# MICROARCHITECTURAL TECHNIQUES TO REDUCE ENERGY CONSUMPTION IN THE MEMORY HIERARCHY

A Dissertation
Presented to
The Academic Faculty

By

Mrinmoy Ghosh

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
in
Electrical and Computer Engineering

School of Electrical and Computer Engineering
Georgia Institute of Technology
May 2009

# MICROARCHITECTURAL TECHNIQUES TO REDUCE ENERGY CONSUMPTION IN THE MEMORY HIERARCHY

Approved by:


Dr. Hsien-Hsin S. Lee, Advisor
*Associate Professor, School of ECE*
*Georgia Institute of Technology*

Dr. Abhijit Chatterjee
*Professor, School of ECE*
*Georgia Institute of Technology*

Dr. Sudhakar Yalamanchili
*Professor, School of ECE*
*Georgia Institute of Technology*

Dr. Santosh Pande
*Associate Professor, College of Computing*
*Georgia Institute of Technology*

Dr. Saibal Mukhopadhyay
*Assistant Professor, School of ECE*
*Georgia Institute of Technology*


Date Approved: April 2, 2009

*To my Mother and Father.*

# ACKNOWLEDGMENTS

This dissertation would not have been possible without the help of many people over the past few years. First of all, I would like to thank my advisor, Dr. Hsien-Hsin S. Lee, for his motivation, excellent technical supervision, and support. I would also like to thank my committee members, Dr. Sudhakar Yalamanchili, Dr. Saibal Mukhopadhyay, Dr. Abhijit Chatterjee, and Dr. Santosh Pande, for their time and encouragement.

I would like to express gratitude for the valuable technical contributions and advice from the past and present student members of MARS: Dr. Weidong Shi, Dr. Taeweon Suh, Dr. Chinnakrishnan Ballapuram, Dr. Joshua Fryman, Fayez Mohamood, Dong Hyuk Woo, Dean Lewis, Richard Yoo, Ahmad Sharif, Eric Fontaine, Vikas Vasisht and Nak Hee Seong.

I am grateful to Emre Ozer, Stuart Biles and Simon Ford of ARM Inc. for providing assistance and guidance during the summer internship in 2005. A lot of credit goes to all my friends, Atri Dutta, Biswajit Mitra, Jacob Minz, Bevin G. Perumana, Prabir Saha, Saikat Sarkar, Payel Paul, Padmanava Sen, Arup Polley, Arindam Basu, Tapobrata Bandyopadhyay, Bhaskar Saha, Rahul Bhatia, Rishi Ranjan, Subrata Govil, Rajarshi and Rachana Mukhopadhyay, Nand and Neha Jha, Koushik and Priya Kundu, and many more for their invaluable assistance during the times of need and making my stay at Atlanta so memorable.

Finally, no person can be successful without the love and support of his family. My deepest gratitude is reserved for my parents, Shri Manik Chandra Ghosh and Smt. Bani Ghosh, who with their love, blessing, and sacrifices have made this possible.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# SUMMARY

This thesis states that dynamic profiling of the memory reference stream can improve energy and performance in the memory hierarchy. The research presented in this theses provides multiple instances of using lightweight hardware structures to profile the memory reference stream. The objective of this research is to develop microarchitectural techniques to reduce energy consumption at different levels of the memory hierarchy. Several simple and implementable techniques were developed as a part of this research. One of the techniques identifies and eliminates redundant refresh operations in DRAM and reduces DRAM refresh power. Another, reduces leakage energy in L2 and higher level caches for multiprocessor systems. The emphasis of this research has been to develop several techniques of obtaining energy savings in caches using a simple hardware structure called the counting Bloom filter (CBF). CBFs have been used to predict L2 cache misses and obtain energy savings by not accessing the L2 cache on a predicted miss. A simple extension of this technique allows CBFs to do way-estimation of set associative caches to reduce energy in cache lookups. Another technique using CBFs track addresses in a Virtual Cache and reduce false synonym lookups. Finally this thesis presents a technique to reduce dynamic power consumption in level one caches using significance compression. The significant energy and performance improvements demonstrated by the techniques presented in this thesis suggest that this work will be of great value for designing memory hierarchies of future computing platforms.

# CHAPTER 1

# INTRODUCTION

The increasing complexity and shrinking feature size of modern microprocessors has caused energy consumption to become a critical design constraint [87]. The demands of the working-set size from increasingly complex applications has led to ever-larger on-chip caches with a slew of read/write ports making it a major consumer of on-chip power. Apart from caches, other components of the memory hierarchy like DRAM also contribute significantly to the power consumption of the overall system . In this chapter, we start with discussing about energy consumption and redundancies in different parts of the memory hierarchy. Then, we explain how this research aims at addressing and eliminating the redundancies to reduce energy at different levels of the memory hierarchy.

## 1.1 Motivation

In this section, we consider energy consumption problems at different levels of the memory hierarchy. We start with the level furthest from the processor, that is DRAMs, and continue to move towards the processor and discuss problems in Level 2 or higher level SRAM caches and finally talk about a couple of energy saving opportunities in the Level 1 cache.

First, we explain the energy consumption problems for DRAMs. DRAMs are used as the bulk of the main memory in computing systems for its high density, high capacity and low cost. Due to the dynamic, leaky nature of a DRAM cell, periodic refresh operations are required for retaining the data. Such regular refreshes account for a large energy consumption in DRAMs even in the *standby* mode [88]. When a DRAM is not being accessed, all the energy consumed in the DRAM is because of the refresh operations. During each refresh operation, the data of every DRAM bit cell is read out and then written back. Since DRAMs reads are destructive, an access to the DRAM does the same operations as a refresh for data retention purposes. Since the DRAM controller does not take into

account access patterns while refreshing DRAMs, a significant number of DRAM refresh operations are redundant. Elimination of these redundant refresh operations will lead to substantial savings in DRAM energy.

As processor designers moving toward the direction of integrating 3D die-stacked DRAM (or 3D DRAM) on a package to alleviate memory latency and bandwidth issues [14, 16, 95], the overhead of the refresh operations will increase. There are two reasons behind this increase. First, a 3D DRAM could be used either as a cache between the last level SRAM-based cache and the system memory or to replace the last level cache entirely. However, a tag array is still needed for such 3D DRAM caches for data lookup and storage. Thus, the refresh operation will become a significant overhead relatively.

Second, since the 3D DRAM is bonded directly on top of the processor using die-to-die vias, the heat dissipated from the processor will be conducted across the DRAM layers, leading to a much higher temperature operation environments for the DRAM. Annavaram *et al.* [27] showed that the operating temperature of a 64MB 3D DRAM will be 90.27°C. Furthermore, the leakage will also increase exponentially with an escalating operating temperature. According to the data-sheet of Micron DRAM [83], the refresh rate must be doubled if the operating temperature exceeds 85°C. Therefore, a 3D DRAM will require double (or more) refreshes, increasing the relative energy overhead substantially. Eliminating redundant refresh operations in 3D DRAMs will be very important for both energy and performance of such systems.

Apart from the DRAMs, the higher level caches(Level 2, Level 3) are also significant contributors to the energy consumption of a computing system. The primary reason for this is that with continuously shrinking CMOS technology, the capacity of single-chip processors has exceeded one billion transistors. To use such an immense amount of available transistors, processor architects tend to allocate more cache space and deepen the level of cache hierarchy. While these caches constitute a major portion of a processor's real estate, they are also the least active components and dominate the leakage power among all other

architectural modules. One redundancy in data content for such large cache hierarchies is the need to maintain multi level inclusion (MLI). MLI requires that if a cache line is present in a lower level cache (say L1), it must be present in all the higher level caches (L2 and beyond). Though MLI is important for efficient implementation of cache coherence, it leads to replication of data in the cache hierarchy. The elimination of this redundant replication without affecting performance can lead to significant leakage energy savings in the level 2 cache.

Another significant trend in the microprocessor industry is to shift towards scalable and simplistic multicore processors like the UltraSPARC T1 processor [110]. One side effect of moving towards simplistic cores is that severe stalls that may occur when a data access misses the last level cache and goes to DRAM memory. Such cache miss events can also be used as a trigger for several microarchitectural energy management processes in the processor. The energy management processes may include but are not limited to putting all caches in a state preserving low power drowsy mode and/or clock-gating or power-gating all or part of the processor core. Architectural techniques to efficiently utilize these energy management activities will lead to significant energy savings.

Along with modern processors having large caches, the caches have increasingly higher associativity [60]. Processors employing highly associative caches consume large amount of energy on every cache lookup. In the case of a cache hit, depending on the implementation, an N way set-associative cache does N tag comparisons, and optionally may read N data lines, and only use one of the cache data lines. For a miss, all tag comparisons and data reads are redundant. Thus most energy consumed in a set-associative cache is redundant and gives ample opportunity for saving dynamic energy.

From an energy consumption perspective, the L2 cache and other higher level caches are primary consumers of leakage power. But, a significant amount of the processor energy is the dynamic energy consumed by the level 1 caches. Virtual caches are employed in the first level memory hierarchy for both high performance and embedded processors to meet

their short latency requirement. However, they also introduce the synonym problem where the same physical cache line can be present at multiple locations in the cache due to their distinct virtual addresses from different processes, leading to potential data consistency issues. To guarantee correctness, common hardware solutions either perform serial lookups for all possible synonym locations in the L1 at the expense of additional energy. Since the occurrence of synonyms is relatively rare, the energy consumed in detecting synonyms is mostly redundant and presents significant opportunities for energy savings.

When considering the dynamic energy consumption of caches, a significant part is drawn by the bitline driver circuitry because the bitlines are densely loaded with a large number of storage cells thus increasing its effective switching capacitance. Also, it has been observed that a significant percentage of data getting stored in the caches have a large number of leading zeroes and ones. Therefore, there is a potential of significant dynamic energy savings if significance compression is performed on the data stored in the cache and the unused bitlines are gated-off.

In this section, we presented a number of facets of the energy consumption problem at all levels of the memory hierarchy. Section 1.2 gives an overview of the contributions of this thesis pertaining to each of the problems highlighted in this section. Section 1.2 constitutes a preview and framework for the rest of this thesis.

## 1.2 Thesis Statement and Contributions

This thesis states that dynamic profiling of the memory reference stream can improve energy and performance in the memory hierarchy. The research presented in this theses provides multiple instances of using lightweight hardware structures to profile the memory reference stream. This profiled information has been used to identify and eliminate redundancies in memory operation primarily with the objective of saving energy consumption and in some cases improving performance. Several simple and implementable ideas to reduce dynamic power in caches, leakage power in higher-level caches, and refresh power

of DRAM were developed as a part of this research. Emphasis was given on several techniques of getting energy savings using a simple hardware structure called the Counting Bloom filter. A modified design of the Counting Bloom filter has also been used to improve performance in multicore systems. The rest of this section gives an overview of the key aspects of this research.

**Smart Refresh: Using Decay Counters to Reduce Energy Consumption in DRAMs**

One facet of this research has been to reduce energy consumption in DRAMs [50]. DRAMs require periodic refresh for preserving data stored in them. The refresh interval of DRAMs depend on the the vendor and the design technology they use. For each refresh in a DRAM row, the stored information in each cell is read out and then written back to itself, as each DRAM bit read is self-destructive. The refresh process often incurs large power and bandwidth overhead. However, it is inevitable for maintaining data correctness.

This research involved construction of an innovative scheme to reduce the refresh overhead in DRAMs. By using a countdown counter for each memory row of a DRAM memory module, all the unnecessary periodic refresh operations were eliminated. The basic concept behind this scheme is that a memory row that has been recently read or written to by the processor (or other devices that share the same DRAM), does not need to be refreshed again by the periodic DRAM refresh operation, thereby eliminating excessive refreshes and the energy dissipated. Based on this concept, we proposed Smart Refresh, a low-cost technique in the design of the memory controller for DRAM power reduction. The simulation results show that our technique can reduce 52% of all refresh operations. This saved 34.5% of the energy consumed for refresh operations. DRAM system energy savings of up to 22% and an average of 9.5% were obtained for SPECint2000, Biobench, and Splash-2 benchmark suites simulated with a 2GB DRAM. We used our Smart Refresh policy on the upcoming 3D die stacked DRAM technology and obtained energy savings up to 21% and 7.2% on an average when the refresh rate is 64 ms.

The rest of this research concentrates on reducing power consumption of SRAM caches.

We present three different techniques to reduce dynamic and leakage power in SRAM caches in the subsequent paragraphs.

**Using Virtual-Exclusion to reduce leakage power in Caches**

Another aspect of our work was focused on architectural techniques to reduce leakage energy in the L2 caches for cache-coherent multiprocessor systems [51]. This research leverages two well-known circuit techniques, gated Vdd and drowsy cache, and proposes a low cost, easily implementable architecture scheme called Virtual-Exclusion. The Virtual-Exclusion scheme saves leakage energy by keeping the data portion of repetitive cache lines "off" in the large higher-level caches, while still manages to maintain Multi-Level Inclusion, an essential property for efficient implementation of conventional cache coherence protocols. By exploiting the existing state information in the snoop based cache coherence protocol, there is almost no extra hardware overhead associated with our scheme. The SPLASH-2 multiprocessor benchmark suite was found to execute correctly under our new Virtual-Exclusion policy. The benchmarks showed up to 72% savings of leakage energy (46% for SMP and 35% for multicore in L2 on average) over a baseline drowsy L2 cache.

**Using Bloom filters to Improve Power and Performance of Caches and Memory.**

The techniques explained above exploited different forms of redundancies in the memory hierarchy to reduce energy consumption. For DRAMs, redundant refresh operations were eliminated to reduce DRAM power. In higher level caches, redundancies were found in the Multi-Level Inclusion policy and the snooping protocol to reduce cache leakage power. The subsequent technique, in contrast, adds a new hardware structure called the "Counting Bloom filter"(CBF) to the memory hierarchy. The emphasis of the research being presented in this thesis are on several innovative uses of the CBF to reduce energy consumption in SRAM caches and sometimes improve performance.

A CBF is an efficient data structure that comprises of a signature of a large data set and indicates the absence of an element in that data set. Applying CBFs to cache lookups result in faster and more power-efficient cache queries. As part of the research on CBFs,

we present four techniques of reducing cache energy using the structure.

Firstly we use CBFs to predict cache misses by tracking addresses of cache lines brought in and evicted from the cache [52] . On every linefill, the block address is hashed to generate an index used to update a bitvector. Also the counter associated with that bit is incremented. During block eviction, the corresponding counter associated with the bit indexed by the hash of the evicted block's address is decremented. On a cache access, the CBF is accessed first, and a zero in the bitvector location obtained by hashing the block address, indicates a cache miss. This prevents access to the larger cache, thus saving power and latency. Our experiments based on SPECint2000 and embedded benchmarks showed that CBFs correctly predict 89% of all L2 Cache misses, and reduces overall energy consumption by 9%.

For the second technique, we present the Way Guard, an efficient hardware structure based on CBFs, for estimating ways in a set-associative cache. Way estimation saves significant amount of unnecessary energy dissipation by reducing lookups going into redundant ways when a set-associative cache is accessed. Our Way Estimation technique required caches to look up only an average of 25-30% of the ways and saved up to 65% of the L2 energy and up to 70% of the L1 cache energy.

Finally, we examine the energy issues due to synonyms in a Virtually Indexed Cache (V-Cache) [119] . V-Caches are used to isolate virtual to physical address translation from the cache access critical path. V-Caches may have the synonym problem, where the same physical cache line can be present at multiple locations in a cache. To maintain data consistency, V-Caches are designed to ensure only one exclusive copy exists in the cache. Towards this, many commercial processors perform serial lookups for all possible synonym locations upon every miss, leading to a large energy overhead. Using a CBF to track physical addresses of cache lines brought in and evicted from the cache, false synonym lookups in the L1 cache are aborted. Using Windows application workloads, the CBF demonstrated up to 27.6% savings in the total cache energy.

**Cache Power Reduction via Hybrid Significance Compression**

This technique tries to reduce dynamic power consumption in caches by identifying redundancies in the data-values stored in the cache [53]. The focus of this technique is to analyze and identify the characteristics of workload behavior, in particular, in the first-level instruction and data caches, for power saving opportunities. We performed data value profiling for a large number of workloads, such as, the SPECint2000 and Mediabench benchmarks and observed that the data values entering the cache consists of long sequence of leading zeros and leading ones in the significance bits, indicating redundancy in the information content.

We then conceived a new significance compression technique, as a part of this effort. The basic idea is that, instead of enabling all the bitlines, the homogeneous data are compressed to a more compact form and only the bitlines representing the compact data will be enabled during cache accesses. The technique called CoolPression, is a dynamic hybrid compression scheme that combines two significance compression methods — (1) CoolCount which involves counting the number of leading zeros or ones and keeping the count by reusing the leading significant bits, and (2) Dynamic Zero Compression (DZC), an existing technique that compresses data in one-byte granularity. CoolPression combines these two schemes and determines the lower power compression scheme on-the-fly. CoolPression reduced dynamic energy consumption by more than 35%, while improving the energy consumption of both the CoolCount and DZC scheme by 5-15%.

This thesis is organized as follows. Chapter 2 gives a detailed description of prior microarchitectural low power techniques for DRAMs and caches. Chapter 3 explains the Smart Refresh technique and its energy savings in the DRAM. Next, we discuss the leakage power reduction technique called Virtual-Exclusion in Chapter 4. Chapter 5 is composed of a description of CBFs and their role in reducing cache energy consumption and way estimation. Chapter 6 consists of the application of CBFs to reduce synonym lookups in virtual caches. This is followed by a presentation of the significance compression technique

called Coolpression in Chapter 7. Finally, Chapter 8 summarizes the contributions of this thesis, talks about potential impact of this work in future microprocessors and enumerates a few directions in which this work may be extended.

# CHAPTER 2

# RELATED WORK

As explained in Chapter 1, our research addresses different aspects of the energy consumption problem at different parts of the memory hierarchy. In this chapter we discuss prior research done on each of the problems considered by this research. Firstly, we discuss prior art in reducing refresh operations in DRAMs. Then we go on to discuss several techniques to reduce dynamic power in caches. Next, we do a survey of static and leakage power reduction techniques. This is followed by an overview of different microarchitectural techniques using Bloom filters. Finally, we discuss prior art on way prediction and estimation in set-associative caches.

## 2.1 Methods of Reducing DRAM Power

We discuss several methods of reducing DRAM energy in this section. Furthermore, we also elaborate a number of reducing redundant refresh operations.

Modern DRAM modules have several low power states for saving energy. For example, Micron DRAMs [7] have temperature controlled self refresh (TCSR), which dynamically changes the refresh interval of a DRAM module based on the ambient temperature of the DRAM component. This saves DRAM energy as the DRAM refresh interval does not need to be conservatively designed for correctness at its maximum operating temperature. Micron DRAMs also have a low power technique called partial array self refresh (PASR). This allows the DRAM module user to select a subset of banks of the DRAM module to be refreshed for data retention. If an application does not need the DRAM for extended periods of time, the DRAM module can be sent to a *deep power down* (DPD) mode, which does not retain data and turns off most of the array power generators. Similarly, RAMBUS DRAM [4] modules typically support three low power modes, namely, *idle*, when the clock distribution is paused, *power down*, when only the clock multiplier is on and a *deep power*

*down* mode that only consumes leakage power.

A number of compiler and microarchitectural techniques have been developed to efficiently use the low power modes supported by DRAM modules. We discuss a few of these techniques in this section. Delaluz *et al.* [41] discusses compiler techniques that performs clustering of data and mode control to detect modules that are not being acceessed. Furthermore, this paper discusses several heuristics implemented in hardware at the memory controller to predict idleness of modules and efficiently switch to a low energy state. The paper reports substantial energy savings of up to 89% .

In [18], the authors save energy in a broadcast based shared memory multiprocessor system by using a regional coherence array to keep track of data present in other processor's cache and reducing access to the DRAM. This reduces DRAM read traffic by 28-32% and reduces DRAM energy consumption by 16-21%.

Hur *et al.* [58] describe three different approaches of saving DRAM energy. The paper describes a simple power down policy to better exploit the low power states of modern DRAMs. It also describes an adaptive history based scheduler and a throttling scheme to save DRAM energy without affecting performance. The paper reports up to 46% DRAM energy efficiency improments using their techniques.

Zheng *et al.* [124] describe a technique that introduces a bridge chip to break a DRAM DIMM into mini DIMMs to reduce the number of devices involved in a single memory access. Experimental results show a 44% improvement in memory efficiency.

A number of DRAM circuit techniques have also been developed to reduce DRAM power. One such technique is explained in [89], in which the authors use ECC in Embedded DRAMs to introduce an Extended Data Retention Sleep mode that increases the data retention time in this mode by more that 8 times and reduces idle time refresh energy. Another circuit technique is explained in [62], where the authors implement a four rank 3D DRAM module using a master and three slave chips. Using a single master enables the authors to remove redundant circuitry like delay locked loops, input buffers and clock

circuitry from the slave chips. This helps this implementation of 3D DRAM to save power compared to conventional quad-die package structures that have 4 ranks per module.

One of the earliest techniques of using countdown timers for tracking DRAM refresh was proposed in a patent disclosure [43]. This patent describes a timer based circuitry to reduce the number of refresh operations in a DRAM based cache. The patent's objective was to invalidate lines (via decay) that have not been accessed for a given time interval in the context of DRAM Caches.

Another similar idea of using counters to reduce refreshes is described in [92]. However, as in the case of [43], the method described in the patent is far from optimal and does not have any technique to solve the burst refresh situation. The patent disclosed by Song *et al.* [108] also described a technique to selectively refresh DRAM rows based on their access pattern. However, the technique based on the limited explanation in the patent can lead to situations where the data of a row may be destroyed because it is not refreshed in time.

Venkatesan *et al.* in [113] introduced RAPID, a retention-aware placement algorithm. This work tries to reduce refresh operations to the DRAM by experimentally identifying that different rows require different refresh times. Kim *et al.* in [69] exploits multiple DRAM refresh times and ECC codes to reduce the number of refresh operations. Ohsawa *et al.* used several techniques in [91] to reduce refresh operations required. One of the techniques used by [91] is to statically declare a line to be dead. This may also be done with the help of the OS. The lines marked as dead in the DRAM are not refreshed. Another scheme is called VRA where counters are used to handle variable data refresh times.

## 2.2 Techniques for Reducing Dynamic Power for Caches

Although leakage power has become more important for higher level caches, dynamic power consumption still dominates the power consumption for L1 caches because of the high access rate. A significant part of the dynamic cache energy is drawn by the bitline

driver circuitry because the bitlines are densely loaded with a large number of storage cells, thus increasing its effective switching capacitance. To address this issue, many low-power cache techniques were proposed including sub-banking, segmented bit-lines [49, 109], and pulsed word-line drivers [22].

Another interesting approach to reduce dynamic power is to perform data compression which allows gating off unused bit-lines, while reading from and writing data to the cache. Kim *et al.* [70] describes a sign compression technique, where the most significant half word is compressed to a sign bit to reduce energy. Canal *et al.* in [32] also applies significance compression to reduce power consumption in all stages of the pipeline. Villa *et al.* [114] describe a Dynamic Zero Compression (DZC) scheme that compresses if an entire byte is zero.

## 2.3 Leakage Reduction Techniques for Caches

There have been a large number of architectural and circuits techniques proposed to reduce leakage power in caches. Powell *et al.* [96, 121] shows that the leakage currents can be dramatically reduced by employing sleep transistors to gate off the supply voltage when the corresponding logic blocks are not in use. A microarchitectural technique called *Cache Decay* was proposed in [63] to exploit this circuit technique in the L1 cache. This technique saved energy by using simple counters to turn off cache lines if they are unlikely to be re-accessed. Although it did mention some implications of applying the decay scheme in large higher level caches, it provided no further in-depth evaluation, in particular, from the cache coherence standpoint, a correctness issue for implementing a multiprocessor (MP) system. One major drawback of the cache decay policy lies in the performance and power trade-offs of the extra misses induced due to the switch-off of decayed lines, which leads to additional accesses to the DRAM. This energy overhead often outweighs the leakage savings from the technique itself.

Another circuit technique for leakage reduction is using the ABC-MT-CMOS memory cell [90]. This circuit technique uses different supply and ground voltage levels to bias the transistors to increase their effective threshold voltage. It reduces leakage current dramatically while preserving transistor state in a lower supply-voltage, i.e., drowsy mode. Memory cells, however, have to incur a small performance penalty for waking up the drowsy cells. Flautner *et al.* [46] proposed an integrated architecture and circuit technique called Drowsy Cache, that implements a simple circuit to dynamically choose between two different supply voltage modes for leakage reduction. They analyzed different architectural policies for turning the L1 lines into drowsy mode. They also showed that they can achieve good leakage power reduction by simply keeping the data portion of all the L2 lines in drowsy mode. A specific data line is reinstated to a normal, high-power mode, only when it is re-accessed with some activation penalty. Since an L2 cache takes tens of cycles to access, adding an extra cycle or two for wake-up will be insignificant to the overall performance.

## 2.4 Microarchitectural Energy Reduction Techniques using Bloom filters

The initial purpose of Bloom filters was to build memory efficient database applications. Since then, Bloom filters have found numerous applications in networking and database areas [30, 99, 42, 72, 35, 39]. Bloom filters were also applied as microarchitectural blocks for tracking load/store addresses in load/store queues. For instance, Akkary *et al.* [19] uses one to detect the load-store conflicts in the store queue. Sethumadhvan *et al.* [103] improved the scalability for load store queues with a Bloom filter. More recently, Roth *et al.* [100] uses a Bloom filter to reduce the number of load re-executions for load/store queue optimizations. The use of Bloom filters as microarchitectural blocks for tracking load/store addresses for resolving load-store conflicts have been demonstrated in [20, 104].

The earliest example of tracking cache misses with a counting Bloom filter is given by

Moshovos *et al.* [86]. They proposed a hardware structure called *Jetty* to filter out cache snoops in SMP systems. Each processing node has a *Jetty* that tracks its own L2 cache accesses, and snoop requests are first checked in the *Jetty* before searching the cache. This is reported to reduce snoop energy consumption in SMP systems. A *Jetty*-like filter is also used by Peir *et al.* [94] for detecting load misses early in the pipeline so as to initiate speculative execution. Similarly, Mehta *et al.* [81] also uses a *Jetty*-like filter to detect L2 misses early so that they can stall the instruction fetch to save processor energy.

Memik *et al.* [82] proposed early cache miss detection hardware techniques encapsulated as *Mostly No Machine (MNM)*, to detect misses early in the multi-level caches below the L1 cache (i.e., L2, L3 etc). Their goal was to reduce dynamic cache energy and to improve performance by bypassing the caches that will miss. The MNM is a multi-ported hardware structure that collects block replacement and allocation addresses from these caches and can be accessed after the L1 access or in parallel with it.

## 2.5 Way Prediction and Estimation Techniques

The most common way prediction mechanism is to predict the MRU way as proposed in [31]. Similar way prediction techniques have been proposed in [59, 66]. Another way prediction scheme that uses the PC to predict ways is proposed in [25]. However, way prediction has the disadvantage of a large performance and energy loss if the prediction is wrong. One alternative to way prediction is way memoization proposed by Ma *et al* in [76]. Way memoization keeps way information in the instruction cache and also has a valid bit that ensures that the way information is correct. However, this technique can only be used in instruction caches. Way prediction has the disadvantage of a large performance and energy loss if the prediction is wrong. This disadvantage is mitigated by way estimation. Way estimation techniques do not predict one single way, but a set of ways where the data is guaranteed to be present for a cache hit. Therefore, way estimation techniques do not incur a large performance loss for a wrong estimation, because a wrong estimate

only results in a lookup in the cache, when it is missing the cache. One way estimation technique is the sentry tag technique proposed in [36]. This technique uses a buffer to hold one tag bit for each line of the cache. On a cache lookup, the last bit of the tag of the address is first compared to the corresponding set in the sentry buffer. On a mismatch for a particular way, the tag comparison for that way is halted, resulting in energy savings. The Way Halting technique [122] is an extension of the concept of sentry tags. This technique explained in Chapter 5.6.5 uses a buffer to hold multiple tag bits for each line of the cache. On a cache lookup, the least significant bits of the tag of the address is first compared to the corresponding set in the sentry buffer. On a mismatch for a particular way, the tag comparison for that way is halted, resulting in energy savings. Another way estimation technique is proposed in [64], in which the authors tried to predict lines that have decayed, because they have not been accessed for a fixed number of cycles. Since this technique incorporates cache decay, it is not suitable for use in the L1 caches as it may increase the miss rate considerably.

The following five chapters explain in detail the research done to exploit redundancies in the memory hierarchy to save power and improve performance. The following chapter explains a DRAM refresh power saving technique.

# CHAPTER 3

# REDUCING DRAM REFRESH POWER WITH SMART REFRESH

Dynamic Random Access Memory (DRAM) is used as the bulk of the main memory in computing systems for its high density, high capacity and low cost. Due to the dynamic, leaky nature of a DRAM cell, periodic refresh operations are required for retaining the data. Such regular refreshes account for a large energy consumption in DRAMs even in the *standby* mode. For instance, a detailed power analysis of the ITSY computer [88] shows that even in the lowest power mode, the refresh power needed accounts for about one third of the total DRAM power dissipated. The refresh rate for DRAMs depends on the memory vendor and the design technology they use. A typical refresh interval is 64ms [1, 2, 3]. The refresh intervals in embedded DRAMs are an order of magnitude shorter. A typical refresh interval for an NEC eDRAM is 4ms [8], and for an IBM eDRAM implementation is 64$\mu$s [71]. During each refresh operation, the data of every DRAM bit cell is read out and then written back. This refresh can incur large power and bandwidth overhead, nonetheless, it is inevitable for the sake of data correctness.

As processor designers moving toward the direction of integrating 3D die-stacked DRAM (or 3D DRAM) on a package to alleviate memory latency and bandwidth issues [14, 16, 95], the overhead of the refresh operations will increase. There are two reasons behind this increase. First, a 3D DRAM could be used either as a cache between the last level SRAM-based cache and the system memory or to replace the last level cache entirely. A tag array is still needed for such 3D DRAM caches for data lookup and storage. For brevity, we simply call this 3D DRAM cache a 3D DRAM hereafter. Thus, the refresh operation will become a significant overhead relatively. Second, since the 3D DRAM is bonded directly on top of the processor using die-to-die vias, the heat dissipated from the processor will be conducted across the DRAM layers, leading to a much higher temperature operation environments for the DRAM. Annavaram *et al.* [27] showed that the operating temperature of a 64MB 3D

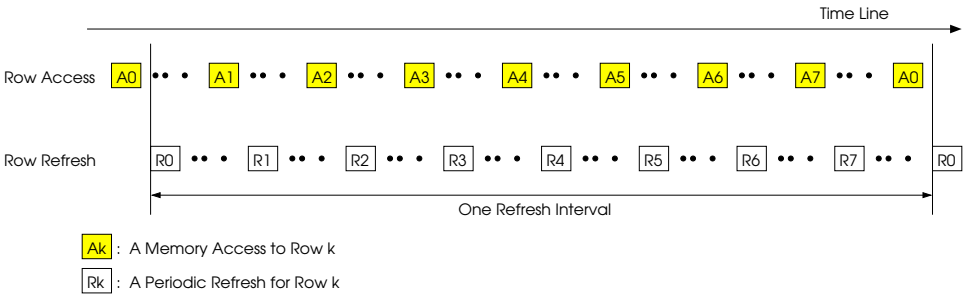



**Figure 1. Best Case for Smart Refresh**

DRAM will be 90.27°C. Furthermore, the leakage will also increase exponentially with an escalating operating temperature. According to the datasheet of Micron DRAM [83], the refresh rate must be doubled if the operating temperature exceeds 85°C. Therefore, a 3D DRAM will require double (or more) refreshes, increasing the relative energy overhead substantially.

To address these issues, in this chapter, we propose a novel technique called *Smart Refresh* to eliminate all the unnecessary DRAM refresh overheads. This technique uses a simple time-out counter for each row in a memory module, tracks the normal memory transactions, and eliminates the excessive refresh operations. The basic concept behind our scheme is that a memory row that has been recently read out or written to does not need to be refreshed again by the periodic refresh mechanism. By simply exploiting such property dynamically, the number of regular row-sweeping refresh operations in both conventional DRAMs and 3D DRAMs can be substantially reduced.

## 3.1 Redundancy in DRAM refresh

To motivate the case for our Smart Refresh technique, a conjured memory access pattern in Figure 1 is used to demonstrate the requirement for refresh operations. To simplify our illustration, we assume that there are only 8 rows in the DRAM.

In this example, we assume that the DRAM is accessed by the processor with a regular access pattern such that each memory row is accessed right before the row is to be refreshed.

For a normal, periodic refresh policy, all the memory rows will be, anyhow, refreshed by the memory controller without the knowledge of these recent accesses. Note that each access to a memory row initiated by the processor, in fact, performs an operation equivalent to a regular refresh from the standpoint of data preservation. In other words, if a row has been recently read or written to, there is no need to refresh the row immediately as shown in this figure. For the above example, in an ideal situation, there is no need to perform refresh at all since these regular memory accesses have already accomplished the same effect.

Our Smart Refresh technique exploits such energy savings opportunities by keeping a time-out counter for each row in the memory controller to minimize the required refresh cycles. Basically, the time-out counters of those rows being accessed will be reset to a default value (e.g. the refresh interval) and any following periodic refresh operation before the counter counts down to zero will be aborted. When applying such mechanism to the access pattern shown in Figure 1, the DRAM will not be refreshed at all by the default periodic refresh, without affecting the correctness. Thus we will be eliminating half of the refresh operations on the DRAM using this technique. So in theory, the best possible energy savings that can be achieved by using Smart Refresh is 50% of the entire DRAM, in which all the periodic refreshes are avoided.

## 3.2 DRAM Refresh Techniques

There are two common refresh modes in commodity DRAMs:

- Burst Refresh: In this scheme, the entire refresh operation of all the rows are done sequentially in a bursty fashion. The scheme is less desirable as it increases the peak power consumption of the DRAM. Moreover, during the time of the refresh operations, the DRAM module cannot handle normal access requests, causing potential performance degradation.
- Distributed Refresh: In distributed refresh, the memory controller spreads out the refresh cycles for different rows evenly across the refresh interval. This method is more

favorable as it refreshes each DRAM row in a timely manner, enables accesses to rows that are not being refreshed, and minimizes the delay of normal memory requests.

In addition, a DRAM refresh cycle can be implemented in two distinct ways [84]. Note that a refresh cycle can be executed in either the distributed mode or the burst mode explained above.

- RAS-only refresh: To perform a RAS-only refresh, a row address is put on the address lines and then the RAS (Row Address Strobe) signal is asserted LOW. When the RAS falls, that row will be refreshed as long as the CAS (Column Address Strobe) signal is held HIGH. It is the DRAM controller's function to provide the addresses to be refreshed and make sure that all rows are being refreshed at the appropriate times. It is important to note that for refresh operations the row order of refreshing does not matter; however, each row must be refreshed before the data stored by the cell is destroyed.
- CAS before RAS refresh: This is often referred to as CBR refresh, and is a frequently used method for refresh because it is easy to use and provides the advantage of lower power. A CBR refresh cycle is performed by setting the CAS signal to LOW (active) before the RAS signal is switched from HIGH to LOW. One refresh cycle will be performed each time the RAS signal falls. The Write Enable (WE) signal must be held HIGH during the period when the RAS signal is falling. The memory module contains an internal address counter which is initialized to a preset value when the device is powered up. Each time a CBR refresh is performed, the device refreshes a row based on the counter value, and then the counter is incremented. When CBR refresh is performed again, the next row indicated by the counter is refreshed followed by an increment in the counter. The counter is wrapped around automatically when it reaches the maximum allowable value equivalent to the number of rows. There is no way to reset the counter once set after initializing. Conventionally, CBR refresh is a more favorable refresh policy as it consumes lower power because the address does not have to be put on the bus. In this chapter we will show that our Smart Refresh technique

is suited to RAS-only Refresh, and despite the overhead over CBR, RAS-only refresh method with our Smart Refresh technique shows significant energy savings over a CBR refresh policy.

## 3.3 Smart Refresh

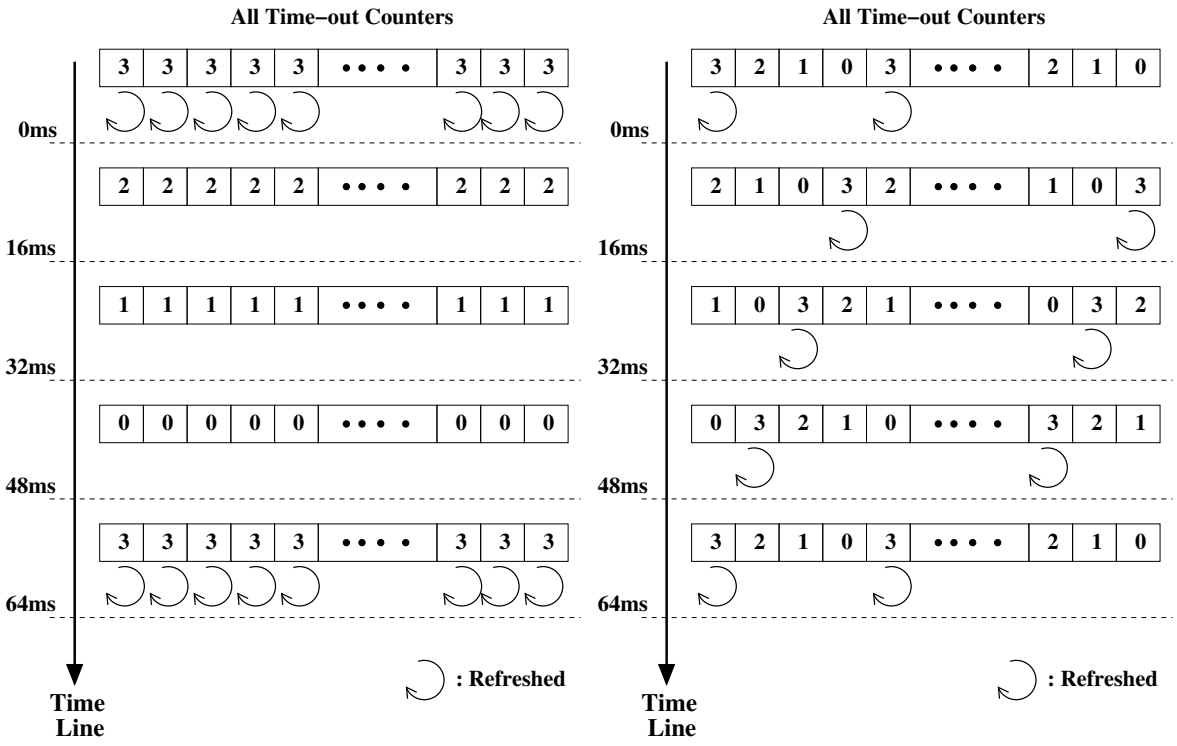
### 3.3.1 Basic Operation

Inspired by the Cache Decay work [63], our *Smart Refresh* technique applies the idea of using time-out counters in the context of the refresh operation of a DRAM to reduce dynamic energy consumption. Before we discuss Smart Refresh, we will discuss the basic operation of a DRAM access in more detail. Any DRAM read or write operation initiated by a bus agent (e.g., the processor) starts with the memory controller selecting a bank and asserting the RAS signal to LOW to be active. It simultaneously posts the row address on the address bus. This causes the corresponding memory module to activate the sense amplifiers for the entire row, and the data from the given row is brought into the sense amplifiers. Note that this read operation essentially destroys the data present in the DRAM cells. Subsequently, the CAS signal is set from HIGH to LOW (active) and the column address is placed on the address bus, which causes the column decoder to multiplex the data out for a read operation. In the case of a write operation, the data on the data bus is written to the correct set of the sense amplifiers. The data for the open row stays in the sense amplifiers until there is an access to another bank or a different row. In either case the data in the sense amps is written back to the original cells and the new row is pre-charged. We know that the refresh operation of a DRAM also involves reading from the cells and writing back to them. Thus we can see that a read or a write to a given row in the DRAM is actually the same as a refresh to that row for data retention purposes. To summarize, whenever a row is accessed, it does not need to be refreshed before another refresh interval is due. If the memory controller can keep track of the rows that have been accessed, then it can potentially delay the refresh of rows that have been recently accessed. This brings us to the concept of Smart Refresh.

The basic idea of our technique is to associate a time-out counter for each *(bank, row) pair* of a memory module. The proposed array of time-out counters is stored and updated in the memory controller. Each time-out counter is simply a 2-bit or 3-bit binary down counter. The time-out counters uniformly count down from its maximum value to zero within the refresh interval of the DRAM. If the value of a counter reaches zero, it indicates that the particular row must be refreshed. The counter is reset to its maximum value whenever the corresponding bank and row in memory is accessed and the row is opened. Since we assume an open page policy in this work, the counter corresponding to an open row is reset again when the page is closed with a precharge operation. This is because during the closing of a page, the values in the DRAM cells of the page are automatically refreshed. The memory controller does not refresh rows whose corresponding counters have a non-zero value. Hence that particular row for the accessed bank will not be refreshed during the regular refresh period. This means that whenever a row is accessed for a normal memory operation (e.g., one induced by a cache miss), the refresh operation for that row is delayed. In the best case, if every row happens to be accessed right before it needs to be refreshed, there will be no need for a separate, default refresh operation.

### 3.3.2 Staggered Countdown

In this subsection we analyze the potential problems of accessing all time-out counters simultaneously. Let us consider a Smart Refresh memory controller that has a 2-bit time-out counter for each row of the DRAM. The array of counters is illustrated in Figure 2(a) horizontally. The refresh cycle in this example is assumed 64ms. For simplicity we assume that there is no access to the DRAM in these examples. The figure shows the counter value for each row of the DRAM as it is being updated by the memory controller. The time-line flows from top to bottom. The 2-bit counter is designed to down-count from 3 to 0 within 64 ms to ensure refresh to all rows are done timely to retain correct data values. If all counters are decremented simultaneously as shown in Figure 2(a), then they will be decremented at times 16ms, 32ms and 48ms respectively. At 48ms, all the counters reach 0 and when the




(a) Timeout counters decremented together

(b) Timeout counters initially staggered

**Figure 2. Down-counting Timeout counters**

memory controller accesses them again at 64ms, all the rows must be refreshed at that time, similar to a burst refresh condition that adversely reduces memory system performance. We should note that even though all the rows need to be refreshed at the same time, they can only be refreshed in a sequential order.

One solution to partially take care of this unwanted burst refresh situation is shown in Figure 2(b). In this figure, the initialization of the time-out counters is staggered.In this case, one quarter of all the counters will decrement to zero at 16ms, another quarter become zero at 32ms, and so on. We have a situation similar to burst refresh where many memory rows need to be refreshed one after another. This staggering at the beginning also incurs some power overhead, because at the beginning even all rows have been refreshed, but 1/4 of the counters are initialized to 0. Therefore they are refreshed again within the first 64ms. This however does not solve our problem. When the rows are accessed during normal

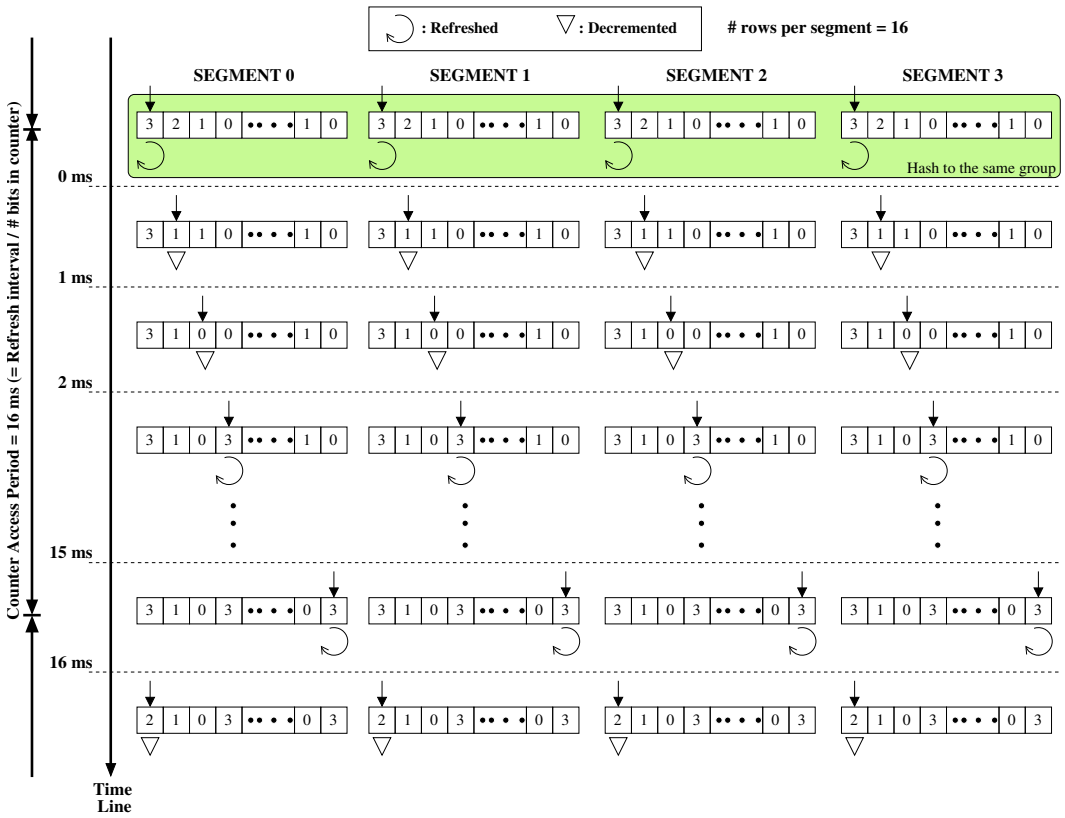



**Figure 3. Countdown counters divided into logical segments and countdown is staggered**

processor reads and writes, their corresponding counters are reset to its maximum value. This could lead to burst refresh like conditions as potentially a large number of counters may have the same value and since they are decremented together, they will all count down to zero at the same time. This problem can be solved only if the decrement to the counters is also staggered along with the initialization.

The solution used in our design is shown in Figure 3. In this scheme, the counters are evenly hashed into $N$ logical segments where $N = 4$ in this illustration. The selection of $N$ segments is based on the size of the *pending refresh request queue* to be explained in Section 3.4. All simulations were done using 8 entry pending refresh queue and 8 segments. The major difference of this technique with previous techniques is that in this solution all the counters will not be accessed by the memory controller simultaneously.

In this new staggered scheme, refresh or counter decrement by the memory controller are only allowed for those four *indexed* counters (with arrows shown on top of the counter

in the figure) at a given time. As a result of the hashing function, only $N$ counters (4 in this case) are active at the same time. The goal of this scheme is to index each counter exactly once within a so-called *counter access period* which is defined as the refresh interval (i.e., 64ms in our example) divided by the size of the counter (= $2^{2bit}$ = 4). In Figure 3, the counter access period is 16ms. The index is advanced to the next counter by a clock period equal to the counter access period divided by the number of time-out counters (i.e., memory rows) within each segment. For example, if there are 16 memory rows for each segment and the refresh period is 16ms, then the counter index will advance by one every 1ms. The update of the counter is the same as previously described. When the value of the indexed counter is zero, at the next time it is indexed again, the counter will be reset back to the maximum value followed by a refresh request for the corresponding memory row; otherwise, the indexed counter value simply decrements by one. The refresh request is immediately sent to the pending refresh request queue for dispatching a refresh operation. Without any memory accesses issued by the memory controller, the refresh policy is similar to a distributed refresh policy, with each refresh operation performing a burst refresh for N memory rows, the same size of pending refresh request queue.

The above solution ensures that the number of counters accessed simultaneously is equal to the number of segments ($N$) chosen. This makes sure that we do not have more than $N$ refreshes pending in the pending refresh queue simultaneously. In Section 3.4 we show that the for DRAMs with refresh time of 32ms, the time interval between accessing counters is enough for completing the refreshes in the pending refresh request queue. This proof for the 32ms case automatically proves the 64ms case. This staggering algorithm also ensures accesses to counters at regular intervals and thus the staggering will not reduce over time, avoiding any possible situation where burst refresh may occur.

Now let us assume there are normal accesses intersperse with refresh operations. Whenever a memory row is accessed by normal reads or writes, the counter corresponding to the row will be reset to the maximum value. Thus refresh operation to the counter will be

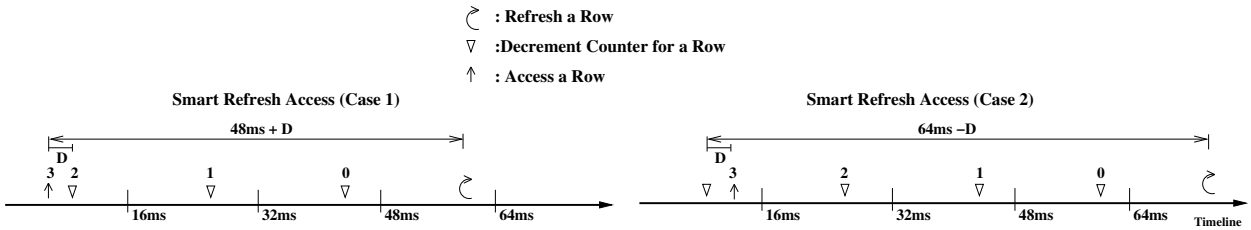



**Figure 4. Smart Refresh Correctness**

delayed until it counts down to zero. Our staggered countdown mechanism guarantees that another refresh only takes place 64ms after the row has been accessed instead of a regular refresh period. This delaying of refresh of memory rows that are being accessed enables Smart Refresh to save significant amount of refresh energy if enough rows are accessed.

The size of the counter is chosen to be 2 bits for our explanation for the technique. We actually used a 3-bit counter for our simulations. The size of the counter determines the granularity with which the refresh operations can be controlled. A larger sized counter will need more steps to count down and allow more finer grained control over how much time the refresh operation can be delayed once the corresponding row is accessed. This will lead to potentially greater power savings at the cost of maintaining and accessing a bigger counter array.

### 3.3.3 Smart Refresh Correctness

We prove that for an arbitrary access pattern the Smart Refresh scheme always refreshes the data within the refresh interval deadline. The proof for this is pictorially described in Figure 4. For the example shown in the figure the refresh interval chosen is 64 ms and the counter is 2 bits wide. The figure just shows the Smart Refresh technique is applied to one particular memory row and its associated counter. The inverted triangles show the times when the counter is decremented. The number above the triangle represent the counter value after it was decremented. As explained earlier, the counter is decremented exactly once within 16ms. We can have only two possible cases for an access to this row. The figure shows that in both cases the row will guarantee be refreshed within 64ms.

In the first case on the left-hand side, the row is accessed *D ms* before it is decremented. The access is denoted by an upward arrow. Note that $D < 16ms$. An access to the row resets the counter to its maximum value 3. After *D ms* the counter is indexed and decremented to 2. From the time-line progression the counter becomes *0* in *D + 32 ms*. Thus when the counter is accessed again at *D + 48 ms*, the memory controller sees a *0* and refreshes the row. Therefore, in this case the row is refreshed after *D + 48 ms* after it is accessed and meets the deadline of 64ms.

The second possible case is that the row is accessed *D ms* after the counter is decremented as shown in the right-hand side. The counter value becomes 3 on the access. It gets decremented to 2 at *16ms -D*, and *0* at *48ms - D* after its access. Finally, it is refreshed at *64ms -D* after it was accessed. Since this is less than 64ms, the refresh is effective.

For a D greater than 16ms, the scenario can be reduced to either Case 1 or Case 2 by subtracting 16ms from D repeatedly until D is smaller than 16ms. Therefore, we show that in all possible access patterns, the row will always be refreshed before its data retention deadline.

### 3.3.4 Optimality of Smart Refresh

We define optimality for refresh as a metric of how close a DRAM row is refreshed to the data retention deadline. Thus an ideal scheme where each row is refreshed exactly after 64ms is said to be 100% optimal. In the Smart Refresh case, the optimality of the scheme depends on the number of bits we use for each counter. We can easily see from Section 3.3.3, if we use a two-bit counter for every row, the least optimal case will be when all the rows are refreshed at *48ms + D* where D is close to zero. Thus the optimality of Smart Refresh for a 2 bit counter is *48/64* = 75 %. Similarly for a 3 bit counter the worst case comes when each row is refreshed at *56ms*. Thus the optimality of Smart Refresh for a 3 bit counter is 87.5%. The general optimality formula is a function of the counting

granularity and can be given by:

$$Optimality = [1 - \frac{1}{2^{N_{bits\_per\_counter}}}] * 100\%$$

### 3.3.5 Smart Refresh Technique for 3D DRAM

3D die stacking is an emerging technology that vertically integrates two or more die with inter-die vias [14, 16, 27, 28, 98]. These vias serve both as a fast communication interface and a stability providing mechanism to the stacked die structure. 3D die stacking reduces wire length and provides tight, high-speed coupling of die designed and manufactured with incompatible technologies. One immediate application is to integrate 3D die-stacked DRAMs with processor cores to alleviate the memory latencies and global wire power consumption by replacing long on-board wires with short, fast inter-die vias [27].

The refresh operation will be a major overhead for 3D DRAMs. The operating temperature of the 3D DRAM will likely be much higher than their conventional DRAM counterpart. As shown in [27], a 64MB 3D DRAM will raise its operating temperature to 90.27°C. According to [83], the refresh rate must be doubled after the temperature exceeds 85°C in order to retain data. Therefore, 3D DRAMs will have a higher refresh rate than conventional DRAMs, increasing the power consumption and potentially the access latency. On the other hand, another design trend could increase the number of accesses to the 3D DRAM compared to a conventional DRAM. As multi-layer DRAM is made possible to be integrated on the same package, it reduces the requirement of having a large L2 on the processor core for area/cost efficiency. The constraints on the size of the 3D DRAM is mainly the number of DRAM cells that can be fitted into the available die layers and the number SRAM tags that can be fitted into the processor die for accessing the 3D DRAM.

Another interesting aspect of 3D DRAMs is that it will be more frequently accessed. Thus the refresh operation will also have a noticeable performance overhead. Our Smart Refresh technique, in fact, uses the more frequent accesses to its advantage to significantly reduce the amount of refreshes required for a 3D DRAM implementation. In Section 3.6.2

we will discuss in more detail the performance and energy benefits of using a 3D DRAM.

### 3.3.6 Smart Refresh for embedded DRAMs

Embedded DRAMs are increasingly being used in modern SOCs. The main reason for their use is compatibility with the logic CMOS process enables them to be integrated on a SOC at a relatively low cost. This allows eDRAMs to offer higher bandwidth with wider on-chip memory buses. However,the refresh intervals in embedded DRAMs are an order of magnitude shorter than conventional DRAM. A typical refresh interval for an NEC eDRAM is 4ms [8], and for an IBM eDRAM implementation is 64$\mu$s [71]. Since the comventional DRAMs have refresh overhead of 64ms, embedded DRAMs need to be refreshed 16 times (for 4ms) or 1000 (for 64$\mu$s) times more frequently. This increases the overhead of refresh of an embedded DRAM by an order of magnitude compared to conventional DRAMs.

However, since embedded DRAMs are faster than their conventional counterparts, they will be typically accessed more often. This increasing access frequency significantly increases the opportunity of our Smart Refresh technique to save more energy. Furthermore, since in the embedded DRAM case the refresh operation is a major impediment to DRAM availability for normal accesses, the Smart Refresh technique will have a big impact in improving DRAM availability and hence system performance.

### 3.3.7 Disabling Smart Refresh

By no means will Smart Refresh always reduce refresh operations in the memory. This happens when the entire data working set fit into L1 (and L2) caches with very infrequent accesses to the DRAM. In this case the refreshing action of Smart Refresh will be degenerated to that of the CBR policy. However, Smart Refresh will consume some extra energy in maintaining those counters and also using the address bus for the RAS-only refresh operation. To avoid this situation we add a simple circuitry that can disable Smart Refresh policy and configure the memory controller to perform a regular CBR policy if accesses to memory is found to be below 1% of the number of rows over the total refresh interval

(64ms or 32ms). The same circuitry will also be responsible for turning Smart Refresh on autonomously, if the accesses to DRAM exceed 2% of the number of rows in it. This turn-off is especially useful for the conventional DRAM below a large 3D DRAM cache, whose size is of the order of 32MB or 64MB as studied in [27]. Also, for the conventional DRAM we checked this policy by simulating an idle OS for 1 billion instructions. We observed that even for the idle OS we got savings of around 10% in refresh energy consumption. With such self-configurability, this feature will exploit dynamic data working set behavior for achieving the best energy management.

### 3.3.8 Area Overhead

We now explain the storage overhead for maintaining the time-out counters. In our design, we refresh one row for a specific bank and rank in a single refresh operation. Hence, we need to maintain a counter for each row and each bank in each rank. First we use the configuration shown in Table 3.4 for a 2GB DRAM module. The number of banks, ranks, and rows for the module is 4, 2, and 16384 respectively. Since we need a counter for every bank, rank and row the number of counters needed are 4 * 2 * 16384 = 131,072. Each of these counters has 3 bits. Thus the area overhead is (131,072 * 3)/(8 X 1024) = 48 KB. If we assume that the memory controller can support up to 32 GB, the counter space needed will be 768 KB. A simple formula is derived below for calculating the area overhead in the counters. In our experiments in Section 3.6, the energy overheads caused by these extra counters were all accounted for.

$$Area = \frac{N_{banks} * N_{ranks} * N_{rows} * N_{bits\_per\_counter}}{8 * 1024}(KB)$$

## 3.4 Smart Refresh Implementation

The schematic of the circuitry controlling refresh operations in the memory controller is illustrated in Figure 5. We can see that the counter update circuitry updates a specific number of time-out counters in a memory controller cycle. If one of the counters that

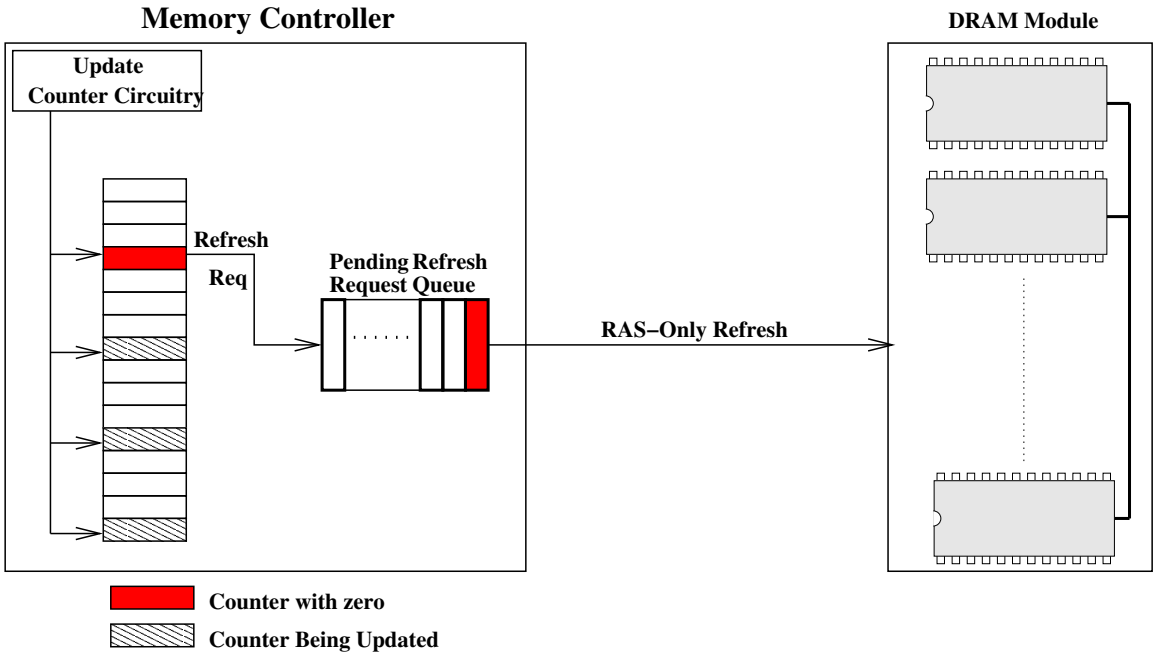



**Figure 5. Smart Refresh Control Schematic**

needs to be updated has counted down to zero, then the row address and bank address corresponding to the counter are inserted into the *pending refresh requests queue*. The memory controller reads the addresses in the pending queue and puts the least recent row address on the data bus and issues an RAS only refresh command. This requires neither changes in the existing DRAM module itself nor the interface between the DRAM module and the memory controller. The only changes are inside the memory controller, making Smart Refresh a highly viable and cost effective technique.

One potential issue of having a queue to store pending refresh operations is to find out any possible case of DRAM access patterns where the queue may overflow. We show that this is not possible. A typical time taken to refresh a row is 70ns [3]. As explained earlier, if the refresh interval is 32ms and there are 8192 rows in the device, the counters are accessed every 4$\mu$s. Now if we choose the size of the refresh pending queue to be 8 entries then we will divide the row into 8 segments. This will guarantee that at most 8 refresh operations are triggered at a time. To avoid overflow for a queue having eight entries, it is essential that all the eight refresh transactions are handled till the next time the counters are accessed. Since refreshing a row takes 70ns and the counters are accessed every 4$\mu$s,

if there is no normal DRAM access, the number of rows that may be refreshed between successive counter accesses will be 57. Nevertheless, in the worst case, we only need to refresh 8 rows in that deadline. Thus a queue of length 8 is sufficient for the purpose and it will never overflow. In the worst case, normal DRAM accesses may get delayed due to at most 8 refresh requests coming one after another. However, our experiments show that for all the benchmark programs considered, since we reduce the refresh operations considerably, any interference refreshes may have with normal accesses is reduced and we always have a slight performance improvement.

We would like to emphasize that Smart Refresh for RAS only refresh does not change the interface between the Memory Controller and the DRAM module. Although CBR refresh is often chosen as the refresh policy for modern DRAMs, we use it as a baseline in our results to show that Smart Refresh provides significant savings even after considering the additional overhead of RAS only refresh.

Another potential issue with "Smart Refresh" is the design of the memory controller with requisite number of counters, as the size of DRAM is not known when the memory controller is designed. This problem can be handled by the memory controller having multiple banks of count-down counters. The total number of counters would be the number of rows for the maximum permissible size supported by the memory controller. The BIOS will turn on requisite number of banks on startup of the system, based on the memory size and configuration.

## 3.5 Evaluation Methodology

Our simulation infrastructure consists of three portions: *Simics* [77], *Ruby* [79] and *DRAMsim* [116]. Firstly, we used Virtutech Simics to execute the benchmark applications. Simics is a full system emulator that can run unmodified production software like full blown operating systems. This infrastructure was used to emulate a "Sun" virtual machine called "sarek" running a version of Solaris 8. We used three different benchmark suites —

**Table 1. DRAM Module and L2 Cache Configuration**

| Parameter | Value |
|---|---|
| *Type* | DDR2 |
| *Size* | 2 GB and 4GB |
| *Rows* | 16384 |
| *Frequency* | 667 MHz |
| *Number of Banks* | 4 and 8 |
| *Number of Ranks* | 2 |
| *Number of Columns* | 2048 |
| *Data Width* | 72 bits (64 data + 8 ECC) |
| *Row Buffer Policy* | Open Page |
| *Refresh Interval* | 64ms |
| *L2 Cache Size* | 1 MB |
| *Number of L2 Port* | 1 |
| *L2 Cache Assoc* | 8 ways |

**Table 2. 3D DRAM Cache Configuration**

| Parameter | Value |
|---|---|
| *Type* | DDR2 |
| *Size* | 64 MB |
| *Rows* | 16384 |
| *Frequency* | 667 MHz |
| *Number of Banks* | 4 |
| *Number of Ranks* | 1 |
| *Number of Columns* | 128 |
| *Data Width* | 72 bits |
| *Row Buffer Policy* | Open Page |
| *Refresh Interval* | 32ms |
| *Ports* | 1 |
| *Associativity* | Direct Mapped |

SPLASH2 [120], SPECint2000 and Biobench [21] for their different memory behaviors. All programs were compiled for the Solaris machine and installed in the virtual disk. Although we could successfully compile all programs, not every benchmark ran for sufficient number of instructions due to limitations and incompatibilities of the simulation infrastructure. We report results for those applications that successfully ran on the simulated system. Except for SPLASH2 that was executed on a 2-processor emulated CMP system sharing a 1MB conventional L2 cache, all other benchmark programs were run on a uni-processor system. We also run a set of experiments where we selectively pair off any two SPECint benchmark programs and run them together to emulate a multi-workload

execution environment. These experiments were performed to observe the effect of more frequent memory look-up's for our technique. Although Simics is a full system emulator, we only use it for functional simulation. To simulate memory and cache behavior in details, the Ruby module developed at University of Wisconsin was loaded into Simics. Ruby leverages the full system infrastructure of Simics and provides timing simulation for the memory hierarchy. However, Ruby does not faithfully simulate the DRAM behavior. The characteristics of DRAM were, on the other hand, simulated using a third simulator called DRAMsim [116] from University of Maryland. DRAMsim can be used either as a standalone trace-driven simulator or as a module that can be integrated into Ruby. The complete implementation of our Smart Refresh technique was done in DRAMsim. Table 3.4 shows the DRAM module and the L2 cache size used in our simulation. We used the conventional DDR2 memory rather than the latest FB-DIMM for our simulation because the conventional DDR2 performs better for benchmarks that are not limited by bandwidth [47], which is the case for our benchmarks. The module configurations were based on actual DRAM specification from [12]. The 3D DRAM Cache configuration is shown in Table 13. The DRAM refresh command policy is one-channel, one-rank, one-bank for all configurations. Each benchmark was simulated for 1 billion instructions after fast-forwarding the first billion instructions.

The calculation of power involves two distinct components: the power consumption of the DRAM module, and the power overhead of the newly proposed time-out counters. To calculate power consumption for the DRAM module we used the power model provided by DRAMsim [116]. For the time-out counters we assume a design consisting of an array of SRAM bits storing the counter values and a logic circuit for the decrement operation. The SRAM bit array has entries equal to the number of rows in the DRAM which is of the order of thousands. Accessing such an array will need a large decoder and very long bit-lines to transfer the data out. In contrast, the counter logic will have tens of gates. Therefore the energy consumption of storing and accessing the array of SRAM bits will be

an order of magnitude larger than the energy consumed by the logic circuitry to decrement the respective values. Thus the energy consumption of the logic circuitry was neglected in our energy calculations. The SRAM array was designed using the *Artisan* 90nm SRAM library to get an estimate on the dynamic energy required to access it. The Artisan SRAM generator is capable of generating synthesizable Verilog code for SRAMs using 90nm process technology. The generated datasheet gives an estimate of the read and write power of the generated SRAM. The counter arrays may be accessed in two different situations. First, when a specific row is accessed and its corresponding time-out counter needs to be reset. This is considered as a write operation to the SRAM array. The second case is that when a counter is checked against zero value for triggering a refresh. When the value is positive, it is decremented. As explained in Section 3.3.2, whenever the counters are accessed for decrementing, eight counters are decremented at the same time. Therefore, in our design we count eight reads and eight writes for each such counter access operation. The results of the simulation will be presented in the next section.

Since Smart Refresh uses RAS-only refresh that consumes relatively more energy than CBR refresh due to the requirement of posting the row address, we assume that the baseline DRAM uses CBR refresh (a lower power baseline) in our experiments, while the Smart Refresh DRAM is based on RAS-only refresh. The extra power for RAS-only refresh is mainly consumed in putting the row address to be refreshed on the bus. To model the power consumption of the bus we use the elementary model explained in [33]. The energy consumption of the bus is given by:

$$\text{Energy} = C * V_{DD}^2 * Width\_of\_Bus * Num\_Accesses$$

Here "C" is the average capacitance of one wire of the bus and is given by:

$$\text{C} = C_{load} + C_{driver}$$

Now for proper impedance matching according to [33], $C_{driver}$ is chosen to be 30 % of $C_{load}$. Thus $C = 1.3 * C_{load}$.

**Table 3. Parameter Values Used in Bus Energy Calculation**

| Parameter | Value |
|---|---|
| *On Chip Length* | 36 mm |
| *Off Chip Length* | 102 mm |
| *On Chip Wire Capacitance* | .21 pF/mm |
| *Off Chip Wire Capacitance* | 0.1 pF/mm |
| *Input Capacitance of Memory Modules* | 3 pF |

$$C_{load} = L_{onchip} * C_{permmonchip} + L_{offchip} * C_{permmoffchip} + \sum_{m \in M} C_{in}(m)$$

where "M" is the number of memory modules (ranks) in the DRAM system, and $C_{in}(m)$ is the input capacitance of each module.

The estimation of wire length on the chip is done using the widely used *"semi perimeter"* method [101]. The on-chip length ($L_{onchip}$) is taken as double the length of one side of the Intel 855PM Chipset MCH die [9]. Typical values of off-chip length was determined from Intel 855PM Chipset design guide [10]. Values of the on-chip capacitance per unit length was obtained from the ITRS Roadmap [11]. The input capacitance of a memory module was obtained from Micron datasheet [1]. The actual values used for these equations have been summarized in Table 3.5.

Apart from evaluating our technique for conventional DRAMs we also performed experiments to evaluate the effectiveness of Smart Refresh for the emerging 3D DRAMs. We extend DRAMsim functionality to simulate the processor using 3D DRAM as another level of cache between L2 and the on-board DRAM. Note that, to access and allocate data on the 3D DRAM cache, an SRAM tag array is still needed on the processor. We implemented Smart Refresh for such a configuration and ran the same benchmark programs for two different sized 3D DRAM (32MB and 64MB). Practically, the 3D DRAMs are level 3 caches integrated directly on top of a processor core with a 256KB Level 2 SRAM cache. The capacity of the 3D DRAM is limited by the core area and the number of DRAM die

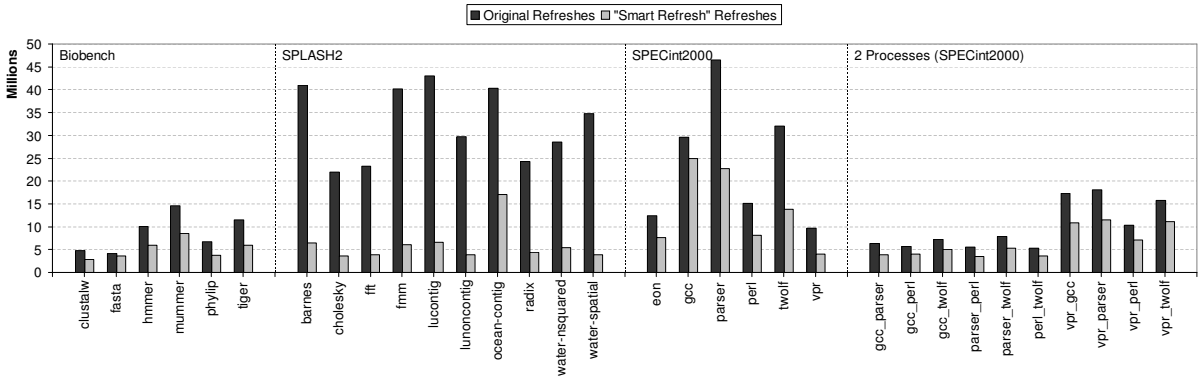



**Figure 6. Comparison of Number of Refreshes per second for a 2GB DRAM**

layers available. We chose the sizes in accordance with the results of the feasibility study given in [27]. According to [27], 3D DRAMs will operate at much higher temperatures (90.27°C) than conventional DRAMs, we performed our experiments using two different refresh intervals (32ms, and 64ms). The following section discusses our results.

## 3.6 Experimental Results

### 3.6.1 Conventional DRAM

Figure 6 shows the number of refresh operations per second taking place for each benchmark program. To show the effectiveness of our technique, we mark the baseline number of refreshes per second required for a memory module with the same configuration.

From Figure 6 we can observe that, though the relative reduction in refreshes heavily depends on the memory behavior of an application, the Smart Refresh technique is very effective in reducing the number of regular refresh operations. The reductions in refresh operations per second range from around 26% for fasta to as high as 85.7% in water-spatial. On average, our technique can reduce more than 59.3% of regular refresh operations over all the benchmark programs.

Figure 7 shows the relative energy consumption for Smart Refresh for refresh operations. We can see that Smart Refresh is successful in saving a significant percentage of energy consumed in refreshing the DRAM. The savings range from 25% in gcc to as much as 79% for radix. On an average Smart Refresh saves 52.57% of energy consumed in

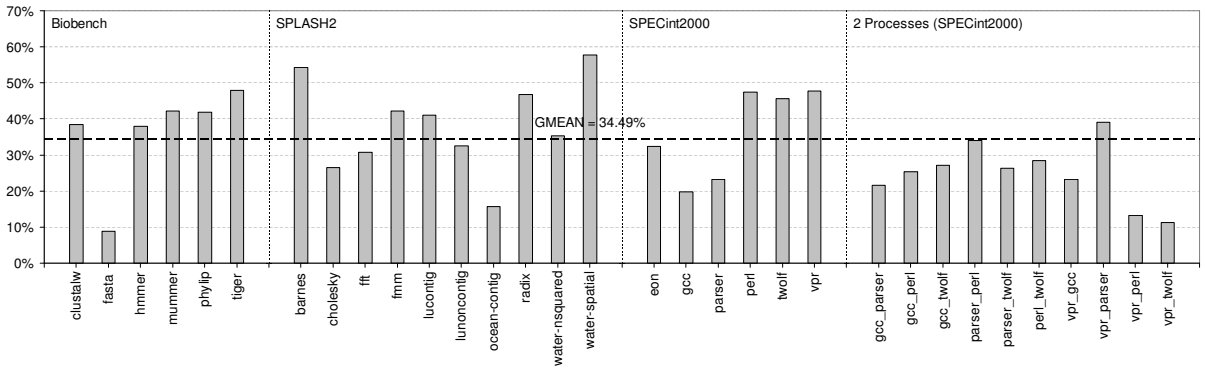



**Figure 7. Relative Refresh Energy Savings for a 2GB DRAM**

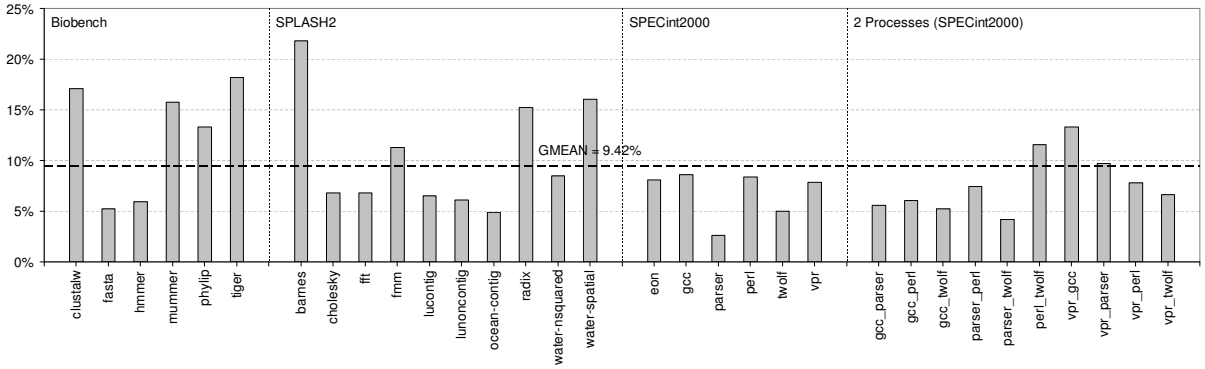



**Figure 8. Relative Total Energy Savings for a 2GB DRAM**

DRAM refresh. We should note that there does not exist a linear relationship between the percentage reduction in the number of refresh operations and the relative reduction in refresh energy. This is because the energy consumed in refreshing a row depends on the state of the bank where the row is being refreshed. For example, if the row of sense amplifiers is in the *precharge* state for the bank where a row is being refreshed, the refresh operation involves bringing the row of data being refreshed to the sense amps, restoring their charge, writing them back to the row and precharging the sense-amps for the next operation. However, if the row of sense amps already has an open page, the refresh operation will involve writing the present open page back to the DRAM cells, precharging the sense amps and then refreshing the row as above. This clearly consumes more energy than the case when the sense-amps were already precharged.

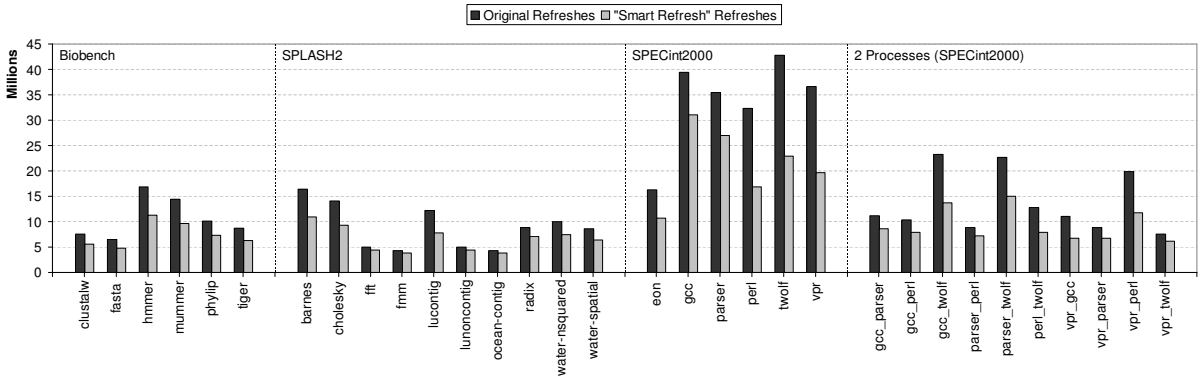



**Figure 9. Comparison of Number of Refreshes per second for a 4GB DRAM**

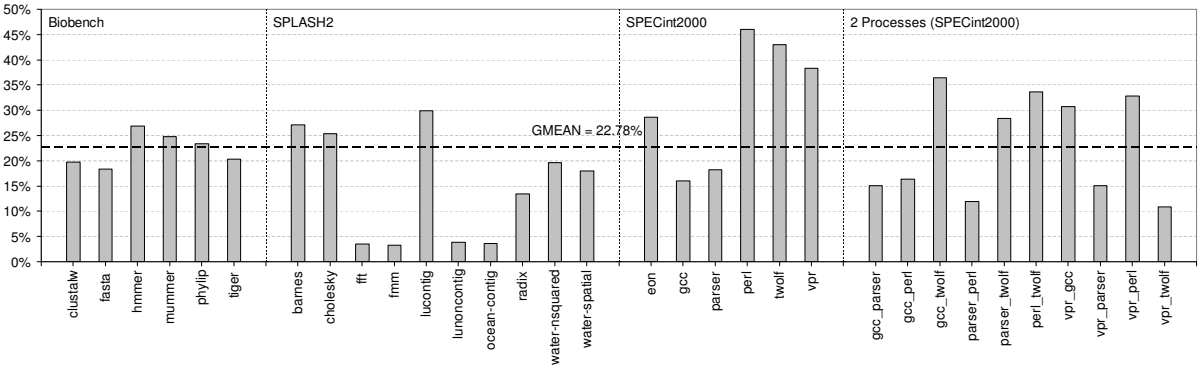



**Figure 10. Relative Refresh Energy Savings for a 4GB DRAM**

Figure 8 shows the relative energy consumption for the DRAM. We took into account the energy overhead of maintaining the time-out counters in the memory controller. We can see that benchmarks that have high refresh energy savings also have large savings in the total energy. Thus benchmark programs such as perl_twolf whose relative refresh energy savings is high, show high total energy savings of 25%. On average, the total savings of DRAM energy is around 12.13%. However, we must also note that there is no exact linear relationship between the relative refresh energy savings and the total DRAM energy savings. This is because the total energy savings depend heavily on what percentage of the total DRAM energy is contributed by refresh energy. This depends on the number of memory references of a benchmark.

Figure 9, Figure 10, and Figure 11 show the number of refresh operations per second, relative refresh energy and relative DRAM energy for a 4GB conventional DRAM. For the baseline CBR refresh technique, we find that the number of refresh operations per

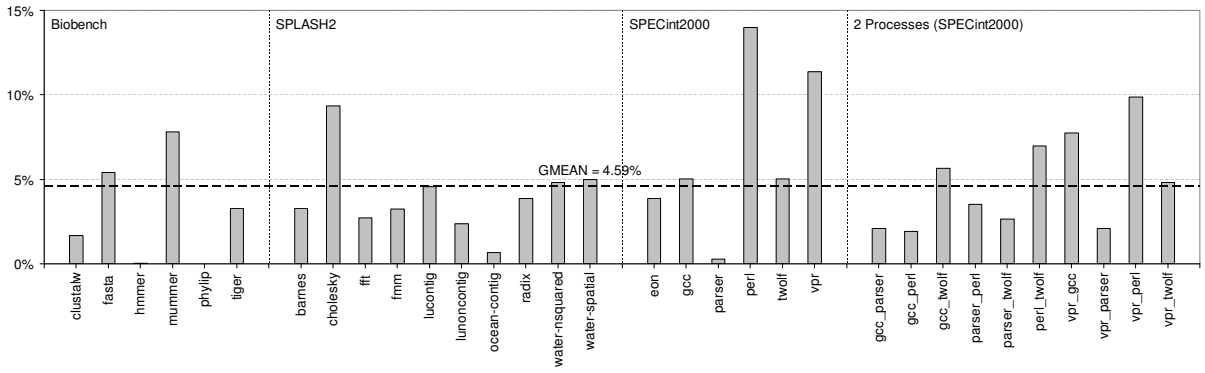



**Figure 11. Relative Total Energy Savings for a 4GB DRAM**

second for every benchmark with the 4GB DRAM module is doubled compared to the 2GB module. This is expected because the 4GB DRAM module has double the number of banks. Since the refresh command policy used in all experiments is one channel/one bank/one rank, the number of rows that need to be refreshed for a 4GB module is twice the number of rows of the 2GB module. As all benchmarks require similar number of cycles to complete in both the 2GB and 4GB configurations, the number of refreshes for the 4GB module is doubled. We observe that the relative reduction in refresh operations is around 40% for a 4GB DRAM. The average reduction in refresh energy is 23.76% and total energy reduction is 9.10%. The energy savings for a 4GB DRAM is generally lower because all the benchmarks simulated have a memory footprint less than 2GB. So on using a 4GB DRAM we increase both the base DRAM energy consumption and also the energy required to maintain double the number of time-out counters. This reduces the savings that can be obtained using the Smart Refresh technique. For example, phylip from Biobench had about 13.3% total energy savings as shown in Figure 8 while the savings dropped down to almost 7.3% as shown in Figure 11. Also, as in the case of the 2GB DRAM module, there is no linear relationship between relative reduction in refresh operations, the refresh energy savings, and the total energy savings.

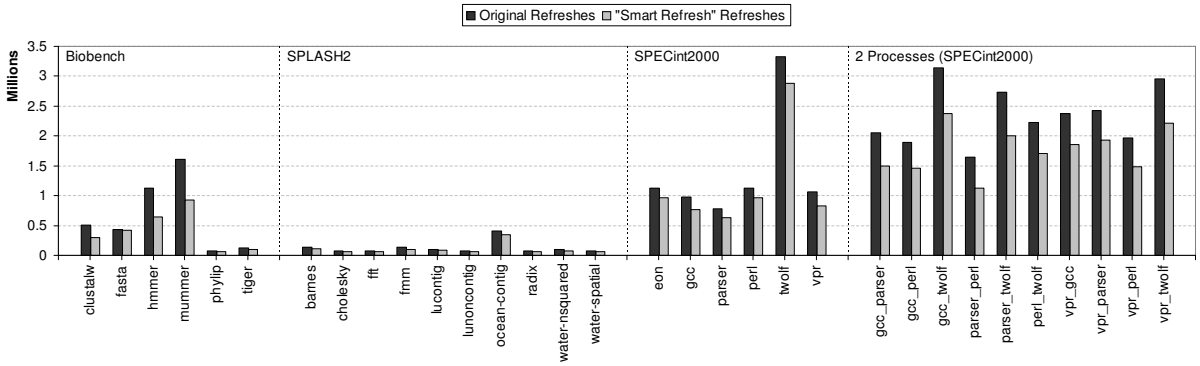



**Figure 12. Comparison of Number of Refreshes for a 64MB 3D DRAM Cache with 64ms refresh rate**

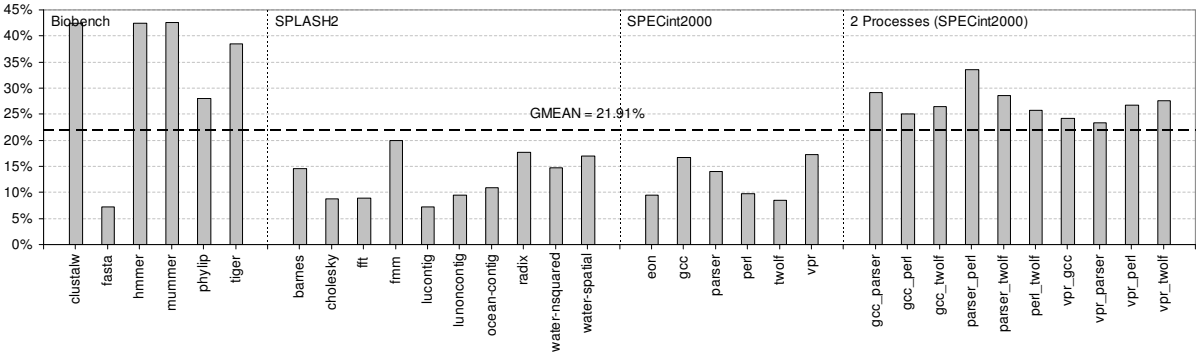



**Figure 13. Relative Refresh Energy Savings for a 64MB 3D DRAM Cache with 64ms refresh rate**

### 3.6.2 3D Die-stacked DRAM

To study the benefit of Smart Refresh for emerging 3D integration technology, we performed experiments by assuming a limited size DRAM bounded with a processor through die-to-die vias. The results for the 3D die-stacked DRAM with a capacity of 64MB total and a 64ms refresh period are shown in Figure 12, Figure 13 and Figure 14. As in the case of the conventional DRAM, the reduction in the number of refresh operations per second

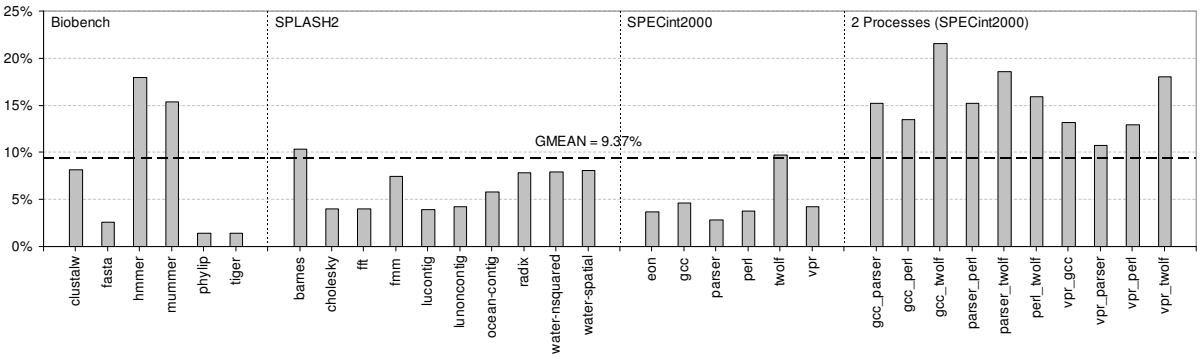



**Figure 14. Relative Total Energy Savings for a 64MB 3D DRAM Cache with 64ms refresh rate**

shown in Figure 12 highly depends on the benchmark considered. The reduction in refresh operations is significant. It ranges from 42% in mummer to 4% in fasta. Figure 13 shows the relative energy savings using Smart Refresh for 3D DRAMs with 64ms refresh rate. Though there is no linear relationship, we observe that the refresh energy savings follow the number of refresh operations reduced per second. The savings range from 42% in the Biobench benchmarks like clustalw, mummer, to as low as 7% in fasta. The geometric mean of refresh energy savings is 21.91%.

Figure 14 shows the total energy savings for the same 3D DRAM configuration. These savings numbers consider that the baseline 3D DRAM cache uses CBR refresh. Thus the power consumption in the wires and vias connecting the memory controller in the processor die and DRAM in the stacked die have been modeled and added as an overhead for the Smart Refresh technique. It can be seen from the savings that refreshes are a significant overhead in 3D DRAMs. We obtain savings of up to 21.5% when running gcc and twolf together. The geometric mean of the savings are 9.37%. A general trend that can be observed from these simulations is that the savings continue to increase for those systems running two processes. One reason for this is that dual process benchmark runs contain less spatial locality of accesses than a single benchmark. So it is more likely for a 2-process benchmark to access different rows rather than having a row buffer hit all the time. Since different rows are accessed, fewer number of rows need to be refreshed and this helps in saving energy. For the 3D DRAM cases, we also had a 2GB conventional DRAM that back up the 3D DRAM which is essentially used as a Level 3 DRAM cache. Since these benchmark programs fit into the 64MB cache and accesses to main memory was negligible, in the order of a few thousands over more than 2 billion cycles for all the benchmarks, thus we did not observe Smart Refresh shows any energy savings for the conventional DRAM with a 64MB 3D DRAM integrated on top of a processor face-to-face. But since Smart Refresh can effectively switch off all the counters and go to CBR refresh mode when less than 1% of the rows are accessed over a whole refresh interval as described in Section 3.3.7,

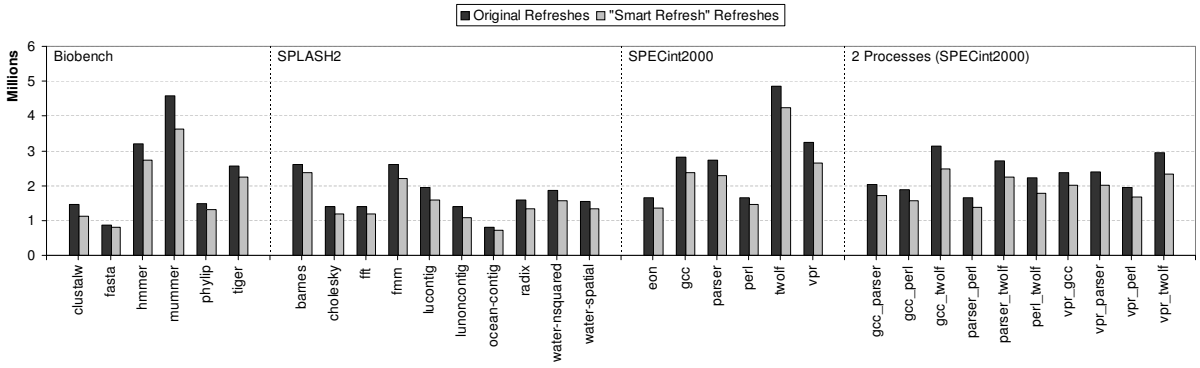



**Figure 15. Comparison of Number of Refreshes for a 64MB 3D DRAM Cache with 32ms refresh rate**

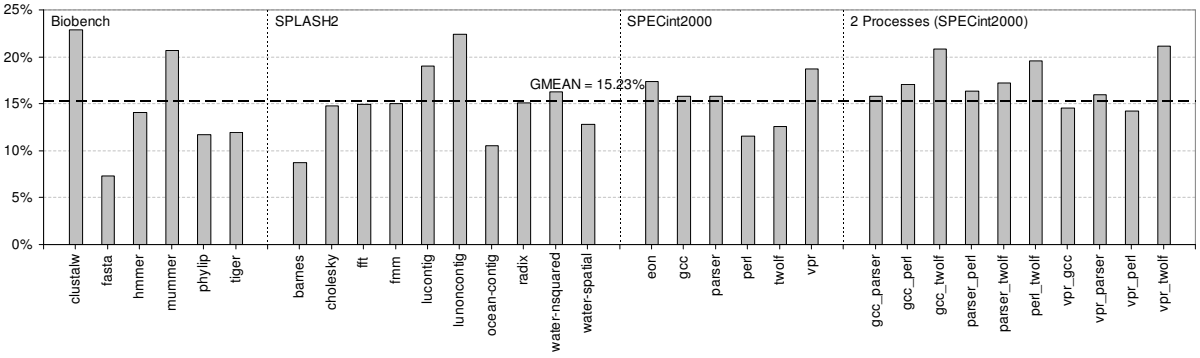



**Figure 16. Relative Refresh Energy Savings for a 64MB 3D DRAM Cache with 32ms refresh rate**

we did not detect any energy loss in the conventional DRAM.

Since the 3D Cache will operate at a temperatures of 90.27°C, the refresh rate required will likely be doubled from the 64ms refresh rate. Therefore, we conducted experiments on the same 64MB 3D DRAM with a faster 32ms refresh rate.

Figure 15 compares the number of refresh operations per second using Smart Refresh with a conventional CBR refresh for the 64MB DRAM with the doubled 32ms refresh rate.

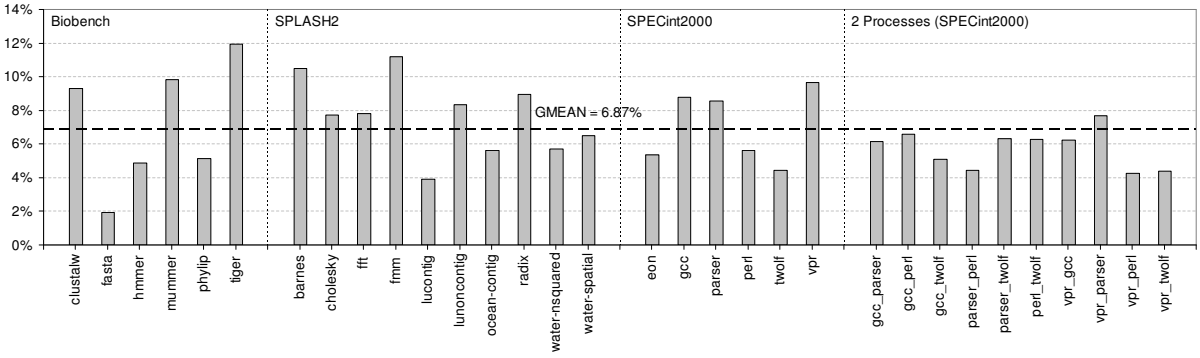



**Figure 17. Relative Total Energy Savings for a 64MB 3D DRAM Cache with 32ms refresh rate**

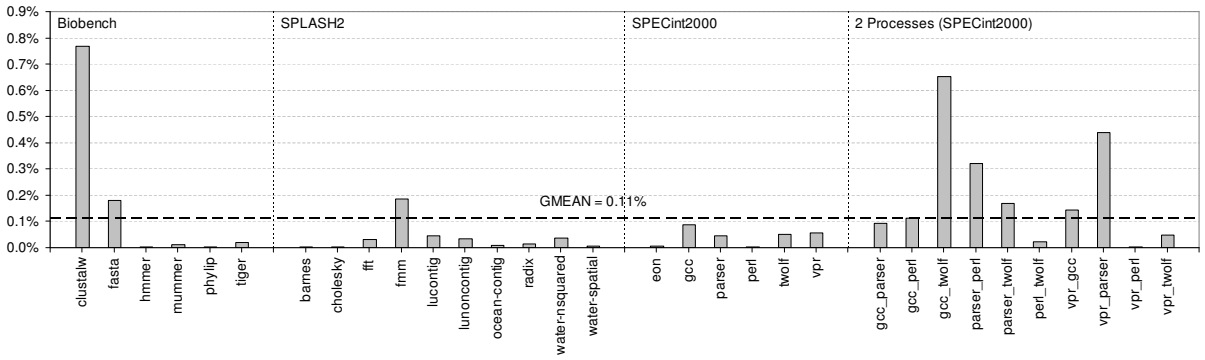



**Figure 18. Performance improvement using Smart Refresh for a 64MB 3D DRAM Cache with 32ms refresh rate**

As expected, the trend in number of refreshes is similar to the 64ms case, but the baseline number of refreshes is scaled up to twice the number of refreshes in the 64ms case. But since the number of accesses is constant, the number of refreshes eliminated is reduced. This is better illustrated in Figure 16, which shows the relative refresh energy savings using Smart Refresh for 3D DRAMs with 32ms refresh rate. Although the trends are similar to the 64ms case, the relative refresh energy savings is in general less than the 64ms case. The geometric mean of refresh energy savings is 15.79%. The total 3D DRAM cache energy savings is shown in Figure 17. We can see that even though the refresh energy savings are modest, we get a decent saving in total energy. The geometric mean of the total energy saved across benchmark suites is 6.87%. One reason for this is even though relatively refresh savings have reduced, refresh energy for the 32ms case accounts for a large part of the total energy. Thus the net energy savings for the 32ms is not much different than the 64ms case.

### 3.6.3 Performance Implication

While the objective of Smart Refresh is aimed at reducing the number of periodic refresh operations, it also shortens the potential memory access delays caused by these redundant refresh operations. Figure 18 shows the performance benefit of using Smart Refresh applied to a 3D DRAM (32ms refresh rate) over a conventional CBR refresh policy. It can be seen

that for all the benchmarks we have very slight (less than 1%) improvement in performance. For all the other DRAM and 3D DRAM configurations with Smart Refresh, we found very similar performance results. This shows that our Smart Refresh technique does not incur any performance degradation, but sometimes has performance improvement.

## 3.7 Summary

In this chapter we presented a simple, low cost technique using time-out counters to save power in DRAMs. This technique did not involve any change in the interface between the memory controller and the DRAM, making it highly feasible. All additional hardware went in the memory controller that controlled and issued the needed refresh operations. We demonstrated that many refresh transactions were indeed not needed because their corresponding rows were recently accessed due to cache misses. This technique saved up to 25% and on an average 12.13% of the energy consumed in DRAMs. Modern computing systems like CMP, CMT, SMP, and SMT would try to exploit MLP and would have increasing number of threads trying to access memory. In this case, the Smart Refresh technique will be instrumental in saving energy as it is very light weight and would increase the bandwidth availability and reduce energy consumption for refresh operations in DRAMs. The emerging 3D stacked ICs [16, 75, 97] will enable the accesses to the DRAM memory at a much lower latency. Also, AMD's licensing of ZRAM technology [5] indicate that future AMD processors may use DRAM type memory using SOI technology for their caches. This work clearly demonstrated that the Smart Refresh technique is very useful for such DRAM type caches. Energy savings of up to 21.5% and 9.4% on an average was obtained when Smart Refresh was used in a 3D DRAM Cache.

Apart from the DRAM, SRAM caches are a major consumer of the power. The subsequent four chapters describe different microarchitectural techniques for saving energy in SRAM caches. The next chapter explains a hybrid technique to save leakage energy in SRAM caches.

# CHAPTER 4

# REDUCING LEAKAGE ENERGY IN CACHES FOR MULTIPROCESSOR SYSTEMS

Owing to the continuing down-scaling of CMOS technology, the threshold voltages have become lower and the gate oxides are getting thinner, both resulting in a significant increase in leakage power. Meanwhile, with technology scaling, the capacity of single-chip processors has exceeded one billion transistors. To use such an immense amount of available transistors, processor architects tend to allocate more cache space and deepen the level of cache hierarchy. While these caches constitute a major portion of a processor's real estate, they are also the least active components and dominate the leakage power among all other architectural modules.

All prior architectural techniques that used Gated-Vdd [63], ignored the implications and correctness issues of maintaining Multi-Level Inclusion (MLI) [24] and cache coherence, voiding their applicability. Switching off an L2 cache line while keeping the same line in the L1 active could either violate the MLI property or complicate the snooping mechanism. With the industry making a paradigm shift to multicores or MPSoC, having a leakage power saving policy for cache-coherent shared-memory Multi-Processor(MP) systems is imperative.

In this chapter we propose a simple, low-cost and viable architectural technique called *Virtual-Exclusion* to reduce leakage energy consumption in the L2 caches. This technique aggressively reduces leakage energy in the L2 or higher level caches while maintaining Multi-Level Inclusion property and cache coherence simultaneously among multiple processors. Virtual-Exclusion is achieved by turning off repetitive but infrequently accessed cache lines in the higher level caches, given locality is already present in the lower level L1. For maintaining Multi-Level Inclusion, small modifications to the MOESI snooping bus coherence protocol are proposed to maintain correctness of the protocol when the power

saving feature is enabled. It does not need any additional hardware support other than the counters for switching off higher-level (e.g. L2) cache lines along with keeping the rest of the lines in the drowsy state. Additionally, Virtual-Exclusion reduces the extra misses from the L2 cache that are introduced by the original Cache-Decay scheme, thereby reducing both the performance penalty and dynamic energy consumption incurred by DRAM memory accesses. This ensures that the leakage energy savings is not offset by the much larger energy consumption of DRAM accesses. In addition, Virtual-Exclusion can be integrated with a conventional Cache-Decay scheme to obtain more leakage energy reduction. In this work, we provide a comprehensive analysis of the leakage energy reduction for a functioning implementation of a cache coherent multiprocessor system based on the Virtual-Exclusion technique. The contributions of this chapter are summarized as follows.

- We provide a viable and low-overhead solution for maintaining Multi-Level Inclusion and coherence for MP systems in the context of saving leakage energy.
- The technique needs only minor changes to traditional snoop-based cache protocols, e.g. MOESI.
- We apply our techniques to two MP architectures: Symmetric Multi-Processor (SMP) and the emerging multicore processors, and demonstrate the advantages.

## 4.1 Multi-Level Inclusion and Cache Coherence

In this section we overview the Multi-Level Inclusion property and describe the architectural policy changes required for having a leakage power management policies in the higher level cache (e.g. L2 or L3), while maintaining multi-level cache inclusion and coherence in a multiprocessor system.

### 4.1.1 Multi-Level Inclusion

A multi-level cache hierarchy consists of a number of levels of caches between the CPU and the main memory, with the lower level caches being closer to the CPU. Multi-Level

Inclusion (MLI), proposed by Baer and Wang in [24], is a property in a cache hierarchy which requires that if a cache line is present in a lower level cache (e.g. L1), it should also be present in all the higher levels (e.g. L2 and beyond). MLI is an important property for facilitating an efficient implementation of cache coherence. Using this property the higher level cache effectively shields the lower level cache from I/O and the snooping bus. Without MLI, the lower-level caches will encounter a large number of queries from the snooping bus. This could lead to substantial performance degradation due to the limited number of ports in small and highly accessed L1 caches.

The baseline cache hierarchy we use to demonstrate MLI in this work contains multiple cores, each with two-level caches communicating via a snooping bus. Each processor has a small L1 data cache, backed by a larger L2 cache, which is connected to the memory through the snooping bus. MOESI protocol is employed in this work to maintain cache coherence across processor cores.

The detailed algorithm and architectural support for maintaining inclusion in such a cache architecture is detailed in [24]. The second level cache needs to have an inclusion bit for every cache line to indicate whether the line is at the previous level. The following are the cache policy changes required to maintain MLI.

- For any line-fill in the L1 cache, the L2 cache sets the inclusion (I) bit for the corresponding line.
- For all evictions (Clean and Dirty) in the L1 cache, the line address is given to the L2 cache and the L2 cache resets the I bit of the corresponding line.
- All invalidation requests for cache lines in the snooping bus are propagated from the L2 cache to the L1. Both the L2 line and the L1 line are invalidated, ensuing necessary writebacks for dirty lines.
- For any write to a line at L1, the line is also marked dirty and written to the L2 cache and the L2 cache sends invalidation requests to the bus.

Each of the afore-mentioned changes to the protocol is an overhead in terms of cache bandwidth. They consume power and can affect performance due to cache contention. That is the reason why a snooping port is typically dedicated to caches [17, 105]. However, these enhancements are required for guaranteeing the correctness of cache coherency protocol and hence are assumed to be part of the baseline cache hierarchy in our work.

### 4.1.2 Leakage Energy Reduction Schemes for Coherent Caches

In a multi-level cache hierarchy, leakage energy reduction schemes are more appealing and profitable for higher level caches for several reasons. One reason is that higher level caches are much larger using most of the on-chip transistors, thereby consuming more leakage power that makes them good candidates for leakage power reduction. Moreover, given the high hit rates of the L1 caches, L2 or higher level caches are not frequently accessed, suggesting that they can stay idle for long periods of time.

Hu et al. [63] discussed briefly the effect of applying their policy on a cache-coherent multiprocessor system. Note that the first design priority of applying leakage power schemes to such systems is *correctness*. For their Cache-Decay technique, even if higher level cache lines can be turned off, it is imperative that the tags and the state of a turned off line must be kept active to maintain MLI and to shield the lower level caches (e.g. L1) from snooping traffic. Another issue not addressed and evaluated in their work is the serious performance degradation and additional power consumption by extra misses going to main memory due to the L2 decayed lines.

The drowsy cache [46] does not suffer from the correctness or MLI issues as it keeps the entire state in drowsy state. As such, the drowsy cache will not introduce additional misses as the Cache-Decay scheme. However, the issue is the increase in the latency for waking up a drowsy cache line. The performance degradation due to this is expected to be negligible since the L2 latency is typically in the order of tens of cycles. Nonetheless, the leakage power savings by using a drowsy cache is expected to be lower than the Cache-Decay scheme as all the drowsy lines still consume some leakage power. In this work we

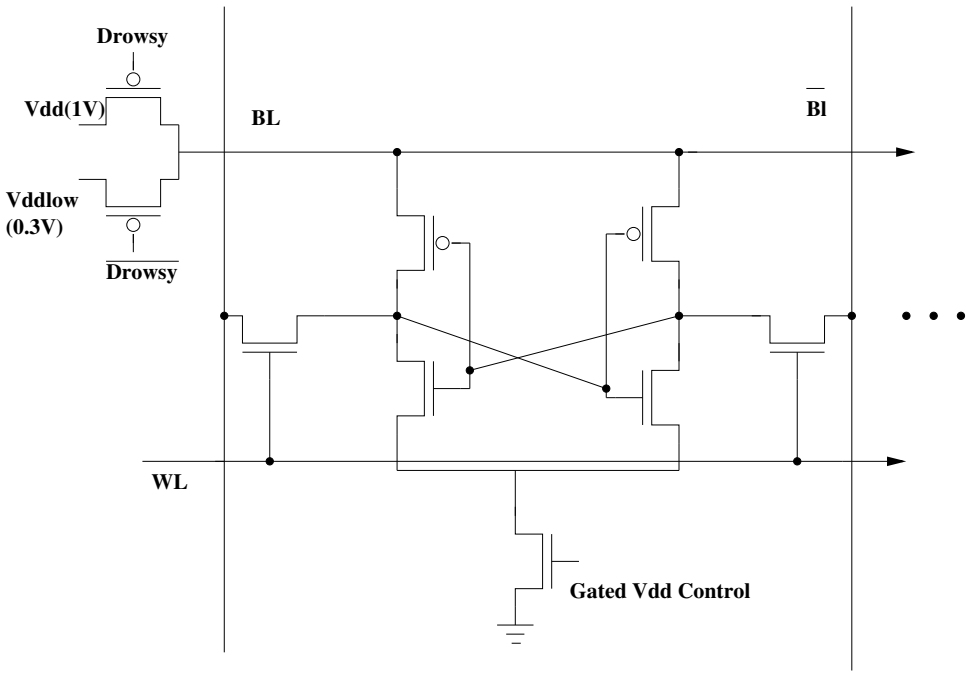



**Figure 19. SRAM cell with both Gated-Vdd and DVS control.**

assume a cache line circuitry where we can control the Vdd reaching the circuit as in [46] as well as gate off the supply voltage as shown in [96]. The schematic of such an SRAM cell is shown in Figure 19. Our proposed architecture technique will exploit this circuit to reduce leakage energy in caches. We discuss this technique in the next section.

## 4.2 Applying Virtual-Exclusion

In this section we explain why Cache-Decay fails to work with an MLI environment. We then describe the concept of Virtual-Exclusion and explain how it can be used to save leakage energy. Finally, we apply the Virtual-Exclusion concept to Cache-Decay and explain how can it help and improve cache performance over simple Cache-Decay and still saves more leakage energy.

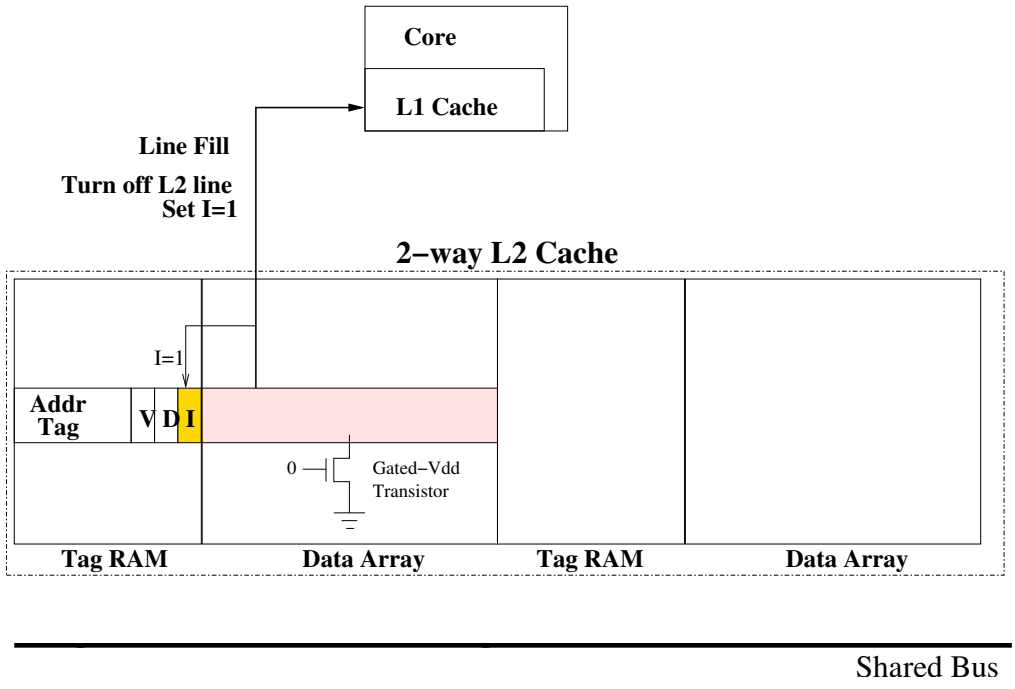
### 4.2.1 Generic Virtual-Exclusion Policy

The drowsy cache paper [46] shows that, for L2 or higher level caches the best and complexity-effective architectural strategy for leakage power control is to keep them in drowsy mode.

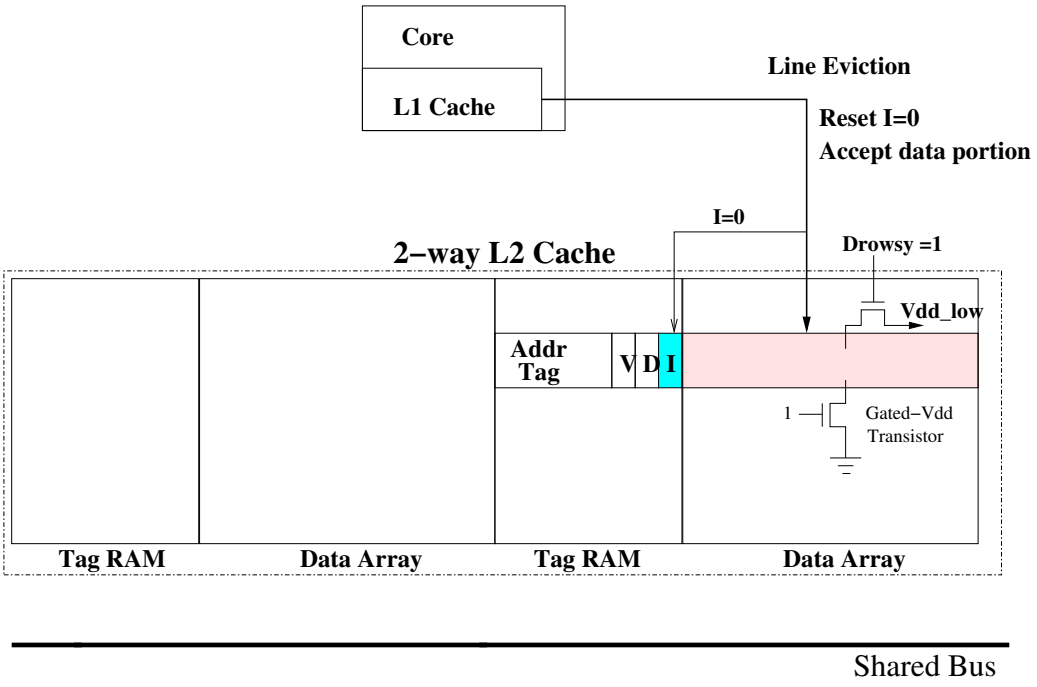
A specific data line is activated, or woken up only when it is accessed. Since the access latency for L2 caches is large, keeping the whole cache drowsy will not incur a large performance penalty as it just adds one or two cycles to the L2 access latency. Our entire L2 cache is initially assumed to be in the drowsy mode before the Virtual-Exclusion algorithm is applied.

The Virtual-Exclusion scheme is added on top of the drowsy cache scheme to allow more data lines to be turned off in the cache hierarchy for saving more leakage energy. To make drowsy higher level caches work with a cache-coherent MP system, it is important to note that the tag arrays of these higher level caches (i.e., L2 in our example) must be kept *on* all the time for supporting a functional cache coherence protocol. The schematic of a cache hierarchy using Virtual-Exclusion is depicted in Figure 20. Each entry in the Tag RAM of the L2 cache contains a physical address Tag (T), a Valid bit (V), a Dirty bit (D), an an Inclusion (I) bit. The state of the I bit indicates the presence of a line in the L1, so as to determine whether the data portion of the line in the L2 should be kept on in drowsy state or be Vdd-gated off. The first simple change for the Virtual-Exclusion scheme is the following. Whenever there is a line-fill into the L1 cache due to an L1 miss, the same line in the L2 cache (or the missed line brought back into the L2 from main memory) will have its corresponding I bit set. This I bit precisely indicates that the data is now present in the L1 cache as well. Subsequently, the corresponding data portion of the line in the L2 is $V_{dd}$ gated off immediately. This turn-off of the data lines in the L2 that are present in the L1 gives our technique its name. We illustrate this mechanism in Figure 20(a).

The L2 lines are turned back on under the following scenarios. Whenever a line is being displaced from the L1 due to a conflict miss, in order to explicitly maintain MLI, the L1 will inform the L2 and forward the line to the L2 for every single L1 line eviction regardless of whether the state of the evicted line is clean or dirty. Note that, since the Virtually Exclusive L2 cache does not have the data portion of a line when the line is present in the L1. Therefore, for each eviction, the L1 cache always supplies the cached, (un)modified




(a) L1 Line Fill.



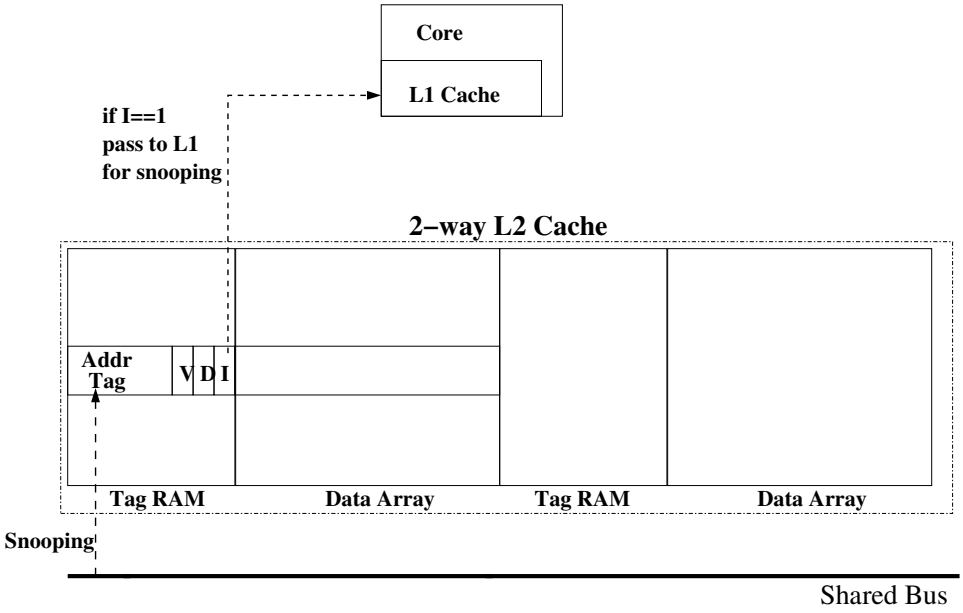

(b) L1 Line Eviction.

**Figure 20. Cache Line Allocation for Virtual-Exclusion.**

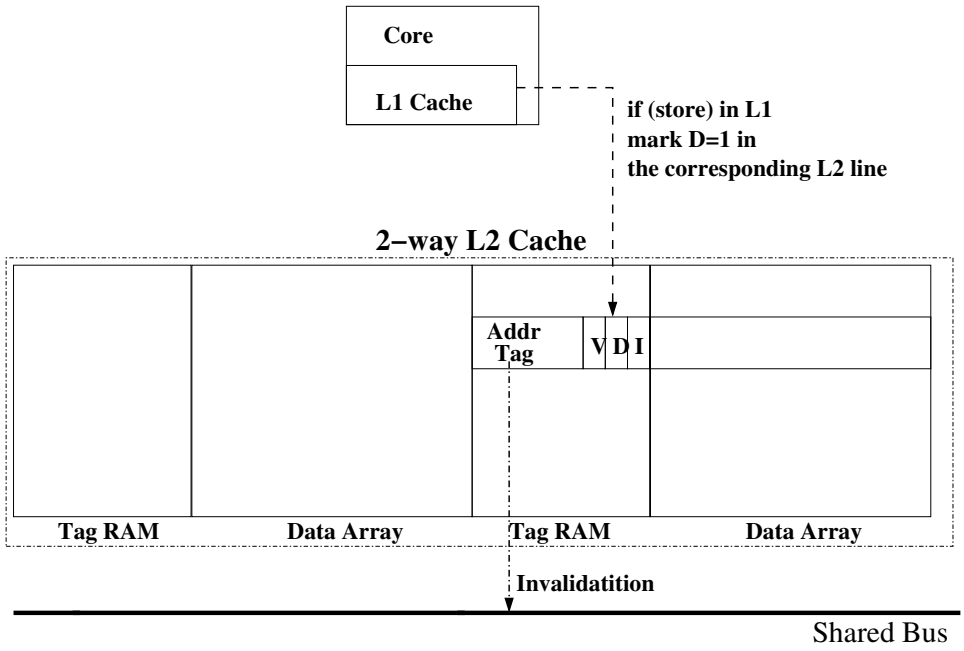
data portion along with its address to the L2. Upon such an eviction, the corresponding L2 cache line is turned back *on* to drowsy state and the I bit is reset to zero simultaneously as depicted in Figure 20(b). Such eviction is completely off the critical path. The only overhead is an increase in power consumption. This power overhead is accounted for in our leakage calculations. Also, it takes place only when there is an L1 miss, thus is unlikely to clobber L2 accesses and impact the performance. Another potential scenario to have data lines in L2 in drowsy state is for architecture that support instructions that prefetch data only into L2, e.g. prefetcht2 in Intel's SSE instruction set. This type of instructions bring data from main memory into L2, but not in the L1, and the data will be kept in drowsy mode for saving leakage energy until the processor makes requests for them.

In addition to the simple changes in the cache line fill and line eviction policy in the caches, some minor changes in the cache coherence protocol are also needed to maintain data consistency. First, any remote request for a cache line with an local L2 hit and its associated I bit set will cause the L2 to pass the request to the L1 cache. This is shown in Figure 21(a). Second, when a line in the L1 cache is being written, the address is provided to the L2 cache for marking the same line as *dirty* without changing any other state. Meanwhile, the L2 also needs to broadcast an invalidation signal for the address on the snooping bus. These operations are illustrated in Figure 21(b). In this way, the states of the same line in the L1 and L2 are kept consistent.

For the scenario depicted in Figure 21(a), considering a remote request in the MOESI protocol, our policy will increase latency if the tag of a line is present in the L2 cache and its I bit is set. In order to maintain correctness, we need a slight change in the protocol to handle this special case. If the requested line has its I bit set, then the only correct copy of the cache line must be present in the L1. Now any remote request pertaining to the line needs the line to be supplied to the bus. This can be done in two ways depending on the cache architecture. If the L1 cache has a direct path to the bus, then the data may be directly supplied by bypassing the L2. Otherwise, the data may be written back to the L2 cache,




(a) Incoming Snooping.




(b) Invalidation upon Write.

**Figure 21. Protocol Changes for Virtual-Exclusion.**

which in turn writes it to the bus. In our simulations, we assume that the line needs to be written to L2 and then to the bus.

While discussing Virtual-Exclusion, we should consider the situation where a line being replaced from the L1 Cache has already been replaced from the L2 Cache. Given the basis of our assumptions of an inclusive cache hierarchy, this situation should never take place. This is because for an inclusive cache the lines with their I bits set will not take part in cache replacement and hence will never be replaced while the line is in the L1.

It is noteworthy, that apart from keeping the L1 lines in L2 *turned off*, we have other opportunities to $V_{dd}$ gate off a few more cache lines. One obvious candidate for turn-off are the invalid lines. A line may become invalid if it is not being allocated or if it has been invalidated by remote snooping activity. These lines, including the tag arrays and data portion, can be safely turned off without incurring any performance loss. Once turned off, they will be turned back when a cache miss to the same locations occurs. Since this event involves an access to either main memory or remote caches, it could take some hundreds of cycles or more. Thus, an additional delay of a few cycles to turn a line on will incur a minute impact on performance.

### 4.2.2 Cache-Decay and Hybrid Virtual-Exclusion Policies

#### *4.2.2.1 Generic Cache-Decay in L2*

The original Cache-Decay scheme proposed in [63] does not address the correctness issue for a cache coherent multiprocessor system where Multi-Level Inclusion property needs to be enforced. The decay scheme turns off cache lines that are not used for a specified number of cycles based on the size of the decay counter employed. Since data will be lost when $V_{dd}$ gating is applied, if a line is allowed to decay in a higher level cache when having a copy in the lower level cache, it will violate the Multi-Level Inclusion property and cause the cache coherency protocol to fail. Here, we first discuss a minor change to the cache decay policy to enable Multi Level Inclusion.

To maintain the Multi Level Inclusion when Cache-Decay is applied, the tags of lines

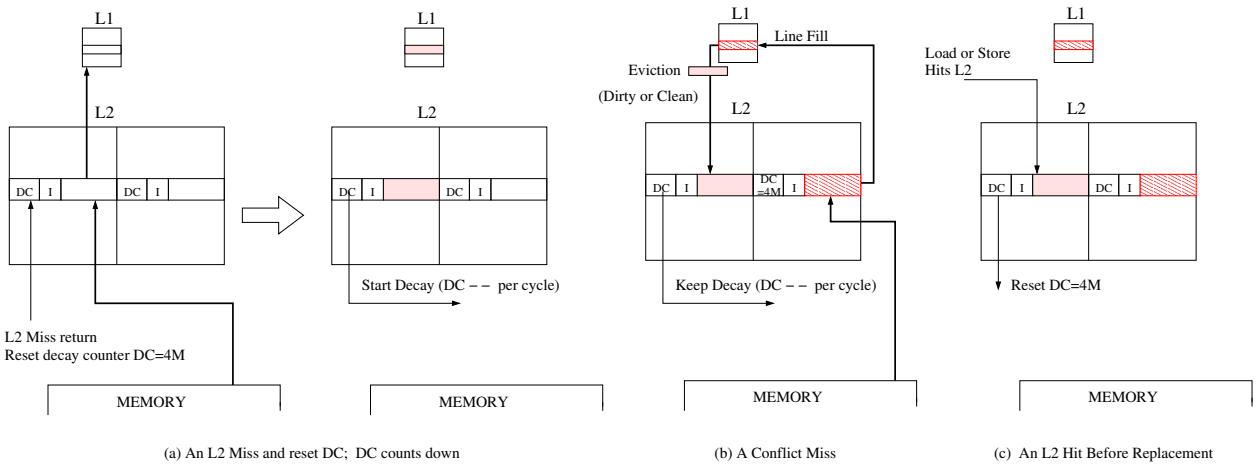



**Figure 22. Cache-Decay Countdown Mechanism.**

that have their I bit set need to be always turned on even if the decay counter indicates that the line can be decayed for not being used for cycles. In our experiments, we always charge up the tag arrays to the normal, high supply voltage even for the invalid lines. This policy potentially decreases the power savings compared to the originally proposed decay policy but is indispensable to preserve the requirement for the MLI and guarantees the correct functioning of the coherence protocol. Figure 22 shows the decaying mechanism. When a line is first brought into the L2 and L1, the corresponding L2 decay counter (DC) is reset to the maximum value, e.g. 4 million as shown in Figure 22(a). Similar to a normal decaying scheme, the DC starts down-counting also shown in Figure 22(a) when the corresponding L2 line is idle. Illustrated in Figure 22(c), when there is a conflict miss causing an eviction of the line, no matter it is clean or dirty, the L2 will keep down-counting. Typically, there is no action if the line was not updated during its lifetime in the L1, otherwise, it needs to be written back to the L2 if it is dirty. Nonetheless, the DC will be untouched in either scenarios. The DC will only be reset back to the maximum value when there is a request that generates an L2 hit. Figure 22(c) shows such a case.

#### *4.2.2.2 Hybrid Virtual-Exclusion Policy*

Now we discuss how to further improve the energy efficiency of the Cache-Decay scheme using our Virtually Exclusive cache architecture. To exploit the advantages of both Cache-Decay and Virtual-Exclusion schemes, we will be able to further reduce the leakage energy consumption. We call our new scheme *Hybrid Virtual-Exclusion* policy. There is a subtle caveat in the generic Cache-Decay scheme. The intuition behind Cache-Decay is that due to temporal locality a line not being used for a long time is unlikely to be used again any time soon. Based on the above, when applying decay technique to higher level L2 cache, lines in L2 is likely to decay when L1 is effective and exhibits high temporal locality. Additionally, to maintain Multi-Level Exclusion in such scenarios, the tag arrays of the L2 cannot be completely gated off for snooping reasons even if the decay counter is already counted down to zero. In other words, so long as the I bit in the L2 is set, decaying (e.g. $V_{dd}$ gate-off) will be disabled for the L2 address tags. According to our Virtual-Exclusion mechanism discussed in Section 4.2.1, when a line is evicted from the L1, the I bit of the same line in the L2 will be reset and the line from the L1 will be copied to its data portion. Upon this point, the decay counter will start counting down.

It is noteworthy to point out that the L2 lines with I bit set do not decay as shown in Figure 23(a), they only start decaying when the I bit is reset in Figure 23(b). Namely, decaying of the L2 lines starts only after they are evicted from the L1, the difference between the hybrid and the decay scheme in the previous section. Note that, for any L2 hit, similar to the generic decay scheme, the decay counter will be reset back to the maximum value. From the above discussion, the decay counter, when starting to count, will always (re)start from the maximum value because (1) the decaying only starts after an eviction due to replacement; (2) any prior L1 line fill, either hit in L2 or miss the first time, must have the decay counter reset back to the maximum value.

Having our Virtual-Exclusion policy applied to the Cache-Decay has the following advantages.

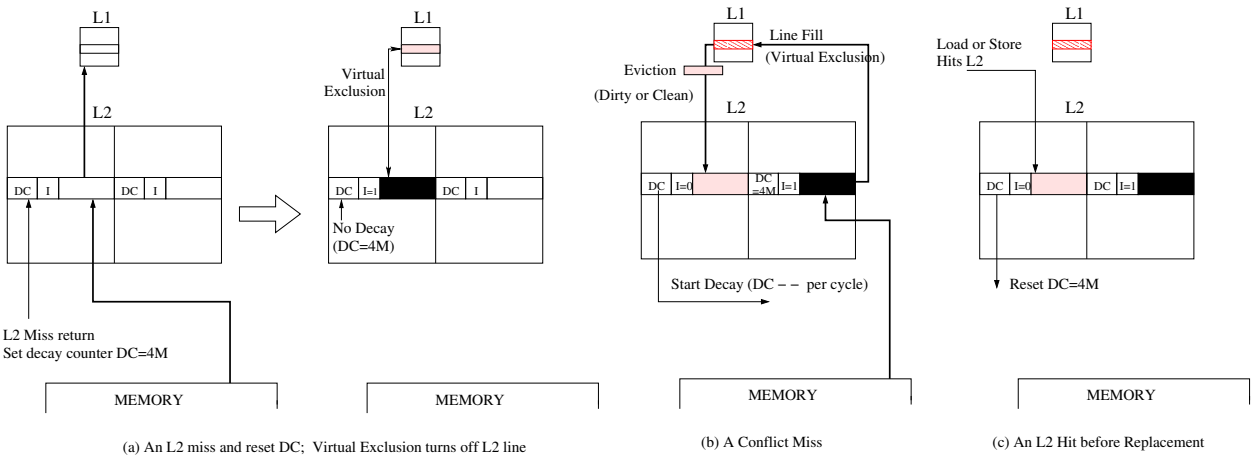



**Figure 23. Hybrid Virtual-Exclusion Countdown Mechanism.**

- It reduces leakage consumption by turning off data portion of lines in the L2 that are in the L1. It does not wait until the line finally decays. In prior work, there was unnecessary leakage current consumption during the 4 million cycles the decay counter is counting down.
- For inclusive lines, the decay counters start counting only when the corresponding L1 line is evicted. This gives us a decaying victim cache, reducing the possibility of decay-induced L2 misses. Reducing a few L2 Cache misses is extremely important, because a L2 cache miss causes a memory access, which in turn consumes more energy in the DRAM and suffer from additional latency of some hundreds of cycles.

Our technique is extremely simple and it only uses the state bits, the dirty bit and the inclusion bit to determine whether to switch a cache line off. Since these state bits are already present in the cache for the purpose of maintaining coherence and inclusion, the only major area overhead will be to maintain different voltage power supplies and the simple cache line driver circuitry. There is also an extra overhead of decay counters that are also present in the original Cache-Decay scheme.

### 4.2.3 Virtual-Exclusion in Multicore Processors

In addition to a traditional multiprocessor system, the Virtual-Exclusion technique can also be applied to the emerging multicore architectures. A modern multicore processor consists of a number of processors sharing a large L2 cache. This L2 cache may be simply a single monolithic structure or may be non-uniformly distributed among processor cores with some type of interconnection network that guarantees coherency [38, 67, 57]. Similarly, in a multicore architecture, the Inclusion bit will be set if any of the L1 caches has a copy of the line. The concept of Virtual-Exclusion is the same as it is in the case of an SMP architecture explained earlier; any line with its "I" bit set will have its data array part switched "off". Since the "I" bit being set guarantees that the line is present in some L1 Cache, any other cache requesting the data may get it through a cache-to-cache transfer. As explained previously, we also apply the decay scheme on top of our Virtual-Exclusion scheme to obtain energy benefits. In multicore type structures, a large number of cores can share the L2 cache. Therefore, more L2 lines will be inclusive in several, distinct L1 caches, thus we have a greater leakage saving opportunity for Virtual-Exclusion — leaving a larger number of L2 data portions to be $V_{dd}$ gated off. Also, due to a larger number of processor cores, the number of accesses to the L2 cache will be greater, too, making decaying lines by a conventional Cache-Decay mechanism more difficult. Our results show that using decay with Virtual-Exclusion in a multicore lead to up to 72% savings in L2 cache leakage power over a baseline drowsy L2 Cache.

## 4.3 Simulation Framework for Virtual-Exclusion

Our experiments were based on the M5 simulator system developed by the University of Michigan [26]. M5 is capable of performing a system level simulation for a snooping bus multiprocessor system. Our baseline architectural parameters along with various cache sizes are listed in Table 4. The processor is chosen to be in-order to be in tune with the

some latest trend in Multicore processors that have multiple cores of simple in-order processors on a chip with an on-chip L2 Cache. The aim of these Multicore architectures is to increase throughput through TLP. An example is the Ultra SPARC T1 (Niagara) processor that contains 8 in-order processor cores on the die [15]. The power estimation tool used for estimating leakage power is based on ECacti [78]. We integrated the ECacti leakage power model into the M5 simulator to analyze both dynamic and leakage power consumptions in caches. The DRAM access energy is estimated from the data sheets of commercial DRAMs offered by Micron [1]. All the simulations are performed on the SPLASH-2 benchmark suite [120] and SPEC CPU2000 Integer benchmark programs. To evaluate the dynamic cost of using the same counters in [63] and the modified bitline and wordline driver circuitry in [46, 63], we use the energy overhead numbers supplied in these papers and scaled down to 70nm technology using conventional technology scaling rules [80].

The simulations were carried out on two types of architectures: multicore processors and SMPs. In the multicore architecture, we simulate using six configurations. These configurations consist of an L2 cache size of either 256 KB or 512 KB, being shared by 2, 4 and 8 processor cores, respectively. For the SMP architecture, each processor contains their own L1 and L2 caches. We simulate the above two L2 cache sizes, with 2, 4, and 8 processors running on a shared bus. We implement and evaluate three distinct energy management policies for the L2 caches. The policies are:

- **Decay**: Cache-Decay policy implemented to work for Multi-Level Inclusion. The decaying policy was detailed in Section 4.2.2.1.
- **Virtual-Exclusion**: Generic Virtual-Exclusion policy described in Section 4.2.1 for the L2 cache.
- **Hybrid**: Virtual-Exclusion implemented on top of the cache decay. The policy was discussed in Section 4.2.2.2 with illustrations.

For all the above configurations we ran the SPLASH-2 benchmarks to completion. We

**Table 4. Architectural Parameters**

| | |
|---|---|
| Processor Core | In-order, stalls on cache misses |
| L1 D Cache Size | 16KB 2-way 64-byte line |
| L1 I Cache Size | 16KB 2-way 64-byte line |
| L1 Access Time | 1 cycle |
| L2 Cache Sizes | 256KB 8-way and 512KB 8-way |
| L2 Access Time | 10 cycles Normal, 12 cycles Drowsy |
| Memory Access Time | 200 cycles |

**Table 5. Spec2000 Benchmark used for simulations**

| | |
|---|---|
| 2-way Multicore | bzip and gzip |
| 4-way Multicore | bzip, gzip, crafty and gap |
| 8-way Multicore | 2 copies each of bzip, gzip, crafty and gap |

also simulated a set of simulations for the multicore architecture that involves running heterogeneous SPEC benchmark programs on different processor cores in a multicore system. These simulations were aimed to analyze the effect of heterogeneous applications running on multiple cores that contain no data sharing. All the SPEC2000 INT benchmark programs were run for 1 billion instructions. The exact SPEC2000int programs used in our simulations are given in Table 5. The reason we did not show all the results is that not all the SPEC2000int programs were successfully ported to M5 simulation framework due to various issues such as unimplemented system calls.

## 4.4 Energy Savings with Virtual-Exclusion

In this section we evaluate our techniques by running SPLASH-2 benchmarks for both SMP and multicore architectures. All results shown are relative savings in the leakage energy over a baseline drowsy cache. All savings numbers take into account the energy consumption overhead. The overhead is different for different cache policies, this discussion encompasses all overheads considered in our analysis. We consider the overhead for the extra circuitry required for maintaining Gated-$V_{dd}$ scheme, the energy consumed for the extra misses by DRAM memory accesses and finally for Virtual Exclusion, the overhead

of bringing a line from L1 on a bus read request, and also writing clean values from L1 to L2 during evictions.

### 4.4.1 SMP Analysis

Figure 24 illustrates the energy savings for a dual processor SMP system and each processor has a 256 KB L2 cache. The percentage reduction calculation is based on the baseline leakage energy (the denominator). The numerator considers both the leakage energy of each scheme and the dynamic energy overhead caused by extra trips to the DRAM memory. The rationale is to evaluate how much energy can be saved with these architectural leakage-reduction techniques. We observe that in almost all the benchmark programs the hybrid scheme shows the best saving results. The reason is that the Cache-Decay scheme incurs a lot of overhead for the extra L2 misses that consume additional DRAM memory energy. This overhead is effectively eliminated by the hybrid scheme because the hybrid scheme transforms a portion of the L2 cache into a decaying victim cache. Also note that the decay scheme for some programs failed to save energy. This happens for the same reason — the large memory access overheads caused by L2 misses. Our simple Virtual-Exclusion scheme does not suffer from such overheads. But since the L1 cache size is a small percentage of the L2, Virtual-Exclusion alone gives an average of 8% leakage energy savings. By combining with decay in our hybrid scheme, the average savings are increased to 20%. Also there is never a case where the hybrid scheme actually encounters energy loss. For FFT and Lu-Contig from SPLASH-2, the pure decay scheme does better than the hybrid scheme. This happens because the Virtual-Exclusion scheme turns "on" the lines that are evicted from the L1 cache. On the other hand, in the decay scheme, the line might have been decayed in the L2 already, therefore, some benchmark programs show better energy savings for the decay scheme. However, as seen from the results, this policy of having a decaying victim cache is useful in reducing L2 misses and its ensuing memory access overheads, if the replacement is transient.

Figure 25 plots the average L2 leakage energy savings for 2, 4, and 8 processor systems

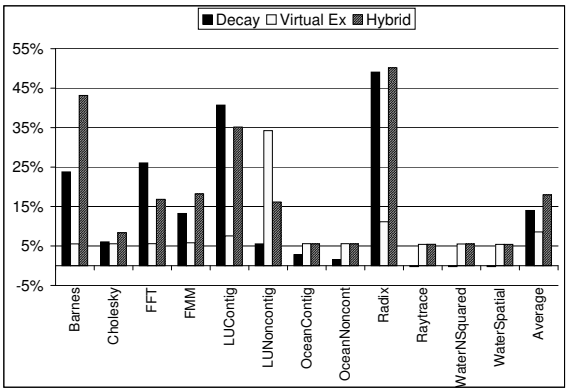



**Figure 24. Leakage Energy Reduction for 2-way SMP (256KB L2).**

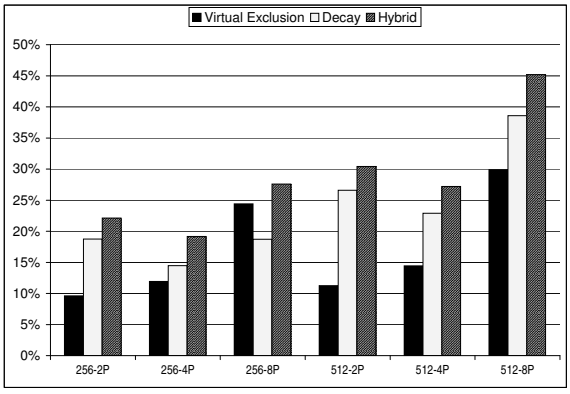



**Figure 25. Average Leakage Energy Reduction for Different SMP Configurations.**

for the entire SPLASH-2 benchmark suite. The average across all the applications clearly reveals that the hybrid method is the best among all techniques. Another obvious trend from the graph is that the leakage energy savings increase with increased cache size. This is because for a given data working set, the larger the cache, the higher likelihood of decaying a line. In overall, the hybrid scheme saves from 19% to as much as 45% of leakage energy consumption of an L2.

### 4.4.2 Multicore Processors Analysis

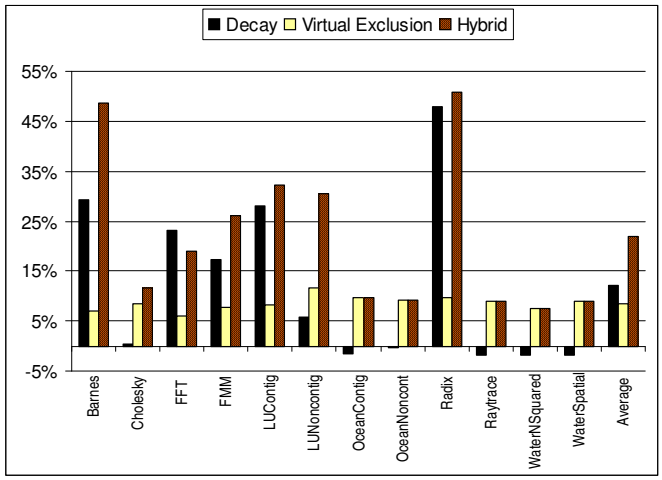
Now, we show the energy results for multicore processors in Figure 26. Using the same metric, we compare the leakage energy savings for each of the three techniques, *Decay*, *Virtual-Exclusion* and *Hybrid* in each figure. Figure 26(a) shows the savings for different SPLASH-2 benchmark programs for a 2-way multicore system with a 16KB L1 data and instruction cache in each processor and a 256KB L2 cache shared by the two processor cores. We can see that the leakage energy savings highly depend on the benchmark characteristics. Similar to the observations made in the SMP analysis, we find that the Cache Decay technique sometimes led to energy loss for more DRAM accesses. The Virtual-Exclusion technique provides around 10% savings across all the benchmark programs. The hybrid technique obtains the best savings of L2 leakage energy, up to 52% in Radix. Neither the Virtual-Exclusion nor the Hybrid technique ever shows any negative savings results.

We further studied the leakage energy savings for a 4- and 8-core system using SPLASH-2. The results in Figure 26(b) and Figure 26(c) demonstrated similar trends to a 2-core system. In fact, as the number of processor cores increases in a multicore system, the relative leakage power savings using the hybrid scheme also increases compared to the decay scheme. This is because as the number of processors increases, the occupancy and activity in the L2 by different workloads also increases. This reduces the opportunity for the generic decay scheme to decay lines.

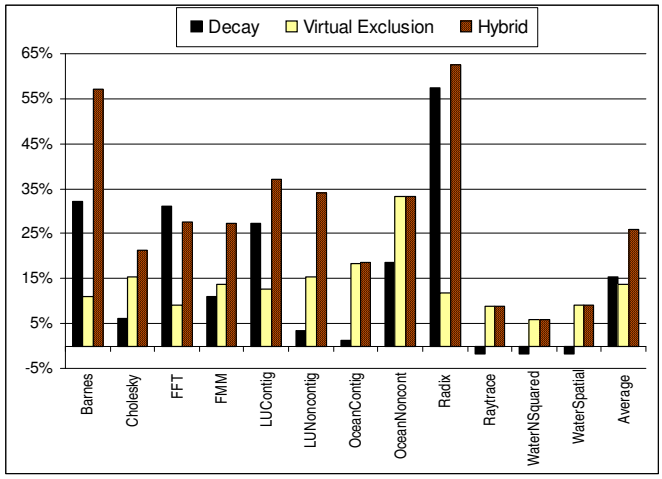
Figure 27 shows leakage energy savings for systems where different SPEC benchmark programs run on different processor cores. The purpose of this experiment is aimed to study the energy impact for heterogeneous applications running on a multicore system, a more realistic scenario for multiple independent single-threaded applications are concurrently executing. Note that, these applications have their respective address spaces. The combinations of SPEC2000int programs for different cores on a 2-, 4- and 8-core system are detailed in Table 5. As mentioned earlier, we subset the results simply because some SPECint programs have not been successfully ported to the M5 simulator yet. We observe that the hybrid scheme provides the best average savings (9%) for all the benchmark programs and configurations we simulated. Unlike the decay scheme, neither the Virtual Exclusion nor the Hybrid scheme ever consumes more energy (i.e., negative savings) than the baseline. As the number of processor cores keeps increasing in future generations of multicore processors, our scheme will become more effective in addressing the leakage issues.

### 4.4.3 Performance Impact

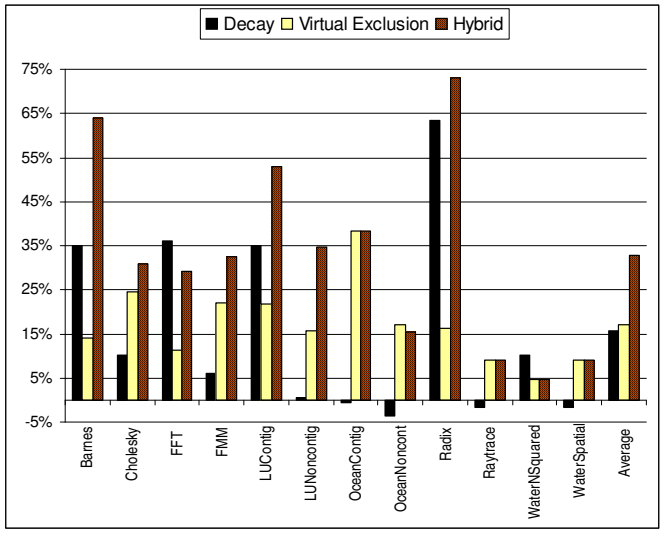
Compared to the baseline MP system with drowsy L2 caches, the performance of our Virtual-Exclusion will mostly be on par. Note that during a snoop hit, the baseline system requires extra cycles to wake up the drowsy lines. On the other hand, the Virtual Exclusion needs to perform an L1 lookup for retrieving the most up-to-date data if the snoop-hit line in the L2 is turned off. In other words, both schemes suffer similar performance overheads.




(a) 2-way Multicore Processor.




(b) 4-way Multicore Processor.




(c) 8-way Multicore Processor.

**Figure 26. Leakage Energy Reduction for Multicore Processors (256KB L2).**

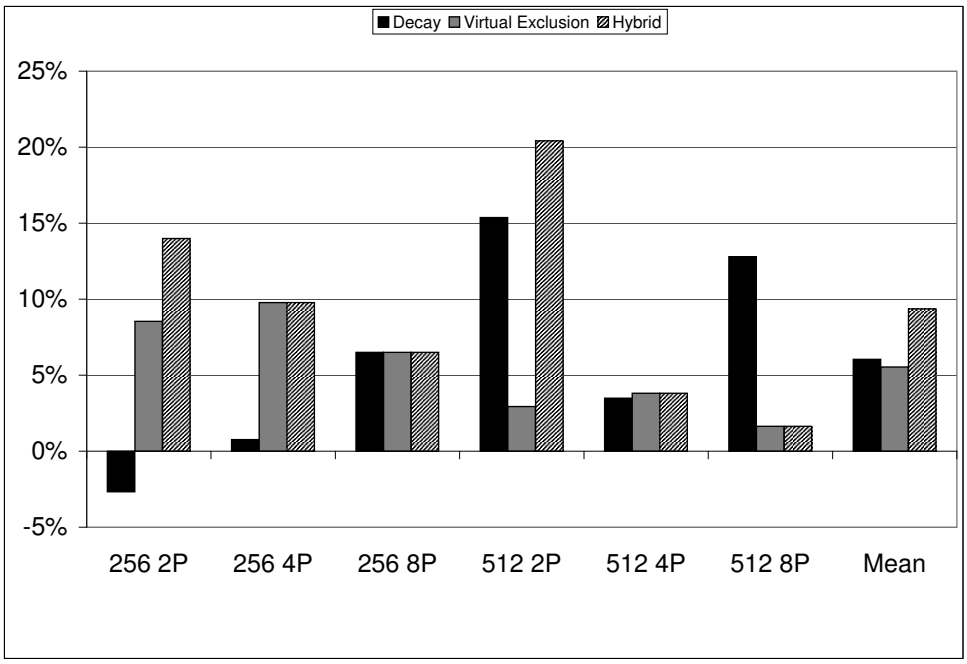



**Figure 27. Leakage Energy Reduction for Multicore Systems (SPEC2000 Integer Benchmarks).**

According to our simulation results, the performance differences between our scheme and the baseline are within the noise range (below 0.00001%) — almost negligible. Therefore, we do not report the performance results in this work.

## 4.5 Summary

Multiprocessor or multicore systems are the current design trend in all processor market segments. All these designs use multiple levels of large on-chip caches, in which leakage control in caches will become highly critical for several looming issues — power management, thermal control, and circuit reliability. However, existing leakage energy saving techniques in multiprocessor systems are limited in scope because cache coherency maintenance for correctness is often neglected in these previously proposed low-power architectural designs. In this chapter, we presented a new, low-overhead architectural technique called Virtual-Exclusion to save leakage energy in higher level caches that simultaneously provided guaranteed Multi-Level Inclusion property for correct operations of cache coherence protocols and saved leakage energy more effectively. Our technique showed that a

significant leakage energy savings of up to 46% in an 8-processor SMP and 35% for an 8-way multicore architecture can be achieved. We envision that such a practical and easy-to-implement technique will be very useful in saving leakage energy for the cache-coherent multicore, multiprocessor systems.

The techniques explained in the previous three chapters exploited different forms of redundancies in the memory hierarchy to reduce energy consumption. For DRAMs, redundant refresh operations were eliminated to reduce DRAM power. In a cache, redundancies were identified in the data storage, the Multi-Level Inclusion policy, and the snooping protocol to reduce leakage energy. The subsequent techniques, in contrast, add a new hardware structure called the "counting Bloom filter"(CBF) to the memory hierarchy. This is the main emphasis of the authors research. The following two chapters describe and evaluate three different applications of the counting Bloom filters in reducing cache energy.

# CHAPTER 5

# USING BLOOM FILTERS FOR EARLY MISS DETECTION AND WAY ESTIMATION

## 5.1 Energy Management for ever larger caches

The increasing complexity and shrinking feature size of modern microprocessors has caused energy consumption to become a critical design constraint [87]. At the same time, also due to shrinking feature size, processor designers are given more transistors for a given die budget at their disposal, leading to large caches with multiple read/write ports. Caches have become a major consumer of both static and dynamic energy in microprocessors. Two general trends have been observed in the microprocessor industry that motivates the use of energy saving techniques described in this chapter. The first trend is the move towards multicore processors leading to simpler cores. The other significant trend is the use of larger caches with increasingly higher associativity.

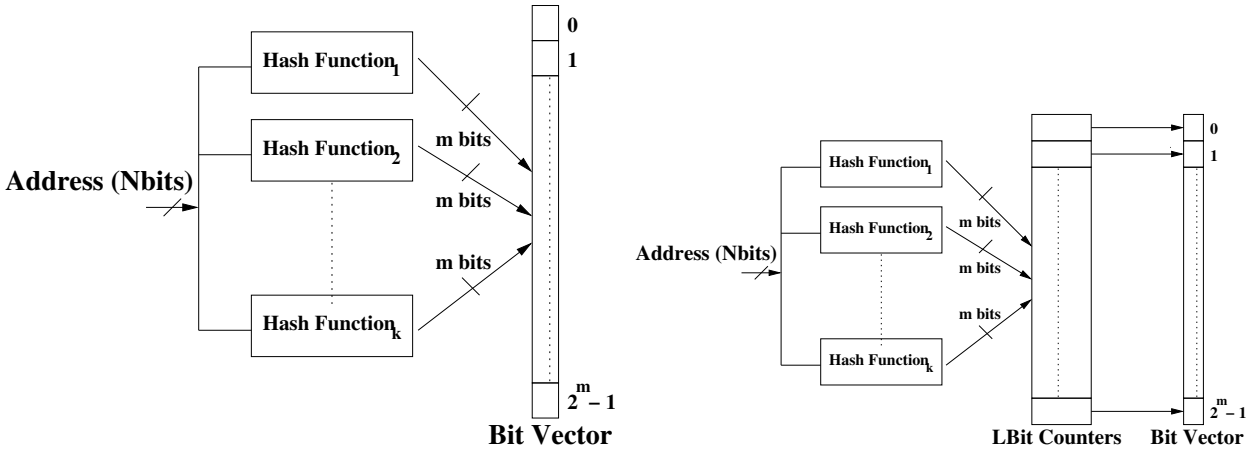
The memory hierarchy of modern processors typically consists of single or multi-level caches implemented with SRAM cells backed up by a large DRAM. With the general trend of the microprocessor industry moving towards multicore processors, each individual core is becoming simpler to focus on parallelism exploitation in exchange for the reduction of the insurmountable design complexity and verification effort. As such, processor architects can incorporate many cores sharing a common, large last-level cache on a single die. A case in point is the UltraSPARC T1 processor [110] that contains eight in-order cores with the support of fine-grained multithreading. In such simplistic cores running at high frequency, a DRAM memory access can be expensive, taking hundreds of cycles. Therefore, significant stalls may occur upon each cache miss. Processors such as UltraSPARC T1 exploit this property by selecting an alternate thread for execution in the event of a stall. On the other hand, such cache miss events can also be used as a trigger for several microarchitectural energy management processes in the processor. The energy management processes may

include but are not limited to putting all caches in a state preserving low power drowsy mode and/or clock-gating or power-gating all or part of the processor core.

Another trend along with having large caches is increasingly higher associativity. For example, the AMD's K8 processor [102] employs a 16-way L2 cache. Some embedded processor such as the Intel XScale [60] implements a 32-way L1 cache. Processors employing highly associative caches will consume an even larger amount of energy on every lookup. N tag comparisons are needed for an N-way cache. Depending on the implementation, the data may need to be retrieved from all N ways before the hit cache line can be "muxed" out. As can be seen, most of the energy consumed for a lookup in a set-associative cache is redundant. Since for a hit, the data can only be present in one particular way. This redundancy provides a good opportunity for saving dynamic energy.

In this chapter, by applying counting Bloom filters, we propose two different techniques to exploit the energy saving opportunities explained above. Bloom filters are simple, fast structures that can eliminate the need of performing associative lookup especially when the lookup address space is huge. They can replace the expensive set associative tag matching with a simple bit vector that can precisely identify addresses that have not been observed before. This mechanism provides early detection of events without resorting to the associative lookup buffers. This leads to significant improvements in energy consumption without adversely affecting performance, considering the fact that Bloom filters are very efficient hardware structures in terms of area, energy consumption and speed.

The first technique previously described in [52] presents an innovative segmented design of the counting Bloom filter that saves energy by detecting a miss in the cache level before the memory. The detection of the miss happens much earlier than the actual request reaches the particular cache level. This early detection allows the processor to respond and initiate the energy management processes in advance in the memory hierarchy. Starting energy saving measures early provides more energy saving opportunities than in the case where the measures are taken after a miss in the lowest cache level is detected.




(a) Original Bloom Filter

(b) Counting Bloom Filter

**Figure 28. Bloom Filters**

For the second technique, we present *Way Guard*, an efficient method for estimating ways using counting Bloom filters in a set-associative cache. Way estimation saves significant amount of unnecessary energy dissipation by reducing lookups going into redundant ways when a set-associative cache is accessed.

## 5.2 Bloom Filters

The structure of the original Bloom filter concept [29] is shown in Figure 28(a). It consists of several hash functions and a bit vector. A given $N$-bit address is hashed into $k$ hash values using $k$ different random hash functions. The output of each hash function is an $m$-bit index value that addresses the Bloom filter bit vector of $2^m$ entries, where $m$ is much smaller than $N$.

Each element of the Bloom filter bit vector contains only 1 bit. Initially, the Bloom filter bit vector is zero. Whenever an $N$-bit address is observed, it is hashed to the bit vector and the bit value hashed by each $m$-bit index is set to one. When a query is to be made whether a given $N$-bit address has been observed before, the $N$-bit address is hashed using the same hash functions and the bit values are read from the locations indexed by the $m$-bit hash values. If at least one of the bit values is 0, it implies that this address has definitely

not been observed before. This is called a *true miss*. On the other hand, if all of the bit values are 1, the address may have been observed if all of the bit values are 1 but there is no guarantee that the address has actually been observed. Despite a Bloom Filter indicating a hit, the event of an address not being observed is called a *false hit*.

As the number of hash functions increases, the Bloom filter bit vector is polluted much faster. On the other hand, the probability of finding a zero during a query increases if more hash functions are used. The major drawback of the original Bloom filter is the high false hit rate because the filter can be quickly filled up with all 1's and start signaling false hits.

To reduce the false hit rate, the original Bloom filter has to be very large. Note that once a bit is set in the filter there is no way to reset it. Thus, as more bits are set in the filter, the number of false hits increase. To improve performance of the original Bloom filter, a mechanism for resetting entries containing one is needed. The counting Bloom filter as shown in Figure 28(b) proposed by *Fan et al.* in [45] for web cache sharing provides such capability of resetting entries in the filter. For a counting Bloom Filter, an array of counters is added along with the bit vector of the classical Bloom Filter. Each L-bit counter has a one-to-one association with each bit in the bit vector. Queries to a counting Bloom filter are similar to the original Bloom filter. The slight modification is the following: when an address is entered into the Bloom filter, each $m$-bit hash index will increment its corresponding counter of the counter array in addition to setting the corresponding bit in the bit vector. Similarly, when an address is deleted from the Bloom filter, each $m$-bit hash index will decrement its corresponding counter. If more than one hash index addresses to the same location for a given address, the counter is incremented or decremented only once. Finally, when a counter is reduced to zero, its associated bit in the bit vector will be reset.

## 5.3 Segmented Bloom Filter Design

We propose an innovative segmented counting Bloom filter as shown in Figure 29 where the counter array of $L$ bits per counter is decoupled from the bit vector and the same hash

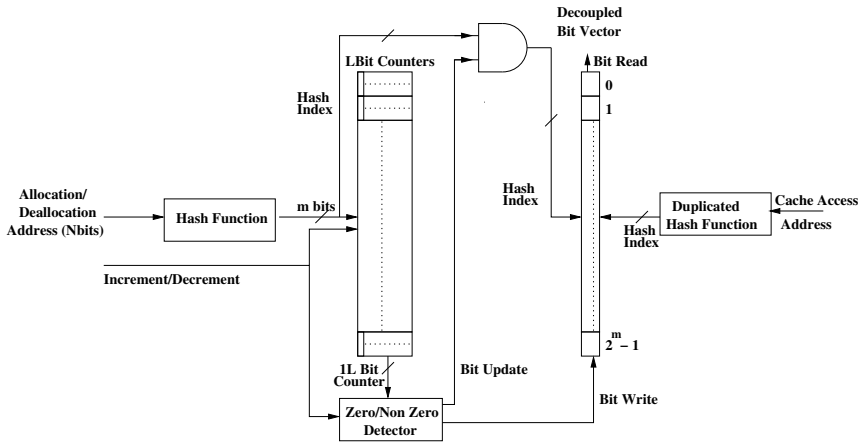



**Figure 29. Segmented Bloom filter**

function is duplicated on the bit vector side. The cache line allocation/de-allocation addresses are sent to the counter array using one hash function while the cache request address from the processor is sent to the bit vector using a copy of the same hash function. The segmented Bloom filter design allows the counter array and bit vector to be in separate physical locations.

A single duplicated hash function is sufficient, as we found in our experiments that the filtering rate of a Bloom filter with two or more hash functions was only slightly better than a single hash function. For the rest of this chapter, we assume only a single hash function is used. The implemented hash function divides a physical address into several chunks of hash index long and bitwise XOR them to obtain a single hash index. The number of bits needed per counter ($L$) depends on how the hash function distributes the indices across the Bloom filter. To deal with the worst case scenario where all cache lines happen to map to the same counter, the bit-width of the counter must be $log_2^{(Num\ of\ Cache\ Lines)}$ to prevent counter overflow. In reality, the required number of bits per counter is much smaller than the worst-case.

The counter array is updated with each cache line allocation and de-allocation operation. Whenever a new cache line is allocated, the address of the allocated line is hashed into the counter array and the associated counter is incremented. Similarly, when a cache

line is evicted from the cache, its associated counter is decremented.

The counter array is responsible for keeping the bit vector up-to-date. The update from the counter array to the bit vector is done only for a single bit location if and only if the counter becomes zero from one during a decrement operation or one from zero during an increment operation. The following are the steps taken for updating the bit vector:

1) The *L*-bit counter value is read from the counter array prior to an increment or decrement operation.

2) The counter value is checked for a zero boundary condition by the zero/nonzero detector to see whether it will become non-zero from zero or zero from non-zero inferred by the increment/decrement line.

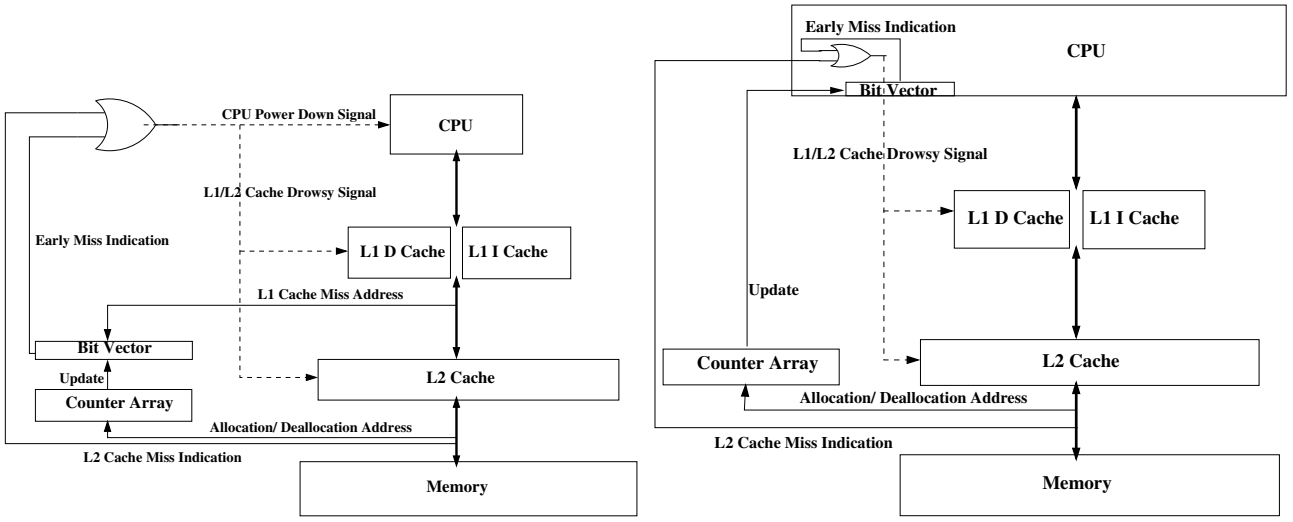
3) If a zero boundary condition is detected, the *Bit Update* signal is asserted, which forwards the hash index to the bit vector.

4) Finally, the *Bit Write* signal is made 1 to set the bit vector location if the counter will become non-zero. Otherwise, the bit write line is made 0 to reset the bit vector location.

5) If there is no zero boundary condition, then the *Bit Update* signal is de-asserted, which disables the hash index forwarding to the bit vector.

When the processor issues a lookup in the cache, the cache address is also sent to the bit vector through the duplicated hash function. The hash function generates an index and reads the single bit value from the vector. If the value is 0, this is a safe indication that this address has never been observed before. If it is 1, it is an indefinite response, i.e., the access can be either a miss or a hit.

There are several reasons for designing a segmented Bloom filter: 1) We only need the bit vector, whose size is smaller than the counter, to know the outcome of a query to the Bloom filter. Decoupling the bit vector enables faster and lower energy accesses to the Bloom Filter. Hence the result of a query issued from the core can be obtained by just looking at the bit vector. 2) The update to the counters is not time-critical with respect to the core. So, the segmented design allows the counter array to run at a much lower frequency




(a) Processor with caches not assumed to be inclusive and the bit vector below the L1 cache

(b) Processor with inclusive caches and the bit-vector inside the CPU

**Figure 30. Bloom Filter Design with Inclusion Property**

than the bit vector. The vector part being smaller provides a fast access time, whereas the larger counter part runs at a lower frequency to save energy. The only additional overhead of our segmented design is the duplication of the hash function hardware. 3) The decoupled bit vector can sit in-between the L1 and L2 caches or can also be integrated into the core. For systems where the L1 and L2 caches are inclusive, the integrated bit vector can also filter out accesses to both the L1 instruction and data caches if an L2 cache miss is detected. This will increase the L1 access time by one cycle and has been modeled in our experiments.

We must note here that the energy savings obtained by powering down the processor and L1 and L2 caches is only possible if the core is a simple inorder processor with blocking caches. This assumption is true for a many embedded processors. Powering down the core is also possible in many-core processors with simple in-order cores like Sun's Ultrasparc T1 and T2 processors. However, superscalar processors cannot be stalled because of a cache miss. Therefore, the savings from our technique will only be the dynamic energy for not accessing the L2 cache on a Bloom filter predicted miss.

## 5.4 Processor Energy Management with Segmented Counting Bloom Filters

This section explains how the segmented Bloom Filter detects L2 cache misses and saves the overall system energy without losing performance in an in-order processor. In an in-order processor with two cache levels, severe stalls may occur for an L2 cache miss leading of an off-chip DRAM memory access. Depending on the processor-memory frequency ratio, a DRAM access may take hundreds of cycles to complete.

By detecting an L2 cache read miss early with a segmented Bloom filter, we can save static energy of the system by turning off all or part of the core and by putting the L1 and L2 caches into drowsy or low-power state-preserving mode until the data returns. The overhead incurred by this technique is turning on and turning off of the core and the caches. This overhead is not much of a concern because the turn-off period overlaps with the memory access, which may take hundreds of cycles. Also, since it is known exactly when the data returns from memory, the turned-off units can be turned on in stages progressively to save energy. In addition to reducing static energy, dynamic energy of the system can also be reduced by preventing an L2 cache access. Not only does this save the dynamic energy of the L2 but also reduces the bus energy consumption due to reduction in bus switching activity.

The segmented Bloom filter is shown in Figure 30(a) for an processor in which the L1 and L2 caches are not inclusive. In such a system, the bit vector is located just below the L1 caches. The processor issues a cache address to the L1 data cache. On a miss, the bit vector snoops the address and signals in a cycle if the L2 cache does not have the cache line. Upon receiving the signal, the CPU is powered down and the L1 I and D and L2 caches can be put into the drowsy mode. The access to the L2 cache is also called off.

Figure 30(b) shows a system where the L2 cache is inclusive with the L1 caches. Here, the bit vector is placed inside the core and can detect L2 cache misses before they are sent to the L1 caches. In a cache system maintaining inclusion property, an L2 miss also indicates

a miss in the L1 cache. Thus, a cache request address can be sent directly to the memory when a miss is detected by the bit vector inside the core.

For both systems, the bit vector may not be 100% consistent with the counter array as there is some delay occurring between updating the bit vector from the counter array. This situation happens if incrementing the counter in the counter array is deferred till the time of a linefill. At that moment, the corresponding bit location in the bit vector might be 0. So, if the counter changes from 0 to 1, the counter array sends an update to the bit vector to set the bit location in the vector. Before this update reaches the bit vector, if the processor accesses the same bit location, then it reads 0 and assumes that this line is not in the cache and therefore forwards the request to memory. This discrepancy is eliminated if the counter is incremented at the time of the miss rather than the linefill. By the time the actual linefill occurs, the bit vector will have been updated by the counter array.

We see that segmenting the Bloom Filter allows the bit vector to be placed in a different physical location leading to more energy saving opportunities. This concept may be extended to cases where there are more than two levels of caches and the segmented Bloom filter is used to filter out requests to the cache that is accessed just before the DRAM memory. In such a case, though the counter array would be updated for the cache before memory, the bit vector may be kept at a place where it would be accessible with any of the previous cache levels, thus providing early miss indication. In the following section we propose another application of the counting Bloom filters in way estimation for set-associative caches.

## 5.5 Way Guard Mechanism

In this section, we describe a novel application of counting Bloom filters to set-associative caches to determine data presence and save lookup energy. The basic idea illustrated in Figure 31 shows the design of a 4-way set-associative cache with our proposed *Way Guard* mechanism. As shown, each Way Guard, structurally the same as the counting Bloom

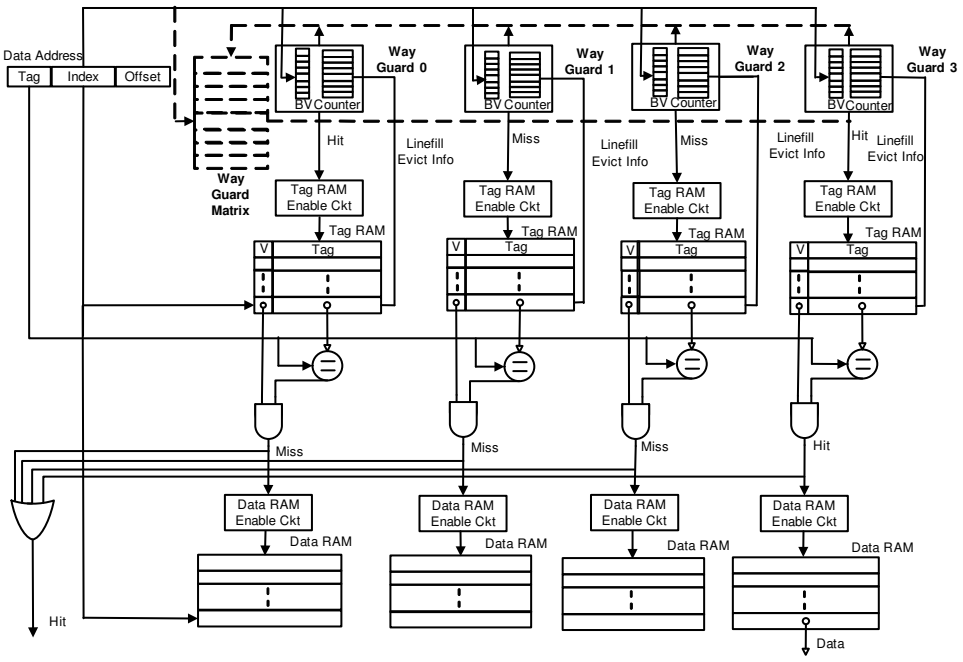



**Figure 31. Way Guard Mechanism — Filtering Out Unnecessary Cache Way Lookup**

Filter, consists of a Bit Vector (shown as BV) and an array of counters. Each way of a set-associative cache is assigned one Way Guard. The purpose and functionality of these filters are very similar to the Segmented Bloom Filters explained earlier. The only difference is that each Way Guard keeps linefill and replacement information of the cache way it is guarding. The following two properties are important in understanding how the Way Guard technique works:

1. If the filter indicates that the data address is not present in the way it is guarding, then the data is certainly not present in that particular cache way.

2. If the filter indicates that the data address is present, then the data may be present in the given cache way.

As such, the filter provides a completely safe indication about the absence of data in the cache way it is guarding. Also, this indication can be performed within a fixed access time, as opposed to previously proposed prediction techniques that contain undesirable variable access times. Figure 31 shows an example scenario of an access to a *Way Guard cache*.

As illustrated in the figure, since the Way Guard filters of Way 0 and Way 3 indicate a possible hit, only their Tag RAMs need to be checked. Thus, for every cache access we only need to check a subset of Tag RAMs which enables the dynamic energy reduction. Even though the scheme incurs extra hardware, the Way Guard only comprises of a bit-vector and counters. Querying a Way Guard only involves checking the bit vector and this consumes much less energy than looking for the address in the Tag RAM it is guarding. Note that, since the filter must be checked prior to the Tag RAMs being selected, there is a potential performance penalty. However, since the filter is fast, the filter access and the Tag RAM selection process can be potentially contained within a cycle. In the worst case, the filter would add one extra cycle to the cache access time.

One implementation variant of the *Way Guard* technique is illustrated by dotted lines in Figure 31. As explained in the previous section the Way Guard are essentially segmented Bloom filters and consist of a bit vector and an array of counters. There will be *"n"* Guards each guarding a way of an n-way cache. However, note that each of these filters is indexed by the same hash function. Given a data address, all the filters will check at the same indices for the presence of the data. Since for all cache accesses all the filters are queried, we propose the following design alternative. In this variant, the vector part of the segmented Bloom filters are coalesced to form the *Way Guard Matrix* shown by the dotted line table. Each row in the matrix contains a bit for each guard filter. This bit will be the bit in the bit vector corresponding to the index of the row. Thus Matrix[i][k] will consist of the $i^{th}$ row of the bit vector of the Way Guard guarding the $k^{th}$ way. During a cache access, this matrix is first queried as shown by the dotted arrows coming from the address and the row of bits obtained for the given index of the address is used to enable only the cache ways that may contain the data. This technique can further reduce energy by using the lower energy matrix structure to filter out unneeded cache lookups going to the Tag RAMs. The bit vectors which consist of a very large number of one-bit entries[1], and the majority of the

[1]The number of entries we chose is four times of the number of cache lines. For a 256KB cache with 32-byte lines, the number of entries in the array is 32768.

access cost to the bit vector structure goes in decoding the index. Using one matrix instead of *n* one bit arrays saved *(n-1)* decoder access costs for every access to the Way Guard.

One notable point about our technique is that it can be extended to other implementations of highly associative caches such as CAM tag caches [123]. This can be done by simply adding an AND gate in the path of every CAM comparator. One input to these AND gates is obtained from the Way Guard result to effectively reduce tag comparisons. The detailed evaluation is outside the scope of this dissertation.

## 5.6 Experimental Results

### 5.6.1 Experimental Framework and Benchmarks

For choosing an experimental framework, we wanted an infrastructure that allowed simulation of a large number of cache configurations at once. Also, we wanted to show the effect of running applications on top of an OS for the way estimation simulations. Therefore, to evaluate the energy savings for both cache miss prediction and way estimation we use a modified version of the Bochs [61]. Bochs is a full system simulator that can boot x86-based operating systems such as Windows NT, XP, and RedHat Linux. For cache miss prediction, the simulation involves running SPEC, Mediabench and MiBench applications on RedHat Linux in Bochs. For way estimation, the simulations further involved running common Windows applications on a Windows NT platform. To collect cache statistics, we integrate the cache simulator from Simplescalar [23] into Bochs. We also enhance the cache simulator with our Bloom filter model. This innovative simulation technique allows us to gather cache statistics of various applications running on top of a full operating system. Previous techniques using Bochs, collected instruction execution traces from Bochs, that were then fed into a microarchitectural simulator like Simplescalar. By integrating the cache simulator into Bochs we remove the time-consuming and cumbersome process of trace collection from the simulation process. This technique is much faster than trace collection and we may simulate a lot more instructions than trace collection techniques which is inherently limited by the size of the trace. Using this technique we can test several

**Table 6. Architectural assumption**

| |
|---|
| Drowsy-mode in/out time = 10 cycles |
| CPU clock gating time = 8 cycles |
| Shutdown Penalty = 16 cycles |
| Bit vector access time = 1 cycle |
| Memory access time = 100 cycles |
| CPU Energy = 2 x L1 Cache Energy |
| Cache Drowsy Energy = 1/6 x Cache Leakage Energy |

memory hierarchies simultaneously.

For cache miss prediction experiments, we model a common embedded processor that executes instructions in program order with blocking caches. We compute the total energy consumption of the on-chip system including the processor, the caches and the Bloom filters. Our baseline model is a system with no Bloom filter. We evaluated twelve applications including *bzip2, gcc, gzip, mcf, parser, vortex* and *vpr* from *SPECint2000*, the *lame* MP3 player application from *MiBench* [56] and *adpcm , epic, jpeg, mesa and pegwit* from *Mediabench* [73]. 4 billion instructions after fast forwarding 1 billion instructions are simulated in the SPECint benchmark programs while *lame* and the Mediabench benchmarks run to completion. Since our technique involves predicting misses in the L2 cache, for proper analysis, we needed benchmarks that stressed the L2 cache. The SPECint, Mibench and Mediabench benchmark programs were chosen, for their stressing L2 cache behavior.

Other pertinent architectural assumptions are listed in Table 6. The following assumptions are made to estimate the energy consumption of the baseline system (i.e., system without the Bloom filter) and a low-power system with the segmented Bloom filter. The time taken to put the caches in drowsy mode is 10 cycles, and it also takes another 10 cycles to resume back to the normal mode. As reported by [46], it takes less than 2 cycles to put a cache line into drowsy mode. Nevertheless, since cache is a large structure, putting the whole cache to a drowsy state simultaneously could lead to high-frequency inductive noise issues [85]. Therefore, we assume that the entire operation is done in five progressive phases, taking a total of 10 cycles. Similarly, turning the cache from the drowsy back to

**Table 7. Architectural Configuration**

| Description | Configuration 1 | Configuration 2 |
|---|---|---|
| L1 I and D cache | 2-way 16KB | 2-way 64KB |
| L2 cache | 4-way 64KB unified | 4-way 256KB unified |
| Bit vector size | 8192 bits | 32768 bits |
| Counter array size | 8192 3-bit counters | 32768 3-bit counters |
| L1 latency (cycles) | 1 | 4 |
| L2 latency (cycles) | 10 | 30 |

normal state also takes 10 cycles in our simulations. We also assume that the time taken to aggresively clock-gate microarchitectural blocks in a processor is 8 cycles. This number is obtained from the clock gating time of the ARM Cortex A8 processor [6]. In our simulations it was assumed that the core that is aggresively clock gated consumes 40% of the actual core energy. This is a reasonable approximation based on the power consumption reported for the C1/C2 states in the Silverthorne processor [48].

The total time for turning clock gating on and off (16 cycles) is called the *shutdown penalty*. This is estimated from the typical cycles taken to aggresively clock gate ARM's Cortex A8 processor and also to wake up from its sleep state. The access time to the bit vector takes one cycle while the memory access time is assumed 100 cycles. We also assume that the processor energy consumption is twice the total L1 instruction and data cache energy consumption which is a realistic assumption shown in some embedded processors [44]. The cache leakage energy in the drowsy mode was estimated one sixth of the cache leakage energy as shown by *Flautner et al.* in [46].

We experiment two different cache architectures as shown in Table 7. The first configuration contains 2-way set-associative 16KB L1 instruction and data caches, 4-way 64KB L2 cache, a 8192-bit Bloom filter bit vector and a Bloom filter counter array of 8192 entries with 3-bit counter per entry. Although the worst case number of bits required per counter is 12, we observe in our experiments that the value of each counter never exceeds 4. Thus we use 3 bits per counter to save energy and have a policy of disabling a particular counter if it saturates. The line size is 32B for both L1 and the L2 caches. The latencies of the

L1 instruction and data caches and L2 cache are 1 and 10 cycles, respectively. This configuration represents low-end market such as industrial and automotive applications in the embedded domain.

The second configuration includes 2-way set-associative 64KB L1 instruction and data caches, a 4-way 256KB L2 cache, each has a 32B line size. The Bloom filter consists of a 32768-bit bit vector and a counter array of 32768 entries with a 3-bit counter per entry. The latencies of the L1 instruction and data caches and the L2 cache are 4 and 30 cycles, respectively. This configuration represents the high-end-market where slightly larger scale applications are targeted, e.g. consumer and wireless applications.

We have chosen the number of Bloom Filter entries to be four times the number of cache lines. We experimented with different BF sizes and found this empirical ratio provides the best results. The area overhead for the Bloom Filters is about 6% of the L2 cache area for both configurations.

The experimental evaluation for "Way Guard" is performed in two stages. In the first stage, only the L2 cache is guarded by the Way Guard filters. So, we used a fixed size of 2-way 16KB L1 caches, and 30 different configurations of the L2 cache. We varied the cache capacity by gradually doubling its size from 64KB up to 2MB. The associativity was varied in the same manner from 2 to 32 ways.

The size of each Way Guard filter was chosen to be four times the number of lines of each cache way. The area overhead for all Way Guards in the L2 Cache is 6% of the L2 cache size. Notice that this 6% relative overhead is irrespective of the cache size, as we always use the heuristic of choosing the Way Guard filter with entries that is four times the number of entries of the number of cache lines it is guarding.[2]

We show energy savings results for both the serial access and parallel access versions of the L2 cache. In a serial access, cache data access follows the tag access only when there is a tag match. In contrast, for a parallel access cache, the data and tag of all ways are

[2]There is also a small overhead of a few logic gates per Way Guard, but the overhead is negligible compared to the size of the filter.

enabled in parallel, and the correct data is "muxed" out.

In the second stage, only L1 caches are guarded by the Way Guard. So, we used a 4-way 128KB fixed sized L2 cache, and 20 L1 configurations. Similarly, the L1 capacity was varied from 8KB up to 64KB and their associativity from 2 to 32 ways. The 32-way L1 is similar to what is employed in Xscale processors. We assume a parallel access L1 cache for all our experiments. We use the a set of seven SPEC benchmarks that were known to stress the L2 cache. In addition, we also used five MS Windows applications used in desktop systems including the booting of Windows NT, Visual Studio compiling the Bochs source code, an MPEG decoder, a MP3 decoder, and a simple web browsing application. All the above applications show sufficient amount of memory activity to properly illustrate our results. Also, the MS Windows benchmarks help us understand how the Way Guard technique will behave in a real multiprocessing environment.

### 5.6.2 Energy Modeling

The L1 and L2 caches, the bit vector and the counter array were designed using the *Artisan* 90nm SRAM library in order to get an estimate for the dynamic and static energy consumption of the caches and the segmented Bloom filter. The Artisan SRAM generator is capable of generating synthesizable Verilog code for SRAMs in 90nm technology. The generated datasheet provides the read and write current of the generated SRAM. This gives us an estimate of the dynamic energy per access of such a structure. The datasheet also provides a standby current from which we can calculate the leakage energy per cycle of the SRAM.

We have two system energy models. The first model is the baseline model in which the dynamic and static energy consumption of the processor, L1 instruction and data caches and the L2 cache are calculated. The second system model is the low-energy system model in which the dynamic and static energy consumptions of the bit vector and counter array are also added to the rest of the system components. Table 8 lists the abbreviation of the variables we will use to formulate the system energy of the baseline and the low-energy

system models.

#### 5.6.2.1 Baseline System Energy Model

$$
\begin{aligned}
Cyc_{off} &= Num_{L2readmiss} * (Lat_{mem} - SP) \\
Cyc_{on} &= Cyc_{tot} - Cyc_{off} \\
E_{cpu}^{base} &= Cyc_{on} * CPU_{dyn} + Cyc_{off} * CPU_{leak} \\
E_{\$}^{base}(type) &= Num_{cacheaccess} * \$_{dyn} + Cyc_{on} * \$_{leak} + Cyc_{off} * \$_{dr} \\
E_{sys}^{base} &= E_{cpu}^{base} + E_{\$}^{base}(I) + E_{\$}^{base}(D) + E_{\$}^{base}(L2)
\end{aligned}
$$

#### 5.6.2.2 Low-energy System Energy Model

We now estimate the energy consumption of the low-energy system model having L1 and L2 caches which are designed with inclusion property with the segmented Bloom filter as follows:

$$
\begin{aligned}
Cyc_{off} &= Num_{L2readmiss} * (Lat_{mem} - SP) + Num_{L2filt} * (Lat_{L2} - Lat_{vector}) \\
Cyc_{on} &= Cyc_{tot} - Cyc_{off} \\
E_{cpu}^{low} &= Cyc_{on} * CPU_{dyn} + Cyc_{off} * CPU_{leak} \\
E_{L2}^{low} &= (Num_{L2access} - Num_{L2filt}) * L2_{dyn} + Cyc_{on} * L2_{leak} + Cyc_{off} * L2_{dr} \\
E_{L1}^{low}(type) &= Num_{L1access} * L1_{dyn} + Cyc_{on} * L1_{leak} + Cyc_{off} * L1_{dr} \\
E_{vector}^{low} &= Num_{L2access} * BV_{dyn} + Cyc_{on} * BV_{leak} + Cyc_{off} * BV_{dr} \\
E_{counter}^{low} &= Num_{L2access} * Counter_{dyn} + Cyc_{on} * Counter_{leak} + Cyc_{off} * Counter_{dr} \\
E_{sys}^{low} &= E_{cpu} + E_{L1}^{low}(I) + E_{L1}^{low}(D) + E_{L2}^{low} + E_{vector}^{low} + E_{counter}^{low}
\end{aligned}
$$

If the L1 and L2 caches are inclusive, then the energy consumption of the L1 cache is determined by the total number of L1 accesses less the number of filtered L2 misses. Also, the number of L2 accesses is replaced by the number of L1 accesses in the bit vector energy equation.

$$\begin{aligned} E_{L1}^{low}(type) &= (Num_{L1access} - Num_{L2filt}) * L1_{dyn} + Cyc_{on} * L1_{leak} + Cyc_{off} * L1_{dr} \\ E_{vector}^{low} &= Num_{L1access} * BV_{dyn} + Cyc_{on} * BV_{leak} + Cyc_{off} * BV_{dr} \end{aligned}$$

For measuring Way Guard energy we divide the L2 cache energy into energy for accessing the tags of one way and that for accessing the data. So for a Way Guard in a parallel access cache:

$$\begin{aligned} E_{L2}^{low} &= (Num_{WayHit} * Num_{L2Hits} + Num_{WayMisses} * Num_{L2Misses}) * (E_{Dyn}^{L2Tag} + E_{Dyn}^{L2Data}) \\ &+ Num_{L2access} * E_{Dyn}^{BV} + (Num_{L2Linefill} + Num_{L2Replace}) * E_{Dyn}^{Counter} \\ &+ Num_{BVUpdate} * E_{Dyn}^{BV} + Cycles * (E_{leak}^{L2} + E_{leak}^{BV} + E_{leak}^{Counter}) \end{aligned}$$

For a serial access cache:

$$\begin{aligned} E_{L2}^{low} &= (Num_{WayHit}^{L2} * Num_{L2Hits} + Num_{WayMisses}^{L2} * Num_{L2Misses}) * E_{Dyn}^{L2Tag} + Num_{L2Hits} \\ &* E_{Dyn}^{L2Data} + Num_{L2access} * E_{Dyn}^{BV} + (Num_{L2Linefill} + Num_{L2Replace}) * E_{Dyn}^{Counter} \\ &+ Num_{BVUpdate}^{L2} * E_{Dyn}^{BV} + Cyc_{tot} * (E_{leak}^{L2} + E_{leak}^{BV} + E_{leak}^{Counter}) \end{aligned}$$

Finally, the percentage savings in the total system (Dynamic + Leakage) energy is defined by the following equation:

$$\% \text{ Savings} = \frac{E_{sys}^{base} - E_{sys}^{low}}{E_{sys}^{base}} \quad (1)$$

### 5.6.3 Cache and Bloom Filter Statistics

For studying the efficacy of early cache miss detection using our segmented Bloom filter, we collected the cache miss rates for the L1 instruction and data caches and L2 cache and the miss filtering rates of the Bloom filter for the two configurations listed in Table 9 and Table 10. Some bechmark programs such as adpcm, gcc, or mesa show higher L2 miss rates in the larger L2 cache configuration. This is because their L1 miss rates were much improved, resulting in fewer accesses to the L2 cache. The miss filtering rates in the last column of the tables are the percentage of the L2 misses that the Bloom filter can detect correctly (i.e., true miss rate). For instance, 86% of the L2 misses can be detected correctly by the Bloom filter in *parser* for Configuration 1. The remaining 14% of them cannot be detected, i.e., the false hit rate. The average true miss rates across all benchmarks are 80% and 81% for both configurations. These rates imply that a great majority of the L2 misses can be captured by the Bloom filter. An 80% filtering of L2 misses also implies that the Bloom filter can reduce accesses to the L2 cache by more than 30%.

### 5.6.4 Energy Consumption Results for Early Cache Miss Detection

Table 11 shows the L2 dynamic energy savings for the two configurations with respect to the L2 cache in the baseline model. Most benchmarks like *gzip*, *jpeg*, *vortex* and *mcf* suffer a drop in the L2 dynamic energy savings in Configuration 2 because of improvements in the L2 miss rates by employing a much larger L2 cache. As the L2 miss rate improves, the number of misses of which the Bloom filter can take advantage to shut down the processor and caches diminishes. The reduction in the L2 energy savings in Configuration 2 are more dramatic in *pegwit*, *vortex* and *vpr* since their miss rates drop significantly in Configuration 2.

In summary, using the segmented Bloom filter provides an average of 34% and 31% savings in the L2 dynamic energy respectively for the two configurations.

Figure 32 plots the static energy savings. The static energy accounts for the leakage energy of

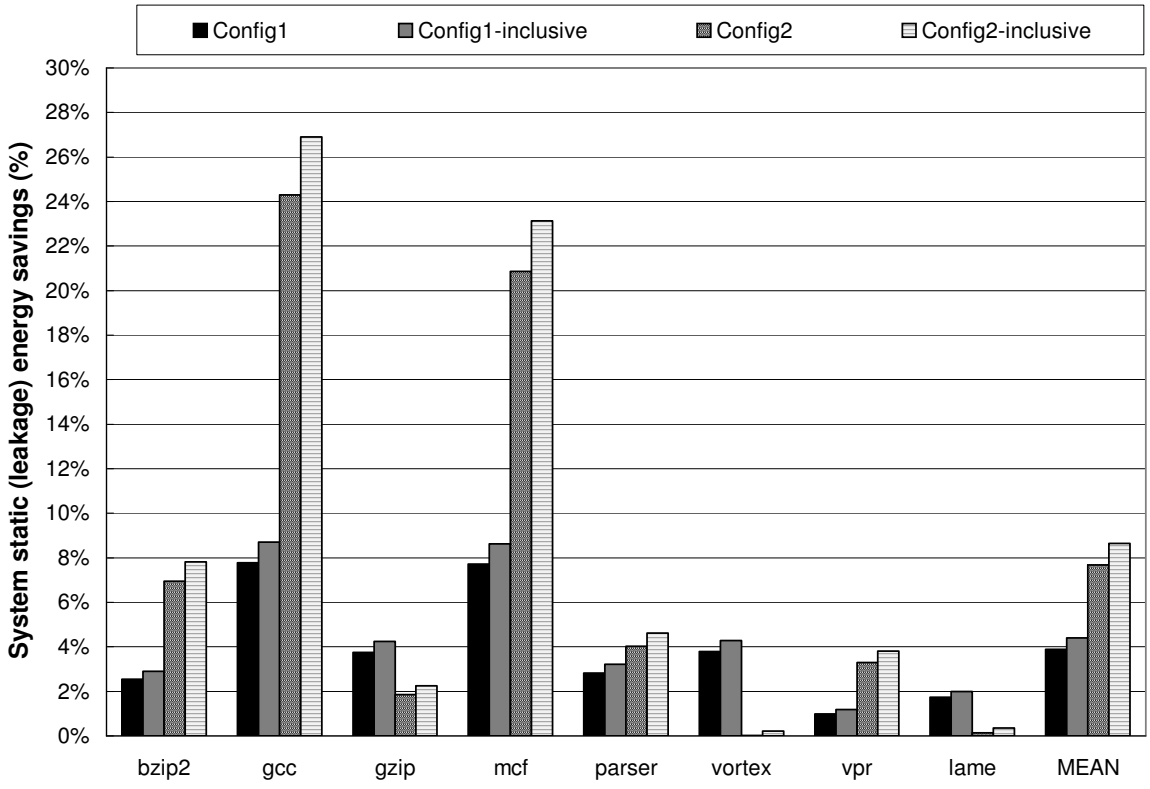



**Figure 32. Static Energy results**

the processor, L1 and L2 caches in the baseline model, and the leakage energy of the bit vector and the counter array. In addition to the two configurations, we also show the results of the inclusive versions for each configuration. In the inclusive version, the bit vector is embedded within the core. This enables the bit vector to filter out accesses to the L1 instruction and data cache.

The percentage increase in the system static energy savings are quite significant for *bzip2* and *mcf* from a smaller configuration to a larger one. For the non-inclusive version, in Configuration 2, 8% and 17% of the static energy consumption can be saved by using the segmented Bloom filter for *bzip2* and *mcf*, respectively. Similar to the L2 dynamic energy results, when switching from a smaller configuration to a larger one,some benchmarks like *adpcm*, *gcc* and *lame* observe some percentage loss in the static energy savings due to lower L2 miss rates. The inclusive versions for

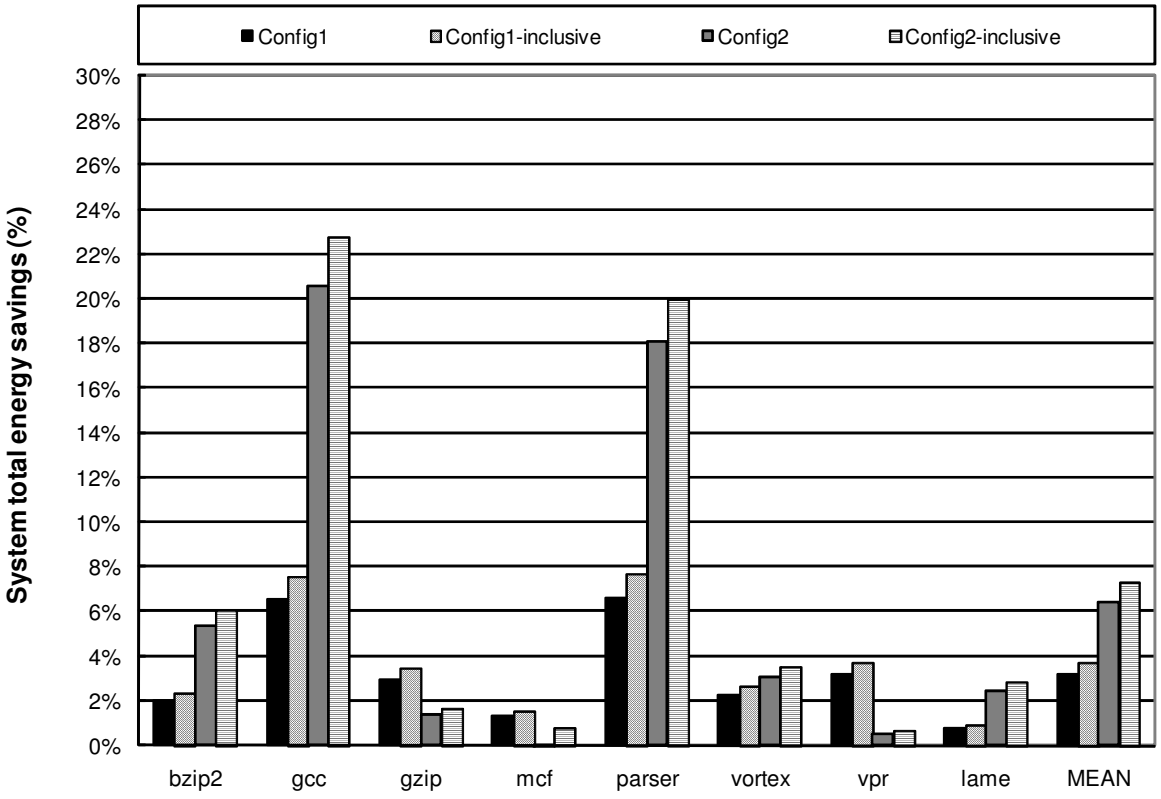



**Figure 33. Total system energy results**

both configurations show slightly better savings for all benchmarks because the inclusive configuration allows early turning off the system components, which reduces the system static energy consumption. The average system static energy savings are 2.93%, 3.82%, 3.14% and 4.12% for Configuration 1, its inclusive version, Configuration 2 and its inclusive version, respectively.

Figure 33 plots the total energy savings in percentage. The total energy is defined as the total dynamic and static energy consumed by the processor, L1 caches, L2 cache for the baseline model. For the Bloom filter enabled cache the total energy also includes the dynamic and static energy consumption of the bit vector and the counter array. Here, we see a very similar trend to the system static energy savings above when changing to a larger configuration from a smaller one. Similar to the static energy reduction, the inclusive versions for both configurations reduces the total energy more than the cases where inclusion property is not applied because of the reduction in the number

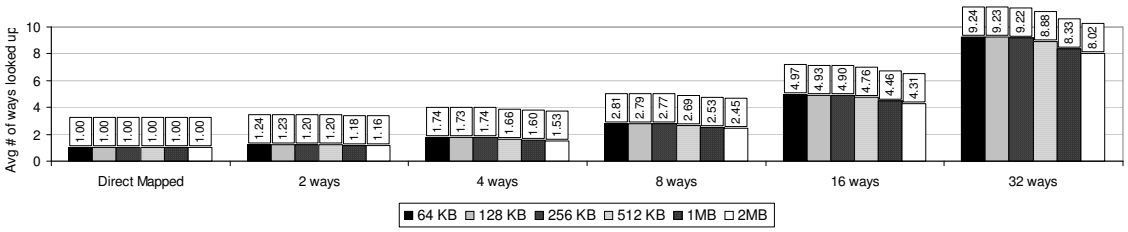



**Figure 34. Average Number of Ways Looked Up for Hits in an L2 Cache**

of L1 cache accesses that reduces the dynamic as well as the static energy consumption.

The average total energy savings for Configuration 1 and its inclusive version are 3.18% and 3.72%, respectively. These rates go to 6.5% and 7.27% for Configuration 2 and its inclusive version.

We should note that the energy savings obtained in the benchmarks above are because of the fact that each of them have a significant L2 cache miss ratio. However, if an application has a very low L2 cache miss ratio, our technique will not lead to energy savings and result in excess energy consumption for maintaining the bloom filter information. Though not evaluated in this chapter, this shortcoming can be addressed in the following way. A separate counter will keep track of the miss ratio of the L2 cache. If the miss ratio goes below a certain threshold, the bloom filter vector will be powered down and the counter will be maintained in the drowsy state. After being in a state preserving mode, if the miss ratio exceeds another threshold, the normal bloom filter operation may be resumed by first powering on the bit vector and then updating its contents from the counters. If there is sufficient hysteresis in choosing the thresholds, this scheme will not have a large overhead.

### 5.6.5 Energy Savings for Way Guard

To illustrate the effectiveness of the Way Guard, we show the average number of ways looked up (for all the benchmark programs) for hits and misses for all the 30 L2 cache configurations in Figure 34 and Figure 35. A notable observation of these results is that the Way Guard does a very good job in filtering out ways where the data is not present. In a typical case of a cache hit for an 8-way cache,

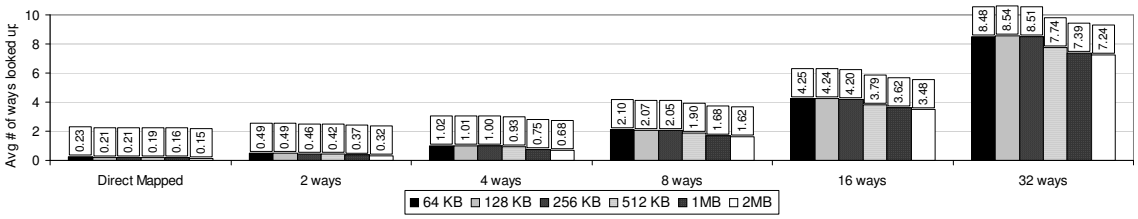



**Figure 35. Average Number of Ways Looked Up for Misses in an L2 Cache**

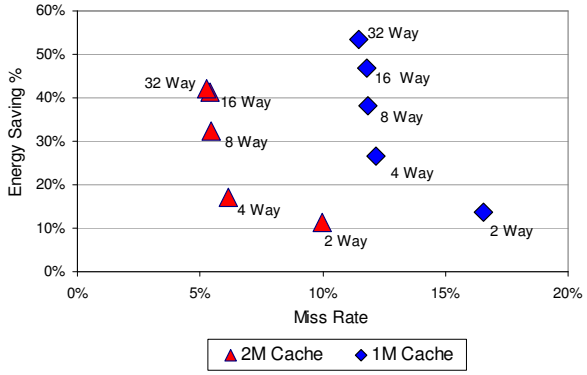



**Figure 36. Energy Savings with respect to Miss Rate**

only 2.77 ways need to be looked up for a data access. The average number of ways needed to determine a cache miss is significantly lower than that for hits. To determine a miss, the Way Guard cache checks less than 25% of the ways. Another interesting trend is that the performance of the Way Guards continues to improve with increasing cache sizes. This has to do with the sensitivity of the counting Bloom filter performance with its size. Our experiments showed that a larger counting Bloom filter always performs better than a smaller one, even though the size of the Bloom filter is always chosen to be four times the number of cache lines in each way. For a given associativity, since larger caches have larger Way Guards guarding their ways, the performance of the filters in larger caches will be better.

In Figure 36 we try to find out how the total energy savings is affected by the miss rate. The energy savings take into account both dynamic and leakage energy consumptions of the cache as well as those consumed by the Way Guards. The baseline is a normal L2 cache without the Way

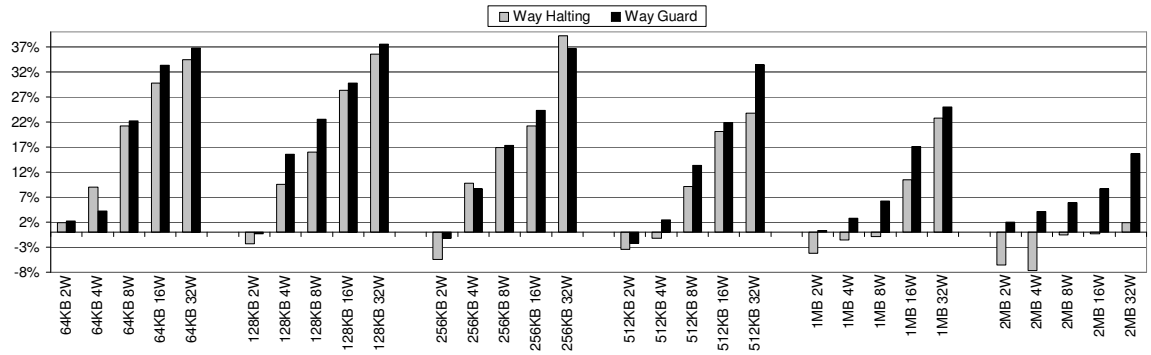



**Figure 37. Comparing Way Halting and Way Guard Energy Savings in a Serial Lookup Cache**

Guard mechanism. The figure shows the savings obtained for two cache sizes (1MB and 2MB) for the "bzip2" from SPEC benchmark. "Bzip2" is chosen for this illustration as a typical benchmark showing trends reflected in all other benchmarks. We find that the miss rate for a fixed cache size is almost the same for associativities greater than 2. We observe that we get significant savings for up to 53% for a 32-way 2 MB cache. As expected for all cache sizes, the savings increase as the associativity increases. In other words, the effectiveness of the Way Guard is increased with higher associativity. The reason behind this is that the Bloom filters are very effective at indicating absence of data, and can predict more than 80% of cache misses [52]. In a set associative cache lookup, most of the accesses to ways result in misses, that counting Bloom filters are usually good in predicting. In the case of a cache hit only one way has the required data and in the case of a cache miss none of the ways have the data. Thus, a higher associativity gives a greater chance for indicating absence, leading to larger energy savings.

By fixing the associativity, the energy savings decrease with increased cache sizes. One reason for this is that for larger cache sizes the overhead of the Bloom filter also increases. Also larger caches contain lower miss rates. So the relative benefits do not completely account for the larger overheads.

Another trend that was observed was that the relative energy overhead of Way Guard increases with associativity and decreases with cache size. We found that for a small 64KB L2 cache the relative energy overhead of Way Guard ranges from 16% for a 2-way cache to 22% for a 32-way cache. Instead, for a 2MB cache, the overhead is only 5% for 2-way to 14% for 32-way. There are two reasons why the Way Guard's overhead increases with associativity. First, as the associativity increases, more bits (one bit for each way) have to accessed for each access to the cache. Second, as the associativity increases, the performance of Way Guard also improves leading to less total energy consumption. The reason for the relative overhead decreasing with increasing cache size is that, as the cache size increases, the relative energy in accessing the cache becomes relatively larger than that of accessing the Way Guard.

We compared our technique with the Way Halting technique described in [122]. Way Halting uses a fully associative buffer to hold four tag bits for each line of the cache. When the cache is looked up, the way halting buffer matches the stored bits for each way of the corresponding set with the least significant tag bits of the address looked up. If these bits do not match for a particular way, then the lookup will surely miss that way, and the tag comparison for that way is halted, resulting in energy savings. We implemented the way halting scheme in our Bochs infrastructure. We also modeled the power overheads for the way halting technique using the Artisan SRAM generator.

The L2 cache energy savings comparing Way Halting and our Way Guard techniques are shown in Figure 37. The baseline cache for these relative energy numbers is a serial lookup cache. In a serial lookup cache, to reduce the lookup energy consumption, tag comparisons and the retrieval of the data portion of a cache line are done serially, similar to what was described in Figure 31. An access involves two steps starting with a tag comparison. Only if there is a tag match will the corresponding data row be accessed to supply the data line. All benchmark programs show similar energy saving trends in the serial lookup cache. Thus we report the geometric means of energy

savings for all the benchmarks in Figure 37. We see that since the data of all the ways are not fired up, the primary savings with the Way Guard lie in the tag comparisons. We find that for all cache sizes considered, the Way Guard technique is not very effective for caches whose associativity is less than 4. The reason behind this is, for a 2-way cache, Bloom filters save at most one tag comparison for a hit and 2 tag comparisons for a miss. This benefit in most cases does not surpass the extra energy cost needed for checking the Bloom filters for each L2 access. In contrast, it shows energy savings of up to 37% for larger associativity caches. Compared to the Way Halting scheme , the Way Guard technique shows much better energy savings for 27 of the 30 cache configurations. In a typical case a 1MB 16-way L2 has a 17% energy savings for a Way Guard while the Way Halting scheme only achieves 6.3% savings. Also for low associativities, the Way Halting technique does not have any energy savings. For a 2MB 4-way cache, Way Halting technique results in a 7% energy loss. The reason behind this is the high overhead of Way Halting for every cache access, that involves comparing four tag bits for every cache way. In contrast, the Way Guard technique only involves reading "n" bits from a bit vector array, where "n" is the associativity.

We also compared the energy savings of our technique against Way Halting, assuming a baseline parallel lookup cache. The results illustrated in Figure 38 assume that the set-associative cache accesses the same data row for all cache ways in parallel using the same set index. It can be easily seen that the Way Guard technique performs much better than the Way Halting technique for 25 of the 30 cache configurations. In a typical case, for a 1 MB 4-way cache, the Way Guard cache shows a saving of 32%, while Way Halting manages to improve the energy by 21%. As expected, the results for a parallel lookup cache show similar trends as the results for a serial lookup cache in terms of sensitivity to cache associativity and cache sizes.

For the L1 cache experiment, we consider the L1 to be a high performance parallel access cache, where data and tag are accessed together to achieving a fast hit latency. We applied our Way Guard

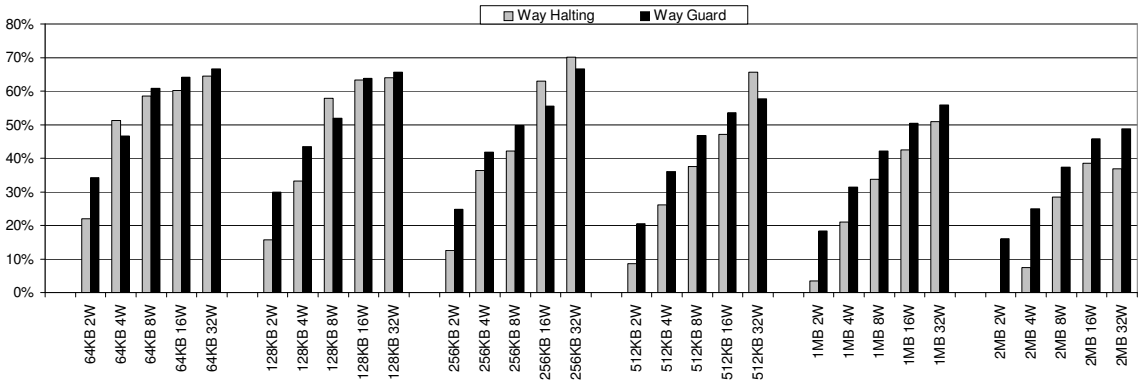



**Figure 38. Comparing Way Halting and Way Guard Energy Savings in a Parallel Lookup Cache**

technique to both the instruction and data caches. For all our experiments we assume a 2-cycle cache access. We assume that our Way Guard lookup can be fit into these two cycles to not affect the L1 cache performance. This assumption is validated by considering the access times of a typical cache configuration (64KB 4-way) and its corresponding Way Guard Bloom filter (2048 entries each way). Using Artisan the access latencies were found to be 0.74ns for the cache and 0.66ns for the Way Guard. The combined access time fits into 2 cycles of an embedded processor like AMD Geode NX 1750 running at 1.4GHz. Note that a normal cache access to this processor will also take 2 cycles, as the 0.74ns access time to the cache is larger than the 0.714ns cycle time.

We first show the geometric means of the normalized energy savings in the L1 I-cache across all the benchmark in Figure 39. In these experiments, our Way Guard technique shows huge benefits up to 68% for a 32-way cache and more than 50% for a 4-way cache. L1 caches show huge benefits because it is almost accessed every cycle during execution. For every access, with the help of Way Guard filters, only 25 to 30% of the ways need to be checked.

We also performed similar experiments for the L1 D-cache. The results are also shown in Figure 39. Similar to the I-cache results, the savings obtained using Way Guard despite the overheads are very impressive. In a typical case of a 32KB 4-way L1 cache, a 52% overall energy saving was

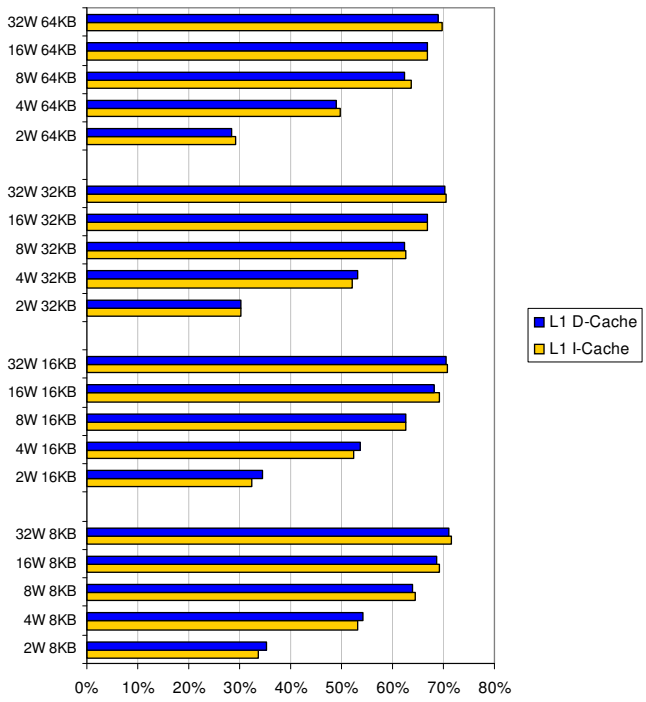



**Figure 39. Average L1 I- and D-Cache Energy Savings**

shown.

## 5.7 Summary

This chapter presents a segmented counting Bloom filter to perform energy management at the microarchitectural level and evaluates its effectiveness in reducing energy. As shown in our experiments, the segmented Bloom filter technique is an efficient microarchitectural mechanism for reducing the total processor energy consumption. A significant part of the total processor energy including L2 dynamic cache energy, L1, L2 and processor static energy can be saved in a system where the multi-level cache hierarchy assumed does not maintain inclusion property. Also,the segmented design is shown to provide even higher energy-efficiency if the multi-level cache hierarchy implements inclusive behavior.This is because the segmented design provides the opportunity to make the bit vector accessible before the L1 cache access and allows for detection of misses much

earlier in the memory hierarchy. The segmented counting Bloom filter is capable of filtering out more than 89% of the L2 misses, causing a 30% reduction in accesses to the L2 cache. This results in a saving of more than 33% of L2 dynamic energy. The results also demonstrated that the overall system energy can be reduced by up to 9% using the proposed segmented Bloom filter.

We also demonstrated that the segmented Bloom filter can be efficiently used as a way estimation technique and saves much more energy than the prior Way Halting technique. We showed that our technique can be efficiently used in all levels of the cache hierarchy obtaining substantial energy savings of up to 70% using Way Guard in both instruction and data L1 caches, and up to 65% for an unified L2 cache.

As future applications demand more memory and shrinking feature sizes allow more one-die transistors, processors would be inclined to have larger caches with higher associativity. Having these longer latency, higher associative caches will provide further opportunities for the segmented design to facilitate microarchitectural energy management earlier in the memory hierarchy and the Way Guard technique to save lookup energy. Therefore, cache miss detection and way estimation techniques in general and the segmented filter design presented in this chapter will play a key role in energy management for future microprocessors.

**Table 8. Abbreviations and their descriptions**

| Abbreviation | Description |
|---|---|
| $Cyc_{tot}$ | Total Number of Cycles |
| $Cyc_{off}$ | Number of Idle Cycles |
| $Cyc_{on}$ | Number of Active Cycles |
| $Num_{cacheaccess}$ | Number of Cache Accesses |
| $Num_{L2readmiss}$ | Number of L2 Read Misses |
| $Num_{L2access}$ | Number of L2 Accesses without filtering |
| $Num_{L1access}$ | Num of L1 Accesses |
| $Num_{L2filt}$ | Number of Filtered L2 Misses |
| $Num_{WayHit}$ | Average Number of Ways looked up in the case of a Cache Hit |
| $Num_{WayMisses}$ | Average Number of Ways looked up in the case of a Cache Miss |
| $Num_{BVUpdate}$ | Number of times the Bit Vector is updated. (Changes from 1 to 0 or 0 to 1 ) |
| $Lat_{mem}$ | Memory Latency |
| SP | Shutdown Penalty |
| $Lat_{L2}$ | L2 latency |
| $Lat_{vector}$ | Bit vector latency |
| $CPU_{dyn}$ | CPU Dynamic Energy per Cycle |
| $CPU_{leak}$ | CPU Leakage Energy per Cycle |
| $\$_{dyn}$ | Cache Dynamic Energy per Cycle |
| $\$_{leak}$ | Cache Leakage Energy per Cycle |
| $Cache_{dr}$ | Cache Drowsy Energy per Cycle |
| $BV_{dyn}$ | Bit Vector Dynamic Energy per Cycle |
| $BV_{leak}$ | Bit Vector Leakage Energy per Cycle |
| $BV_{dr}$ | Bit Vector Drowsy Energy per Cycle |
| $Counter_{dyn}$ | Counter Array Dynamic Energy per Cycle |
| $Counter_{leak}$ | Counter Array Leakage Energy per Cycle |
| $Counter_{dr}$ | Counter Array Drowsy Energy per Cycle |

**Table 9. Cache miss and miss filtering rates for configuration 1**

| Benchmark | L1 I | L1 D | L2 | Bloom Filter |
|---|---|---|---|---|
| adpcm | 0.74% | 0.78% | 36.25% | 79.6% |
| bzip2 | 0.27% | 3.4% | 67.44% | 77.12% |
| epic | 0.29% | 1.13% | 36.20% | 83.01% |
| gcc | 1.55% | 2.61% | 21.47% | 81.29% |
| gzip | 1.35% | 2.64% | 40.47% | 79.48% |
| jpeg | 0.46% | 0.89% | 37.23% | 79.82% |
| lame | 2.34% | 10.11% | 26.42% | 81% |
| mcf | 0.09% | 19.27% | 51.66% | 82.94% |
| mesa | 0.42% | 0.77% | 30.86% | 77.13% |
| parser | 0.17% | 2.85% | 40.73% | 86.34% |
| pegwit | 0.47% | 7.83% | 34.55% | 71.62% |
| vortex | 0.78% | 2.76% | 27.61% | 85.6% |
| vpr | 0.16% | 3.84% | 34.73% | 81.91% |
| **MEAN** | **0.7%** | **4.53%** | **37.36%** | **80.53%** |

**Table 10. Cache miss and miss filtering rates for configuration 2**

| Benchmark | L1 I | L1 D | L2 | Bloom Filter |
|---|---|---|---|---|
| adpcm | 0.17% | 0.46% | 51.75% | 78.49% |
| bzip2 | 0.07% | 3.07% | 51.06% | 78.72% |
| epic | 0.06% | 0.74% | 22.90% | 89.30% |
| gcc | 0.21% | 0.84% | 42.76% | 80.43% |
| gzip | 0.49% | 1.78% | 38.87% | 78.91% |
| jpeg | 0.12% | 0.36% | 34.98% | 79.23% |
| lame | 0.87% | 6.47% | 31.36% | 79.84% |
| mcf | 0.02% | 16.60% | 48.85% | 78.56% |
| mesa | 0.07% | 0.51% | 38.78% | 76.73% |
| parser | 0.05% | 1.82% | 32.92% | 80.91% |
| pegwit | 0.08% | 4.19% | 9.66% | 81.64% |
| vortex | 0.11% | 2.15% | 33.67% | 84.35% |
| vpr | 0.02% | 2.14% | 12.62% | 89.75% |
| **MEAN** | **0.18%** | **3.18%** | **34.63%** | **81.30%** |

**Table 11. L2 cache energy savings**

| Benchmark | Configuration 1 | Configuration 2 |
|---|---|---|
| adpcm | 28.85% | 40.61% |
| bzip2 | 52.01% | 51.06% |
| epic | 21.46% | 22.90% |
| gcc | 36.20% | 42.75% |
| gzip | 40.47% | 38.86% |
| jpeg | 37.22% | 34.97% |
| lame | 22.37% | 20.64% |
| mcf | 51.65% | 48.85% |
| mesa | 30.86% | 38.78% |
| parser | 40.73% | 32.92% |
| pegwit | 34.55% | 9.66% |
| vortex | 18.09% | 5.79% |
| vpr | 34.73% | 12.62% |
| **MEAN** | **34.55%** | **30.80%** |

# CHAPTER 6

# REDUCING VIRTUAL CACHE ENERGY CONSUMPTION BY USING BLOOM FILTERS TO DETECT SYNONYMS

Virtual caches are often chosen over physical caches as the first level cache because they do not need an address translation path from the virtual addresses generated by the processor. This can potentially reduce the critical path to the cache by eliminating the address translation stage through the Translation Lookaside Buffer (TLB), thereby reducing the cache access time significantly.

However, the major problem with the virtually-indexed caches is the *synonym* or *aliasing* problem [107], where multiple virtual addresses can map to the same physical address. The synonym problem occurs when the cache index bits include some bits from the virtual page number without TLB translation. In such cases, the cache block may reside in multiple cache locations, and all the tags in these locations must be looked up. However, to maintain cache consistency, only one copy of a cache line can be present in the cache. So the tag lookup must be done in multiple cache sets to check for possible synonyms.

Synonyms can be avoided by either software, hardware or a combination of both. Software techniques involve OS intervention that restricts address mapping. For instance, page coloring [65] that aligns pages so that their virtual and physical addresses point to the same cache set. However, page coloring can have an effect on system performance such as page fault rate and restricts memory allocation. Another software solution to the synonym problem is that the OS flushes the caches on context switches to prevent synonyms, at a cost of degrading the overall system performance.

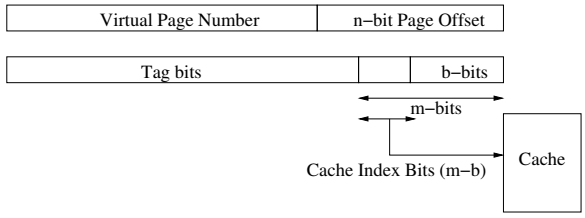
Hardware solutions, on the other hand, can detect synonyms without restricting the system software. Some of these solutions use duplicated tag arrays or reverse maps [54, 68, 107, 118], however, these techniques increase the die area and the power budget. Instead, designers prefer a simpler approach that only a single copy of a cache line is kept in the cache at any time [40,

112]. Synonyms are eliminated by the cache controller on a cache miss by searching for a possible synonym in every cache line belonging to the synonymous sets. In this chapter, we assume that this simpler approach is used to detect synonyms.

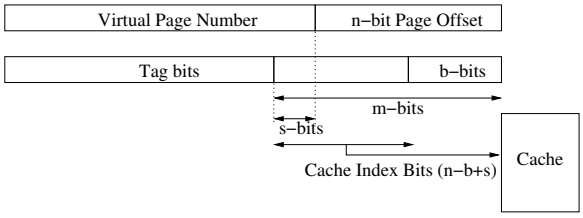
Performing multiple tag lookups for finding a synonym consumes a considerable amount of cache dynamic energy. The increasing dynamic energy consumed by tag lookups in highly associative caches pushes the designers to consider ad hoc techniques such as way prediction [59]. These techniques can predict the way that the cache line may reside in by using the address history information. Only the tag at the predicted way is compared against the outstanding address rather than initiating all-way lookup. Such techniques are not useful to reduce energy consumed by synonym lookups because synonyms are rare events and can be hard to predict using history-based predictors.

Even though the occurrence of synonym is very infrequent, the potential locations in the cache still need to be looked up for every access to maintain correctness. In fact, all we need is a mechanism that can guarantee that there is no synonym in the cache so that no further tag lookup is necessary. Such mechanism will be functionally sufficient and potentially beneficial to dynamic energy consumption.

In this chapter, we propose Synergy — an early synonym detection mechanism based on Bloom filters to reduce SYNonym lookup enERGY — to filter out unnecessary synonym lookups. A Bloom filter can provide a definitive indication that a data item or an address has not been encountered before. We call this a negative recognition. However, it cannot definitively indicate that the data item or address has been observed before. So, a positive recognition is not guaranteed in a Bloom filter. We will exploit the property of negative recognition in Bloom filters to filter synonym lookups in virtual caches for reducing cache dynamic energy consumption.




(a) No Synonym ($m \leq n$)




(b) With Synonym ($2^S$ lookups)

**Figure 40. Virtual-to-Physical Addressing**

## 6.1 Redundancy in Synonym Lookup in Virtual Caches

Figure 40(a) shows an example where there is no synonym problem. The sum of cache index and line offset bits ($m$) is smaller than or equal to the page offset bits, $n$. No synonym exists in this configuration as none of the cache indexing bits falls in virtual space. On the other hand, Figure 40(b) shows a case where synonyms can exist in the cache. When the sum of index and offset bits ($m$) is greater than $n$, $s$ bits from the virtual page number are used to index the cache without going through a virtual-to-physical address translation. The synonym problem occurs when these $s$ bits or synonym bits are used to index the cache. The synonym bits determine the number of synonym cache sets in which the potential data synonym can reside.

One of the basic hardware approaches for handling synonyms in virtual caches is to keep only a single copy of a cache line in the cache [34, 93]. In this approach, there is no extra lookup penalty or energy consumption when there is a cache hit. However, when there is a cache miss, the memory

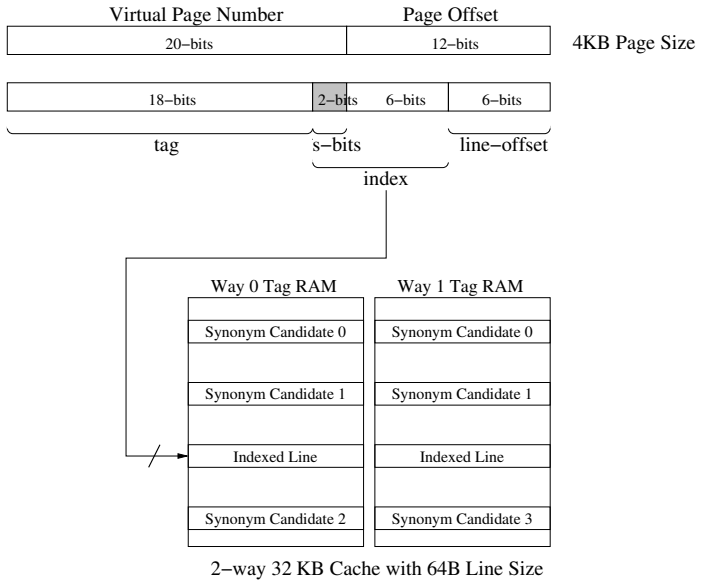



**Figure 41. Synonym problem and tag lookup energy**

request is sent to the lower-level memory while all cache lines in the synonym sets are looked up for a potential synonym. For virtually-indexed physically-tagged caches, all physical tags in the synonym sets are looked up for a synonym, the memory request is aborted and the cache line is moved to the indexed set if a synonym is found. For virtually-indexed virtually-tagged caches, every virtual tag must be translated into physical tags and then each must be compared with the tag of the missed cache address. Again, if a synonym is found, the memory request is aborted and the cache line is remapped or retagged and the line can also be moved to the indexed set.

### 6.1.1 Synonym Lookup and Tag Energy

Figure 41 shows the effect of the synonym problem on the number of tag lookups and energy consumption illustrated by an example. In this example we assume a 32-bit address space, 4KB pages, and a 2-way set associative 32KB cache with 64B line size. The page offset has 12 bits and the rest of 20 bits go through address translation. The cache consists of 256 sets indexed by 8 bits, 6 of which come from the page offset. The remaining 2 bits are taken from the virtual page number

without being translated. After getting translated the value of these two bits can lead to 4 different values. Thus the requested cache line can be in one of 4 different sets or 8 different lines in a 2-way cache.

When a cache request is made, the cache is accessed with the given virtual index and both ways are looked up for a hit. If there is a hit, then it means that there is no synonym in the cache as the cache keeps a single copy. If there is a miss, this may or may not be a true miss because the requested cache line might be in the other 3 synonym cache sets. The miss request is sent to the L2 cache and in the mean time, the tag lookup in the L1 continues serially until a synonym is found. After 3 serial lookups, if there is a hit, a synonym is found in the cache. The lookup in the L1 is then stopped and the miss request to the L2 cache is also aborted. Afterward, the synonym is moved to the location that generated the miss.

If a synonym is not found after 3 serial lookups, this is a true miss and the cache does not have a synonym with the given address. As such, six additional tag lookups were performed to determine if there is a true miss. This certainly increases the cache dynamic energy consumption.

Given the fact that synonyms are infrequent, performing the additional search for a synonym is generally a waste of dynamic energy. We can eliminate the extra synonym lookups if we know for certain that there is no synonym in the cache. For this purpose, we propose to use a Bloom filter associated with the cache that can safely indicate that there is no synonym in the cache. A detailed description of Bloom filters is given in Section 5.2.

## 6.2 Synergy: Early Synonym Detection with a Decoupled Bloom Filter

If finding synonyms is a rare event per cache miss, then the Bloom filter can be quite successful for fast and low-power detection of synonyms in virtual caches. It is fast because it indicates whether there is no synonym by checking a single bit location in the vector. In contrast, the baseline serial

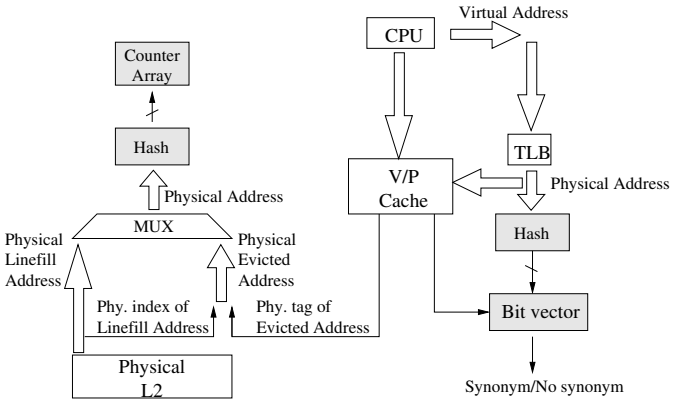



**Figure 42. Early synonym detection in a V/P cache**

lookup cache detection may also be detrimental to performance in some cases. If a cache has 3 synonym bits, it would have 8 possible locations to search for a synonym. If each search takes 2 cycles and the L2 cache access time is 10 cycles, searching for a synonym serially is worse than going to the L2 cache. It is low-power because *(w*S)* tag lookup comparisons can be eliminated where *w* is the cache associativity and *S* is the number of synonym sets. The frequency of synonyms depends on the number of shared pages. When also considering the fact that cache misses are less frequent than hits, the frequency of finding synonyms per cache miss is likely to be low, so the virtual cache system certainly benefits from using a Bloom filter to filter out several tag lookups.

### 6.2.1 Virtually-indexed Physically-tagged Caches

Figure 42 shows an early synonym detection mechanism with a loosely-coupled Bloom filter for a virtually-indexed physically-tagged (VIPT) cache. The Bloom filter keeps track of physical addresses rather than virtual ones so that it can provide a quick response regarding synonym existence with a given physical address. The counter array is updated with the physical addresses of the linefill from the physical L2 cache and the evicted line. The physical address of the evicted line is formed from the tag that contains the usual tag bits and the translated synonym bits, appended with the

lower part of the index of the set.

When there is a miss in the VIPT cache for a lookup, the physical address is sent to the L2 cache to service the miss. At the same time, the physical address is hashed into the bit vector to check for any synonym. If the bit location is zero, there is no synonym in the cache. In that case, no synonym lookup is needed in the cache. If the bit location is 1, this implies that there may be a synonym in the cache, and a serial synonym lookup process is launched to all synonym sets for synonym detection.

### 6.2.2 Virtually-indexed Virtually-tagged Caches

For virtually-indexed virtually-tagged (VIVT) caches, the Bloom filter counters are updated by the physical address of the linefill during a linefill. For eviction, however, the evicted address must be translated into the physical address by the TLB. The dirty evicted lines are already translated by the TLB before sending it to the physical memory. The only extra TLB translations are needed for the clean evicted lines. When a cache miss occurs, the Bloom filter vector is accessed by the physical address acquired from the TLB. Similar to the VIPT caches, a true miss in the Bloom filter vector (value returned is zero) means that there is no synonym in the cache associated with this physical address, and a false hit (value returned is one) means that all synonym sets must be looked up for a synonym in the cache. This involves in translating all virtual tags in all the synonym sets into physical tags and then compare them to the physical tag of the outstanding address. Because the Bloom filter keeps track of signatures of physical addresses, its functionality remains correct regardless of context switches.

## 6.3 Counter Overflow Issues

### 6.3.1 Probability of Overflow

One important concern with Synergy is whether an L-bit counter is big enough to prevent it from overflowing. As the data access pattern of applications are unpredictable and corresponding hashing

results are difficult to analyze, this is not easy to tell. From our vast amount of experiments with tens of billion instructions simulated, we found that a 3-bit counter is good enough for the Bloom filter with $\alpha = 4$. In this case, as the number of entries of this Bloom filter is four times more than the number of cache lines of the L1 data cache, overflow is unlikely to occur. For example, 32KB 2-way set associative cache with 32B line contains total 1024 cache lines. Corresponding Bloom filter contains 4096 entries. Since the presence of an address in a cache line counts as one increment of a counter, the sum of all counter values is limited to the number of cache lines (1024), thus the expected value of each counter is 0.25 (=1024/4096), which is very small compared to 7, the biggest number that 3-bit counter can count.

In this section we try to show mathematically that the probability of overflow of a counter is indeed very small. We define our probabilistic experiment as follows:

Experiment $E$: Select a cache line from the cache and note corresponding Bloom filter index of this cache line

The sample space for this experiment is $S = \{A, B\}$, where A corresponds to the event, "the $i^{th}$ cache line maps to the $j^{th}$ index of the Bloom filter", and B to the event, "the $i^{th}$ cache line does not map to the $j^{th}$ index of the Bloom filter".

The number of entries of the Bloom filter, $n_b$, is $\alpha \times w \times n_s$, where $w$ is associativity of the cache, $n_s$ is the number of sets of the cache, and $\alpha$ is the design-dependent constant that determines the size of Bloom filter. Then, the number of cache lines, $n_c$, is represented as $w \times n_s$.

Random variable $X_i^j$ is defined as 1 if the $i^{th}$ cache line maps to the $j^{th}$ index of the Bloom filter, or 0 otherwise, where $0 \leq i < n_c$ and $0 \leq j < n_b$. We assume that the hash generated by the XOR hashing function can be uniformly spread across the entire Bloom filter indices, so the probability

of the $i^{th}$ cache line mapping to the $j^{th}$ index of the Bloom filter is represented as follows.

$$P[X_i^j = 1] = \frac{1}{n_b} = \frac{1}{\alpha \times w \times n_s} = p$$

On the other hand, the probability of the $i^{th}$ cache line mapping to indices other than the $j^{th}$ index of the Bloom filter is represented as follows.

$$P[X_i^j = 0] = 1 - P[X_i^j = 1] = 1 - p$$

Now, let $S_{n_c}^j$ be the sum of all the random variables of the form $X_i^j$, for all cache lines with respect to the Bloom filter index $j$:

$$S_{n_c}^j = X_0^j + X_1^j + \cdots + X_{n_c-1}^j$$

Note that $S_{n_c}^j$ represents the counter value of the $j^{th}$ Bloom filter index. Assuming that $X_i^j$ is an independent identically distributed random variable, now $S_{n_c}^j$ becomes binomial random variable. Thus, the probability that $S_{n_c}^j$ is equal to $x$ can be represented as follows:

$$P[S_{n_c}^j = x] = \begin{pmatrix} n_c \\ x \end{pmatrix} p^x (1-p)^{n_c - x}$$

Then, the probability of overflowing a 3-bit counting Bloom filter can be represented as follows:

$$P[S_{n_c}^j > 7] = 1 - \sum_{x=0}^{7} P[S_{n_c}^j = x] = 1 - \sum_{x=0}^{7} \begin{pmatrix} n_c \\ x \end{pmatrix} p^x (1-p)^{n_c - x}$$

Because $n_c$ is fairly big, we can apply the central limit theorem to use the uniform Gaussian distribution as an approximation to the discrete binomial distribution [74].[1] The mean of $S_{n_c}^j$ is

$$m_s = n_c p = \frac{1}{\alpha}$$

, and its variance is

$$\sigma_s^2 = n_c p(1-p) = \frac{\alpha n_c - 1}{\alpha^2 n_c}$$

[1]The smallest $n_c$ in our experiments is 256.

Then, now the probability of overflow can be represented as follows:

$$P[S^j_{n_c} > 7] = \frac{1}{\sqrt{2\pi}\sigma_s} \int_7^\infty e^{-\frac{(s-m_s)^2}{2\sigma_s^2}}\, ds$$

If we normalize this Gaussian distribution, using

$$Z^j_{n_c} = \frac{S^j_{n_c} - m_s}{\sigma_s}$$

, then finally the probability of overflow of $j^{th}$ index is represented as follows:

$$P[S^j_{n_c} > 7] = P[Z^j_{n_c} > \frac{7 - m_s}{\sigma_s}] = \frac{1}{\sqrt{2\pi}} \int_{\frac{7-m_s}{\sigma_s}}^\infty e^{-\frac{z^2}{2}}\, dz = \frac{1}{\sqrt{2\pi}} \int_{\frac{7-\frac{1}{\alpha}}{\sqrt{\frac{\alpha n_c - 1}{\alpha^2 n_c}}}}^\infty e^{-\frac{z^2}{2}}\, dz$$

In all our cache configurations where $\alpha = 4$, $z = \frac{7-\frac{1}{\alpha}}{\sqrt{\frac{\alpha n_c - 1}{\alpha^2 n_c}}}$ is found to be bigger than 13. Thus, $P[S^j_{n_c} > 7] < P[Z^j_{n_c} > 13]$. Unfortunately, $P[Z^j_{n_c} > 13]$ is very small number, so we are not able to calculate this number using either normal distribution table or MATLAB. For reference, we provide some other probability numbers that MATLAB calculates with smaller $z$ in Table 12. In conclusion, the probability is so small that the counters will overflow very rarely.

**Table 12.** $P[Z^j_{n_c} > z]$

| $z$ | $P[Z^j_{n_c} > z]$ |
|---|---|
| 5 | $2.8665 \times 10^{-7}$ |
| 6 | $9.8659 \times 10^{-10}$ |
| 7 | $1.2799 \times 10^{-12}$ |
| 8 | $6.6613 \times 10^{-16}$ |
| 9 ~ | 0.0000 |

### 6.3.2 Simple Solutions to the Overflow Problem

Nonetheless, the overflow of a counter will affect the correctness of the filter and thus should not be ignored. There are several solutions that can solve the correctness problem. The first solution is disabling Synergy once it overflows. For example, once one entry of the Bloom filter overflows, the following memory lookup's that map to this entry will not rely on Synergy. Instead, they need to look up all possible synonym sets as in a conventional design. Only after the system is rebooted, the full functionality of Synergy can be recovered. The second solution is flushing the cache once one entry of the Bloom filter overflows. Considering the fact that this is a very rare event, this

will not affect the system performance a lot, and seems to be a good design trade-off in terms of performance, energy, and area. The third solution is to use larger counters. In the previous example with 1024 cache lines in total, we can prevent the overflow problem if all counters are 11 bit wide, which can count up to 1024, the worst case where all cache lines map to the same entry. However, considering the fact that this is a extremely rare event, this is a design overkill.

## 6.4 Overflow-free Bloom Filter

Here, we propose an overflow-free Bloom filter design, although this may be more expensive in terms of hardware and energy. This solution is to use a more predictable hash function and estimate the required number of bits in a virtual cache. Instead of XOR-ing, hashing can be done by simply taking the least significant bits (LSBs) of the physical memory address excluding line offset bits. For example, when using a 2-way set associative 32KB cache with 32B line, and a 4096-entry Bloom filter, this hash function uses physical address bit[16:5] to index the Bloom filter as shown in Figure 43.

We now show that using the simplistic hash function described above, we may obtain a mathematical bound on the maximum value a counter can reach. Since the bits [11:5] of the example shown in Figure 43 is part of the page offset, it is used to index both the cache and the Bloom filter. Thus, only different combinations of the s-bits (bit[13:12]) can make different cache lines to be mapped to the same Bloom filter entry. Furthermore, owing to the cache allocation/eviction mechanism, only two valid cache lines can map to the same set of the 2-way cache at any given

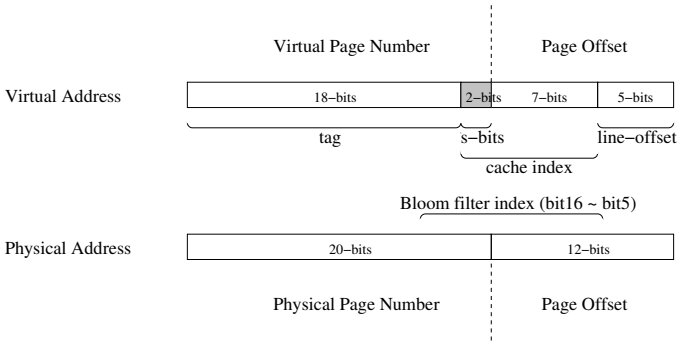



**Figure 43. 2-way Set Associative 32KB Cache with 32B Line**

time. Consequently, with this simplistic hash function, these 8 ($= 2 \times 2^2$) virtually-indexed cache lines can all map to the same Bloom filter entry under the worst case scenario.

In general, to prevent an overflow, theoretically, we need $1 + log_2(w \times 2^s)$ bit counters where $w$ is associativity of the cache, and $s$ is the length of synonym bits. This can be easily proved as follows.

DEFINITION 6.4.1. *Let w be the associativity of the cache, s be the length of synonym bits, i be the length of cache index bits, b be the length of cache block offset bits, and r be the length of hashed values of the new hashing function.*

DEFINITION 6.4.2. *For any virtually-indexed cache, the bits that are both a part of the page offset and the cache index bits are defined as the non-synonym index bits.*

DEFINITION 6.4.3. *Let $V = \{x : x \in \mathbb{N}\}$ and $W_r = \{x : x \in \mathbb{N}, x < 2^r\}$, where r is a positive integer. Modulo hash function (MHF), $Mod_b^r(x)$, is a function from V to $W_r$, defined by*

$$Mod_b^r(x) = (\frac{x}{2^b}) \ mod \ 2^r$$

*where $b, r \in \mathbb{N}$ and $r > i$. In the context of caches, this function right-shifts a given physical address by the amount of $log_2(blocksize)$, and takes r least significant bits.*

LEMMA 6.4.4. *Non-synonym index bits of one virtual address are identical to non-synonym index bits of the corresponding physical address.*

PROOF. From the definition of non-synonym index bits, they are part of the page offset, thus not affected by address translation. Therefore, non-synonym index bits of one virtual address are identical to non-synonym index bits of the corresponding physical address. □

LEMMA 6.4.5. *For a counting Bloom filter using $Mod_b^r(x)$, any two virtual addresses with different non-synonym index bits are mapped to different indices of the Bloom filter.*

PROOF. From Lemma 6.4.4, any two virtual addresses with different non-synonym index bits would be translated into physical addresses with different non-synonym index bits. Because non-synonym index bits become the suffix of hashed value of $Mod_b^r(x)$, these two virtual addresses would be mapped to different indices of the counting Bloom filter. □

LEMMA 6.4.6. *In a virtually indexed cache, the maximum number of addresses with the same non-synonym index bits is given by the expression* $w \times 2^s$.

PROOF. By the definition of cache index bits, there exists $2^s$ cache indices with the same non-synonym bits. Now for every cache index, by the definition of cache associativity, there can be at most $w$ cache lines which are mapped to the same index and are present in the cache simultaneously. Thus, the maximum number of cache lines that can have the same non-synonym index bits is $w \times 2^s$. □

THEOREM 6.4.7. *The counter value of any counter in a counting Bloom filter using* $Mod_b^r(x)$ *is less than or equal to* $w \times 2^s$.

PROOF. From Lemma 6.4.5, all cache lines that differ in the non-synonym index bits are mapped to different indices in the Bloom filter. Thus, the maximum number of cache lines that can be mapped to one Bloom filter counter is bounded by the maximum number of cache lines that have the same non-synonym index bits. From Lemma 6.4.6, the maximum number of cache lines having the same non-synonym index bits is $w \times 2^s$. Consequently, it is proved that the possible maximum value of the counter is $w \times 2^s$. □

Because this counter should be able to count numbers from 0 to $w \times 2^s$ not to overflow, each counter should be $1 + log_2(w \times 2^s)$ bits long. We have thus presented a design of the counting bloom filter that will never overflow. In the following section we present results obtained using both the conventional and overflow free counting bloom filters.

## 6.5 Experimental Results

### 6.5.1 Simulation Environment

We use Bochs [61] to perform full-system simulations. Each simulation involves running one or more common applications on a Windows NT platform. To collect cache statistics, we integrate Simplescalar's cache simulator in Bochs. We also integrated our Bloom filter model within the cache simulator. This simulation technique allows us to gather cache statistics of various applications running on top of a full-blown operating system. Previous techniques [115] used Bochs to collect instruction execution traces and fed them into a micro-architectural simulator. By integrating

**Table 13. L1 data cache configuration**

| # of sets | # of ways | Line size (B) | Cache size (KB) | Bit vector (b) | Counter array* (b) | s-bits | $S$** |
|---|---|---|---|---|---|---|---|
| 256 | 1 | 32 | 8 | 1k | 3k | 1 | 2 |
| 256 | 2 | 32 | 16 | 2k | 6k | 1 | 2 |
| 256 | 4 | 32 | 32 | 4k | 12k | 1 | 2 |
| 256 | 8 | 32 | 64 | 8k | 24k | 1 | 2 |
| 512 | 1 | 32 | 16 | 2k | 6k | 2 | 4 |
| 512 | 2 | 32 | 32 | 4k | 12k | 2 | 4 |
| 512 | 4 | 32 | 64 | 8k | 24k | 2 | 4 |
| 1024 | 1 | 32 | 32 | 4k | 12k | 3 | 8 |
| 1024 | 2 | 32 | 64 | 8k | 24k | 3 | 8 |
| 2048 | 1 | 32 | 64 | 8k | 24k | 4 | 16 |
| 256 | 1 | 64 | 16 | 1k | 3k | 2 | 4 |
| 256 | 2 | 64 | 32 | 2k | 6k | 2 | 4 |
| 256 | 4 | 64 | 64 | 4k | 12k | 2 | 4 |
| 512 | 1 | 64 | 32 | 2k | 6k | 3 | 8 |
| 512 | 2 | 64 | 64 | 4k | 12k | 3 | 8 |
| 1024 | 1 | 64 | 64 | 4k | 12k | 4 | 16 |

* Size of one counter is 3 bits
** $S$ is the number of synonym sets

the cache simulator into Bochs we remove the time-consuming and cumbersome process of trace collection from the simulation process. This technique is much faster than trace collection and we may simulate a lot more instructions than trace collection techniques, as trace collection inherently limits the size of the trace that may be collected.

We choose the size of each page to be 4KB and use 16 different configurations for L1 data cache as shown in Table 13. These 16 cache configurations are the only configurations possible for cache sizes less than 64KB with a page size of 4KB to demonstrate the synonym problem.[2] We simulate both VIPT caches and VIVT caches with 64-bit virtual memory space. The L1 instruction cache was also modeled using the same configuration with the L1 data cache. To estimate power consumption for the caches and the Bloom filter, we use Artisan 90nm SRAM library. The Artisan SRAM generator is capable of generating synthesizable Verilog code for SRAMs in 90nm technology. A datasheet is also generated that gives an estimate of the read and write power of the generated SRAM. The datasheet also provides a standby current from which we can estimate the leakage power of the SRAM.

The energy reduction results reported are only for the data cache. Synonyms in I-Cache do not affect correctness as they are read only. Also hit rates of I-Cache is extremely high. So we assume no hardware technique for detecting synonyms for the I-Cache. Our simulation system runs Windows

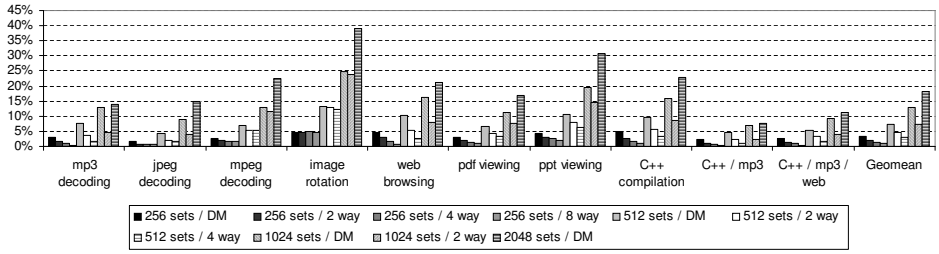
[2]We exclude four configurations with 128 sets that Artisan 90nm SRAM library does not support.

NT and we uses eight applications to evaluate our approach. These applications include typical everyday workloads like mp3/jpeg/mpeg decoding and pdf/powerpoint viewing. Since it would be difficult to exactly reproduce actions that involve human interaction like clicking the mouse, we decided to simulate all 16 cache configurations simultaneously. We instantiated 16 cache simulators and fed them with the instruction stream produced by Bochs. Throughout all simulations that we performed, we evaluated the advantage of our Bloom filter enhanced cache structure compared with the conventional cache structure.

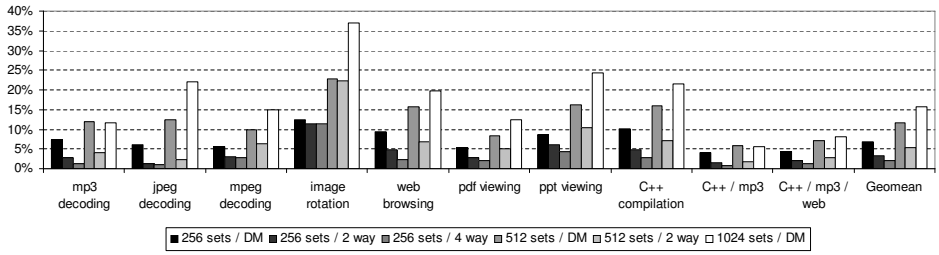
We accurately model the delay and energy overhead for the Bloom filter. To estimate dynamic and static power consumption, we synthesize the Bloom filter structures using the same 90nm Artisan SRAM generator used for synthesizing the caches. Since we have implemented the loosely coupled design, the bit vector and the counter array are synthesized separately. We add the bit vector access energy every time the Bloom filter is queried for the presence or absence of an address. This is done for every cache miss to check for synonyms. The counter array access energy is added every time the structure is updated. This happens when a cache line is evicted from the cache and also when a line-fill occurs. Since our bit vector is a relatively small structure than the cache, we assume the access latency to be one cycle. However, this access latency affects performance only on cache miss events in the presence of synonyms. Since the occurrence of synonyms is very rare it has negligible effect on the performance of the system. For VIVT caches, we also model the additional energy overhead for accessing the TLB (Section 6.2.2).

### 6.5.2 Results

First, we analyze the number of synonyms detected during our simulations. Here, we define *synonym ratio* as the number of synonyms detected normalized to the number of cache misses. All our ten simulation scenarios show that synonyms are rarely present. The harmonic mean of synonym ratio is merely 0.00003%, meaning that only three out of ten million cache misses are identified as synonyms. Note that, even though the occurrence of synonyms is rare, it is nonetheless required for the baseline processor to examine all synonym sets on every cache miss — leading to large, superfluous dynamic energy consumption just for synonym detection. This observation justifies our motivation for finding a light weight hardware alternative to alleviate the unnecessary energy




(a) Dynamic Energy Savings with 32B-Line Caches




(b) Dynamic Energy Savings with 64B-Line Caches

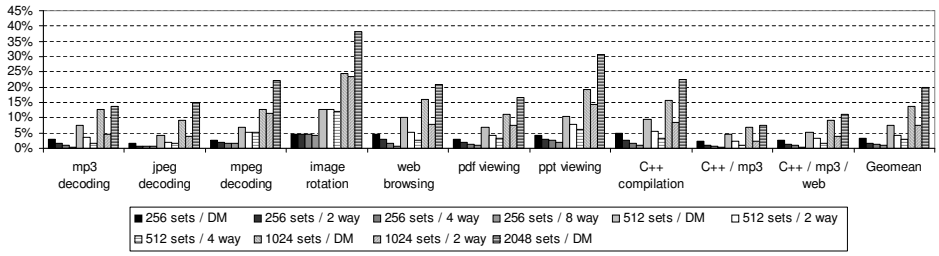
**Figure 44. Dynamic Energy Savings (VIPT)**

consumed due to synonym problem.

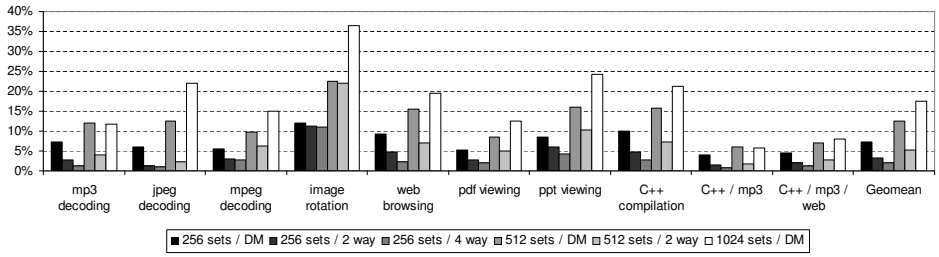
One reason of the synonym ratio being extremely low is that the number of operations between successive context switches is very large. Since the L1 data cache is very small, the cache would only contain data pertaining to the presently running process. This reduces the probability of having shared data from another process, making the number of synonyms negligible. Synonyms would only be detected at the initial phase of one context switching period, after which the cache would be warmed up and filled with data from the current process.

Figure 44 shows the dynamic energy saved in VIPT caches with 32 and 64 byte cache lines, using XOR-based 3-bit counting Bloom filter. Unless otherwise stated, we use this Bloom filter model by default. The amount of dynamic energy reduction of Synergy ranges from 0.3% to 38.9% depending on the cache configuration and the characteristics of the application. Note that these savings take into account the dynamic energy overheads of the bit vector and the counter array.

Figure 45 shows that the amount of total energy reduction of L1 data cache with Synergy ranges



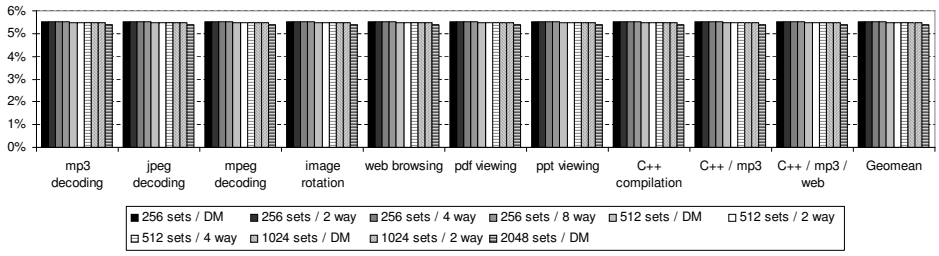

(a) Overall Energy Savings with 32B-Line Caches



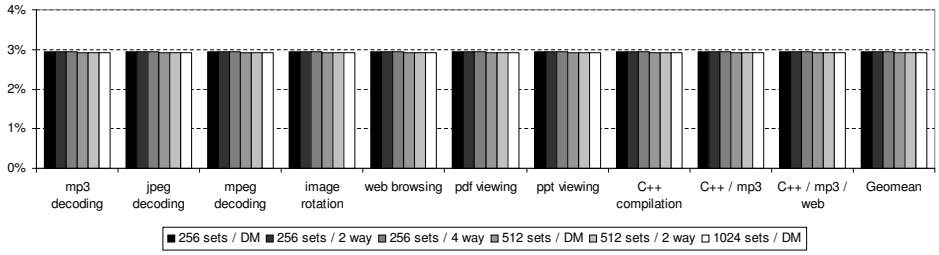

(b) Overall Energy Savings with 64B-Line Caches

**Figure 45. Overall Energy Savings (VIPT)**

between 0.3% and 38.2%. To facilitate the estimation of the leakage energy, we assume a 4GHz in-order processor with blocking cache. The latencies of L1, L2 and DRAM are 3, 20 and 250 cycles. We assume the average micro-ops to x86 instruction ratio is 1.4 as shown in [106]. Note that our approach is very conservative because a longer execution time of our simulation model will lead to more pessimistic leakage energy consumption although our solution is aimed for reducing dynamic energy consumption. Thus, the actual total energy reduction using our scheme in out-of-order processors with non-blocking caches is expected to be higher. This savings take into account the dynamic and leakage energy consumption (Figure 46) of the added hardware structures, namely the bit vectors and the counters. The leakage energy overhead of the additional hardware, compared to overall dynamic and leakage energy consumption, is found to be 0.04% over all the benchmark programs. Since the additional hardware structures are very small compared to the original cache, and the dynamic energy consumption of L1 caches dominates the leakage energy consumption, the leakage energy overhead of Synergy is negligibly small.




(a) Leakage Energy Overhead for 32B-Line Caches




(b) Leakage Energy Overhead for 64B-Line Caches

**Figure 46. Leakage Energy Overhead (VIPT)**

To further understand the source of energy savings, we use the *PowerPoint* workload running on 32-byte line caches for our analysis. The reason for choosing *Powerpoint* is that the simulation result with this benchmark is very sensitive to different cache configurations and thus helps us to get a better understanding of the source of energy savings. Figure 47 plots the normalized dynamic energy consumed by hits and misses in the baseline cases, and by true misses, false hits of Synergy. We may easily see that Synergy's miss handling contributes to a significant amount of L1 data cache dynamic energy. In some cases it is as high as 44.4% (for the configuration of 2048 sets, Direct-Mapped). The figure also shows that Synergy can significantly reduce dynamic energy consumption for miss handling, resulting in an overall 33.9% energy reduction for this configuration. In addition to energy consumption of the cache itself, dynamic energy consumed by bit vectors and counter arrays are also shown in the figure, but they are too small to be recognized. It is found that the energy consumed by the bit vector and the counter array account only for 0.06% and 0.16% on average (geometric mean) respectively.

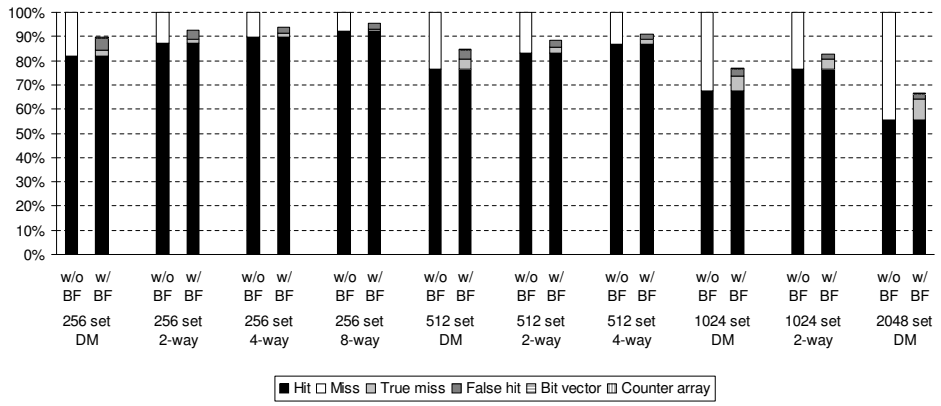



**Figure 47. Relative Dynamic Energy Consumption of *PowerPoint* for 32B-Line Caches**

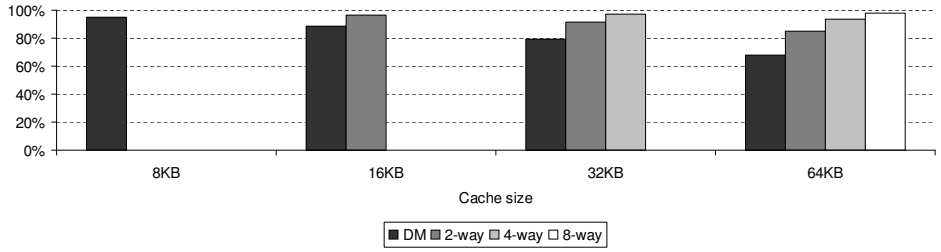
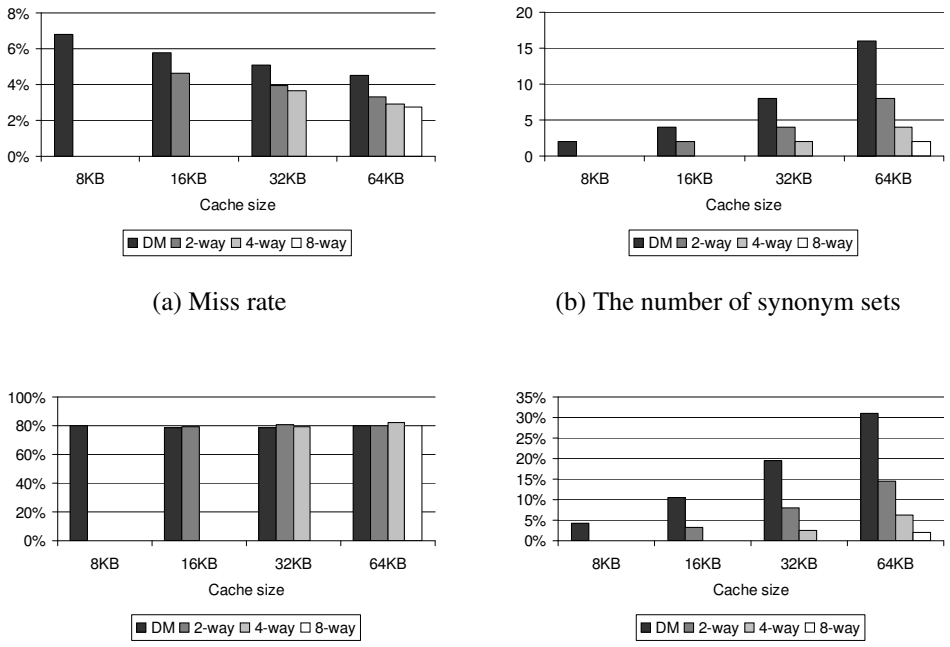



**Figure 48. Relative Number of L1 D-cache accesses of *PowerPoint* for 32B-Line Caches**

Figure 48 shows the relative percentage of L1 cache accesses of Synergy compared to the baseline cache. In this figure, for every cache miss, we count *S* accesses to the baseline cache with each n-way access, where *S* is the number of synonym sets. We see that Synergy can reduce cache accesses by up to 32.1%. In the extreme case, the image rotation algorithm, Synergy can reduce cache accesses by 40.7%. We know that L1 cache performance is critical to overall system performance. The contention for the L1 cache would be more of a critical factor in superscalar and SMT systems where multiple cache accesses may occur in one single cycle. Our simple hardware technique will be extremely useful in such systems as it would reduce a large number of accesses and contention for the L1 cache.

Figure 49(d) shows the effect of cache configuration on the effectiveness of our approach. From the figures we see that energy reduction decreases when associativity increases. One reason for this is that increasing associativity drastically reduces the number of misses as shown in Figure 49(a)




(a) Miss rate

(b) The number of synonym sets

(c) True miss rate

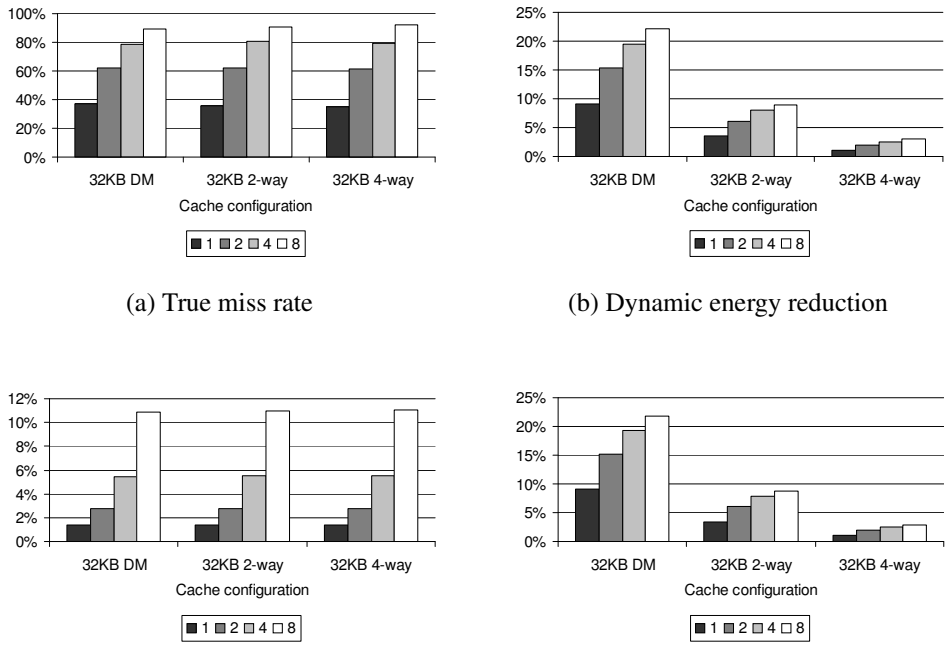
(d) Dynamic energy reduction

**Figure 49. Simulation Results of *PowerPoint* for 32B-Line Caches**

and thus the opportunity of Synergy to save energy. Another reason can be found by looking at Figure 49(b). We see that for cache configurations having the same cache and line sizes, the number of synonym sets drops linearly as the associativity increases. Consequently, the opportunity of energy saving also drops. The design trend of modern L1 caches, however, are in favor of lower associativity to meet cycle time constraints. Another observation that can be made from these figures is that energy reduction decreases as the cache size is reduced. This is because even though miss rates decrease with larger caches, the number of synonym sets also increase. This gives a significant energy saving opportunity for our technique over the baseline.

The last factor that affects the benefit of Synergy is the *true miss rate*. This indicates how efficiently the Bloom filter can filter out unnecessary synonym lookups. However, the effect of the true miss rate is not clearly revealed in the set of simulations shown in Figure 49(c), because true miss rate of these simulations are similar.

To analyze the effect of true miss rate, we vary the number of entries of the Bloom Filter. In this simulation, it is set to $\alpha \times (\#of\,sets) \times (associativity)$ where $\alpha$ =1, 2, 4 or 8, and corresponding




(a) True miss rate

(b) Dynamic energy reduction

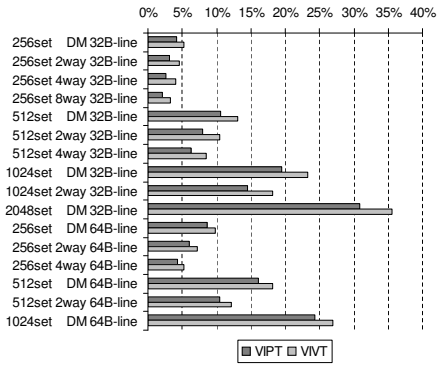
(c) Leakage energy overhead

(d) Total energy reduction

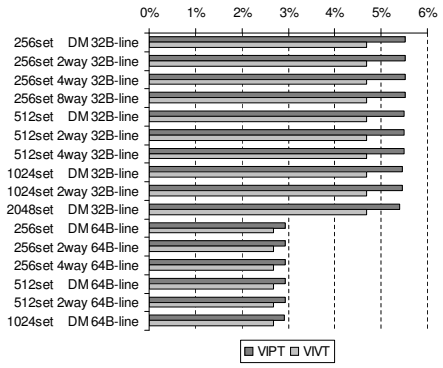
**Figure 50. Sensitivity Study of the Bloom Filter Size with *PowerPoint* (32B-line Caches)**

different energy overheads are modeled. In all previous simulations $\alpha$ was uniformly set to four (Table 13). Figure 50 shows the effect of the Bloom filter size for a 32KB cache with 32B lines. As expected, as the $\alpha$ value increases from one to eight, the true miss rate increases (Figure 50(a)), because the probability of two or more different physical addresses being mapped to the same index of the Bloom filter decreases. Consequently, the amount of dynamic energy saved, also increases (Figure 50(b)). However, it does not scale well, because the dynamic overhead grows as $\alpha$ increases. Furthermore, leakage overhead increases linearly with $\alpha$ (Figure 50(c)). Thus, the total energy saving that we can achieve with a larger Bloom filter gradually levels off as shown in Figure 50(d). Although the good choice of the Bloom filter size depends on the cache configuration, the Bloom filter with $\alpha = 4$ seems to be reasonable, considering the fact that the advantage of the Bloom filter with $\alpha = 8$ is not significant, and it consumes larger area.

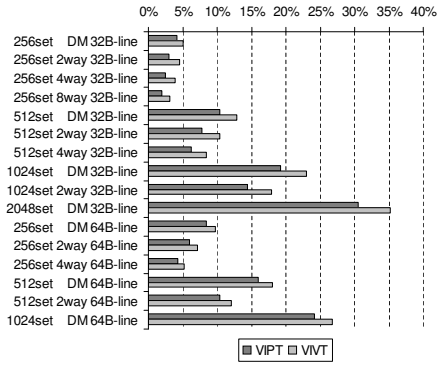
In addition to VIPT caches, we also evaluate the advantage of Synergy with VIVT caches. Figure 51 shows simulation results with VIVT caches. Although we performed simulations with all workloads, we only show the simulation results with the *PowerPoint* workloads, because the




(a) Dynamic Energy Saving




(b) Leakage Energy Overhead




(c) Overall Energy Saving

**Figure 51. Simulation Result with VIVT Caches using** *PowerPoint*

trend shown by this workload is representative of other workloads used. For easier comparison, previous simulation results with VIPT caches are shown as well. In comparison to VIPT caches, for VIVT caches, Synergy needs to perform an additional TLB access on clean evictions, as mentioned in Section 6.2.2. However, as shown in Figure 51(a), additional accesses to the TLB upon clean eviction on a VIVT cache do not hurt dynamic savings of Synergy. Rather, it was found that Synergy on a VIVT cache can save more dynamic energy than Synergy on a VIPT cache. It is mainly caused by following facts: (1) Upon every cache miss, a baseline VIVT cache needs to access the TLB to detect synonyms, thus it consumes more power on miss handling than a baseline VIPT cache. (2) Because Synergy mainly saves power consumed upon cache miss, Synergy on a VIVT cache has more chances to save power. (3) Consequently, additional power overhead of Synergy on a VIVT

cache (due to additional TLB access upon clean eviction) becomes relatively small. Since the tag array of VIVT caches is bigger than that of VIPT caches, due to bigger virtual tags and additional process ID information to solve the homonym problem, the leakage energy overhead of Synergy on VIVT caches is found to be smaller than that on VIPT caches (Figure 51(b)). Owing to these reasons, overall energy saving of Synergy does not change a lot either (Figure 51(c)).

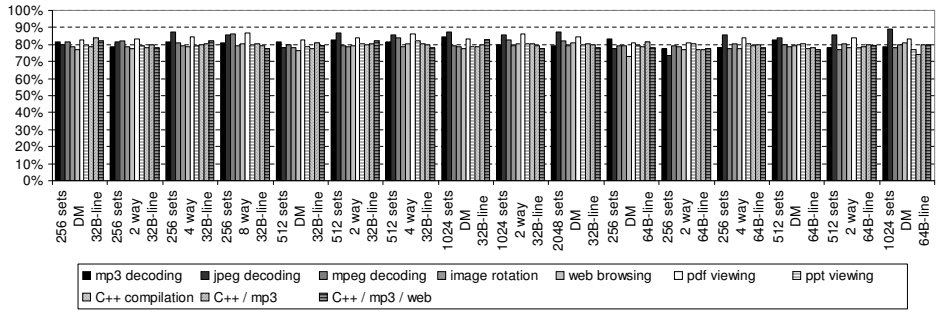
Finally, we also quantify the energy saving of overflow-free Synergy described in Section 6.4. Table 14 shows the number of bits required to build overflow-free Synergy. As shown in the table, area overhead of overflow-free Synergy usually is bigger than that of XOR-based 3-bit counter Synergy, but in some cases, e.g. 256 sets, directly-mapped 8KB cache, overflow-free Synergy is more area-efficient. Note that, in both cases, area overhead of Synergy is negligibly small compared to corresponding L1 data cache.

**Table 14. 3-bit Counter vs. Overflow-free Counter**

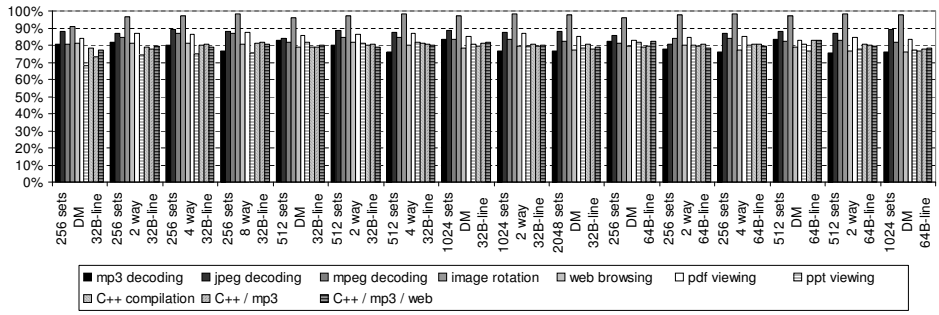
| # of sets | # of ways | Line size (B) | s-bits | Bit vector (b) | 3-bit counter array (b) | Overflow-free counter array (b) |
|---|---|---|---|---|---|---|
| 256 | 1 | 32 | 1 | 1k | 3k | 2k (2-bit counters) |
| 256 | 2 | 32 | 1 | 2k | 6k | 6k (3-bit counters) |
| 256 | 4 | 32 | 1 | 4k | 12k | 16k (4-bit counters) |
| 256 | 8 | 32 | 1 | 8k | 24k | 40k (5-bit counters) |
| 512 | 1 | 32 | 2 | 2k | 6k | 6k (3-bit counters) |
| 512 | 2 | 32 | 2 | 4k | 12k | 16k (4-bit counters) |
| 512 | 4 | 32 | 2 | 8k | 24k | 40k (5-bit counters) |
| 1024 | 1 | 32 | 3 | 4k | 12k | 16k (4-bit counters) |
| 1024 | 2 | 32 | 3 | 8k | 24k | 40k (5-bit counters) |
| 2048 | 1 | 32 | 4 | 8k | 24k | 40k (5-bit counters) |
| 256 | 1 | 64 | 2 | 1k | 3k | 3k (3-bit counters) |
| 256 | 2 | 64 | 2 | 2k | 6k | 8k (4-bit counters) |
| 256 | 4 | 64 | 2 | 4k | 12k | 20k (5-bit counters) |
| 512 | 1 | 64 | 3 | 2k | 6k | 8k (4-bit counters) |
| 512 | 2 | 64 | 3 | 4k | 12k | 20k (5-bit counters) |
| 1024 | 1 | 64 | 4 | 4k | 12k | 20k (5-bit counters) |

Since overflow-free Synergy uses several LSBs of virtual addresses excluding line offset bits as its hashing function (Section 6.4), the true miss rate of overflow-free Synergy is different from that of XOR-based 3-bit Synergy. As previously explained, true miss rate is one of the critical factors that affects overall energy saving. Figure 53(a) shows the true miss rate of normal Synergy and overflow-free Synergy. As shown in the figure, XOR-based hashing works well regardless of cache configurations and workloads, while the true miss rate of overflow-free Synergy is more dependent on cache configurations and workloads. The XOR hash function generates a more uniform distribution of indices, and eliminates aliasing of the Bloom filter indices more efficiently. However,

we should note that the true miss rate of overflow-free Synergy is as good as that of XOR-based Synergy on an average.



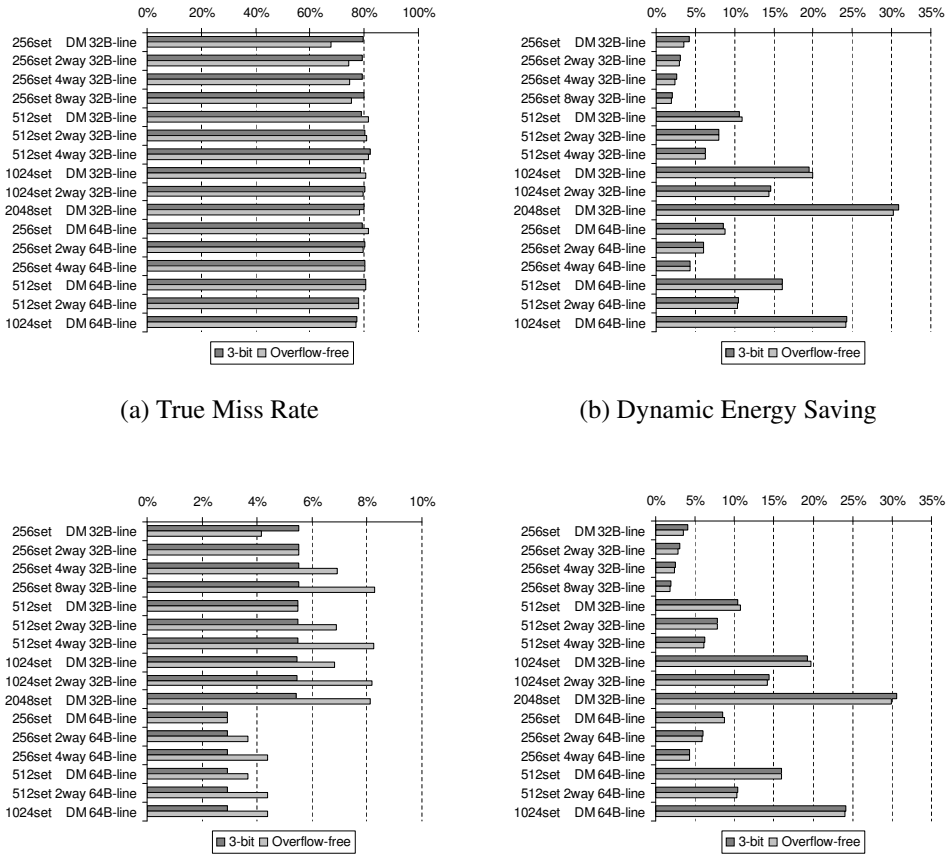

(a) XOR-based 3-bit Synergy




(b) Overflow-free Synergy

**Figure 52. True Miss Rate Comparison**

Figure 53 shows energy saving of overflow-free Synergy. For easier comparison, energy saving of XOR-based 3-bit Synergy is also shown. As shown in Figure 53(b), overflow-free Synergy works as well as XOR-based 3-bit Synergy. Although overflow-free Synergy uses bigger counters in most cases, the amount of energy savings is dominated by the true miss rate (Figure 53(a)), rather than the greater overhead of dynamic energy consumption of bigger counters. This is because power consumption of the Bloom filter is very small compared to that of the cache itself. Figure 53(c) shows leakage energy overhead of overflow-free Synergy. Clearly, in most cases, it consumes more leakage energy due to its bigger counters. However, since dynamic energy consumption of the cache dominates leakage energy consumption, overall energy saving is not affected much by the additional leakage overhead, as shown in Figure 53(d).




(a) True Miss Rate

(b) Dynamic Energy Saving

(c) Leakage Energy Overhead

(d) Overall Energy Saving

**Figure 53. Simulation Result of Overflow-free Synergy using** *PowerPoint*

### *6.5.2.1 Effect of Different Hash Functions on True Miss Rate*

Another experiment is performed to see the effect of the hash functions on the true miss rate of a counting Bloom filter. Four hash functions which can be easily implemented are evaluated. The first hash function is XOR as explained before, and the second one is using XNOR instead of XOR operations. The third hash function inverts and reverses the the bits before doing the XOR. The fourth hash function is just simply taking the least significant bits (LSBs) of the physical memory address excluding line offset bits. For example, when using the cache with 32B line, and the 4096-entry Bloom filter, this hash function takes bits between bit16 and bit5 as the Bloom filter index of this physical address. Figure 54 shows that for most of cache configurations, there is almost no appreciable difference in true miss rate among the three different hash functions. We may conclude

from this that a sufficiently random hash function, even the one using the LSBs can give very high true miss rates.

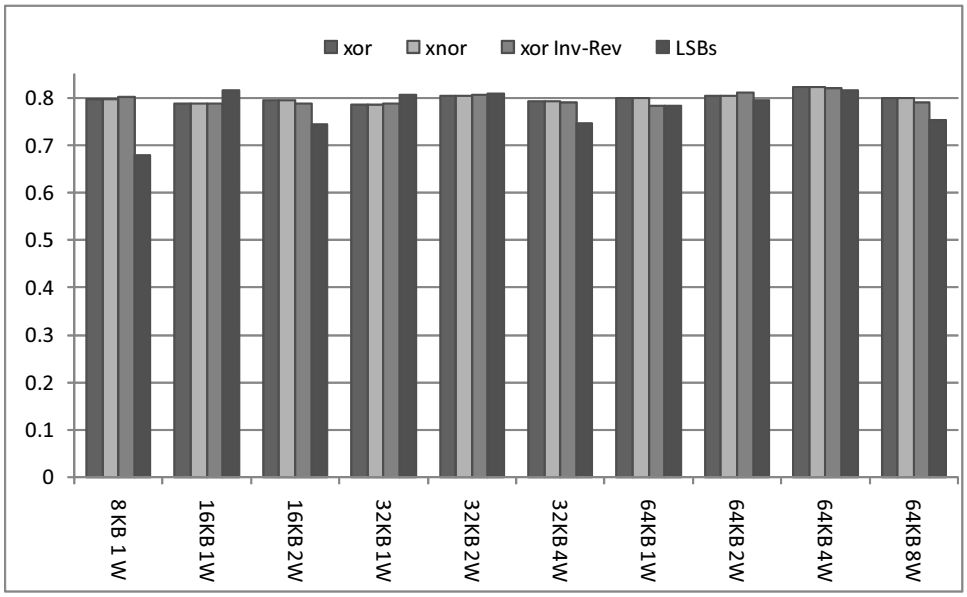



**Figure 54. True Miss Rate using Different Hashing Functions**

## 6.6 Summary

Implementing virtual caches can accelerate L1 cache accesses while introducing the synonym problem. Even though a synonym hit is an infrequent event, to guarantee correctness, the processor still needs to perform additional lookups for all possible synonym locations upon each L1 miss, thereby consuming more energy. In this chapter, we propose Synergy, an early synonym detection mechanism using counting Bloom filters. It is shown to be fast, effective, and consumes lower power. By tracking and checking the address signature in the filters, we are able to exclude unnecessary lookups for addresses that were never accessed. Furthermore, we also analyze the overflow probability of counting Bloom filters using the probability theory, and propose a novel overflow-free Bloom filter design.

To evaluate Synergy, we performed thorough simulations using different sizes of both VIPT and VIVT caches and different sizes of Bloom filters. Our full-system simulation results show that

Synergy can effectively reduced the total cache accesses by up to 40.7%. The dynamic energy consumption and the overall energy consumption (including leakage energy) in the L1 can be reduced by up to 38.9% and 38.2%, respectively.

# CHAPTER 7

# REDUCING DYNAMIC ENERGY CONSUMPTION OF CACHES WITH COOLPRESSION

Continuous shrinking of transistor feature size and demands of the working set size from increasingly complex applications has led to ever-larger on-chip cache design with a slew of read/write ports making it a major consumer of on-chip power. A significant part of the cache energy is drawn by the bitline driver circuitry because the bitlines are densely loaded with a large number of storage cells thus increasing its effective switching capacitance. The focus of this chapter is to identify redundancy in data values stored in caches to reduce the energy consumed in precharging bitlines during cache operation.

In order to identify redundancy in data values stored in level one caches, we performed data value profiling for a large number of workloads, such as, the SPECint2000 and Mediabench benchmarks and observed that the data values entering the cache consists of long sequence of leading zeros and leading ones in the significance bits. On identification of the redundancy we came up with a novel significance compression technique. The basic idea is that, instead of enabling all the bitlines, the homogeneous data are compressed to a more compact form and only the bitlines representing the compact data will be enabled during cache accesses.

We propose *CoolPression*, a hybrid significance compression technique by using Villa's DZC as a basis along with a novel technique called *CoolCount*, and then dynamically determine the more energy-efficient way to minimize the number of instances of driving data bitlines. The CoolPression circuitry monitors all accesses to the cache and compresses/encodes any data written to the cache, using either of the two compression schemes according to the compressibility of the given data. An extra bit is used to indicate which compression scheme was used. For every read it decodes the data before sending it. The CoolCount technique uses a novel priority encoder and XOR gates, and can count both leading ones and leading zeroes. The novelty of the counting technique lies in the fact that this scheme compresses information[1] in the granularity of bits, while all prior schemes applied compression in the granularity of bytes at best, losing the saving opportunities across byte

[1]Information hereafter represents both instructions and data in general.

boundaries. In addition, CoolCount is able to exploit data with leading ones, primarily the small negative integer numbers. Reusing the most significant byte for book-keeping purposes also avoids the area overheads, thus no extra dynamic and leakage energy is consumed. As shown in our experiments, our compression scheme can save 35% of the cache energy on an average over a baseline cache. As the cache size keeps increasing, the CoolPression will demonstrate even more benefits.

## 7.1 Redundant Leading Zeroes and Ones

It has been shown in [37] that more than 70% of bits read from or written to the cache are all zeroes. Also more than 75% of the values used are rather small, having a large number of leading ones or zeroes.

As shown in Figure 55, we conducted a study on the data accesses for SPECint2000 benchmark to profile how many data accesses to the Dcache has "x" (1< "x" < 64) number of leading zeroes and ones. In the two figures, Y-axis plots the number of leading zeroes (or ones) from 1 to 64 while the X-axis shows the number of instances. The triangle in the plots represents the average number of instances. Each vertical bar shows the range of the number of instances from the 8 SPECint2000 integer benchmark programs. As shown in Figure 55(a), the average number of times we access a piece of data which has "x" number of leading zeroes is quite uniform across the board, for 1<x<64. A similar trend is also observed for the number of leading ones as shown in Figure 55(b). The encoding techniques proposed in [70, 114], are unable to adequately capture instances in which leading zeroes are not in multiples of 8. Therefore, we will be losing a lot of energy saving opportunities if we only consider compressing data in the granularity of bytes rather than in bits. Based on this analysis, we introduce a compression scheme where we count the leading bits, and keep the count instead of all the bits. We would need 6 counting bits for counting 64 bits of data. We also need another bit indicating whether we counted leading ones or zeroes. If we add 7 bits for every 64 bits of data in the cache, our area overhead would become prohibitively large. We thus propose to reuse the most significant byte of the data to keep the count. Using this method we would need to have only one extra bit for every 64 bits to indicate whether we have employed the counting scheme. If the data being accessed has "count" number of leading zeroes or ones, we

need to enable only 64 minus count bitlines to read or write the actual data and append them with leading zeroes or ones. We detail our approach in the following section.

## 7.2 CoolPression Cache

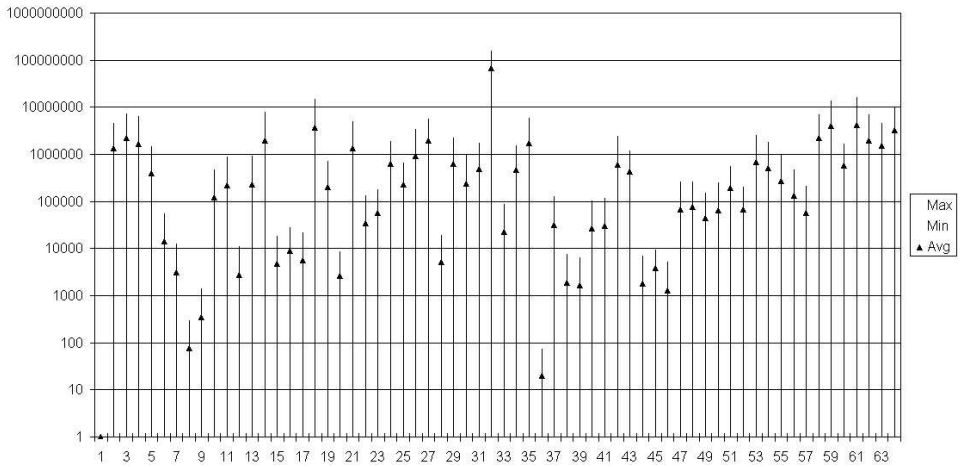
The CoolPression cache is illustrated in Figure 56. It employs two compression schemes — Dynamic zero compression to capture the zero bytes, and a new CoolCount technique to exploit compression opportunities in bit-level granularity. To explain our approach we would first explain each one individually before we demonstrate the hybrid approach.

### 7.2.1 Dynamic Zero Compression (DZC)

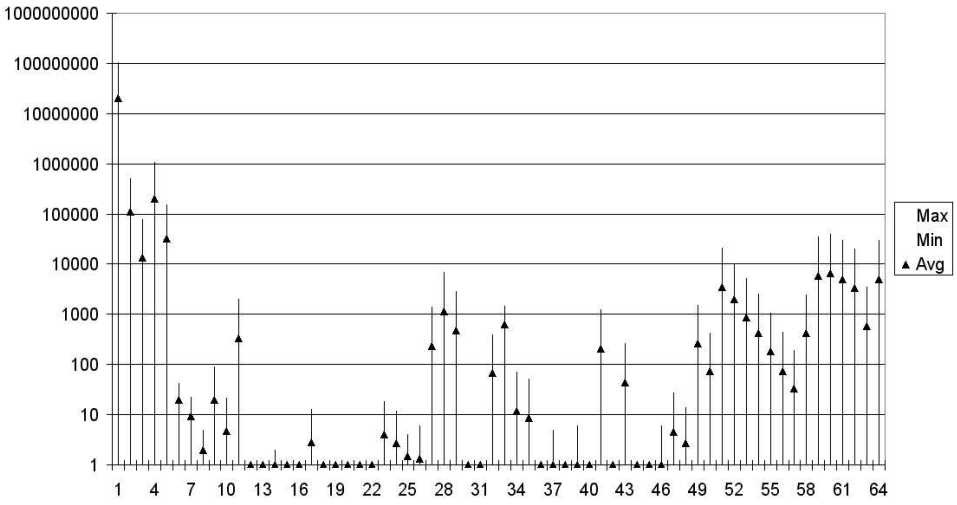
The DZC in [114] uses an extra bit, known as the Zero Indicator Bit (ZIB), for each data byte in the cache. On every data write, it is checked whether any of the eight data bits being written are all 0's, if so, the ZIB is enabled and the write for the eight bits is disabled. If the data bits are not all zeroes then the ZIB is cleared and the data is written to the cache as normal. On a cache read, if the ZIB for a byte is enabled, the corresponding bit-lines are gated off and a zero byte is emitted, through a bank of NOR gates. If the ZIB is zero for a byte then a normal cache read operation occurs.

### 7.2.2 CoolCount

Our proposed technique, *CoolCount*, counts the number of leading 0's or 1's and reuses the most significant byte to record the count. On a cache write the CoolCount circuit counts the number of leading 0's or 1's. If the count is more than eight, it asserts the Count Enable (CE) bit high. The most significant bit is used to store whether we counted leading 0's or 1's. The next six bits from the most significant byte are used to keep the count. Along with this, (64 - count) least significant bitlines are enabled to store the actual data. If the number of leading 0's or 1's is lower than eight, the cache performs a normal write. The cache read is illustrated in Figure 57(a). Reading is a two step but pipelined process. On a cache read if the CE bit is enabled, the most significant 6 bits are read to get the number of leading 0's or 1's. Next the least significant bit-lines are enabled to get the actual data appended with the leading bits to obtain the final data. However if the CE bit is zero, the CoolCount cache behaves exactly like a normal cache for a read.




(a) Number of Leading Zeroes.




(b) Number of Leading Ones.

**Figure 55. Leading 0's and 1's for SPECint2000.**

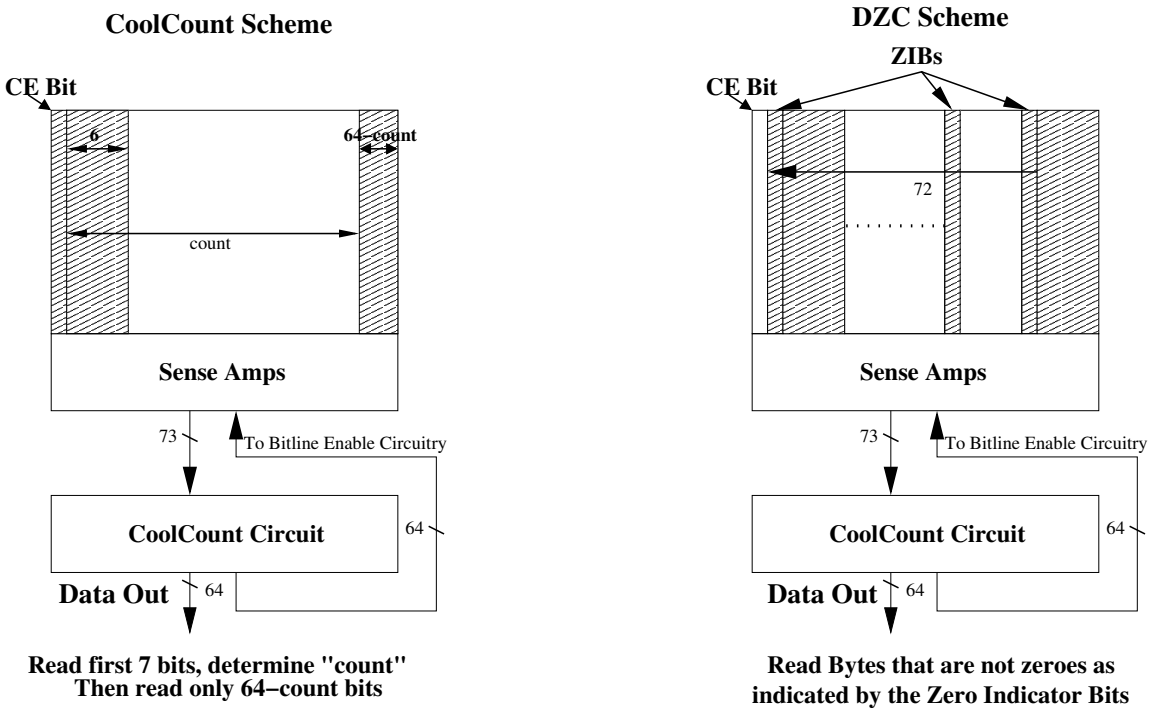


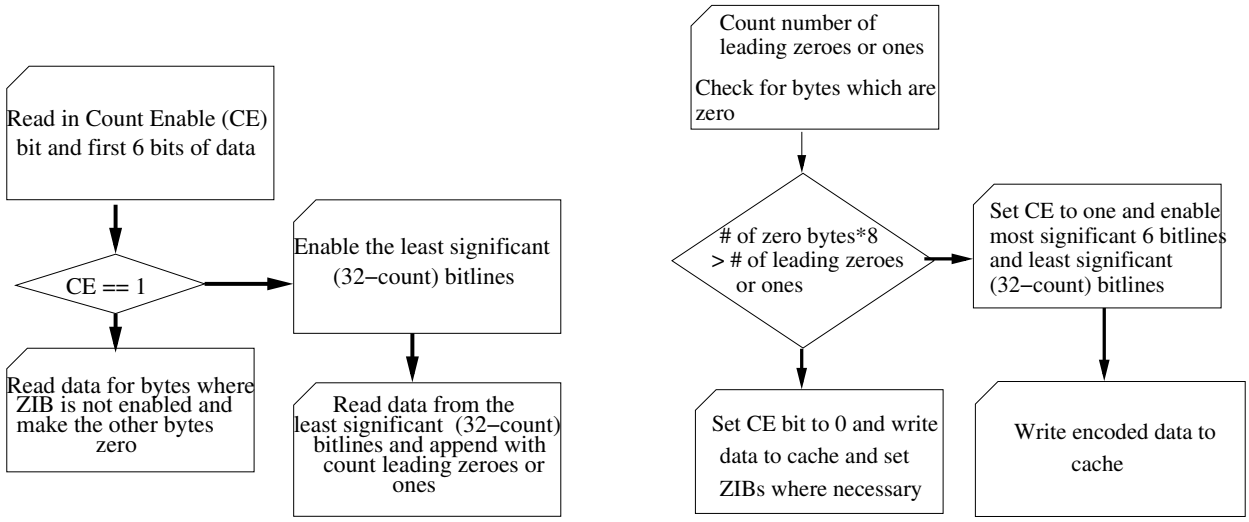

**Figure 56. CoolPression Cache.**

### 7.2.3 Hybrid Compression Scheme

Our evaluation indicated that both techniques can lead to significant energy savings when operated independently. However, there are certain cases where the CoolCount scheme outperforms the DZC and in some cases the DZC does better. For instance, CoolCount can capture energy saving opportunities at finer granularity while DZC can exploit the hidden opportunities where zero bytes are embedded in the middle of a data word. Based on this observation, we propose a hybrid compression scheme which employs both DZC and CoolCount and exploits all possible opportunities in a dynamic manner.

The operations occurring in the CoolPression cache for reads and writes are illustrated by the flow-charts in Figure 57(a) and Figure 57.

On each cache write, one circuit will count the leading 0's or 1's, while another circuit counts how many bytes of the data are zero. These circuits and their energy impact are detailed in Section 7.3. The results are compared to determine which scheme is better. If the CoolCount is better, the CE bit is turned on, all the ZIB's are cleared and the CoolCount scheme is followed. If DZC is found better, the CE bit is cleared, the corresponding ZIB's are enabled and the DZC is used. For reading data from the cache, after address decoding is completed, the CE bit, the most significant




(a) Read Data from Caches.

(b) Write Data to Cache.

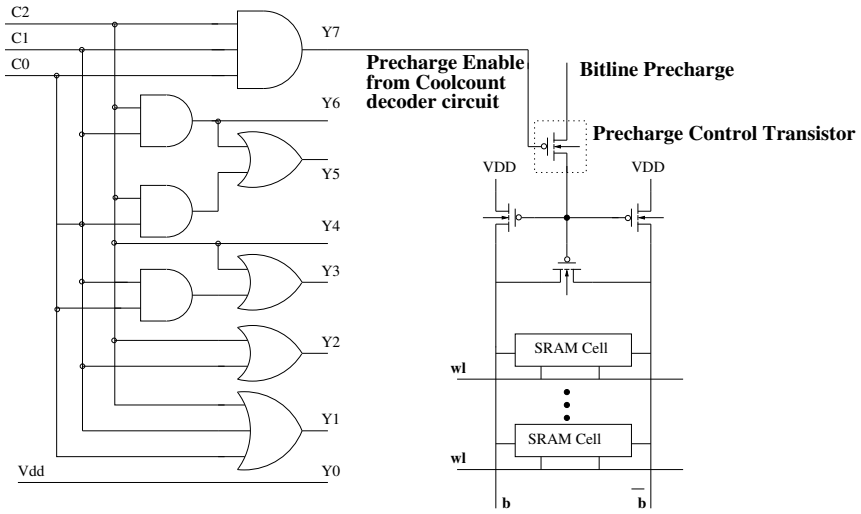
**Figure 57. Flowchart for Reading and Writing Data to the CoolPression Cache.**

6 bits and the ZIBs are read in the first cycle of the cache read. The CE bit enables the Coolcount decoder shown in Figure 58 which uses the *count* bits to precharge the least significant *64 - count* bits before the start of the second cycle. If the CE bit is disabled, the ZIBs are used to precharge only the relevant *"byte"* lines. The precharged bit lines are read in after the end of the second cycle and properly appended with zero or one bits to form the final data.

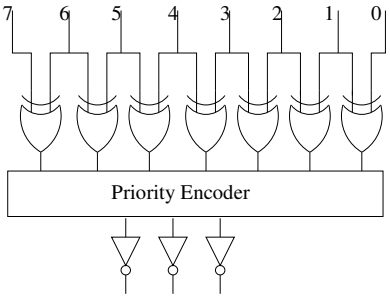
## 7.3 CoolPression Implementation

Figure 58 illustrates the schematic of our logic circuits for an efficient implementation for counting the leading 0's or 1's. For illustration purposes, the circuit shown in Figure 58(b) is a stripped-down version using a 8-to-3-line priority encoder for an 8-bit data set. Depending on the data size supported by the targeted ISA, for example, the CoolCount circuit will have a 64-to-6-line priority encoder for a 64-bit data set. Note that as shown in the schematic, the adjacent bits of the data are XORed together and fed into the priority encoder. This design is to determine when the data first changes from 0 to 1 or from 1 to 0, as we scan the bits from the most significant bit. The position of this bit is reported by the priority encoder. The number of leading zeroes or ones is obtained by negating the above result.

Figure 58(a) illustrates the decoding circuits to enable only the required bitlines depending on




(a) Decoder and Precharge Control Circuit.




(b) Encoder.

**Figure 58. Decoding/Encoding Logic Circuits.**

the number of leading 0's or 1's. The circuit illustrated is again for 8-bit data only. The CoolCount circuits will be scaled for a 64 bit data set. The output of the decoding circuit are used to disable the Bitline Precharge signal using the Precharge Control Transistor as shown in the Figure 58(a). This mechanism allows only the required bitlines to precharge, when reading or writing compressed lines to the cache. We explain the overheads in terms of delay and extra power consumed for the CoolPression circuit in the following sub-section.

### 7.3.1 Overheads

**Delay Overheads:** To consider the effect of delay overheads we note that the decoding circuit

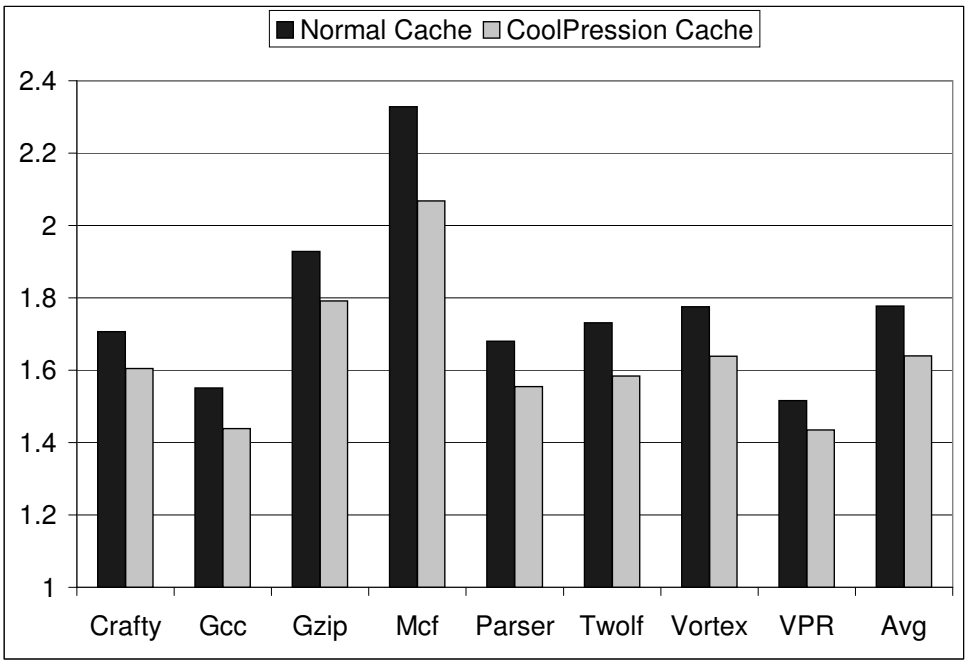



**Figure 59. Impact on IPC.**

Figure 58(a) is extremely simple, a tiny overhead compared to the cache array, and the delay associated with it is minimal. Since reading is a two stage process as shown in Figure 57(a), we assume that a L1 cache read would take 2 cycles instead of one, even though we could have headroom to customize the entire read into one cycle for a lower frequency processor. The 2 cycle latency can be easily pipelined by dividing the cache access into address decoding and data transfer stage. This disadvantage is also omnipresent in other cache compression schemes [70, 114]. This is an indispensable overhead in any cache compression since extra time is always needed for ascertaining whether the data were compressed prior to being stored. Figure 14 studies the effect if a 2-cycle pipelined Instruction and Data Cache is to be designed against a single cycle uncompressed cache. It is observed that the average IPC can be degraded by 7.8% for a 35% savings in energy consumption of the cache. The processor architects have to weigh the trade-off when making such a design decision. The delay associated with the Priority Encoder, is in the order of 6 gate delays. Nevertheless, since the delay is only associated with a write and not a read, it will be hidden in practice in the interval between a write and the next read of the same cache line.

**Power Overheads:** The majority of the delay and energy consumption associated with the CoolCount circuit is caused by the Priority Encoder (PE). The PE design used for this study is

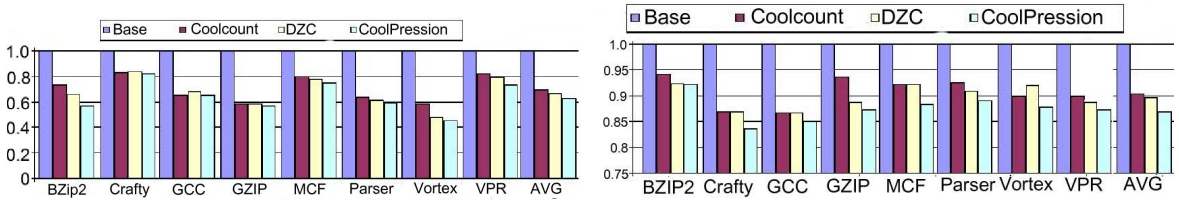
an energy efficient high speed PE design taken from [117]. The reported power numbers have been scaled down to the current process technology parameters using standard technology scaling rules [80]. The cache energy consumption numbers were reported using Wattch. All numbers cited are for $0.1\mu$m technology. The CoolPression circuit only consumes 0.1% of the 64KB cache power as reported by Wattch.

## 7.4 Energy Savings and Performance Implications of Coolpression

We integrated the CoolPression technique into SimpleScalar to quantify the energy savings. The processor model is similar to an Alpha 21264. Wattch and Cacti were used to model total cache energy dissipation in each application and the energy consumed in each cache array. A two level cache was used and our technique was applied to both levels. We simulated SPECint2000, each for 1 billion instructions.

Figure 60(a) shows the energy consumption in a 16K direct mapped L1 data cache using the CoolCount, DZC and the hybrid CoolPression schemes, taking a word size of 64 bits. All the results are normalized to the baseline cache with no compression. We observe that the DZC and CoolCount are on par within 3% savings, and there is no obvious trend as to which one is better. Since the CoolPression scheme can dynamically choose the better technique, the energy savings are better for CoolPression. The CoolPression beats the DZC from 3 to 15%. In overall, the CoolPression provides an energy savings of 36% over the baseline cache. Figure 60(b) illustrates a similar analysis on a 16K L1 Instruction cache which shows that the CoolPression generates considerable savings even though it is not as significant as the data cache. One reason for lower energy savings in the Icache is that the data in the Icache consists of instructions whose encodings would have a higher entropy in terms of the distribution of 0's and 1's, while data, mostly, show a large number of small positive and negative numbers, i.e., many consecutive 0's and 1's, for integer programs. Our technique did not show any substantial improvement for the unified L2 cache since there are very few accesses to the L2 cache in the benchmarks we studied, and the L2 energy consumption is largely dominated by the leakage energy due to its size.

In Figure 61, we plot the energy consumption for 32 KB and 64KB L1 Dcache and Icache, showing CoolPression reduces cache energy by as much as 50%. It is observed that the CoolPression




(a) Norm. Energy in a 16KB L1 D-Cache.

(b) Norm. Energy in a 16KB L1 I-Cache.

**Figure 60. Norm. Energy in a 16KB L1 D- and I-Cache.**

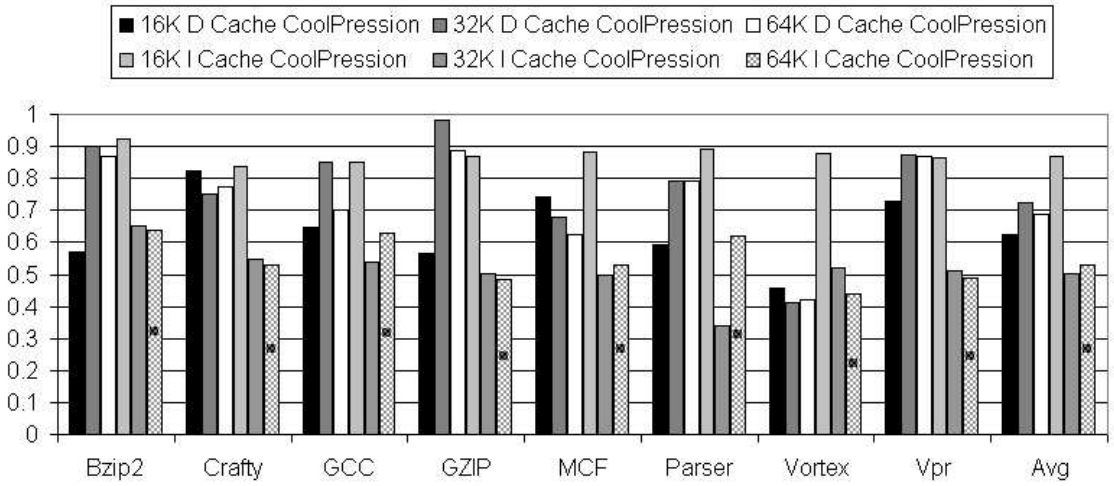



**Figure 61. Norm. Energy in a 32k/64k L1 I/D-Cache.**

cache gives substantial energy savings for the larger caches. In addition, what we do not show due to space contraint is that the CoolPression also consistently provides more saving than the DZC and the CoolCount when applied alone. The trend is similar to what was shown in Figure 60(a) and Figure 60(b).

## 7.5 Summary

In this chapter, we presented *CoolPression*, a hybrid hardware-based data compression mechanism that reduced energy consumption in caches by eliminating unnecessary bitline switchings. CoolPression consists of a dynamic zero compression technique with a novel CoolCount scheme to exploit both byte and bit-level compressibility. The CoolPression scheme is system transparent in the sense that the rest of the system does not change their interface with the CoolPression cache. The CoolCount circuitry used a small amount of hardware to encode data stored in the cache, and

succeeded in reducing the energy consumption by a significant amount. Based on our simulation results using SPECint2000, the CoolPression improves dynamic energy consumption by more than 35% against a baseline cache, while having energy savings ranging from 5 to 15% compared to the dynamic zero compression or CoolCount applied individually.

The novel encoding scheme discussed in this chapter may be extended to save energy at all places wherever data is being transferred. Some important locations that may be considered are the pipeline latches. We may consider using this technique for data transfer from L2 cache to memory and memory to the disk. Effects of reducing the bus transfer power at certain places, where we would encode the data using this scheme and transfer only the required data, gating-off the other bits may also be considered.

Although dynamic power consumption is an important problem for L1 Caches, a major concern for modern microprocessor designers is the leakage power consumption of L2 or higher-level caches. The following chapter describes in detail a leakage power reduction technique especially suited for multiprocessor systems.

# CHAPTER 8

# CONCLUSIONS

This dissertation presented several techniques to identify redundancy in several aspects of memory operation and eliminate those redundancies to save energy and improve performance. Most of the techniques involved introduction of lightweight hardware structures in the memory hierarchy to help capture and eliminate redundant memory operations. In this chapter, we summarize the contributions made by this dissertation, analyze their impact in modern computing platforms, and enumerate a few areas where this work may be extended.

## 8.1 Summary of Contributions

First, we identified redundancy in the DRAM refresh operation and presented a simple, low cost technique using time-out counters to eliminate the redundancies and save power in DRAMs. This technique does not involve any change in the interface between the memory controller and the DRAM, making it highly feasible. All additional hardware goes in the memory controller that controls and issues the needed refresh operations. The work demonstrates that many refresh transactions are indeed redundant for their corresponding rows were recently accessed due to cache misses. This technique saved up to 25% and on an average 12.13% of the energy consumed in DRAMs. Modern computing systems like CMP, CMT, SMP and SMT would try to exploit MLP and would have increasing number of threads trying to access memory. In this case, the Smart Refresh technique will be instrumental in saving energy as it is very light-weight and would increase the bandwidth availability and reduce energy consumption for refresh operations in DRAMs. The emerging 3D die stacked ICs will enable the accesses to the DRAM at a much lower latency. Also, AMD's licensing of ZRAM technology indicates that future AMD processors may use DRAM type memory using SOI technology for their caches. Apart from having redundancy in DRAM refresh operations, the memory hierarchy also has redundancy in the way data is stored in the caches.

The next contribution of this dissertation was to identify redundant data in caches supporting muli-level-inclusion. We presented a new, low-overhead architectural technique called Virtual-Exclusion to save leakage energy in higher level caches that simultaneously provided guaranteed

Multi-Level Inclusion property for correct operations of cache coherence protocols and saved leakage energy more effectively. Our technique showed that a significant leakage energy savings of up to 46% in an 8-processor SMP and 35% for an 8-way multicore architecture can be achieved. We envision that such a practical and easy-to-implement technique will be very useful in saving leakage energy for the cache-coherent multicore, multiprocessor systems.

The techniques explained in the previous two paragraphs exploited different forms of redundancies in the memory hierarchy to reduce energy consumption. For DRAMs, redundant refresh operations were eliminated to reduce DRAM power. In a cache, redundancies were identified in the data storage, the Multi-Level Inclusion policy, and the snooping protocol to reduce dynamic and leakage power. The subsequent techniques, in contrast, add a new hardware structure called the "counting Bloom filter"(CBF) to the memory hierarchy. The CBF in contrast to the previously employed techniques exploits redundancies in the method of accessing a memory location in conventional caches. This is the main emphasis of the authors research, and the following paragraphs summarize the contributions of using counting Bloom filters in reducing cache energy.

Counting Bloom filters provide a signature of the cache and prevent redundant cache lookups. In this thesis we presented a new segmented design for the counting Bloom filters to perform energy management at the microarchitectural level and evaluates its effectiveness in reducing energy. As shown in our experiments, the segmented Bloom filter technique is an efficient microarchitectural mechanism for reducing the total processor energy consumption. A significant part of the total processor energy including L2 dynamic cache energy, L1, L2 and processor static energy can be saved in a system where the multi-level cache hierarchy assumed does not maintain inclusion property. Also,the segmented design is shown to provide even higher energy-efficiency if the multi-level cache hierarchy implements inclusive behavior. This is because the segmented design provides the opportunity to make the bit vector for the L2 Cache accessible before the L1 cache access and allows for detection of misses much earlier in the memory hierarchy. The segmented counting Bloom filter is capable of filtering out more than 89% of the L2 misses, causing a 30% reduction in accesses to the L2 cache. This results in a saving of more than 33% of L2 dynamic energy. The results also demonstrated that the overall system energy can be reduced by up to 9% using the proposed segmented Bloom filter.

We also demonstrated that the segmented Bloom filter can be efficiently used as a way estimation technique and saves much more energy than the prior Way Halting technique. We showed that our technique can be efficiently used in all levels of the cache hierarchy obtaining substantial energy savings of up to 70% using Way Guard in both instruction and data L1 caches, and up to 65% for an unified L2 cache.

As future applications demand more memory and shrinking feature sizes allow more one-die transistors, processors would be inclined to have larger caches with higher associativity. Having these longer latency, higher associative caches will provide further opportunities for the segmented design to facilitate microarchitectural energy management earlier in the memory hierarchy and the Way Guard technique to save lookup energy. Therefore, cache miss detection and way estimation techniques in general and the segmented filter design presented in this thesis will play a key role in energy management for future microprocessors.

Other than estimating ways and predicting misses CBFs can also be used to detect synonyms in virtual caches. Implementing virtual caches can accelerate L1 cache accesses while introducing the synonym problem. Even though a synonym hit is an infrequent event, to guarantee correctness, the processor still needs to perform additional lookups for all possible synonym locations upon each L1 miss, thereby consuming more energy. In this thesis, we proposed Synergy, an early synonym detection mechanism using counting Bloom filters. It was shown to be fast, effective, and consumed lower energy. By tracking and checking the address signature in the CBFs, we were able to exclude unnecessary lookups for addresses that were never accessed. Furthermore, we also analyzed the overflow probability of counting Bloom filters using the probability theory, and proposed a novel overflow-free Bloom filter design.

To evaluate Synergy, we performed thorough simulations using different sizes of both VIPT and VIVT caches and different sizes of Bloom filters. Our full-system simulation results show that Synergy can effectively reduce the total cache accesses by up to 40.7%. The dynamic energy consumption and the overall energy consumption (including leakage energy) in the L1 can be reduced by up to 38.9% and 38.2%, respectively.

The final contribution of this dissertation was to identify redundancy in data values moving in the memory hierarchy. To eliminate the redundancy in data values, we present *CoolPression*, a

hybrid hardware-based data compression mechanism that reduces energy consumption in caches by eliminating unnecessary bitline switchings. CoolPression consists of a dynamic zero compression technique with a novel CoolCount scheme to exploit both byte and bit-level compressibility. Like the Smart Refresh technique, CoolPression is also system transparent in the sense that the rest of the system does not change their interface with the CoolPression cache. The CoolCount circuitry used a small amount of hardware to encode data stored in the cache, and succeeded in reducing the energy consumption by a significant amount. Based on our simulation results using SPECint2000, the CoolPression improves dynamic energy consumption by more than 35% against a baseline cache, while having energy savings ranging from 5 to 15% compared to the dynamic zero compression or CoolCount applied individually. The novel encoding scheme discussed in this dissertation may be extended to save energy at all places wherever data is being transferred. Some important locations that may be considered are the pipeline latches. We may consider using this technique for data transfer from L2 cache to memory and memory to the disk. Effects of reducing the bus transfer power at certain places, where we would encode the data using this scheme and transfer only the required data, gating-off the other bits may also be considered.

## 8.2 Future Work

As processor designers continue to add more cores in a die, the major bottleneck to proper utilization of the available computing resources will be the interconnect among the cores and also between the cores and off-chip memory. The emerging many core processor is likely to have a hierarchical interconnect design. At the top level, the many core processor will have a mesh network similar to [111] [55] among clusters of processing elements. Each processing element will have a set of two to four cores connected by a high speed bus or crossbar. Each core will have its local L1 and L2 caches. The off-chip communication of such this many core die will be using the traditional I/O pins at the chip's periphery and also the through silicon vias (TSVs) to communicate with a 3D die stacked DRAM. In this section, we present a number of aspects of the research presented in this dissertation that will be useful in improving performance and saving energy in the many core processor of the future.

The refresh operation will become a major overhead for 3D die stacked DRAMs. The Smart

Refresh technique will be invaluable to increase availability and reducing power of such systems. Modern DRAM systems have temperature compensated self refresh mechanism, that adjusts the refresh rate automatically with changes in DRAM temperature. The Smart Refresh scheme can be adapted for DRAMs having TCSR by having a configuration register that is set to reflect the current data retention deadline. A simple logic circuit will determine the frequency of counter updates based on the data retention deadline set at the configuration register.

To reduce coherency communication between cores in a many core processor, the caches will be inclusive [13]. Since a very large portion of the many core chip will be SRAM caches, the Virtual Exclusion scheme explained in this dissertation will have significant opportunity to save leakage energy.

The bandwidth of the interconnect in future many core systems will be a major performance bottleneck. As this many core processor is likely to have a shared last level cache, the segmented bloom filter may be used by individual cores to predict misses to the last level cache. Once a miss is predicted the data will be accessed directly from memory bypassing the last level cache. This will significantly reduce traffic in the interconnect between the cores and the shared last level cache. Another opportunity to reduce traffic in the interconnect will be to use compression techniques like CoolPression for all data transfers in the interconnect.

This dissertation presented several techniques to demonstrate that using lightweight hardware structures for dynamic profiling of the memory reference stream can improve energy and performance in the memory hierarchy. The significant energy and performance improvements demonstrated by the techniques presented here suggest that this work will be of great value for designing future computing platforms.

# REFERENCES


[1] 128Mb: x32 SDRAM data sheet. http://download.micron.com/pdf/datasheets/dram/sdram/128MbSDRAMx32.pdf Date: May 2007.

[2] 16 M-Word by 64-Bit DDR Synchronous Dynamic RAM Module Unbuffered Type Specification. http://www.nec.com Date: May 2007.

[3] 512Mb D-Die DDR SDRAM Specification. http://www.samsung.com Date: May 2007.

[4] Advanced Power State Management. http://www.rambus.com/us/patents/innovations/detail/apsm.html Date: March 2009 .

[5] AMD licenses Innovative Silicon's SOI memory. http://www.eetimes.com/news/latest/showArticle.jhtml? articleID=177101749 Date: January 2007.

[6] Cortex A8 Technical Reference Manual. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0344c/index.html Date: March 2009 .

[7] DRAM Power Saving Features. http://download.micron.com/pdf/technotes/tn4612.pdf Date: March 2009.

[8] Easy Integration of Embedded DRAMs. http://www.am.necel.com/process/ edram-sequences.html Date: March 2009.

[9] Intel 855PM Chipset Memory Controller Hub (MCH) DDR datasheet. ftp://download.intel.com/design/chipsets/datashts/25261303.pdf.

[10] Intel 855PM Chipset Platform Design Guide. http://download.intel.com/design/mobile/desguide/25261403.pdf Date: November 2005.

[11] ITRS Roadmap 2006 Interconnect Update. http://www.itrs.net/Links/2006Update/FinalToPost/09_Interconnect2006Update.pdf Date: January 2007.

[12] Micron DDR2 SDRAM Registererd DIMM 2GB and 4GB data sheet. http://download.micron.com/pdf/datasheets/ modules/ddr2/HTF36C256_512x72.pdf Date: April 2007.

[13] Next Generation Intel Microarchitecture(Nehalem). http://www.intel.com/technology/ architecture-silicon/next-gen/whitepaper.pdf Date: April 2009.

[14] Samgsung Develops 3D Memory Package that Greatly Improves Performance Using Less Space. http://www.samsung.com/PressCenter/PressRelease/PressRelease.asp?seq=20060413_0000246668 Date: May 2007.

[15] Sun's Niagara Pours on the Cores. http://www.mdronline.com/mpr/h/2004/0913/183702.html Date: August 2005.

[16] Tezzaron Semiconductor, FaStack Technology. http://www.tezzaron.com/technology/FaStack.htm Date: May 2007.

[17] Advanced Micro Devices, Inc. *AMD Athlon Processor Architecture*, 2000.

[18] Nidhi Aggarwal, Jason F. Cantin, Mikko H. Lipasti, and James E. Smith. Power-efficient dram speculation. In *14th International Conference on High-Performance Computer Architecture (HPCA-14), 16-20 February 2008, Salt Lake City, UT, USA*, pages 317–328.

[19] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proceedings of the 36th International Symposium for Microarchitecture*, December 2003.

[20] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *Proceedings of IEEE/ACM 36th International Symposium on Microarchitecture*, pages 423–434, 2003.

[21] K. Albayraktaroglu, A. Jaleel, X. Wu, M. Franklin, B. Jacob, C.-W. Tseng, and D. Yeung. Biobench: A benchmark suite of bioinformatics applications. In *Proceedings of the 2005 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2005)*, pages 2–9, 2005.

[22] B. Amrutur and M. Horowitz. Techniques to Reduce Power in Fast Wide Memories. In *Proceedings of the International Symposium on Low Power Electronics*, pages 92–93, October 1994.

[23] Todd M. Austin. Simplescalar tool suite. http:/www.simplescalar.com Date: October 2003.

[24] J.-L. Baer and W.-H. Wang. On the Inclusion Properties for Multi-Level Cache Hierarchies. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 73–80, 1988.

[25] B. Batson and TN Vijaykumar. Reactive-Associative Caches. In *PACT '01: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, September*, 2001.

[26] N. Binkert, E. Hallnor, and S. Reinhardt. Network-Oriented Full-System Simulation using M5. In *Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads*, 2003.

[27] Bryan Black, Murali Annavaram, Ned Brekelbaum, John DeVale, Lei Jiang, Gabriel H. Loh, Don McCaule, Pat Morrow, Donald W. Nelson, Daniel Pantuso, Paul Reed, Jeff Rupley, Sadasivan Shankar, John Shen, and Clair Webb. Die stacking (3d) microarchitecture. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 469–479, 2006.

[28] Bryan Black, Donald W. Nelson, Clair Webb, and Nick Samra. 3D Processing Technology and Its Impact on iA32 Microprocessors. In *Proceedings of the 2004 IEEE International Conference on Computer Design*, pages 316–318, 2004.

[29] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(4), July 1970.

[30] A. Border and M. Mitzenmacher. Network application of bloom filters: A Survey. In *40th Annual Allerton Conference on Communication, Control, and Computing*, 2002.

[31] B. Calder, D. Grunwald, and J. Emer. Predictive sequential associative cache. In *Proceedings of the Second Annual Symposium on High Performance Computer Architecture*, pages 244–253, 1996.

[32] Ramon Canal, Antonio Gonzalez, and James E. Smith. Very low power pipelines using significance compression. In *Proceedings of the 33rd annual ACM/IEEE International Symposium on Microarchitecture*, pages 181–190. ACM Press, 2000.

[33] F. Catthoor. *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design*. Kluwer Academic Publishers, 1998.

[34] M. Cekleov and M. Dubois. Virtual-Address Caches Part 1: Problems and Solutions in Uniprocessors. *IEEE Micro*, 17(5):64–71, 1997.

[35] F. Chang, W. Feng, and K. Li. Approximate caches for packet classification. In *INFOCOM 2004: Proceedings of the Twenty-Third Conference of the IEEE Communications Society*, volume 4, pages 2196–2207, March 2004.

[36] Yen-Jen Chang, Shanq-Jang Ruan, and Feipei Lai. Sentry tag: An efficient filter scheme for low power cache. In Feipei Lai and John Morris, editors, *Seventh Asia-Pacific Computer Systems Architectures Conference (ACSAC2002)*, Melbourne, Australia, 2002. ACS.

[37] Yen-Jen Chang, Chia-Lin Yang, and Feipei Lai. A Power-Aware SWDR Cell for Reducing Cache Write Power. In *Proceedings of the 2003 International Symposium on Low Power Electronics and Design*, pages 14–17, August 2003.

[38] Zeshan Chishti, Michael D. Powell, and T. N. Vijaykumar. Distance Associativity for High-Performance Energy-Efficient Non-Uniform Cache Architectures. In *Proceedings of IEEE/ACM 36th International Symposium on Microarchitecture*, pages 55–66, 2003.

[39] S. Cohen and Y. Matias. Spectral bloom filters. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, 2003.

[40] Compaq Computer. *Alpha 21264 Microprocessor Hardware Reference Manual*.

[41] V. Delaluz, N. Vijaykrishnan, A. Sivasubramaniam, and M. J. Irwin. Hardware and software techniques for controlling dram power modes. *IEEE Transactions on Computers*, 50:1154–1173, 2001.

[42] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. Deep Packet Inspection using Parallel Bloom Filters. In *Hot Interconnects 12: Proceedings of the 12th Annual IEEE symposium on High Performance Interconnects*, pages 44–51, 2003.

[43] Philip George Emma, William Robert Reohr, and Li-Kong Wang. Restore Tracking System for DRAM, U.S Patent No 6,839,505 B1, 2002.

[44] D. Fan, Z. Tang, H. Huang, and G. R. Gao. An energy efficient tlb design methodology. In *Proceedings of the International Symposium on Low Power Electronics and Design*, 2005.

[45] L. Fan, P. Cao, J. Almeida, and A. Broder. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[46] 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*, 2002.

[47] B. Ganesh, A. Jaleel, D. Wang, and B. Jacob. Fully-Buffered DIMM Memory Architectures: Understanding Mechanisms, Overheads and Scaling. *Proceedings of the 13th International Symposium on High Performance Computer Architecture*, 2007.

[48] G. Gerosa, S. Curtis, M. D'Addeo, B. Jiang, B. Kuttanna, F. Merchant, B. Patel, M. Taufique, H. Samarchi, and A. Intel. A Sub-1W to 2W Low-Power IA Processor for Mobile Internet Devices and Ultra-Mobile PCs in 45nm Hi-K Metal Gate CMOS. In *Proceedings of IEEE International Solid-State Circuits Conference*, pages 256–258, 2008.

[49] Kanad Ghose and Milind B. Kamble. Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line segmentation. In *Proceedings of the 1999 International Symposium on Low Power Electronics and Design*, pages 70–75, 1999.

[50] Mrinmoy Ghosh and Hsien-Hsin S. Lee. Smart Refresh: An Enhanced Memory Controller Design for Reducing Energy in Conventional and 3D Die-Stacked DRAMs. In *Proceedings of IEEE/ACM 40th International Symposium on Microarchitecture*, pages 134–145, November 2007.

[51] Mrinmoy Ghosh and Hsien-Hsin S. Lee. Virtual Exclusion: An architectural approach to reducing leakage energy in caches for multiprocessor systems. In *Proceedings of the 13th International Conference on Parallel and Distributed Systems*, December 2007.

[52] Mrinmoy Ghosh, Emre Özer, Stuart Biles, and Hsien-Hsin S. Lee. Efficient System-on-Chip energy Management with a Segmented Bloom Filter. In *Proceedings of the 19th International Conference on Architecture of Computing Systems*, pages 283–297, March 2006.

[53] Mrinmoy Ghosh, Weidong Shi, and Hsien-Hsin S. Lee. CoolPression-a hybrid significance compression technique for reducing energy in caches. In *Proceedings of the 17th IEEE International SOC Conference*, pages 399–402, 2004.

[54] James R. Goodman. Coherency for Multiprocessor Virtual Address Caches. In *Proceedings of the 2nd Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 72–81, 1987.

[55] Paul Gratz, Changkyu Kim, Karthikeyan Sankaralingam, Heather Hanson, Premkishore Shivakumar, Stephen W. Keckler, and Doug Burger. On-chip interconnection networks of the trips chip. *IEEE Micro*, 27(5):41–50, 2007.

[56] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *the IEEE 4th Annual Workshop on Workload Characterization*, Austin, TX, Dec. 2001.

[57] Jaehyuk Huh, Changkyu Kim, Hazim Shafi, Lixin Zhang, Doug Burger, and Stephen W. Keckler. A NUCA substrate for Flexible CMP Cache Sharing. In *Proceedings of the 19th Annual ACM International Conference on Supercomputing*, pages 1028–1040, 2005.

[58] Ibrahim Hur and Calvin Lin. A comprehensive approach to dram power management. In *14th International Conference on High-Performance Computer Architecture (HPCA-14 2008), 16-20 February 2008, Salt Lake City, UT, USA*, pages 305–316.

[59] Koji Inoue, Tohru Ishihara, and Kazuaki Murakami. Way-predicting set-associative cache for high performance and low energy consumption. In *ISLPED '99: Proceedings of the 1999 international symposium on Low power electronics and design*, pages 273–275, New York, NY, USA, 1999. ACM Press.

[60] Intel. Intel XScale Microarchitecture. ftp://download.intel.com/design/intelxscale/XScaleDatasheet4.pdf.

[61] K. Lawton. Welcome to the Bochs x86 PC Emulation Software Home Page. http://www.bochs.com Date: February 2005.

[62] Uksong Kang, Hoe-Ju Chung, Seongmoo Heo, Soon-Hong Ahn, Hoon Lee, Soo-Ho Cha, Jaesung Ahn, DukMin Kwon, Jae-Wook Lee Jin Ho Kim, Han-Sung Joo, Woo-Seop Kim, Hyun-Kyung Kim, Eun-Mi Lee, So-Ra Kim, Keum-Hee Ma, Dong-Hyun Jang, Nam-Seog Kim, Man-Sik Choi, Sae-Jang Oh, Jung-Bae Lee, Jei-Hwan Yoo Tae-Kyung Jung, and Changhyun Kim. 8Gb 3D DDR3 DRAM Using Through-Silicon-Via Technology . In *Proceedings of the International Solid State Circuits Conference, 2009.*, pages 130–132, 2009.

[63] Stefanos Kaxiras, Zhigang Hu, and Margaret Martonosi. Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power. In *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, pages 240–251, New York, NY, USA, 2001. ACM Press.

[64] Georgios Keramidas, Polychronis Xekalakis, and Stefanos Kaxiras. Applying Decay to Reduce Dynamic Power in Set-Associative Caches. In *2007 International Conference on High Performance Embedded Architectures and Compilers*, January 2007.

[65] R. E. Kessler and M. D. Hill. Page placement algorithms for large real-indexed caches. *ACM Tran. on Computer Systems*, 10:338–359, 1992.

[66] RE Kessler, R. Jooss, A. Lebeck, and MD Hill. Inexpensive Implementations Of Set-Associativity. In *ISCA '89: Proceedings of the 16th Annual International Symposium on Computer Architecture, 1989.*, pages 131–139.

[67] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches. In *Proceedings of the 10th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 99–107, 2002.

[68] J. Kim, S. L. Min, S. Jeon, B. Ahn, D.-K. Jeong, and C. S. Kim. U-cache: A Cost-effective Solution to the Synonym Problem. In *Proceedings of the First Annual Symposium on High Performance Computer Architecture*, pages 243–254, 1995.

[69] Joohee Kim and Marios C. Papaefthymiou. Dynamic memory design for low data-retention power. In *PATMOS '00: Proceedings of the 10th International Workshop on Integrated Circuit Design, Power and Timing Modeling, Optimization and Simulation*, pages 207–216, London, UK, 2000. Springer-Verlag.

[70] Nam Sung Kim, Todd M. Austin, and Trevor N. Mudge. Low-Energy Data Cache Using Sign Compression and Cache Line Bisection. In *2nd Annual Workshop on Memory Performance Issues*, 2002.

[71] T. Kirihata, P. Parries, D.R. Hanson, H. Kim, J. Golz, G. Fredeman, R. Rajeevakumar, J. Griesemer, N. Robson, A. Cestero, et al. An 800-MHz Embedded DRAM With a Concurrent Refresh Mode. *IEEE JOURNAL OF SOLID-STATE CIRCUITS*, 40(6):1377, 2005.

[72] A. Kumar, J. Xu, J. Wang, O. Spatschek, and L. Li. Space-Code Bloom Filter for Efficient Per-Flow Traffic Measurement. In *INFOCOM 2004: Proceedings of the Twenty-Third Conference of the IEEE Communications Society*, volume 3, pages 1762–1773, 2004.

[73] C. Lee, M. Potkonjak, and WH Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia andcommunications systems. *Proceedings of the Thirtieth Annual IEEE/ACM International Symposium on Microarchitecture 1997*, pages 330–335, 1997.

[74] Alberto Leon-Garcia. *Probability and Random Processes for Electrical Engineering*. Addison-Wesley Publishinc Company, Inc., 1994.

[75] C. C. Liu, I. Ganusov, M. Burtscher, and S. Tiwari. Bridging the Processor-Memory Performance Gap with 3D IC Technology. *IEEE Design and Test of Computers*, pages 556–564, November-December 2005.

[76] A. Ma, M. Zhang, and K. Asanovic. Way memoization to reduce fetch energy in instruction caches. In *WCED '01: Proceedings of the Workshop on Complexity Effective Design held in conjunction with the 28th International Symposium on Computer Architecture*, page 31, 2001.

[77] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, February 2002.

[78] Mahesh Mamidipaka, Kamal Khouri, Nikil Dutt, and Magdy Abadir. Analytical models for leakage power estimation of memory array structures. In *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 146–151, 2004.

[79] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005.

[80] Akira Matsuzawa. RF-SoC - Expectations and Required Conditions. In *IEEE Transactions on Microwave Theory and Techniques*, pages 245–253, 2002.

[81] N. Mehta, B. Singer, R. Iris Bahar, M Leuchtenburg, and R Weiss. Fetch halting on critical load misses. In *Proceedings of the The 22nd International Conference on Computer Design*, 2004.

[82] G. Memik, G. Reinman, and W. H. Mangione-Smith. Just say no: Benefits of early cache miss determination. In *Proceedings of the Ninth International Symposium on High Performance Computer Architecture*, 2003.

[83] Micron. DDR2 SDRAM SODIMM 1GB 2GB Data Sheet. http://download.micron.com/pdf/datasheets /modules/ddr2/HTF16C64_128_256x64HG.pdf Date: October 2006.

[84] Micron. Various Methods of DRAM Refresh. http://download.micron.com/pdf/technotes/DT30.pdf Date: March 2006.

[85] Fayez Mohamood, Michael B. Healy, Sung Kyu Lim, and Hsien-Hsin S. Lee. A floorplan-aware dynamic inductive noise controller for reliable processor design. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 3–14, Washington, DC, USA, 2006. IEEE Computer Society.

[86] A. Moshovos, G. Memik, B. Falsafi, and A. Choudhary. Jetty: Snoop filtering for reduced power in smp servers. In *Proceedings of International Symposium on High Performance Computer Architecture (HPCA-7)*, January 2001.

[87] Trevor N. Mudge. Power: A First-Class Architectural Design Constraint. *IEEE Computer*, 34:52–58, 2001.

[88] M.Viredaz and D. Wallach. Power Evaluation of a Handheld Computer: A Case Study. Technical report, Compaq WRL, 2001.

[89] T. Nagai, M. Wada, H. Iwai, M. Kaku, A. Suzuki, T. Takai, N. Itoga, T. Miyazaki, H. Takenaka, T. Hojo, et al. A 65nm low-power embedded DRAM with extended data-retention sleep mode. In *Proceedings of the IEEE International Solid-State Circuits Conference,*, pages 567–576, 2006.

[90] K. Nii, H. Makino, Y. Tujihashi, C. Morishima, Y. Hayakawa, H. Nunogami, T. Arakawa, and H. Hamano. A low power SRAM using auto-backgate-controlled MT-CMOS. In *Proceedings of the 1998 International Symposium on Low Power Electronics and Design*, pages 293–298, November 1998.

[91] Taku Ohsawa, Koji Kai, and Kazuaki Murakami. Optimizing the DRAM refresh count for merged DRAM/logic LSIs. In *ISLPED '98: Proceedings of the 1998 international symposium on Low power electronics and design*, pages 82–87, New York, NY, USA, 1998. ACM Press.

[92] J. Thomas Pawlowski. Intelligent refresh controller for dynamic memory devices, U.S Patent No 5,890,198 B1, 1999.

[93] J.-K. Peir, W. W. Hsu, and A. J. Smith. Implementation Issues in Modern Cache Memory. *IEEE Transactions on Computers*, 40(2), 1999.

[94] Jih-Kwon Peir, Shih-Chang Lai, Shih-Lien Lu, Jared Stark, and Konrad Lai. Bloom filtering cache misses for accurate data speculation and prefetching. In *Proceedings of the 16th International Conference of Supercomputing*, pages 189–198, 2002.

[95] LA Polka, H. Kalyanam, G. Hu, and S. Krishnamoorthy. Package Technology to Address the Memory Bandwidth Challenge for Tera-Scale Computing. *Intel Technology Journal*, 11(03), 2007.

[96] M.D. Powell, S.H. Yang, B. Falsafi, K. Roy, and T.N. Vijaykumar. Gated-vdd: A circuit technique to reduce leakage in cache memories. In *Proceedings of the 2000 International Symposium on Low Power Electronics and Design*, pages 90–95, 2000.

[97] Kiran Puttaswamy and Gabriel H. Loh. Implementing caches in a 3d technology for high performance processors. In *ICCD '05: Proceedings of the 2005 International Conference on Computer Design*, pages 525–532, Washington, DC, USA, 2005. IEEE Computer Society.

[98] A. Rahman and R. Reif. System Level Performance Evaluation of Three-Dimensional Integrated Circuits. *IEEE Tranactions on VLSI*, 8(6):671–678, 2000.

[99] S. Rhea and J. Kubiatowicz. Probabilistic Location and Routing. In *INFOCOM 2002: Proceedings of the Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 3, pages 1248–1257, 2002.

[100] A. Roth. Store vulnerability window (svw): Re-execution filtering for enhanced load optimization. In *Proceedings of the 32th International Symposium on Computer Architecture (ISCA-05)*, June 2005.

[101] S.M. Sait and H. Youssef. *VLSI Physical Design Automation: Theory and Practice*. McGraw-Hill, 1995.

[102] Sandpile. AA-64 Implementation: AMD K8. http://www.sandpile.org/impl/k8.htm Date: February 2006.

[103] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, and S. W. Keckler. Scalable hardware memory disambiguation for high ilp processors. In *Proceedings of the 36th International Symposium for Microarchitecture*, 2003.

[104] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, and S. W. Keckler. Scalable Hardware Memory Disambiguation for High ILP Processors. In *Proceedings of IEEE/ACM 36th International Symposium on Microarchitecture*, pages 399–410, 2003.

[105] Tom Shanley. *Pentium Pro Processor System Architecture*. MindShare, Inc, 1997.

[106] B. Slechta, B. Fahs, D. Crowe, M. Fertig, G. Muthler, J. Quek, F. Spadini, S. Patel, and S. Lumetta. Dynamic optimization of micro-operations. In *Proceedings of the Ninth Annual Symposium on High Performance Computer Architecture*, 2003.

[107] Alan J. Smith. Cache Memories. *ACM Computing Surveys*, September 1982.

[108] Seungyoon Peter Song. Method and system for selective DRAM refresh to reduce power consumption, U.S Patent No 6,094,705 B1, 2000.

[109] Ching-Long Su and Alvin M. Despain. Cache design trade-offs for power and performance optimization: a case study. In *Proceedings of the 1995 International Symposium on Low Power Design*, pages 63–68, 1995.

[110] Sun. OpenSPARC T1 Microarchitecture Specification. http://opensparc-t1.sunsource.net/specs/OpenSPARCT1_Micro_Arch.pdf Date: January 2006.

[111] Michael Bedford Taylor, Walter Lee, Jason Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jason Kim, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ilp and streams. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 2–13, 2004.

[112] S. T. Tucker. The IBM 3090 System: An Overview. *IBM Systems Journal*, 1986.

[113] R. Venkatesan, S.Herr, and E. Rotenberg. Retention-aware placement in dram (rapid):software methods for quasi-non-volatile dram. In *Proceedings of the Twelfth Annual Symposium on High Performance Computer Architecture*, pages 155–165, November 2006.

[114] Luis Villa, Michael Zhang, and Krste Asanovic. Dynamic zero compression for cache energy reduction. In *Proceedings of IEEE/ACM 33rd International Symposium on Microarchitecture*, pages 214–220, 2000.

[115] Stevan Vlaovic and Edware S. Davidson. TAXI: Trace Analysis for X86 Interpretation. In *Proceedings of the 2002 IEEE International Conference on Computer Design*, pages 508–514, 2002.

[116] David Wang, Brinda Ganesh, Nuengwong Tuaycharoen, Kathleen Baynes, Aamer Jaleel, and Bruce Jacob. Dramsim: a memory system simulator. *SIGARCH Comput. Archit. News*, 33(4):100–107, 2005.

[117] Jinn-Shyan Wang and Chung-Hsun Huang. High-speed and low-power cmos priority encoders. *Journal of Solid-State Circuits*, 35(10):1511–1514, 2000.

[118] Wen-Hann Wang, Jean-Loup Baer, and Henry M. Levy. Organization and Performance of a Two-Level Virtual-Real Cache Hierarchy. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 140–148, 1989.

[119] Dong Hyuk Woo, Mrinmoy Ghosh, Emre Özer, Stuart Biles, and Hsien-Hsin S. Lee. Reducing energy of virtual cache synonym lookup using bloom filters. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 179–189, New York, NY, USA, 2006. ACM Press.

[120] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH–2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA)*, pages 24–36, June 1995.

[121] S. H. Yang, M. D. Powell, B. Falsafi, K. Roy, and T. N. Vijaykumar. An Integrated Circuit/Architecture Approach to Reducing Leakage in Deep-Submicron High-Performance I-Caches. In *Proceedings of the Seventh Annual Symposium on High Performance Computer Architecture*, page 147, 2001.

[122] C. Zhang, F. Vahid, J. Yang, and W. Najjar. A way-halting cache for low-energy high-performance systems. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2(1):34–54, 2005.

[123] M. Zhang and K. Asanovic. Highly-associative caches for low-power processors. in Kool Chips Workshop, held in conjunction with the 33rd International Symposium on Microarchitecture, 2000.

[124] H. Zheng, J. Lin, Z. Zhang, E. Gorbatov, H. David, and Z. Zhu. Mini-rank: Adaptive DRAM architecture for improving memory power efficiency. In *Proceedings of the 41st IEEE/ACM International Symposium on Microarchitecture*, pages 210–221, 2008.