed within accepted levels of tolerance. These are based on the miss rate of the cache, the overall IPC of the system and the branch frequency of the application. If the cache does not perform within the tolerance levels, the size is increased to the next largest size, with this process continuing until the cache reaches the maximum size or the working set fits within the current configuration. The

				Subarray/Way Allocation (L1 or L2)															
Cache Configuration	L1 Size	L1 Assoc	L1 Acc Time	Subarray 2				Subarray 0				Subarray 1				Subarray 3			
				W3	W2	W1	W0	W3	W2	W1	W0	W0	W1	W2	W3	W0	W1	W2	W3
	256-1	256KB	1 way	2.0	L2	L2	L2	L2	L2	L2	L1	L1	L2	L2	L2	L2	L2	L2	L2
	512-2	512KB	2 way	2.5	L2	L2	L2	L2	L2	L1	L1	L1	L1	L2	L2	L2	L2	L2	L2
	768-3	768KB	3 way	2.5	L2	L2	L2	L2	L2	L1	L1	L1	L1	L1	L1	L2	L2	L2	L2
	1024-4	1024KB	4 way	3.0	L2	L2	L2	L2	L1	L1	L1	L1	L1	L1	L1	L2	L2	L2	L2
	512-1	512KB	1 way	3.0	L2	L2	L2	L1	L2	L2	L2	L1	L1	L2	L2	L2	L1	L2	L2
	1024-2	1024KB	2 way	3.5	L2	L2	L1	L1	L2	L2	L1	L1	L1	L1	L2	L2	L1	L1	L2
	1536-3	1536KB	3 way	4.0	L2	L1	L1	L1	L2	L1	L1	L1	L1	L1	L1	L2	L1	L1	L1
	2048-4	2048KB	4 way	4.5	L1	L1	L1	L1	L1	L1	L1	L1	L1	L1	L1	L1	L1	L1	L1

Figure 2.3: Possible configurations for the virtual two-level, physical one-level cache for a single 512KB structure [6].

configuration will be considered stable when the current number of misses and branches are similar to the previous interval and should the configuration change to unstable, the cache is set to the smallest configuration. To save energy, the authors chose to use a serial tag-data lookup, as opposed to the faster, but higher energy parallel tag-data lookup. Overall this technique was able to reduce CPI by 15% when configured for the L1/L2 setup and reduce energy in the cache by 43% when configured for L2/L3 setup.

2.1.3 Accounting Cache

This work all came together in creating the *Accounting Cache*, where Dropsho *et al.* [11] leveraged least recently used (LRU) state information to calculate the ideal configurations within the cache hierarchy. To determine the next configuration, counters are added to the cache to record most recently used (MRU) statistics. The total number of counters required are $n+1$, where n is the number of ways and an additional counter for the misses. The *Accounting Cache* keeps track of the performance and energy of the system over time, allowing it to save or spend according to the set performance degradation threshold (PDT),

which was set to either 1.25%, 6.2% or 25%, 1/64, 1/16, or 1/4 respectively.

Figure 2.4: Possible configurations of a 4-way cache and swapping of cache blocks [12]

The access protocol is as follows: The initial access is made to the A partition and if the required block is located, the associated counter is updated and the block is returned. If the block is not found in the A partition, a subsequent, parallel search is made to both the B partition and the next level in the memory hierarchy. If the block is found in the B partition, the associated counter is updated and the block is swapped into the A partition, as well as being returned. If the block has to be brought in from the next level, it is placed into the A partition and the displaced block is moved into the B partition, with the displaced block in the B partition either being overwritten if clean, or written back if dirty.

Due to the design of cache, the tag array is also able to be controlled in the same manner as the data array, providing even more possibilities for energy savings. Thus, the tag array has two options: either mirror the data array partitioning which will help to reduce energy consumption, or keep the tag array full which consumes more energy but allows for faster access. The final option is to either access the tag and data arrays in parallel, or to access the tag array first, followed by the data array. The serial access provides some performance loss, but the data array will not be accessed if the required cache line is not found, reducing

energy consumption.

2.1.4 Phase-Adaptive Cache

Dropsho *et al.* [12] further refined the *accounting cache* into a *phase-adaptive* cache that would dynamically adjust the size and speed of the cache based on past usage, with the goal of increasing performance. To accomplish this a globally asynchronous, locally synchronous (GALS) design was used, allowing for the different operating frequencies between the domains within a processor, shown in Figure 2.5. Using this method, the latency to access the A partition was always took the same number of cycles, but the operating frequency changed. When the A partition was at its smallest, the operating frequency was the highest. Depending on the size of the A partition, the latency to access the B partition varied greatly. When sized equally, the latency to access each partition is the same, but when the A partition is as small as possible, the latency to access the B partition can be three or more times longer than an access to the A partition.

Figure 2.5: GALS clock domains [12]

To compare the *phase-adaptive* cache, offline simulations were performed to determine

Flautner *et al.* proposed a new technique to reduce subthreshold leakage within SRAM cells called a *drowsy* mode [13], as seen in Figure 2.6. In essence, the SRAM cell is either supplied the nominal supply voltage or supplied a lower, state-preserving voltage. Note that the voltage controller, seen in Figure 2.6, creates a virtual supply voltage (VV_{DD}). In [18], Kim *et al.* explain how dynamic voltage scaling (DVS) can be used as a state-preserving method to reduce leakage currents within the cache hierarchy. Whenever there is a bit being stored within an SRAM cell, there will inevitably be two transistors within the cross-coupled pair that are in the off-state. This leads to leakage paths when the transistors are operating in weak inversion, modeled with 2.1 [27]:

$$I_D = I_S e^{\frac{V_{GS}-V_T}{nkT/q}} \left(1 - e^{-\frac{V_{DS}}{kT/q}}\right) (1 + \lambda V_{DS}) \quad (2.1)$$

where λ is the parameter for channel-length modulation. Equation 2.1 leads to equation 2.2 that defines the leakage current in an SRAM cell [18]:

$$I_L = ((I_{SN} + I_{SP}) + (I_{SN}\lambda_N + I_{SP}\lambda_P) V_{DD}) \times \left(1 - e^{-\frac{V_{DD}}{nkT/q}}\right) \quad (2.2)$$

where N and P are the values for nMOS and pMOS transistor types respectively and I_{SN} and I_{SP} are the nMOS and pMOS off-transistor current factors independent of equation 2.1. From equation 2.2, it can be seen that a slight reduction in V_{DD} will have a profound affect on the leakage current within the SRAM cell. Kim *et al.* were able to see a reduction in leakage by almost 80% by supplying a voltage 50% higher than the threshold, in this case a voltage of 300 mV was used for the technology node of 70 nm. To control each cache line, additional logic must be included between the decoders and the SRAM cells, as seen in figure 2.7. To prevent accidental destruction of data, wordline gating is used to stop such an occurrence, which requires a single AND gate for each wordline.

Figure 2.7: Additional logic required for the implementation of a drowsy cache line [13]

Because the drowsy cache is state preserving, the penalty for putting a cache line into the drowsy state does not incur the same performance hit that gated- V_{DD} has. If the cache tag array is designed to not use the drowsy circuits, there is only a 1 cycle penalty to 'wake up', or raise the voltage in the cache line, whereas when the tag array is designed with drowsy circuits, it takes a total of 3 cycles to access the data in a drowsy line. The authors used a fixed window size of 4000 cycles, at which time the entire cache was put into a drowsy state. This policy provides the benefit of having just a single hardware counter for the cycle count, at the expense of requiring extra cycles to wake up lines when they need to be accessed. Altogether, the authors were able to reduce the total energy usage of the data cache by 50% with only incurring a very slight increase in run time.

2.1.6 Temporal Locality for Drowsy Caches

Petit *et al.* [25] extended the work on drowsy caches to look to determine the best overall policy based on temporal locality. They found that within a DL1 cache with 4 ways that more than 92% of the hits are to the MRU line, and when looking at the MRU and 2nd MRU lines, more than 98% of the hits are found there. These two policies, most recently used on (MRO) and two most recently use on (TMRO) were compared against Flautner *et al.*'s *simple* policy [13], where all of the cache lines are periodically, every 4096 cycles, put into a drowsy state. Petit *et al.* proposed the use of *reuse* information to determine which state to put cache lines into, which leads to the policy of reused most recently used on (RMRO). Usage statistics are gathered during windows of execution, and when the window is closed, the information gathered is used to determine if the line should be placed into the drowsy state, keep one line awake and the rest in the drowsy state (MRO policy), or keep two lines awake and the rest in the drowsy state (TMRO policy).

2.2 Related Work

There have been several other techniques explored that look for different ways to accomplish the same task. An easy way to effectively stop subthreshold leakage current is to use power gating [26], but this technique is not state preserving, whereas the drowsy circuit allows for low-leakage operation while keeping the state of the SRAM cells intact. Power gating also has the unfortunate consequence of incurring increased energy usage due to the need to bring the discarded data back into the cache from the higher level, be it a cache or main memory.

2.2.1 MorphCache

Srikantaiah *et al.* proposed MorphCache [30], a reconfigurable adaptive multi-level cache hierarchy. Instead of altering the sizes of individual cache structures, they proposed modifying the topology of the cache. The L2 and L3 caches are effectively shared between multiple cores, but they can be partitioned into varying sizes depending on the application workload. As seen in Figure 2.8, the possible topologies of the cache do not necessarily need to be symmetric. This provides the opportunity to have both single threaded workloads and parallel workloads running at the same time with optimal cache configurations.

When combining slices of cache, care must be taken to ensure that the new size of L2 is not larger than L3. Another issue that needed to be handled is the coherence when combining slices, which was solved by invalidating duplicated data after an access to one of the cache blocks. To help determine the configuration of the cache hierarchy, the authors use the *Active Cache Footprint* (ACF) as a metric as opposed to the traditional miss rate, throughput or access latency. The ACF is a measure of unique cache line accesses within a time period, but it only tracks the cache lines that were brought in during the current period, not previous ones.

Figure 2.8: Various topologies of the MorphCache [30]

2.2.2 DRG-Cache

The DRG-Cache, or Data Retention Gated-Ground Cache, from Agarwal *et al.* looks at building upon the gated-ground technique [26] to further reduce leakage in SRAM [1]. The gating transistor takes advantage of the stack effect [36] to reduce leakage current. To control the DRG-cache, the transistor is connected to the row decoder to signal which cells are put into standby leakage mode and which cells are active. While this technique can drastically reduce leakage within SRAM, it requires an increase in dynamic energy whenever a cell is read or written, it also has the ability to destroy the data being stored within the SRAM cell. The biggest drawback to this technique is determining the optimal size of the gated transistor. A large width transistor improves access time and data retention at the expense of reducing the amount of leakage saved. Since the purpose of cache is to provide fast access to data, stability of the data is paramount, so the amount of saving possible with this technique is limited.

2.2.3 Smart Cache

Sundararajan *et al.* presented a design they designated as the *Smart Cache* [31], where both the number of sets in the cache and the associativity can be reconfigured. The sets are controlled through the use of size selection bits that are ANDed with index bits, which are the lower order tag bits, that vary in size based on the cache size. The ways are controlled using the last two bits of the tag, depending on the size of the cache, and two control bits. The complete circuit can be seen in Figure 2.9, where both selection circuits are run in parallel with other functions, as to not affect performance. Issues arise with this technique where a small cache size requires all of the dirty cache lines in the unused portions to be flushed, requiring a costly write-back to the next level. Increasing the size of the cache can also incur a penalty as some cache blocks may map to a different set, requiring a write-back

of the dirty data and a read to put the correct data in the correct location.

Figure 2.9: Selection logic of the Smart Cache architecture [31]

To control the next configuration, a decision tree is used that is determined through machine learning techniques. The decision tree relies on two statistics gathered during runtime: stack distance and dead set counts. A set is considered dead when the set has not been accessed more than three times within a predetermined interval, through the use of 2-bit saturating counters. Should the number of dead sets get large enough, a smaller cache size can be utilized.

The authors compared a static cache with not only their *Smart Cache* design, but also an oracle based system, which always picks the optimal solution. The strange outcome is that there are times in which the *Smart Cache* out performs the oracle system. If the oracle always picks the best solution, then it should not be possible for the *Smart Cache* to use less energy.

2.2.4 Adaptive Width Data Cache

Instead of partitioning sets, ways or slices of the cache hierarchy, Jiongyao *et al.* dynamically partitioned the cache lines themselves based on the length of the data being stored

[17]. Their technique, an *Adaptive Width Data Cache* (AWDC), gates off sections of a cache line based on the size of the data, either 4 bits, 16 bits, or a full 32 bits. A data type detector is used to determine the length of the data and which sections of the cache block can be turned off accordingly. This requires an additional two bits for every cache block, adding to the total power consumption of the SRAM. On reads, the lowest four bits of the cache block are initially read along with the two control bits and depending on the value, additional pieces of the cache block can be read out.

2.2.5 Reconfiguration Management Algorithm

Chen et al [9] looked at a three-dimensional reconfigurable cache organization that modifies the capacity, line size and associativity for embedded systems. The authors introduce an algorithm they designate as a reconfiguration management algorithm (RMA) to dynamically adjust the cache. This technique requires the use of a search heuristic to reduce their space from 48 possible configurations to an average of 10. This work only considered dynamic energy, which currently is not the major concern of computer architects.

2.2.6 Comparisons

Unlike the work just presented, this thesis work requires very minimal modifications to the cache hierarchy to save power, where the AWDC requires detectors and control logic, the MorphCache requires modifications to the interconnection network, and the RMA modifies three dimensions of the cache. The proposed design is also state-preserving, unlike the DRG-Cache, or any other power gating techniques. Unlike the SmartCache, this design works without the need to profile a program, which works great under the assumption of a single program running completely in a processor, but not great under context switching environments.

Summary

This chapter presented previous work that is directly related to the design of this thesis work along with alternative designs that aimed to either accomplish the same goal, saving static and dynamic power with minimal reduction of performance, or a subset of it. The next chapter explains the high level design of this thesis work and how the previous work is put together.

Chapter 3

Drowsy Phase-Adaptive Cache

The drowsy phase-adaptive cache presented in this thesis is a combination of the accounting cache and phase-adaptive cache, which both strived to save dynamic energy within the cache while incurring a slight performance penalty, and integration of the drowsy cache that aimed to save leakage power. The accounting cache splits the cache into two partitions, where each partition is always a subset of the number of ways within the cache, and reduced dynamic power consumption by accessing the A partition before accessing the B partition.

This thesis work is performed within a single core of a tiled CMP, where the first and second level cache are private and the third level is shared between each core. Because the first level cache (IL1 & DL1) is small, vital to performance, and has comparatively little energy consumption, this work is focused on the second level cache (L2), with an equal share of the third level (L3) that is kept constant.

3.1 Accounting Cache Protocol

The accounting cache employed in this work was originally proposed in [3], and it is divided into two partitions, each one containing a subset of the total ways. The number of ways contained in each partition depends on the cache configuration, and in this thesis, ranges from one to eight. To reduce the state space of possible configurations, this thesis

restricts the resizing of the A partition to powers of two: 1/7, 2/6, 4/4, and 8/0 ways in the A/B partitions, Table 3.1. These configurations will be referred to as C0, C1, C2 and C3, respectively. The base configuration (smallest size in the A partition) acts as a 32 KB direct-mapped L2 Data Cache, which can be upsized by increasing the associativity of the A partition up to 8 ways (corresponding configuration C3, 8/0). To determine the next configuration, counters are used to keep track of the hits within each MRU state along with a counter for the misses. Assuming a four way cache, if the majority of accesses are to the first and second MRU states, the cache will be partitioned into a two way A partition and two way B partition, whereas if the majority of accesses are only in the first MRU state, the cache will be partitioned into a one way A partition and three way B partition.

Table 3.1: Possible L2 cache configurations with associated latency

Name	A partition	B partition
C0	1-way 6-cycle	7-way 8-cycle
C1	2-way 7-cycle	6-way 8-cycle
C2	4-way 8-cycle	4-way 8-cycle
C3	8-way 8-cycle	0-way 0-cycle

Figure 3.1 shows the four possible configurations listed in Table 3.1 that is used in this thesis work. The yellow blocks represent the active A partition while the green blocks represent the drowsy B partition. Independent of the configuration, the general set associativity does not change, the L2 cache used will always be an 8-way set associative cache, but due to the access protocol, searching the partitions can act as different set associativity. The A partition of configuration C0 acts as a direct mapped cache, with the B partition acting as a 7-way set associative cache.

The protocol that the phase-adaptive design uses is as follows: first, the cache searches

Figure 3.1: Representation of L2 cache configurations

the A partition for the requested block and if it is located, the corresponding MRU counter is updated and it is returned. If the block is not found in the A partition, both the B partition and the next level cache is searched. If it is found in the B partition, the access to the higher level is canceled, the MRU counter is updated and the block is swapped into the A partition. If the block is located in the next level cache, the block is brought into the A partition, and a block from the A partition is swapped into the evicted block's location from the B partition. Last, the MRU states are set to reflect the changes, with the requested block being set to MRU_0 and the other blocks increasing accordingly.

Figure 3.2: Example access pattern

Figure 3.2 shows an example access pattern of a 4-way cache partitioned into a 2-way A partition, 2-way B partition. The blocks of color at the top represent the current MRU state that a cache block is in: yellow is MRU state 0, red is MRU state 1, magenta is MRU state 2, and black is MRU state 3 or the LRU block. On the first access, block B is hit, and the corresponding MRU counter is increased. On the second access, block C is hit, and the corresponding MRU counter is increased. Because block C is in the B partition, it is swapped into the A partition. On the third access, block C is hit and the MRU counter is increased, which is now MRU[0]. Finally, block D is accessed, and MRU[3] is increased.

This design, similar to the original accounting cache, saves dynamic power by avoiding the surrounding logic needed to access data belonging to the B partition. This includes the bitline precharge circuits, row and column decoders, sense amplifiers and comparators. By putting the B partition into the drowsy state, leakage current is reduced, saving static power. The downside of this design is two-fold: first, because of the access protocol, data found in the B partition takes extra cycles to be accessed because the partitions are searched serially, reducing the IPC. Second, putting the B partition into the drowsy state requires an addition cycle to raise the voltage before the data can be accessed, further reducing the IPC. On the other hand, the design takes advantage of the inherent predictability of programs, specifically the temporal locality of data, since the most recently used block is always located within the A partition due to the swapping. In this work, the tag array is first searched, so determining where the requested cache block is located can be accomplished faster and uses less energy. This method also provides a minimal difference in the number of cycles required to access either partition.

3.2 Phase-Adaptive Cache

Similar to the work in [11, 19, 20], this thesis work relies on a set number of instructions to function as the windows or *phases* to collect statistics. A phase of 15,000 instructions was used [20] as opposed to a phase of 100,000 instructions [11], which provides granularity without causing performance degradation. To ensure that the cache does not constantly change configurations, a warmup period of one phase is used after each configuration change, reducing possibilities of abnormal phase behavior. Figure 3.3 shows an example of this phase behavior. Assuming a four way cache, initially it cache is set to the full four way A partition. After 15,000 instructions, the initial warmup period, the cache is partitioned in half, with the A partition in yellow and the B partition in green. During phase 1 and phase 2, there is no change in behavior of the program. During the phase 3, behavior changes and at the start of phase 4 the cache structure is changed to a one way A partition, three way B partition to reflect the past access pattern.

Figure 3.3: Example phase behavior

The next best configuration is always calculated regardless of the current configuration and costs for all of the possible configurations are found at the end of each phase. This is possible from the use of the MRU counters and a simple decision tree. Through this method, the number of possible configurations are limited by the set associativity of the

cache. For this thesis work, the A partition is always a power of 2, as seen in Table 3.1, but it is possible to have the A partition set to any subset of the total number of ways within the cache, providing more granularity in the performance/energy consumption tradeoff.

3.3 Cost functions

The generic cost function, shown below, was presented by Dropsho *et al.* [11], where the cost was based either on delay or energy. López *et al.* [20] defined the cost as the amount of time required to access the data normalized to the frequency. The general cost function is as follows:

$$Cost = hits_A * cost_A + hits_B * cost_B + misses * cost_{misses} \quad (3.1)$$

where $hits_A$ and $hits_B$ are the number of hits in the A and B partitions, $misses$ are the number of misses in the particular cache level, and $cost_A$, $cost_B$ and $cost_{misses}$ is the cost, which can be either power, latency or both, of accessing the partitions or a miss. To calculate $hits_A$ and $hits_B$ for a particular configuration, the MRU state counters are added together based on the possible configurations listed in Table 3.1.

This work looks primarily at energy, with the energy cost function being defined as:

$$\begin{aligned} EnergyCost_i = & hits_A * DynamicEnergy_A + hits_B * DynamicEnergy_B \\ & + misses * cost_{misses} + LeakageEnergy_A + LeakageEnergy_B \\ & + swaps * (DynamicEnergy_A + DynamicEnergy_B) \end{aligned} \quad (3.2)$$

with each possible configuration being calculated at the end of each phase. All of the possible values can be found in either Table 4.3 or Table 4.4. The $LeakageEnergy_{[A,B]}$ can

be ignored for the phase adaptive cases, as there is no savings, but the energy is included in the final energy calculations to compare against the baseline, and $cost_{misses}$ is the cost of accessing the next level cache. There are four possible configurations, with the A partition always being a power of 2, as seen in Table 3.1. The associated latency for each configuration are included, ignoring the added cost to wake up the drowsy partition.

$$DelayCost = hits_A * Latency_A + hits_B * Latency_B + misses * Latency_{misses} \quad (3.3)$$

For the Energy-Delay simulations, the final equation is the multiplication of 3.2 and 3.3.

$$Cost = EnergyCost * DelayCost \quad (3.4)$$

The next logical cost function is the Energy-Delay², but testing showed no noticeable difference between that and the Energy-Delay function at the expense of needed extra logic to compute two multiplications.

The energy for the tag array is not included with the cost functions because the tag array is not partitioned. Because of this, the tag energy is added into the total energy consumption to reflect the difference in energy consumption between the simulations and a conventional cache design.

3.4 Drowsy Cache

In order to save leakage power, the B partition is put into the drowsy mode. This is accomplished by lowering the operating voltage of the cache lines. Because 92% of the hits within a cache are to the MRU line and 98% of the hits are to the two MRU lines [25], putting a large portion of the cache lines into the drowsy state does not effect the performance much. Whenever the data requested is located within the B partition, the voltage is

raised to the normal operating voltage, and the data is swapped into the A partition. This process incurs a slight penalty, one cycle additional latency, to raise the voltage so the data stored is not lost.

Summary

This chapter presented the various techniques that are combined to make this thesis work possible. A combination of the drowsy cache, accounting cache and phase-adaptive cache will be used to look towards achieving reduced static and dynamic power consumption while minimally affecting performance. In the next chapter, the methods of determining the values, and the values themselves, required for the simulations are shown.

Chapter 4

Methodology

To perform the proposed work, several preliminary simulations had to be done. SPICE simulations were first performed to ensure that an SRAM cell could retain data at a lowered operating voltage. CACTI was then used to gather parameters for the cost functions at the intended technology node of 32 nm. Finally, all of the information is added to a modified simulation framework, Multi2Sim, to test a superscalar, out-of-order processor.

4.1 SPICE

To ensure that the SRAM cells would be able to retain the stored data after lowering the voltage to 0.7V, SPICE simulations had to be done. Figure 4.1 represents the circuit used in the SPICE simulation of the SRAM cell to ensure that the drowsy mode would be state-preserving. The SRAM cell itself is represented by transistors P_1 , P_2 , N_1 , N_2 , N_3 , and N_4 . Transistors P_3 and P_4 and the inverter connected to the *precharge* signal represent the precharge circuit. Transistors N_5 , N_6 , N_7 and N_8 , and the bottom inverter and the *write* and *data* signals represent the write driver.

To read data from the SRAM cell, the bitlines are precharged high, then the wordline is pulsed high, allowing the data to be read out at one of the access transistors, with the compliment at the other access transistor. For writes, the desired value is put on the bitline,

Figure 4.1: Representation of 6T SRAM cell and peripheral circuitry used in SPICE simulation

then the wordline is pulsed, and the desired bit is stored within the cell [34].

Using ngspice [23] to perform the simulations, the following details the plots in Figure 4.2. V_{DD} starts off at 0.9V for 7ns, then falls to 0.7V at 7.33ns at a gradual, interpolated slope, using the SPICE command *PWL* standing for Piece-wise Linear. After V_{DD} reaches 0.7V, it holds there for until 19ns, at which case the voltage is raised to the nominal 0.9V over a period of 0.33ns. This ensures that the data can be kept at the drowsy voltage and also read back out after returning to the active state. The wordline is pulsed from ground to V_{DD} back to ground between 3ns and 5ns. At this point, the desired bit of data is written into the cell and from 7ns to 8ns, the voltage falls to 0.7V and then holds steady for the rest of the simulation. While it is possible to continue to lower the voltage in hopes of saving leakage power, it would come at the cost of introducing defects due to the manufacturing process. Further simulation showed it is possible to continue to reduce the operating voltage, saving increased amounts of leakage power, but this would only work in an ideal world.

Figure 4.2: Simulation of 6T SRAM cell with Drowsy voltage of 0.7V

4.2 CACTI

First, the physical parameters of the cache had to be determined. To accomplish this, CACTI 6.5 [22], was used. CACTI is a program that models access time, cycle time, area, leakage power and dynamic energy of a cache by utilizing data from the International Technology Roadmap for Semiconductors (itrs) to help accurately model both present and future cache architectures. Before any simulations could be ran a few initial values had to be found. As mentioned previously, this work focuses on the second level cache within a single core of a CMP. The split first level cache is vital to performance and its small size doesn't consume much energy. The last level cache is assumed to be shared equally

between the multiple cores, and extending this work to the third level is left for future work.

Uniform cache access (UCA) was used instead of non-uniform cache access (NUCA) so the access time to any block would be the same. Two exclusive read and two exclusive write ports were used which provides multiple access opportunities at the expense of increased cell size and power usage[16]. Once the parameters that would remain fixed were set, the banking had to be determined. To do this, the smallest subset of the cache, *i.e.* an 32KB, 1-way partition of the L2 cache and simulations were ran with varying amounts of UCA banks.

Table 4.1: Various L2 banking options with associated pertinent values

Banks	Access Time (ns)	Dynamic Read (nJ)	Leakage/Bank (mW)	Total Leakage (mW)
1	1.14156	0.0465552	18.279	18.279
2	0.927271	0.0531252	10.4309	20.8618
4	0.917925	0.112982	9.93614	39.74456

As seen in Table 4.1, two banks per way was chosen for L2 as the best overall design point, access time does not significantly improve when increasing the number of banks, but both dynamic energy and leakage power are reasonable compared to one bank per way. Table 4.2 shows the simulation results for L3, in which four banks, the highlighted row, is chosen as the best balanced number. One limiting factor ended up being the maximum number of banks for the size of the cache, as too many banks causes an error within CACTI, where it cannot find a valid tag organization. To keep energy consumption to a minimum, the goal of this thesis work, the sequential access mode was used, which first searches the tag array and then searches the data array. It was determined through simulations that the parallel tag-data lookup is faster but consumes additional energy.

Table 4.2: Various L3 banking options with associated pertinent values

Banks	Access Time (ns)	Dynamic Read (nJ)	Leakage/Bank (mW)	Total Leakage (mW)
1	4.2884	1.29704	1300.01	1300.01
2	4.24018	1.36753	883.007	1766.014
4	3.81596	1.34924	446.064	1784.256
8	3.5818	1.46238	307.196	2457.568
16	3.65983	1.43017	192.145	3074.32
32	3.0623	1.19136	84.3839	2700.2848
64	3.09218	0.966831	74.7693	4785.2352
128	2.27054	0.96097	57.8941	7410.4448

4.2.1 Energy & Power

With the banking determined, the next step was to gather the energy and power numbers for each possible configuration. Because CACTI only allows for numbers expressible as powers of two for the number of ways, graphical interpolation was performed to find the best approximate values for the configurations that cannot be found through CACTI, values corresponding to the B partitions. To make sure that the best function was found, the coefficient of determination, R^2 , value that was as close to 1 as possible was the function used. Linear interpolation was used to find the dynamic energy in the normal L2, as seen in 4.3, while the leakage power in the normal L2 was found using a power function, as seen in 4.4. The dynamic energy interpolation was almost exact with the R^2 value at 0.9997, and the leakage power interpolation was not quite as good, but still excellent, with the R^2 value of 0.9969. Note that only the equations that were ultimately used are included within the graphs, as each option was tested to find the optimal fit. Table 4.3 shows the values used for the simulations in the phase-adaptive cases. Because the A partition is always set to a power of 2, 3-way and 5-way values are left out. Using the same number

Figure 4.3: Interpolation and line fitting of dynamic energy for L2

Table 4.3: L2 energy and power numbers

Ways	Dynamic Read (nJ)	Leakage (mW)
1	0.0487877	17.85836
2	0.141499	56.9864
4	0.209668	133.9328
6	0.2735	260.6771385
7	0.3064	327.1112511
8	0.338417	403.5232

Table 4.4: L2 drowsy energy and power numbers

Ways	Dynamic Read (nJ)	Leakage (mW)
4	0.0653267	74.92824
6	0.0851	146.0103765
7	0.0952	183.3090019

of banks per way, the drowsy cache parameters were also found, with CACTI having been altered to use 0.7V as V_{CC} instead of the normal 0.9V for 32nm itrshp devices. This was the only change made to the source code of CACTI. As seen in Figures 4.5 and 4.6, the R^2 values are the same for both the dynamic energy and leakage power as the full voltage cases, at 0.9997 and 0.9969 respectively. Note that this thesis work assumes that the energy required to access a block of data is an average over the entire cache or way depending on the configuration.

Figure 4.4: Interpolation and line fitting of leakage power for L2

4.2.2 Latency

Also found from CACTI were the numbers for the latency to access each of the partitions, shown in Table 4.5. The number of cycles associated with an access to each partition is under the assumption of a 3 GHz processor. The latency between the partitions is not all that different and because the tag array is initially searched before accessing the data array, an access to the B partition is just the number of cycles listed. Again, it takes one additional cycle to access the drowsy partition, to raise the voltage and safely read the data.

Table 4.5: L2 cache configurations with associated latency

Name	A partition	B partition
C0	1-way 6-cycle	7-way 8-cycle
C1	2-way 7-cycle	6-way 8-cycle
C2	4-way 8-cycle	4-way 8-cycle
C3	8-way 8-cycle	0-way 0-cycle

Figure 4.5: Interpolation and line fitting of dynamic energy for L2

Figure 4.6: Interpolation and line fitting of leakage power for L2

4.3 Multi2Sim

Multi2Sim [33] is an application only, heterogeneous simulation framework that was modified to implement the phase-adaptive cache design. The simulator is highly configurable, with the ability to implement an out-of-order, superscalar processor that can have either SMT, CMP or both. The simulator also provides a highly configurable memory hierarchy, one that allows for the simulator to model a standard split L1, private L2, to more exotic designs that are only limited by having the correct connections between levels.

A modified version of Multi2Sim 3.2 was used for this thesis work, where the cache library initially had to be updated to fix a bug within the cache coherence code. Care was taken to follow the coding style of the authors. Outside of the cache library, the only change to the CPU architecture that was made was code to implement the evaluation of the counters approximately every 15k instructions [20]. The largest variance observed was 5 instructions past 15k, due to the maximum number of instructions that could be committed every cycle.

The general overview of the cache library can be seen in Figure 4.7, with the orange blocks indicating the structures that were modified for this thesis and the arrows showing how each structure is connected. Due to the nature of the code, the configuration file was modified to read in a new input parameter, *PhaseAdaptive*, which accepted either 0 or 1. This modification not only provides the ability to select which level or levels of the cache to be put into the phase-adaptive behavior, but also allowed for debugging to ensure that any modifications made would not affect the original operation of the cache library. This input parameter had a corresponding integer variable added to be used as a flag in the code, allowing for the integration of the phase-adaptive code without unnecessary modification to the existing, working code.

Within the cache structure, various counters were added to keep track of the MRU accesses, hits, misses and swaps. All of these counters are used to calculate the energy used at each interval and the next best configuration. The counters are cleared at the end of each calculation period to ensure the correct information is kept. Variables were also added to keep a running total of the dynamic energy and leakage energy during the running of the simulations. Due to the difference between the usage of leakage energy in this thesis and leakage power as given by CACTI, the number of cycles are recorded for each instruction

Figure 4.7: Multi2Sim Cache Design

interval to convert the power to energy by Equation 4.1:

$$Leakage_{Energy} = \frac{Leakage_{Power} * CycleCount}{3GHz} \quad (4.1)$$

The `cache.blk` structure had a variable added to keep track of the current MRU state, aptly called `mru_state`, along with a `partition` variable to keep track of the location of the partition and a `swapped` variable to signal to the simulator to add the latency cost associated with the swapping of blocks when needed.

Whenever the simulator looks to access a block, the `mru_counter` is updated, then if it is located within the B partition it is swapped into the A partition and the block's MRU state is set to 0. Following that, the MRU states are all updated to reflect the recent access.

On eviction and replacement of a block, if the status of a cache block within a set is *invalid*, the block is first swapped into the `MRU0` state as the new block will be the most recently used block. If none of the cache blocks within the set are invalid, the cache block with the highest MRU state is returned.

The simulator uses the *MOESI* cache coherence protocol [32] to keep data consistent between multiple caches. Initially, the coherence code was updated to reflect a bug fix in

a more recent release of the source, where the LRU states were updated too often in the incorrect states. After this, the cache coherence protocol had a slight modification to add the increased latency on the swapping of blocks, which used another added variable, lat_B , to represent the different possible B partition latency based on the values found in Table 3.1.

The control algorithm was added to the cachesystem structure, which records all of the statistics for the just finished phase, determines the next cache configuration, sets the new latency for the B partition, and records the energy usage of the past phase. After the statistical information was recorded, the configuration costs were determined, using Equation 3.2 and potentially Equation 3.3. The determination of the next configuration is based on simple comparison between all of the possible energy usages from the recently finished phase. After all the evaluations have finished, the counters are reset and the next phase begins.

4.4 Simulation Configuration

With the aim to design a technique that can be applied to production processors, the cache hierarchy and CPU configuration was roughly based on the Intel Nehalem architecture [21]. This work is based on the assumption that we are using an one core of a 4 core processor, and an equal slice of the L3 cache, with 2MB of the possible 8MB is used, as seen in Table 4.6 with the CPU specific configuration is shown in Table 4.7. IL1 was made perfect to ensure correct and consistent results, and L2 was the cache was implemented to have a phase-adaptive behavior, with L3 being held to a consistent size.

Table 4.6: Memory Hierarchy Configuration

Parameter	Value
General	LRU, 64B line, 2 Read ports, 2 Write ports
L1 Cache	Split, 32KB, 4 way, 3 cycle
L2 Cache	256KB, 8 way, 8 cycle
L3 Cache	2MB, 16 way, 20 cycle

Table 4.7: Processor Configuration

Parameter	Value
Fetch queue	64 bytes
Decode width	4 instructions
Branch Predictor	Combined, 1024 entry BTB, 1024 entry Bimodal, Two level 8K history table
Return Address Stack	16 entry
Issue & Commit width	4 instructions
Reorder Buffer size	128

4.5 Benchmarks

The SPEC2006 benchmark suite [10] were used for the single core simulations. A representative set was chosen that included both CINT2006 and CFP2006 with varying cache access patterns, as seen in Table 4.8. For all of the benchmarks the reference set of inputs were used. All of the simulations ran for 2 billion instructions with 500,000 instructions fast-forwarded to warm up each cache. All of the benchmark executables were precompiled for the x86 architecture, provided by the authors of Multi2Sim, with the only requirement being a valid license to the input sets used. Each test was run using a command similar to:

```
./m2s --cpu-config cpu.ini --cpu-cache-config cache.ini
--cpu-sim detailed --max-inst 2000000000
--report-cpu-cache reports/astar-report.out
```

—ctx—config astar.ctx

Table 4.8: Names and descriptions of SPEC2006 tests used

Test	Description
<i>SPECINT</i>	
400.perlbench	Perl V5.8.7 running SpamAssassin
401.bzip	Modified bzip2 to run in memory opposed to I/O
403.gcc	Generates code based on GCC version 3.2
456.hmmer	Protein sequence analysis using Markov models
458.sjeng	An artificial intelligence program that plays chess
462.libquantum	Simulates a quantum computer running Shor's factorization algorithm
464.h264ref	Reference implementation of H.264 video encoder
471.omnetpp	Models a large ethernet network using OMNet++
473.astar	Path finding algorithm for 2-D maps
<i>SPECFP</i>	
433.milc	Quantum Chromodynamic simulator
434.zeusmp	Fluid dynamic simulation of astrophysical phenomena
436.cactusADM	Einstein evolution equation solver using staggered leapfrog method
444.namd	Simulator of a large biomolecular system
450.soplex	Linear program simulator using a simplex algorithm and sparse linear algebra
454.calculix	Finite element code for linear and nonlinear 3D structures
465.tonto	Open source quantum chemistry package
482.sphinx3	Speech recognition program from Carnegie Mellon University

Summary

This chapter gave the background on all of the methods and values found to perform this thesis work. Starting with circuit level simulations using SPICE, moving to cache level

simulations using CACTI and finally using the values found to put into the architectural simulator Multi2Sim. Listed at the end are the benchmarks from SPEC2006 that were chosen for the simulations. In the next chapter, the results of this work are presented.

Chapter 5

Results

As mentioned previously, the drowsy phase-adaptive cache has a few interesting aspects, all of which will be explained in this section:

- Reduce dynamic energy consumption by partitioning the cache, reducing the use of the surrounding logic required to access and evaluate the cache lines.
- Reduce leakage power consumption by putting the B partition into a *drowsy* state, by lowering the the operating voltage of the cache lines from 0.9V to 0.7V.
- A performance penalty is incurred due to the sequential access to the partitions and the additional latency required to awaken the drowsy cache lines.
- Performance is not hurt terribly due to the extremely high temporal locality associated with cache, as up to 98% of accesses are to the two most recently used cache lines.

5.1 Experiments

Five different configurations were ran for each of the 18 SPEC2006 benchmarks, as listed in Table5.1. The configurations names: *Baseline*, *Phase*, *Drowsy*, *PhaseED*, and *DrowsyED*,

will be used in all following graphs.

Table 5.1: Simulation configuration names and descriptions

Configuration name	Description
Baseline	The cache configuration was held unpartitioned
Phase	The cache configuration is determined on the MRU statistics, energy and leakage.
Drowsy	The cache configuration is determined on the MRU statistics, energy and leakage with the B partition being put into the drowsy state.
PhaseED	The cache configuration is determined on the MRU statistics, and the energy-delay product.
DrowsyED	The cache configuration is determined on the MRU statistics, and the energy-delay product with the B partition being put into the drowsy state.

5.2 Performance

Of the two typical aspects of computer architecture design, performance and power, performance has traditionally been in the forefront. When talking about general purpose processors, if the design fails to improve on the performance of the previous generation, then the odds of the processor succeeding is minimal. Performance is an easily quantifiable value, one that, generally, means if one processor has a higher rating over another, it is an easy decision to pick the better processor. This section is broken down into two parts, the first shows the percentage of time each simulation spent in the various configurations. Second, the performance metrics of the simulations, Speedup and IPC, are shown.

5.2.1 Configuration Time

The amount of time spent within the different configurations, as shown in Table 3.1, determines the power/performance tradeoff in the system. Figures 5.1, 5.2, 5.3, and 5.4 show the percentage of time spent in each of the four possible configurations. Based on the different simulations, as explained in Table 5.1, the optimal configurations are picked. The only test to show any sign of phase behavior, where different configurations were picked during testing, based on the configuration parameters, was the *PhaseED* case. As shown in Figure 5.2, the majority of the tests spend time in either C0 or C3, with the overwhelming majority of the time spent in the full configuration C3 state. The average amount of time spent in the two configurations is 15% for C0 and 85% for C3. *omnetpp* spent close to half of the time between the two configurations, 44% and 56%, whereas *libquantum* and *milc* spent the entire time in C3.

Figure 5.1: Configuration time for Phase

Diametrically opposed to the results shown above, *Drowsy* and *DrowsyED*, shown in Figures 5.3 and 5.4, the percentage of time spent in the smallest, C0 state is practically 100%. It is important to notice the scale of the y-axis, as three and four significant figures are required. This shows how profound the energy cost plays into the decision of picking

Figure 5.2: Configuration time for PhaseED

Figure 5.3: Configuration time for Drowsy

the best configuration for the drowsy tests, as the addition of the *delay* cost, Equation 3.3, to create the *energy-delay* equation, Equation 3.4, still overwhelms the decision. The amount of time spent in the C3 configuration can be contributed to it being the starting configuration, and after the initial phase, the cost function determines that the optimal configuration is C0.

Based on the choices in configurations, the average set associativity of the simulations can be seen in Figure 5.5. Unsurprisingly, *Phase* is exactly 8-way set associative. Likewise, both *Drowsy* and *DrowsyED* are ever so slightly above 1-way set associative, or direct

Figure 5.4: Configuration time for DrowsyED

mapped. *PhaseED*, due to some of the tests configured for the smallest configuration, C0, at times, is just under 7-way set associative for the average.

Figure 5.5: Average associativity of the simulations

5.2.2 Accesses, Hits, & Misses

Based on the differences in configuration time for the simulations, it is important to point out that the average misses compared to the *Baseline* are -0.44%, -2.04%, -1.55% and -2.34% for *Phase*, *Drowsy*, *PhaseED* and *DrowsyED* respectively. So while each simulation had an increase in the number of misses over the same number of instructions, the

difference is not outrageous, and if anything it would be expected based on the chosen configurations and the swapping of blocks.

Figure 5.6: Average percent of Hits and Misses in the A partition

Figure 5.6 shows both the percent of the number of hits in the A partition, and the percent of the number of misses for each of the simulations. Because the *Phase* simulation always picks configuration C3, as shown in Figure 5.1, the number of hits in the A partition is the same as the *Baseline*. Similarly, *PhaseED* shows a slight reduction in the average, but overall the hits are less than 5% off from the baseline, due to a few of the tests switching configurations between C0 and C3, as seen in Figure 5.2. Because both the *Drowsy* and *DrowsyED* simulations pick the smallest configuration, C0, the percent of hits in the A partition compared to the total number of accesses is small, but this does not factor in the swaps.

As can be seen on the second half of the plot, the average percent of the misses between all the simulations is practically the same, with the smallest being the *Baseline* at 18.6% and the largest being *DrowsyED* at 18.89%. Even with the discrepancy between the number of hits in the A partition and the size of the partition for the *Drowsy* and *DrowsyED* simulations, the performance is not affected by much, as shown in the next section. As mentioned earlier, the expectation was to have more than 90% of the hits to MRU state 0,

but this data was found using SPEC2000 benchmarks, and this thesis work was performed using SPEC2006, which have different properties than the previous set. Instead, this work found that 25% of the hits were to MRU state 0, with the remaining 75% of the hits to the other MRU states. Due to the cost functions, it was determined that the best policy was to only focus on MRU state 0 for *Drowsy* and *DrowsyED*, as opposed to utilizing MRU state 0 and MRU state 1.

Figure 5.7: Percentage of Hits in the A partition

Figure 5.8: Percentage of Hits in the B partition

Figures 5.7, 5.8, and 5.9, show the percentage of Hits in A, Hits in B, and misses for each benchmark of each of the simulations. These correspond directly the amount of

Figure 5.9: Percentage of Misses

both dynamic energy and leakage energy expended during the simulations and the performance of the simulations. The differences between the hits in the A partition for *Phase* and *PhaseED* can be seen in the hits in the B partition for *PhaseED*.

The amount of hits in the B partition, as seen in Figure 5.8 is vital to the energy savings of the *Drowsy* and *DrowsyED* simulations. Since the cache is configured to have the B partition with 7-ways, it holds most of the data, but at a lower operating voltage, allowing for the large savings in leakage energy, seen later in Section 5.3.2, and also this does not hurt performance that much, as will be seen in the next section.

Finally, it is important to reiterate that the percentage of misses for each of the simulations is almost exactly the same, with Figure 5.9 showing the misses for each benchmark, with the overall average having been shown in the second part of Figure 5.6. This shows that the design does not adversely effect the miss rate of the cache, which is vital to performance and energy consumption. The cost functions take the cost of a miss into account, specifically the energy related to an access to the next level in the memory hierarchy, L3. If the miss ratio was higher than the baseline, the energy consumption would also be higher.

5.2.3 Speedup & IPC

Arguably, the most important metric of this thesis work is the speedup of the simulations compared to the baseline. If the new designs cannot perform within an acceptable variance from the baseline, then the design itself is flawed, as there are already a litany of techniques to significantly reduce energy and power usage at the expense of performance. The results are shown in Figure 5.10, with all of the tests have been normalized against the baseline. A general observation can be made that the *Phase* and *PhaseED* tests are generally similar in Speedup, along with the *Drowsy* and *DrowsyED*. Though there is a difference visible in Figure 5.10, with the values normalized against the baseline the difference between simulations is less than 1%. Notice that the y-axis is from 0.95 to 1.01.

Figure 5.10: Speedup relative to the baseline

The *Phase* and *PhaseED* tests average out to be just as fast as the non-partitioned *Baseline*. For the *Phase* case this is because the configuration is the same as the *Baseline*, as shown in Figure 5.1. For the *PhaseED* case, even with some of the tests spending time in the C0 state, this proves to slow down the simulations by an average of 0.38%, hardly noticeable.

The *Drowsy* tests, on average, were just 1.28% slower than the *Baseline*, with *soplex*

having the worst performance at 3.89% slower than the baseline, while *namd* having the best relative performance, running just 0.25% slower than the *Baseline*. The *DrowsyED* tests performed almost exactly as the *Drowsy* tests, but ever so slightly slower, on average less than one hundredth of a percent.

Figure 5.11: IPC of all the simulations

From the figures in Section 5.2.1, it can be understood why the performance of *Phase* and *PhaseED* are so close to the *Baseline*, while *Drowsy* and *DrowsyED* are not as close, but still well within acceptable limits. *Phase* only uses the largest A partition, configuration C3 (8/0), which means it has the same latency as the *Baseline*. It is not exactly the same due to the swapping of cache blocks into the MRU_0 way, which at times can lead to increased performance, as in *astar* and *hmmer*, or decreases in performance, as in *perlbench*. *PhaseED* has a handful of benchmarks that switch between the smallest A partition, C0 (1/7), and the largest A partition, C3 (8/0), completely disregarding both C1 (2/6) and C2 (4/4), but still perform just slightly slower than the *Baseline*. This shows that the energy-delay product for this simulation has some importance, but it still does not overcome the increase in latency to save power.

Meanwhile, both *Drowsy* and *DrowsyED* both pick the smallest A partition, configuration C0 (1/7), showing that the decision of saving the most energy is the overwhelming factor. While there is some difference in dynamic energy between the normal operating voltage and the drowsy voltage, the difference in leakage is such that it drives the decision to pick the smallest A partition. This reinforces the fact that temporal locality is an important aspect of cache performance.

For comparison and completeness, the IPC graph can be seen in Figure 5.11. Most of the differences between IPCs are almost imperceptible, further illustrating the effectiveness of the design. As shown, it is hard to distinguish the amount of difference in IPC between most of the tests, further reflecting how little impact this design has on performance.

5.3 Energy & Power

The goal of this thesis work was to reduce energy/power consumption within the cache hierarchy with minimal performance loss. As shown in Section 5.2, the average of all the simulations was just 0.67% slower than the *Baseline*, with the slowest simulation benchmark running 3.89% slower, with a few simulation benchmarks running 0.52% faster than the *Baseline*. The following section will discuss the energy and power aspects of the design.

5.3.1 Dynamic Energy

Dynamic power is defined by the following equation, where α is the switching activity factor:

$$P = \alpha CV^2 f \quad (5.1)$$

The phase-adaptive cache is primarily used to lower dynamic energy usage within the cache, due to only needing to access a subset of the cache ways, reducing the amount

of logic surrounding the cache that needs to be utilized. The dynamic energy usage of all the simulations can be seen in Figure 5.12 while the dynamic energy savings compared to the *Baseline* can be seen in Figure 5.13.

Figure 5.12: Dynamic energy usage of all the simulations

The total dynamic energy usage of the L2 cache over two billion instructions is generally very small, with the highest being just over 0.05 joules for *soplex*, and the smallest energy usage being 0.2 mJ for *sphinx3*. Looking at Figure 5.13, the *Drowsy* and *DrowsyED* simulations save a good portion of dynamic energy, with an average of 17.9% savings over the *Baseline*. Again, this is primarily due to the fact that when the simulations run almost exclusively in the C0 (1/7) state, much of the logic required to access the remaining seven ways of the cache is not used.

There is practically no dynamic energy savings for the *Phase* simulations, as it only picks the largest A partition, C3 (8/0), which is the same configuration as the baseline. Because *PhaseED* does switch to the smallest A partition, C0 (1/7), for some periods of time, there is a slight amount of savings. Due to both *Drowsy* and *DrowsyED* always picking the smallest A partition, C0 (1/7), they are both able to save some dynamic energy, due to the avoidance of the logic required to access all of the ways, but there is some additional energy usage due to the swapping required. It should be pointed out the *milc*

Figure 5.13: Dynamic energy savings of the simulations

shows no savings, whereas *tonto* shows close to 40% savings. This difference is due to the discrepancy between the number of hits in each partition. While the number of hits in the B partition is 6 times the number of hits in the A partition for *milc*, *tonto* has just 1.9 times more hits in the B partition than hits in the A partition, and relative to the *Baseline*, the number of hits in the A partition for *tonto* is higher than the number of hits in the A partition for *milc*.

5.3.2 Leakage Energy

Because leakage current is the dominant portion of energy usage within a processor, as mentioned in Section 2.1, and seen in Figure 5.14, the amount of leakage that can be possibly saved is important. With the addition of the drowsy circuitry, massive amounts of leakage current can be reduced as opposed to traditional cache designs. The smallest overall energy usage is still 50% more than the largest amount of dynamic energy usage. It should be noted that all of the graphs and values associated with leakage power have been converted to energy, under the assumption of the processor running at 3 GHz.

With an average savings of over 44%, seen in Figure 5.15, between both *Drowsy* and

Figure 5.14: Leakage energy usage of the simulations

DrowsyED, the design performs extremely well, and referring back to Figure 5.10, with minimal loss in performance, the design works extremely well. While not exactly the same in leakage savings, the differences can be contributed to slight differences in the total number of cycles that each simulation ran, corresponding to the speedup difference between the simulations.

Figure 5.15: Leakage energy savings of the simulations

Again, based on the figures in Section 5.2.1, specifically Figures 5.3 and 5.4, it is very easy to see why the *Drowsy* and *DrowsyED* simulations save a large amount of leakage,

both pick the smallest A partition, C0 (1/7). This shows that the choice between either energy or energy-delay is not important, as only energy is considered for the *Drowsy* and *DrowsyED* simulations, as opposed to the *PhaseED*, which switched between C0 (1/7) and C3 (8/0).

5.3.3 Total Energy

The total energy consumption in the L2 cache is shown in Figure 5.16. The similarity between this Figure and Figure 5.15 is uncanny in that leakage dominates the total energy usage, and consequently the savings. This is unsurprising due to the high amount of leakage within the system, further illustrating the point that leakage power is the more important aspect of power consumption within a processor than dynamic energy.

Figure 5.16: Total energy savings of all the simulations

Considering the amount of saving achieved from both the *Drowsy* and *DrowsyED* simulations and the decision by both to always chose the smallest A partition, C0 (1/7), it is clear that there is no need for any additional hardware to calculate costs, count hits and misses, or change the partitions. The only additions required is the logic to change the voltage to the drowsy voltage, and the logic to swap blocks between the set partitions. All

of the savings found in the *PhaseED* simulations can be contributed to the same benchmarks that spent a significant amount of time in configuration C0 as opposed to spending most of the time in configuration C3.

5.3.4 Summary

Through various simulations utilizing industry standard benchmarks, it is determined that the optimal solution is to statically partition the cache into an active 1-way A partition and a drowsy 7-way B partition. Therefore, there is no need to dynamically adapt the number of ways of the A partition to the phase of the running application, or to implement any cost functions to that purpose. This greatly reduces the amount of additional logic required to save almost 45% leakage energy and almost 20% dynamic energy on average, while the performance is just 2% slower than the non-partitioned configuration. Instead of one counter for each way, one for misses, and control logic to modify the partitions, this technique only requires the logic to swap blocks, with the addition of the drowsy circuitry. This design also removes the need for logic required to calculate the costs, which also would save the performance penalty of having to calculate the next configuration.

Chapter 6

Conclusions

This thesis work presented a novel cache design, a drowsy phase-adaptive cache, that aimed to save both dynamic and static power within the cache hierarchy with minimal performance loss. Taking advantage of the temporal locality that is inherent in programs and the advantages of the accounting cache to place most recently used blocks in the first accessed A partition, the design accomplished all aspects of the motivation.

Through testing, it can be seen that with the inclusion of the drowsy cache structure, the ability to dynamically reconfigure the cache becomes a moot point, as practically 100% of the time the A partition is held in the smallest state, saving the most leakage power possible. This leads to the determination that there is no need for the decision logic to calculate the next best configuration as only one configuration was used. With the aim to reduce dynamic and static power, any additional logic added is counter to what the goals are, but with the simple partitioning scheme of a 1-way A partition and 7-way B partition and some swapping logic, the design accomplishes all the goals set forth. This work looked at some simple, yet useful, design choices to save power within the cache hierarchy. Achieving savings of almost 20% for dynamic energy and almost 45% for leakage energy, all the while keeping performance within 2% of the *Baseline* on average.

6.1 Future Work

Using this work as a starting point, there are several opportunities for future research. First, because this work only focused on the second of three levels of cache, testing to see how this design would work over multiple levels should be explored. Due to the increased size of level 3 cache in modern processors, there is opportunity for immense energy savings. Another would be to apply this work to a complete chip multiprocessor to see how the sharing of data across cores and caches would affect the cost function decisions. The modification of the design to include either multiple cores or multiple levels might require a more thorough search of possible cost functions. As L3 is the backing store for L2 and cache coherence protocols get involved, the flat partitioning model might not handle well. Finally, an exploration into new, post-CMOS technologies should be done. Moore's law cannot continue forever, a contradiction in itself, and the need for new technologies means a need to continue to save power.

Bibliography

- [1] Amit Agarwal and Kaushik Roy. DRG-cache: a data retention gated-ground cache for low power. In *Proceedings 2002 Design Automation Conference (IEEE Cat. No.02CH37324)*, pages 473–478. ACM, 2002.
- [2] Vishwani D. Agrawal. Multi-core parallelism for low-power design. 2006.
- [3] D.H. Albonesi. Selective Cache Ways: On-demand Cache Resource Allocation. In *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 248–259. IEEE Comput. Soc, 1999.
- [4] R. E. Aly and B. R. Nallamilli. Variable-way set associative cache design for embedded system applications. In *2003 IEEE 46th Midwest Symposium on Circuits and Systems*, pages 1435–1438, December 2003.
- [5] J.-L. Baer and W.-H. Wang. On the inclusion properties for multi-level cache hierarchies. In *[1988] The 15th Annual International Symposium on Computer Architecture. Conference Proceedings*, pages 73–80. IEEE Comput. Soc. Press, 1988.
- [6] R. Balasubramonian, D.H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 245–257, Dec. 2000.
- [7] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.
- [8] Mark T. Bohr, Robert S. Chau, Tahir Ghani, and Kaizad Mistry. The high-k solution. *IEEE Spectrum*, Oct. 2007.
- [9] Liming Chen, Xuecheng Zou, Jianming Lei, and Zhenglin Liu. Dynamically Reconfigurable Cache for Low-Power Embedded System. In *Third International Conference on Natural Computation (ICNC 2007) Vol V*, number Icnc, pages 180–184. IEEE, August 2007.

- [10] Standard Performance Evaluation Corporation. SPEC CPU2006 Benchmark Suite, v1.1. Available at <http://www.spec.org/cpu2006>, Jun. 2008.
- [11] S. Dropsho, A. Buyuktosunoglu, R. Balasubramonian, D.H. Albonesi, S. Dwarkadas, G. Semeraro, G. Magklis, and M.L. Scottt. Integrating adaptive on-chip storage structures for reduced dynamic power. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 141–152, Sept. 2002.
- [12] S. Dropsho, G. Semeraro, D. H. Albonesi, G. Magklis, and M. L. Scott. Dynamically Trading Frequency for Complexity in a GALS Microprocessor. In *37th International Symposium on Microarchitecture*, pages 157–168, 2004.
- [13] K. Flautner, N.S. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy Caches: Simple Techniques for Reducing Leakage Power. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 148–157, May. 2002.
- [14] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*, 4th ed. Morgan Kaufmann Publishers, San Francisco, CA, 2007.
- [15] Seongmoo Heo and Krste AsanoviC. Power-optimal pipelining in deep submicron technology. In *Proceedings of the 2004 international symposium on Low power electronics and design - ISLPED '04*, page 218, New York, New York, USA, 2004. ACM Press.
- [16] Bruce Jacob, Spencer Ng, and David Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2008.
- [17] Y.E. Jiongyao. An Adaptive Width Data Cache for low power design. *2009 International SoC Design Conference (ISOC)*, pages 488–491, 2009.
- [18] Nam Sung Kim, Krisztian Flautner, David Blaauw, and Trevor Mudge. Circuit and Microarchitectural Techniques for Reducing Cache Leakage Power. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(2):167–184, February 2004.
- [19] S. Lopez, S. Dropsho, D.H. Albonesi, and O. Garnica. Dynamic Capacity-Speed Tradeoffs in SMT Processor Caches. In *HiPEAC'07 Proceedings of the 2nd International Conference on High Performance Embedded Architectures and Compilers*, pages 136–150, Jan. 2007.

- [20] S. Lopez, O. Garnica, D.H. Albonesi, S. Dropsho, J. Lanchares, and J.I. Hidalgo. Adaptive Cache Memories for SMT Processors. In *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, pages 331–338, Sept. 2010.
- [21] Daniel Molka, Daniel Hackenberg, Robert Schone, and Matthias S. Muller. Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System. *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 261–270, September 2009.
- [22] N. Muralimanohar, R. Balasubramonian, and N.P. Jouppi. CACTI: An Integrated Cache and Memory Access time, Cycle Time, Leakage and Dynamic Power Model. <http://www.hpl.hp.com/research/cacti/>.
- [23] Paolo Nenzi, Holger Vogt, and Dietmar Warning. ngspice, Jan. 2012.
- [24] D.A. Patterson and J.L. Hennessy. *Computer Organization and Design, 3rd ed.* Morgan Kaufmann Publishers, Burlington, MA, 2007.
- [25] S. Petit, J. Sahuquillo, J.M. Such, and D. Kaeli. Exploiting Temporal Locality in Drowsy Cache Policies. In *Proceedings of the 2nd conference on Computing Frontiers*, pages 371–377, May. 2005.
- [26] Michael Powell, Se-Hyun Yang, Babak Falsafi, Kaushik Roy, and T. N. Vijaykumar. Gated-V dd. In *Proceedings of the 2000 international symposium on Low power electronics and design - ISLPED '00*, pages 90–95, New York, New York, USA, 2000. ACM Press.
- [27] J. Rabaey, A. Chandrakasan, and B. Nikolic. *Digital Integrated Circuits: A Design Perspective, 2nd ed.* Prentice-Hall, Englewood Cliffs, NJ, 2002.
- [28] P. Ranganathan, S. Adve, and N. P. Jouppi. Reconfigurable caches and their application to media processing. In *ACM SIGARCH Computer Architecture News*, pages 214–224, 2000.
- [29] Gary Silcott. AMD Cache Sizes. Personal E-Mail, Dec 2010.
- [30] S. Srikantaiah, E. Kultursay, T. Zhang, M. Kandemir, J. Irwin, and Y. Xie. MorphCache: A Reconfigurable Adaptive Multi-level Cache hierarchy. In *2011 IEEE*

- 17th International Symposium on High Performance Computer Architecture (HPCA)*, pages 231–242, 2011.
- [31] Karthik T. Sundararajan, Timothy M. Jones, and Nigel Topham. Smart cache: A self adaptive cache architecture for energy efficiency. *2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pages 41–50, July 2011.
 - [32] P. Sweazey and A. J. Smith. A class of compatible cache consistency protocols and their support by the IEEE futurebus. In *ACM SIGARCH Computer Architecture News*, volume 14, pages 414–423. IEEE Computer Society Press, June 1986.
 - [33] R. Ubal, J. Sahuquillo, S. Petit, and P. López. Multi2Sim: A Simulation Framework to Evaluate Multicore-Multithreaded Processors. In *Proc. of the 19th Int’l Symposium on Computer Architecture and High Performance Computing*, Oct. 2007.
 - [34] Neil H. E. Weste and David Harris. *CMOS VLSI Design: A Circuits and Systems Perspective, 3rd ed.* Pearson Education, 2005.
 - [35] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, Jun. 1995.
 - [36] Y. Ye, S. Borkar, and V. De. A new technique for standby leakage reduction in high-performance circuits. In *1998 Symposium on VLSI Circuits. Digest of Technical Papers (Cat. No.98CH36215)*, pages 40–41. IEEE, 1998.
 - [37] C. Zhang, F. Vahid, and J. Yang. A way-halting cache for low-energy high-performance systems. In *ACM Transactions on*, page 126, 2005.

Appendix A

CACTI configuration for L2 cache

```
# Cache size
// -size (bytes) 4096
// -size (bytes) 8192
// -size (bytes) 16384
// -size (bytes) 32768
// -size (bytes) 65536
// -size (bytes) 131072
-size (bytes) 262144

# Line size
-block size (bytes) 64

# To model Fully Associative cache, set associativity to zero
// -associativity 0
// -associativity 1
// -associativity 2
// -associativity 4
-associativity 8

-read-write port 0
-exclusive read port 2
-exclusive write port 2
-single ended read ports 0

# Multiple banks connected using a bus
// -UCA bank count 1
```

```

//UCA bank count 2
//UCA bank count 4
//UCA bank count 8
UCA bank count 16
//UCA bank count 32
-technology (u) 0.032

# following three parameters are meaningful only for main memories
-page size (bits) 8192
-burst length 8
-internal prefetch width 8

# following parameter can have one of five values
# (itrs-hp, itrs-lstp, itrs-lop, lp-dram, comm-dram)
//Data array cell type - "comm-dram"
-Data array cell type - "itrs-hp"
//Data array cell type - "itrs-lstp"
//Data array cell type - "itrs-lop"

# following parameter can have one of three values
# (itrs-hp, itrs-lstp, itrs-lop)
-Data array peripheral type - "itrs-hp"
//Data array peripheral type - "itrs-lstp"
//Data array peripheral type - "itrs-lop"

# following parameter can have one of five values
# (itrs-hp, itrs-lstp, itrs-lop, lp-dram, comm-dram)
-Tag array cell type - "itrs-hp"
//Tag array cell type - "itrs-lstp"
//Tag array cell type - "itrs-lop"

# following parameter can have one of three values
# (itrs-hp, itrs-lstp, itrs-lop)
-Tag array peripheral type - "itrs-hp"
//Tag array peripheral type - "itrs-lstp"
//Tag array peripheral type - "itrs-lop"

# Bus width include data bits and address bits required by the decoder
-output/input bus width 256

```

```

// 300–400 in steps of 10
-operating temperature (K) 350

# Type of memory – cache(with a tag array) or ram(scratch ram similar to a register file)
# or main memory(no tag array and every access will happen at a page granularity
# Ref: CACTI 5.3 report)
-cache type "cache"
//–cache type "ram"
//–cache type "main memory"

# to model special structure like branch target buffers , directory , etc .
# change the tag size parameter
# if you want cacti to calculate the tagbits , set the tag size to "default"
//–tag size (b) "default"
-tag size (b) 17

# fast – data and tag access happen in parallel
# sequential – data array is accessed after accessing the tag array
# normal – data array lookup and tag access happen in parallel
#           final data block is broadcasted in data array h-tree
#           after getting the signal from the tag array
//–access mode (normal, sequential, fast) – "fast"
//–access mode (normal, sequential, fast) – "normal"
-access mode (normal, sequential, fast) – "sequential"

# DESIGN OBJECTIVE for UCA (or banks in NUCA)
-design objective (weight delay,dynamic power, leakage power, cycle time, area) 0:0:0:0:0

# Percentage deviation from the minimum value
# Ex: A deviation value of 10:1000:1000:1000:1000 will try to find an organization
# that compromises at most 10% delay.
# NOTE: Try reasonable values for % deviation. Inconsistent deviation
# percentage values will not produce any valid organizations. For example,
# 0:0:100:100:100 will try to identify an organization that has both
# least delay and dynamic power. Since such an organization is not possible , CACTI will
# throw an error. Refer CACTI–6 Technical report for more details
//–deviate (delay,dynamic power,leakage power,cycle time,area) 60:10000:10000:10000:100000

```



```

-deviate (delay,dynamic power,leakage power,cycle time,area) 1000:1000:1000:1000:1000

# Objective for NUCA
-NUCAdesign objective (weight delay,dynamic power,leakage power,cycle time,area) 0:0:0:0:0
-NUCAdeviate (delay,dynamic power,leakage power,cycle time,area) 10:1000:1000:1000:1000

# Set optimize tag to ED or ED^2 to obtain a cache configuration optimized for
# energy-delay or energy-delay sq. product
# Note: Optimize tag will disable weight or deviate values mentioned above
# Set it to NONE to let weight and deviate values determine the
# appropriate cache configuration
//--Optimize ED or ED^2 (ED, ED^2, NONE): "ED"
//--Optimize ED or ED^2 (ED, ED^2, NONE): "ED^2"
--Optimize ED or ED^2 (ED, ED^2, NONE): "NONE"

--Cache model (NUCA, UCA) -- "UCA"
//--Cache model (NUCA, UCA) -- "NUCA"

# In order for CACTI to find the optimal NUCA bank value the following
# variable should be assigned 0.
--NUCA bank count 0

# NOTE: for nuca network frequency is set to a default value of
# 5GHz in time.c. CACTI automatically
# calculates the maximum possible frequency and downgrades this value if necessary

# By default CACTI considers both full-swing and low-swing
# wires to find an optimal configuration. However, it is possible to
# restrict the search space by changing the signalling from "default" to
# "fullswing" or "lowswing" type.
--Wire signalling (fullswing, lowswing, default) -- "Global_10"
//--Wire signalling (fullswing, lowswing, default) -- "default"
//--Wire signalling (fullswing, lowswing, default) -- "lowswing"

--Wire inside mat -- "global"
//--Wire inside mat -- "semi-global"
--Wire outside mat -- "global"

--Interconnect projection -- "conservative"

```

```

//--Interconnect projection -- "aggressive"

# Contention in network (which is a function of core count and cache level) is one of
# the critical factor used for deciding the optimal bank count value
# core count can be 4, 8, or 16
//--Core count 4
--Core count 1
//--Core count 16
--Cache level (L2/L3) -- "L2"
--Add ECC -- "true"

//--Print level (DETAILED, CONCISE) -- "CONCISE"
--Print level (DETAILED, CONCISE) -- "DETAILED"

# for debugging
--Print input parameters -- "true"
//--Print input parameters -- "false"
# force CACTI to model the cache with the
# following Ndbl, Ndw1, Nspd, Ndsam,
# and Ndcn values
//--Force cache config -- "true"
--Force cache config -- "false"
--Ndw1 64
--Ndbl 64
--Nspd 64
--Ndcn 1
--Ndsam1 4
--Ndsam2 1

```

Appendix B

SPICE file for SRAM Cell

SRAM cell with peripheral elements with BSIM4 mosfet models

```
.include 32nm_HP.pm

.param lambda=16n
.param h_power=0.9V
.param l_power=0.7V

* Voltage Source Definition
vdd dd 0 pwl(0 h_power 7ns h_power 8ns l_power)
vpre pre 0 pulse(0V h_power 0 10ps 10ps 0.5ns 0s)
vw1 w1 0 pulse(0V h_power 3ns 10ps 10ps 2ns 0s)
vdata data 0 dc h_power
vwrite write 0 dc h_power

*
x2 pre 3 dd inv
x3 data 4 dd inv
cbl bit 0 100fF
cnbl bitb 0 100fF

* Precharge transistors
mp3 bit 3 dd dd pmos l={2*lambda} w={64*lambda}
mp4 bitb 3 dd dd pmos l={2*lambda} w={64*lambda}

* Write transistors
mn5 bit write 5 0 nmos l={2*lambda} w={64*lambda}
```

```

mn7 5      4      0 0 nmos l={2*lambda} w={64*lambda}
mn6 bitb write 6 0 nmos l={2*lambda} w={64*lambda}
mn8 6      data  0 0 nmos l={2*lambda} w={64*lambda}

* Inverter
.subckt inv in out vdd
mp out in vdd vdd pmos l={3*lambda} w={3*lambda}
mn out in 0 0 nmos l={2*lambda} w={8*lambda}
.ends

* Cross-coupled Inverters
mp1 1 2 dd dd pmos l={3*lambda} w={3*lambda}
mn1 1 2 0 0 nmos l={2*lambda} w={8*lambda}
mp2 2 1 dd dd pmos l={3*lambda} w={3*lambda}
mn3 2 1 0 0 nmos l={2*lambda} w={8*lambda}

* Pass transistors
mn2 1 w1 bit 0 nmos l={2*lambda} w={4*lambda}
mn4 2 w1 bitb 0 nmos l={2*lambda} w={4*lambda}

* Initial conditions
.ic v(bit)=0V
.ic v(bitb)=0V
.ic v(1)=0V
.ic v(2)=h_power

* Analysis and Plot
.tran 10ps 30ns
.control
run
plot v(1) v(2) v(dd) v(w1)
.endc
.end

```