UNIVERSITY OF CINCINNATI

Date: July 18th, 2008

I, Jason Nemeth,

hereby submit this work as part of the requirements for the degree of:

Master of Science

in:

Computer Engineering

It is entitled:

Location Cache Design and Performance Analysis

for Chip Multiprocessors

This work and its defense approved by:

Chair: Dr. Wen-Ben Jone

Dr. Hal Carter

Dr. Yiming Hu

Location Cache Design and Performance Analysis for Chip Multiprocessors

A thesis submitted to the

Division of Research and Advanced Studies
of the
University of Cincinnati

in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in the Department of
Electrical and Computer Engineering
of the College of Engineering
July 18, 2008

by

Jason Nemeth
B.S. (Computer Engineering), University of Cincinnati, June 2005

Thesis Advisor and Committee Chair: Dr. Wen-Ben Jone

For Nicole

Abstract

As it becomes increasingly difficult to improve the performance of a microprocessor by simply increasing its clock speed, chip makers are looking towards parallelism in the form of Chip Multiprocessors (CMPs) to increase performance. Indeed, recent research at Intel suggests that chips with hundreds of cores are possible in the not-so-distant future [1]. As the number of cores grows, so does the size of the cache systems required to allow them to operate efficiently. Caches have grown to consume a significant percentage of the power utilized by a processor. In this research, we extend the concept of a location cache to support CMP systems in combination with low-power L2 caches based upon the gated-ground technique. The combination of these two techniques allows for reductions in both dynamic and leakage power consumption. In this work we will present an analysis of the power savings provided by utilizing location caches in a CMP system. The performance of the cache system is evaluated by extending the capability of CACTI and Simics using the SPLASH-2 and ALPBench benchmark suites. These simulation results demonstrate that the utilization of location caches in CMP systems is capable of saving a significant amount of power over equivalent CMP systems that lack location caches.

Acknowledgement

I wish to extend my most sincere thanks and gratitude to my advisor, Dr. Wen-Ben Jone. He may not realize it, but we exchanged well over *1,400* e-mails while developing this work. With a sharp eye and a door that was always open, we often had thoughtful discussions long into the night. I thank you, Dr. Jone, for all that you have done for me and for this work. This wouldn't have been possible without you.

I would like to thank my committee members, Dr. Hal Carter and Dr. Yiming Hu, for taking the time out of their very busy schedules to review this work. Special thanks goes to Bin Qi for kindly offering to lend a section from his thesis to this work.

I'd also like to thank my parents, Deborah and Joseph, and my sister, Jessica, for their endless support through all these years. Finally, I express sincere gratitude to my wife, Nicole, for her constant support and love.

Thank you all.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years microprocessor companies have had difficulty increasing the performance of the CPUs by simply increasing their clock frequency. Research has moved to parallelism in an effort to maintain performance increases [2]. It is now increasingly common for multiple processing cores to be included on a single silicon die, creating what is called a Chip Multiprocessor (CMP). These CMPs typically contain multiple cores operating at the same clock frequency, and those cores tend to share at least part of their cache system with the other processors on the chip. An example die photograph of an Intel Xeon MP CMP is shown in Figure 1.1. In this photograph it can be seen that this processor contains a pair of processing cores, and includes a cache system consisting of private L1 and L2 caches in addition to a large L3 cache that is shared between the cores.

There are a number of issues relating to the design of cache systems in CMP systems. One major problem is that of contention [4]. Since each processing core is permitted to execute instructions independently of the others, situations arise

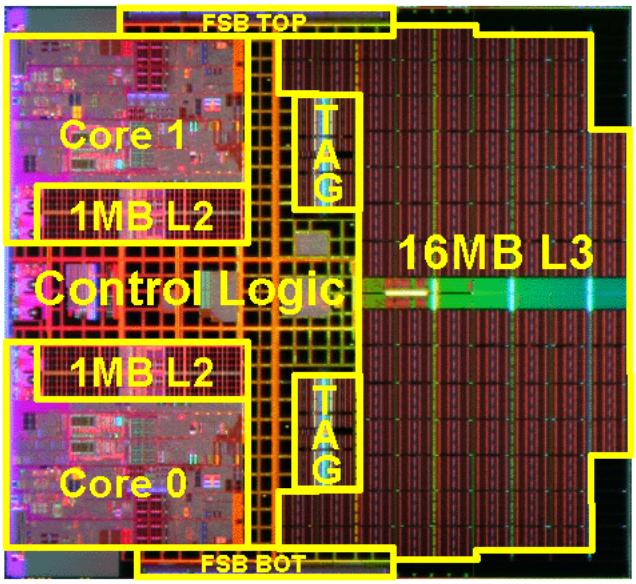


Figure 1.1: Die photo of an Intel Xeon Chip Multiprocessor [3].

where multiple cores need to access a single cache line at the same time. The effect of this issue can be greatly minimized by constructing the shared caches such that they have multiple banks and dedicated read/write ports for each core, though the occasional simultaneous accesses to the same bank must still force a processor to *back off* and wait for another core's access to complete.

Another issue in CMP cache design is that of coherency [5]. Data written to a private cache cannot be accessed by other cores on the same chip. When multiple caches have different data for the same memory address, they are said to be *incoherent*. Protocols such as MESI [6] must be put in place to ensure that such a condition cannot occur. While these protocols add overhead to cache system, they are necessary to ensure that all of the cores in a CMP system have access to the most recent data.

As cache systems have grown in size to satisfy the additional needs of these

CMP systems, so does the amount of power they consume. Several techniques are commonly used to reduce the amount of dynamic power used by a cache. *Subbanking* partitions the cache into smaller subbanks, which saves power by accessing only the subbank that actually contains the requested information [7, 8]. Techniques also exist to segment the bitline, discharging only the bitline segments necessary to perform an access [7]. Another approach to power savings is utilizing a *phased cache*, where the tag array is accessed prior to the data array, allowing only the correct way in the data array to be accessed at the expense of increased access time [9, 8, 10].

While the dynamic power used for read and write accesses still plays a large part in overall power usage, leakage has grown to dominate the power consumed by the cache system [11]. While continually reducing the process size increases speed and reduces area, it also increases the sub-threshold leakage power consumed by the chip. In addition, the increased power consumption may also lead to thermal issues on the chip, and design must proceed carefully in order to eliminate potentially-damaging hot spots. Several different techniques have been presented to reduce sub-threshold leakage power. The concept of *gated-*Vdd [12] completely shuts off an unused cell's connection to its power source. While this reduces the amount of leakage power consumed, it has the side effect of losing its state when driven into sleep mode. *Drowsy caches* were introduced in [13], allowing a cell to enter a low leakage mode and also retain its data by reducing the voltage across Vdd. Finally, *DRG-caches* achieve very low subthreshold leakage in addition to data retention by exploiting the properties of transistor stacking [14].

In an attempt to save both dynamic and sub-threshold leakage power in the face of shrinking process sizes, the concept of a location cache was introduced [15]. A location cache is a small direct-mapped cache that stores information relating an address to its location in the *target* cache. In a standard set-associative cache such as an L2, a given address can be stored in any of a number of ways equal to the cache's associativity. The way number of the data's location is then passed to and stored in the location cache. During future accesses, if way information for a given address is present in the location cache, the *target* (L2) cache can be accessed as if it were direct-mapped. This capability can save dynamic power upon cache reads and writes. More importantly, this behavior is capable of being exploited when used in combination with gated-ground techniques to save a significant amount of leakage power.

CACTI is capable of providing detailed power, timing, and area estimates for a cache system [16, 17, 18, 19, 20]. Given a set of user-provided parameters, CACTI searches for an optimal cache configuration. Power and timing analysis is provided on a component-by-component basic, easily allowing the user to determine which portions of the cache may benefit most from a given optimization. In addition, CACTI 5.0's updated technology modeling no longer uses linear scaling of its original 0.8 micron technology, allowing more accurate estimation of dynamic and leakage power for the 65nm process used in this work [20].

Once statistics on power consumption were calculated, a way to simulate the workloads was required. Virtutech's Simics [21] provides an ideal environment for cache development, and allows for booting a variety of operating systems, and is consequently capable of executing a variety of benchmark suites. Unfortunately,

Simics does not have any built-in capability for power estimation.

In this work we propose a method of extending the concept of a location cache to support CMP systems. We utilize CACTI to provide both dynamic and leakage power measurements for all the caches in this work. CACTI does not natively support location caches or caches utilizing low leakage techniques, so it was extended to support these cache architectures. Simics is extended to provide the capability for cycle-by-cycle power estimation for traditional caches, location caches, and gated-ground caches. These extensions provide a framework for testing a variety of cache systems by running benchmarks on actual operating systems. Models of both the location caches and the gated-ground low leakage L2 caches are created and simulated using CACTI and Simics. The power utilization of the cache system is presented for a number of possible configurations using the SPLASH-2 and ALPBench multithreaded benchmark suites, and a discussion of the results is provided.

The following Chapters of this thesis are arranged as follows:

Chapter 2 reviews background information for gated-ground cache design, CACTI cache architectures, and the Simics system simulator.

Chapter 3 covers previous work involving location caches in single processor systems, and how it was extended to work in CMP systems.

Chapter 4 discusses low leakage cache design, and how CACTI was utilized to support gated-ground low leakage caches and location caches.

Chapter 5 presents our use of Simics to provide power estimation statistics for our low leakage and location cache modules.

Chapter 6 describes the experimental environment used in this work, and provides the power estimation results for two benchmark suites in a variety of cache configurations.

Chapter 7 provides the conclusions of this thesis and discusses possible future work.

Chapter 2

Background

In this Chapter we will briefly discuss background information related to this work. We will begin in Section 2.1 by discussing a low-power cache implementation called *gated-ground*. Section 2.2 presents Virtutech's Simics, which is used to implement our cache and location cache models. Finally, in Section 2.3 the CACTI cache simulator, used to measure power consumption of the various caches in this work, is discussed.

2.1 Gated-ground Cache

One technique proposed to reduce the amount of subthreshold leakage energy consumed by a cache system is the DRG-cache [14]. This technique involves inserting a single NMOS sleep transistor in between the ground plane and the SRAM cell, as shown in Figure 2.1. Such an implementation exploits the stacking effect of chaining multiple NMOS transistors in series [22], and allows for a significant reduction of leakage energy while in the sleep mode.

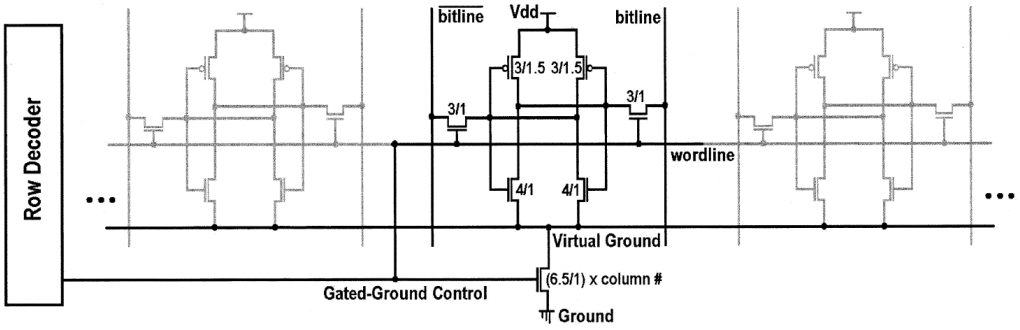


Figure 2.1: A typical implementation of a *DRG-cache* [23].

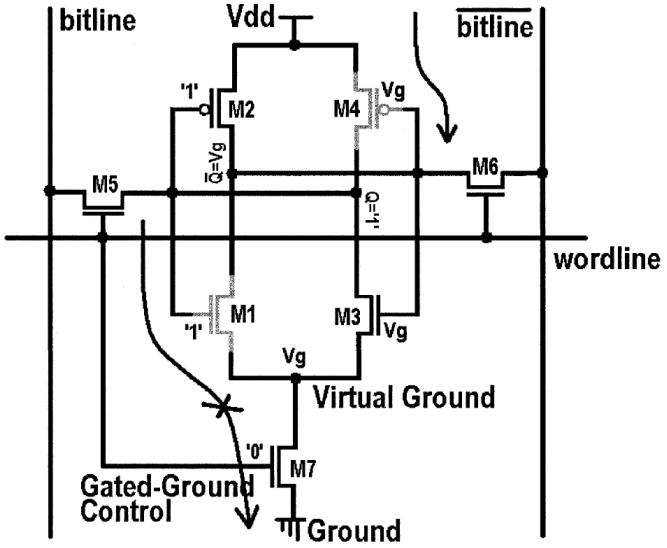


Figure 2.2: Data retention capability of a *DRG-cache* SRAM cell [23].

When altering the ground voltage it is important to consider the stability of the SRAM cell, and its capability to retain a written value when the virtual ground voltage is raised. Figure 2.2 presents a single SRAM cell from a *DRG-cache*. In an example provided in [23] a logic 1 is written on line Q, and the voltage at $\overline{Q}$ goes to the saturation voltage Vg where Vg is the voltage present at the virtual ground in the SRAM cell's sleep mode. Vg is set small enough so that transistor M4 remains on, which ensures that Q remains a logic 1 and transistor M1 also remains on. With transistor M1 on, Vg follows the voltage at $\overline{Q}$, causing transistor M3 to turn off because its $Vgs = 0$.

While in active mode, the sleep transistor is turned on allowing direct access to the ground plane. When the transistor is turned off, the SRAM cell is driven into sleep mode, allowing the *virtual ground* shown in Figure 2.1 to float up slightly in voltage to about $0.3V$. The voltage present on the *virtual ground* while in the sleeping mode is dependent on the size of the sleep transistor. The sleep transistor can be controlled by the row decoder, which would allow the cells to be activated and put into sleep mode automatically without any additional circuitry required. Using such an implementation, a *DRG-cache* can achieve an energy savings of nearly 50% in the 70nm and 100nm process sizes compared to a conventional cache [14].

The *DRG-cache* is also being used in commercially-available devices. Intel has begun using a gated-ground technique in its processors [3], and the Massachusetts Institute of Technology has even experimented with its use in FPGAs [24]. The gated-ground technique's real-world usage in Intel's modern multi-core chips made it a logical selection as the leakage reduction technique used in this

work.

2.2 Virtutech Simics System Simulator

Virtutech's Simics is a full system simulator capable of simulating an entire computing system, including processors, caches and memories, graphics and networking cards, hard disks, and many types of removable media [21]. This kind of flexibility allows the simulation of many different hardware architectures and the ability to boot a variety of different operating systems. Better yet, the ability to boot these operating systems means that there are a variety of benchmarking suites available to test system optimizations.

Simics provides a built-in cache system called *g-cache* that allows individual cache modules to be attached to a processor. Using these cache modules it is possible to build up a model of the entire cache system, including simulating accesses to main memory. This is particularly useful for cache-related works such as our own, as it provides a working starting point for building up a cache architecture that fully integrates with the simulator.

The *g-cache* implementation even provides support for a built-in coherency protocol called MESI [25], which is used in a many of Intel's microprocessors. While this implementation of MESI is specifically intended for cache systems utilizing write-through L1 caches and write-back L2 caches [26], it can be modified to work for other configurations. MESI, which stands for *Modified Exclusive Shared Invalid*, provides a method for indicating the status of lines within the cache. Limiting the number of states to four requires that only two bits be added

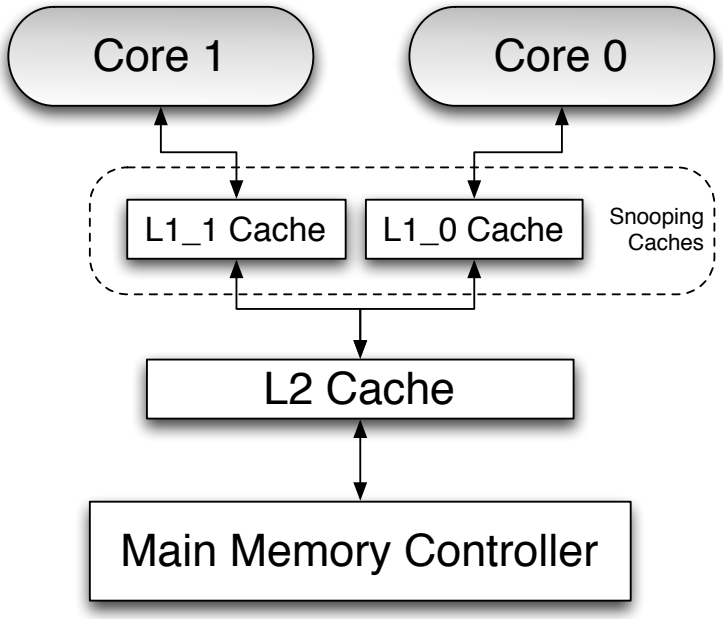


Figure 2.3: A simple cache system illustrating snooping and the MESI protocol.

to to each line in the cache, resulting in a relatively small storage overhead. Each cache line is capable of being in any one of the four states, described as follows:

1. **Modified** - This is the only cache that contains this data entry, but it has been modified and main memory is not up-to-date.

2. **Exclusive** - This is the only cache that contains this entry and main memory is up-to-date.

3. **Shared** - Two or more caches are sharing a copy of this entry and it is consistent with main memory.

4. **Invalid** - This line is invalid and cannot be used.

Caches are kept coherent by utilizing the concept of *snooping*. *Snooping* is

a technique where a cache monitors the address lines of other caches, looking in particular for accesses to addresses that it also contains. Assume a cache configuration like the one shown in Figure 2.3, where L1_0 and L1_1 are *snooping* on each other. Core 0 initiates an access to a memory address that is not present in either L1_0 or L1_1, and the result of this access is placed in the L1_0 cache and the line in L1_0 is marked *Exclusive*. The L1_0 cache must now monitor the L1_1 cache for accesses to this address. If such a read access to this address occurs, both lines will be marked *Shared*. This process of monitoring accesses to other related caches is vital to the operation of the MESI protocol.

With the basics of *snooping* covered, let us now consider a more complex example. Again, assume a cache of the configuration shown in Figure 2.3, and that the L1 caches both contain a cached copy of the same address (marked *Shared*), as in the previous example. Now, however, Core 1 initiates a write to to this address. MESI initiates a *Read for Ownership*, which causes the copy in L1_0 to be marked *Invalid*. Now writing to the cache may proceed as normal, with the state of the line in L1_1 being marked *Modified*. While the address is still cached in both L1_0 and L1_1, L1_0's copy can no longer be used as it is now incoherent, and will be overwritten at the next opportunity. When the cached copy stored in L1_1 is eventually written to L2, and subsequently main memory, the status of the line in L1_1 will be returned to *Exclusive*.

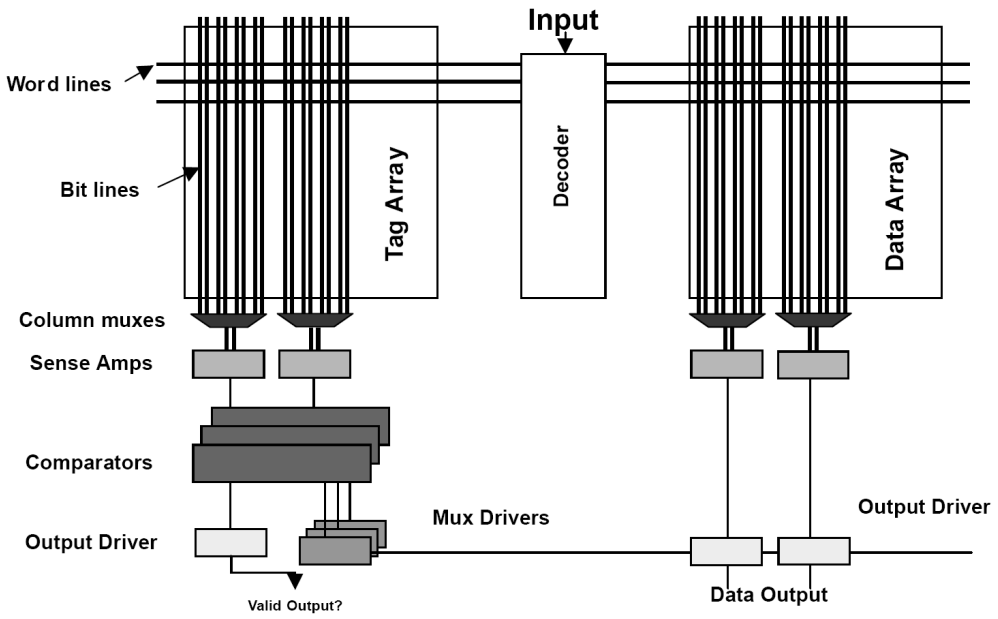


Figure 2.4: The cache architecture utilized by CACTI [17].

2.3 CACTI Cache Simulator

CACTI is a tool that can be used to estimate the access time, power, and area of cache systems [16, 17, 18, 19, 20]. CACTI utilizes an analytical model to determine the optimal configuration of a cache system given design parameters, at a minimum, cache size, associativity, and block size. The configuration produced by CACTI adheres to the architecture shown in Figure 2.4. While CACTI is capable of providing estimates of access time and area, we will concentrate on its power estimation capabilities in this work.

CACTI breaks a cache down into two separate sets of SRAM cells, a data array and a tag array. These two arrays are sized independently so that each may be fully optimized [16]. Figure 2.5 shows how a cell array is partitioned in CACTI. Each array can be assigned any number of banks, which is provided to CACTI as a

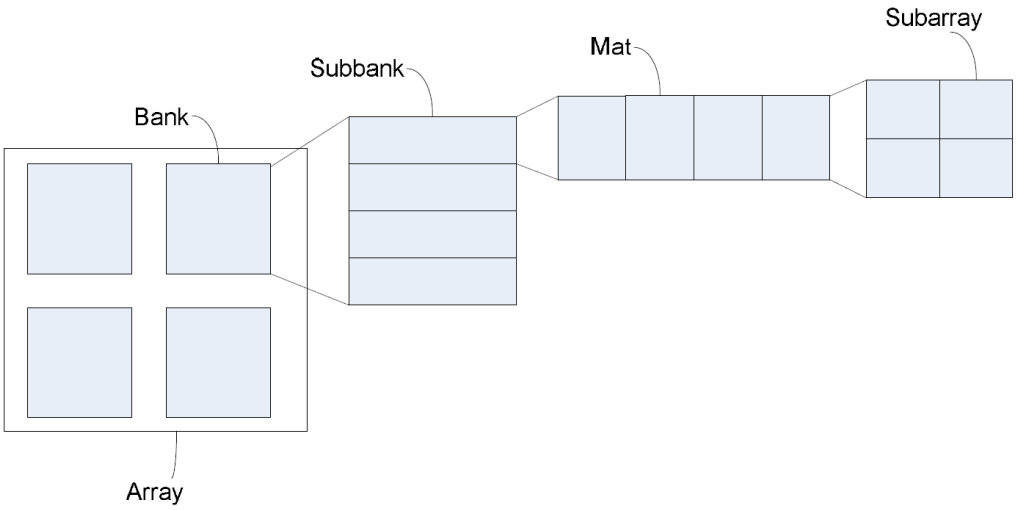


Figure 2.5: The CACTI cell array hierarchy [20].

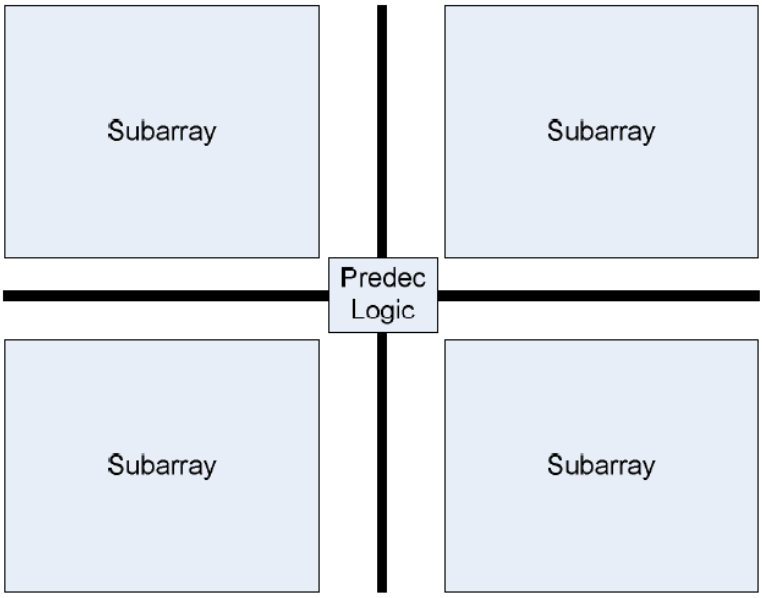


Figure 2.6: An overview of a CACTI mat [20].

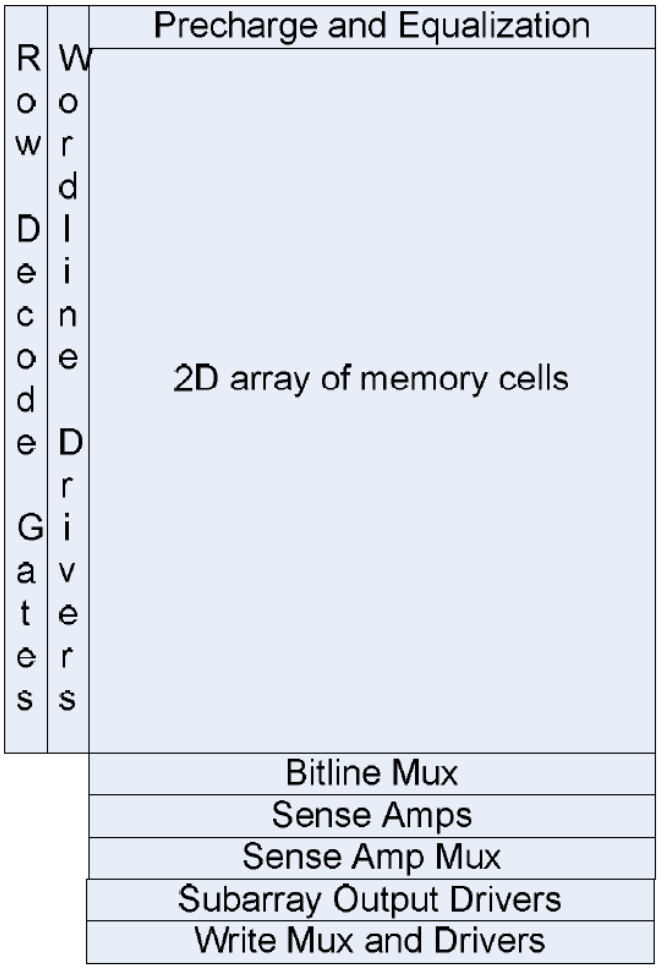
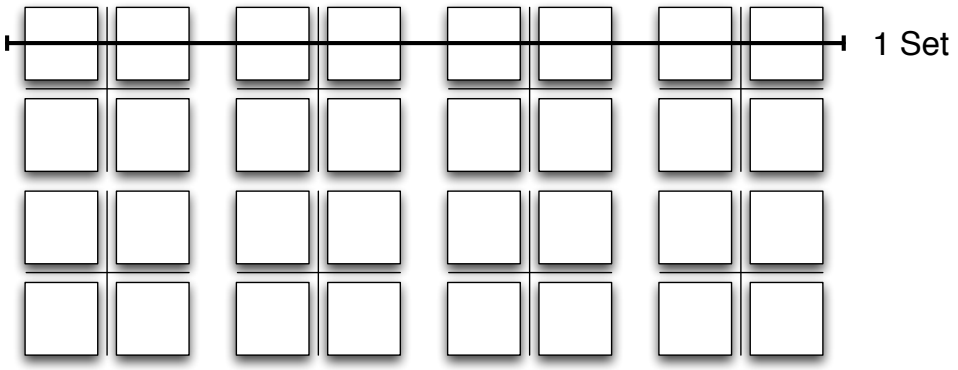


Figure 2.7: An overview of a CACTI subarray [20].

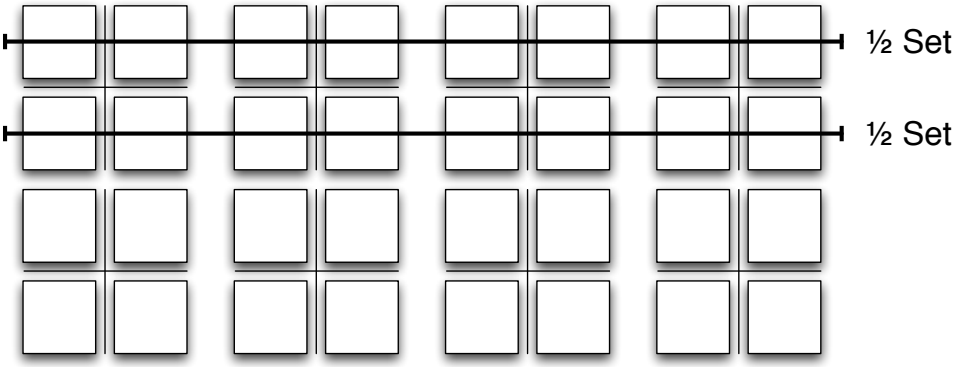
user-defined parameter. From there, CACTI determines the optimal arrangement of the rest of the hierarchy. Each bank has a number of subbanks, each of which occupies the entire width of the cache. This implies that subbanks are arranged vertically within a bank, as can been seen in Figure 2.5. Each subbank contains a number of horizontally-arranged mats, each of which occupies the entire height of the subbank. Further, each mat contains four subarrays arranged in a 2x2 pattern along with the pre-decoder and decoder logic that is shared among each subarray as shown in Figure 2.6. The subarrays contain the actual SRAM cells and related decoders, multiplexers, drivers, and sense amplifiers required to utilize them, as illustrated in Figure 2.7.

The division of the cache system into subarrays is important, as it will be exploited in Chapter 4 to allow for power optimizations when paired with a location cache. The optimal configuration of each bank can be fully described by the parameters *Ndbl*, *Ndwl*, and *Nspd*. *Ndbl* is the number of divisions in the bitlines, which describes the number of subbanks present in the array. *Ndwl* is the number of divisions of the wordlines, which stipulates the number of subarrays arranged along the entire width of the array. Since each mat is two subarrays wide, the number of mats along the width of the array can be described as $\frac{Ndwl}{2}$. *Nspd* defines the number of sets (where each set contains a number of ways equal to the associativity of the cache) per cache line, where fractional numbers indicate that a set is spread out across multiple cache lines.

For example, shown in Figure 2.8 are two different cache systems with *Ndwl* = 8 and *Ndbl* = 4. In Figure 2.8(a) $Nspd = 1$, which indicates that an entire set is accessible by using a single line of the cache. In Figure 2.8(b) we have



(a)



(b)

Figure 2.8: The effect of *Nspd* on the cache configuration.

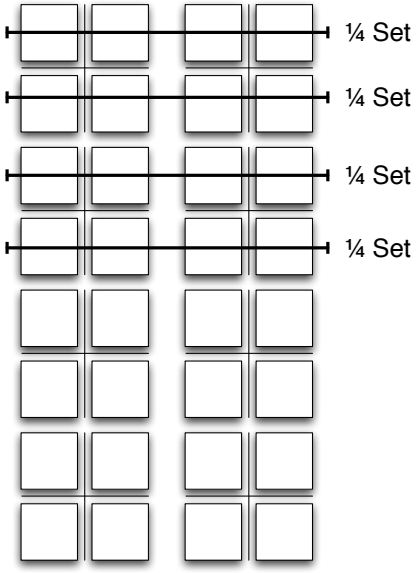


Figure 2.9: A CACTI cache configuration with *Ndwl* = 4, *Ndbl* = 8, and *Nspd* = 0.25.

$Nspd = 0.5$, which means that only half of a set is present on any given cache line. Therefore, in order to access an entire set two lines of the cache are accessed, which requires accessing a total of $\frac{Ndwl}{Nspd}$ different subarrays.

A second example, shown in Figure 2.9, shows a cache configuration where *Ndwl* = 4, *Ndbl* = 8, and *Nspd* = 0.25. As we can see from this example, since $Nspd = 0.25$ each cache line contains only $\frac{1}{4}$ of a set, requiring total of $\frac{Ndwl}{Nspd} = \frac{4}{0.25} = 16$ subarrays to access an entire set.

Chapter 3

Location Cache Design

This Chapter discusses the principals and design of the location cache concept, and the modifications made to allow it to support CMP environments. Section 3.1, background information on the principals of location caches, is an excerpt from Bin Qi's thesis *Performance Analysis of Location Cache for Low Power Cache Systems* [27].

3.1 Location Caches in Single Processor Systems

In order to reduce the miss rate of an L2 cache, normally the L2 cache is a large set-associative cache with multiple ways. For instance, the L2 cache in the Intel Itanium 2 family is a 256KB, eight ways set-associative cache. Accessing a set-associative cache wastes power because multiple data and tag ways are probed simultaneously, but only one way carries the required data. To resolve this problem, a new cache architecture, called *location cache*, has been proposed in [15]. In this section, first the structure of a location cache is introduced, and then the

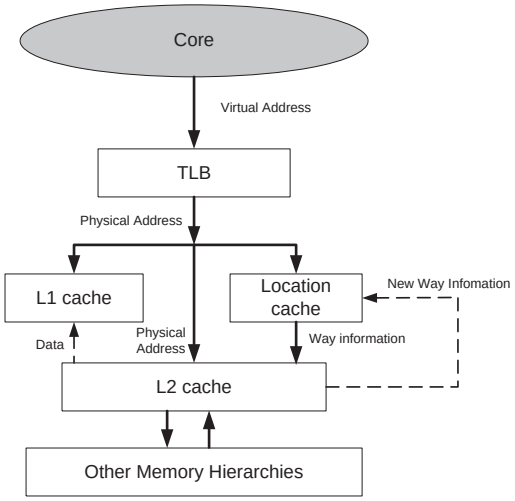


Figure 3.1: Physically addressed location cache architecture.

working principle and timing analysis of the location cache are described.

3.1.1 Structure of Location Cache

The location cache shown in Figure 3.1 is a small direct-mapped cache, using address affinity information to provide the accurate location information for L2 cache references [15]. The proposed location cache technique reduces the L2 cache power consumption, when compared with a conventional set-associative L2 cache. Depending on the L2 cache architecture described above, a location cache can be physically addressed or virtually addressed. Fig. 3.1 illustrates the revised L2 cache system architecture with a location cache, which is physically addressed.

In this physically addressed cache system, the location cache is physically addressed as well. It caches the access way location information of the L2 cache (the way number in one set where a memory reference falls). This cache works in parallel with the L1 cache. As a location cache tries to cache the L2 location in-

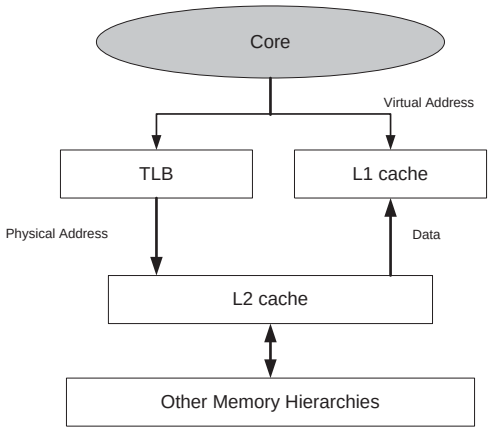


Figure 3.2: Virtually addressed L2 cache architecture.

formation, the block address (composed of the index address and the tag address) of the location cache should be of the same length as that of the L2 cache. For instance, in Intel Itanium 2 the physical address is 50 bits and the L2 cache block size is 128 bytes (instead of 64 bytes of block size for L1), so the block address of the location cache has 43 (50-7) bits. If the location cache has 512 entries (i.e., the index contains 9 bits), then each tag array entry will have 34 (43-9) bits.

The location cache can also be virtually addressed based on the architecture shown in Fig. 3.2. The revised L2 cache system architecture with a location cache, which is virtually addressed, is illustrated in Fig. 3.3. In this virtually addressed cache system, the location cache is virtually addressed too. This cache works in parallel with the TLB and the L1 cache. On an L1 cache miss, the physical address (physical tag and index) translated by the TLB and the way information provided by the location cache are both presented to the L2 cache. Since the location cache tries to cache the location information of the entire block in the L2 cache, as in the physically addressed location cache, the location cache should have the same

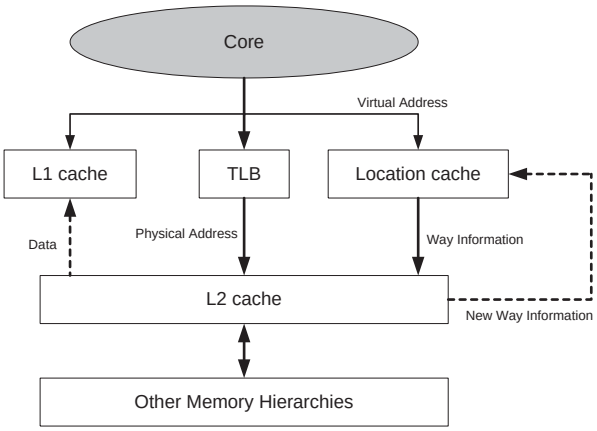


Figure 3.3: Virtually addressed location cache architecture.

block address as the L2 cache (instead of the L1 cache). For instance, in Intel Itanium 2 the virtual address is 64 bits and the L2 cache block size is 128 bytes, so the block address of the location cache will be 57 (64-7) bits. If the location cache has 512 entries, then each tag array entry will have 48 (57-9) bits. When compared with the physically addressed location cache, each entry of the tag array in a virtually addressed location cache will have 14 (64-50) more bits.

The data array of a location cache is used to store the way location information of the L2 cache. If the L2 cache has N ways, then the maximum number of bits for storing this information will be N bits. As the way number is normally a power of 2, we can also use binary encoding, which needs $log_2 N$ bits to store this information. We emphasize that the L2 cache is accessed based on physical address, while the L1 and location caches can be accessed by either physical address or virtual address depending on the implementation strategy. Further, there is an unified location cache which stores the way information in L2 for data and instructions.

3.1.2 Working Principle of Location Cache

One interesting issue arises here: the locations for which references should be cached? Obviously, the location cache should catch the references which turn out to be L1 misses. The equation below defines the optimal (largest) coverage of the location cache:

$$Opt_{Coverage} = L2_{Coverage} - L1_{Coverage} \tag{3.1}$$

However, the actual coverage of a location cache might be smaller than the optimal coverage, and will increase as the size of the location cache increases. But, unfortunately, the access time, and the access power and leakage power of the location cache will be increased too. This will be further explored in the following sections.

The proposed cache system works in the following way. The location cache is accessed in parallel with the L1 cache. If the L1 cache sees a hit, then the result obtained from the location cache is discarded. If there is a miss in the L1 cache and a hit in the location cache, the L2 cache is accessed as a direct-mapped cache. If both the L1 cache and the location cache see a miss, then the L2 cache is accessed as a conventional set-associative cache. When there is a hit in the location cache and a miss in the L1 cache, the access power of the L2 cache will be greatly reduced. As opposed to the way-prediction methods [28, 29, 30], the cached location is not a prediction. Even if there is a location cache miss, we do not see any extra delay penalty as seen in way-prediction caches.

The content (i.e., the new way information) in the location cache is updated

when both L1 miss and location miss occur. The flow diagram for the location cache content update is shown in Fig. 3.4. When the location cache stores the location information of the L2 cache, it uses the same block address as the L2 cache, instead of the L1 cache. Normally, the block size of the L2 cache is larger than that of the L1 cache; for instance, in Intel Itanium 2, the L1 block size is 64 bytes while the L2 block size is 128 bytes. Due to this difference in L1 and L2 block sizes, the location cache can still catch many references which are L1 misses but location cache hits. For example, in Intel Itanium 2 the physical address is 50 bits and the L2 (L1) cache block size is 128 (64) bytes as discussed above. Given an address, the L1 (L2) cache will interpret the address as 38 (35) bits for tag, 6 (8) bits for index, and 6 (7) bits for offset in the block by the L1 (L2) cache. Assume one byte is to be accessed with the last seven bits of its address equal ”0111111” in the binary form. Thus, the entire block (64 bits) containing this byte is accessed from the L2 cache to the L1 cache. Also, the corresponding way location of this access is stored into the location cache. In the next memory access, assume the next byte is accessed by the CPU. The last seven bits of the address thus contains ”1000000” again in the binary form. For the L1 cache, the index has changed one bit and an L1 cache miss might occur. However, for the location cache, the index is not changed and the location cache hits the memory access successfully. Note that the index field of the location cache is not changed for the new address, since it has the same block address as the L2 cache.

Even when the L2 block size is the same as the L1 block size the location cache still can hit many memory accesses with L1 miss. The reason comes from the fact that the location cache entry number (e.g. 512) is generally allocated to be

much larger than that of the L1 cache (e.g. 64) without exceeding the access time of the L1 cache. This is because the location cache data array (which contains the way information) and tag array are both smaller when compared with the L1 data array (which contains data or instructions) and tag array. As a result, the location cache still can catch many L1 misses due to its larger entry number, when the block size in the L1 cache is the same as that in the L2 cache.

By the drowsy cache technique in [13], a cache array can be put into drowsy mode when it is not accessed for a period of time. In drowsy mode, the leakage power is significantly reduced when compared with normal mode. As the L1 cache hit rate is normally high, the L2 cache can also be put into drowsy mode when it is idle for a period of time. In a simple L2 cache system, all ways of the L2 cache are waken up, when there is an access to the L2 cache. With the way information stored in the location cache, when there is a hit (miss) in the location (L1) cache, only the hit way of the L2 cache is waken up, while other ways can still be kept in drowsy state. So, in the location cache system, separate multiplexors for selecting V_{dd} or V_{dd_low} are used for each way, and they are controlled by the location information provided by the location cache. The drowsy and wake-up policy can be cache-line-based for data cache, and subbank- (e.g., way-) based for instruction cache [13]. Since L2 cache is an unified cache, we use the subbank-based drowsy and wake-up policy. That is, the entire way of the L2 cache is waken up (drowsy) when the way is accessed (idle). The same idea for location cache can be applied to other data-retention low-leakage cache designs such as the gated-GND cache [14, 3].

The location cache is organized as a direct-mapped normal cache with a data

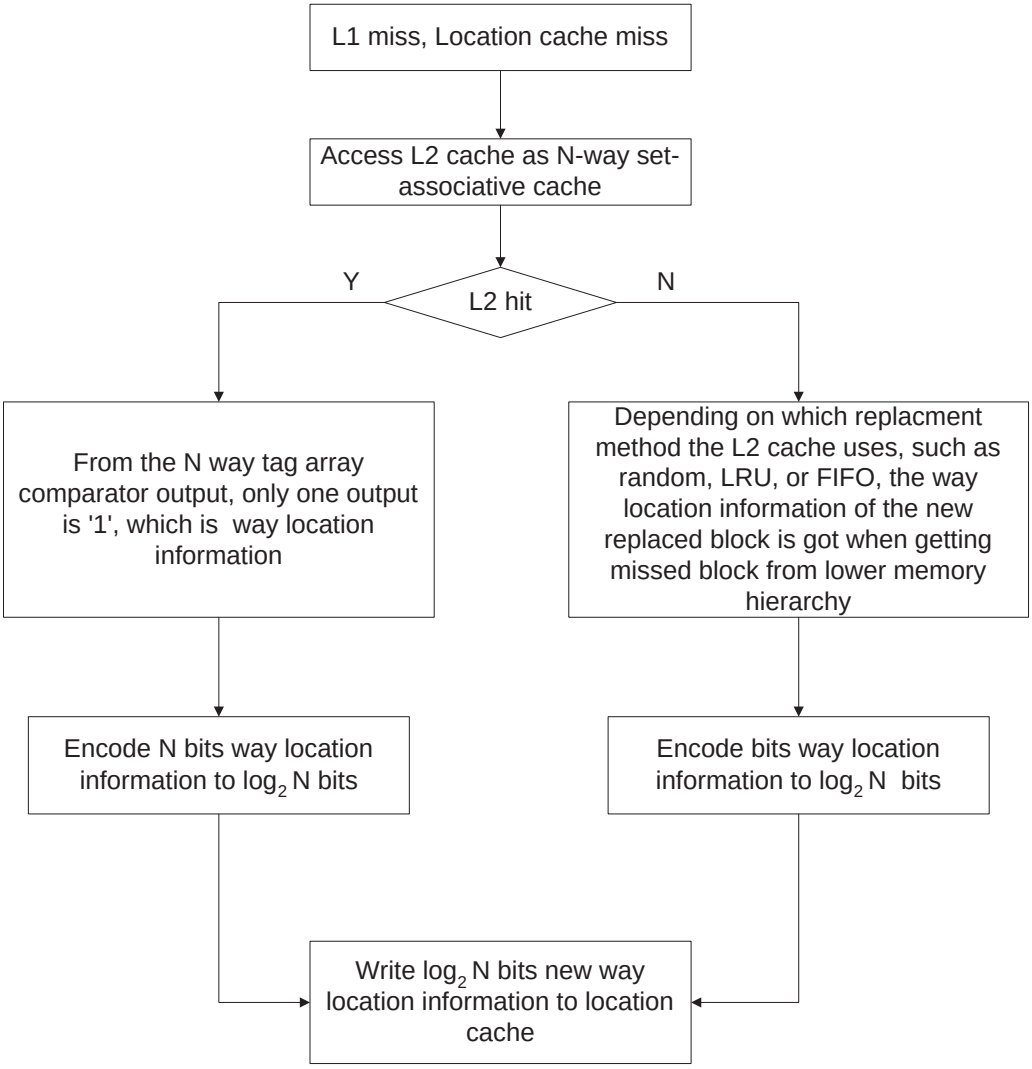


Figure 3.4: Flow diagram for location cache content update.

array and a tag array. For the data array of the location cache, each entry has $log_2 N$ bits where N is the way number of the L2 cache. As the tag array of the location cache has the same block address as the L2 cache, the entry width (W) of its tag array can be calculated by the following equation:

$$W = T - log_2 B - log_2 E \tag{3.2}$$

where we have the following notation:

T: the total access address bit width,

E: the entry number of the location cache,

B: the block size of the L2 cache in byte.

Normally the block size of an L2 cache is fixed, so each entry of the location cache tag array will have fewer bits if more entries are used in the location cache. For instance, in Intel Itanium 2 the physical address is 50 bits and the L2 cache block size is 128 bytes. If the location cache has 256 entries, then each of its tag array entry will have 35 (50-7-8) bits. However, if the location cache has 512 entries, then each of its tag array entry will have 34 (50-7-9) bits.

3.2 Location Caches in CMP Systems

Previous works utilizing location caches have been limited to single processor systems. With multicore chips becoming increasingly prevalent, the concept of a location cache needed to be adapted to these new types of systems. The following sections describe several approaches to creating location caches capable

of functioning within CMP systems.

3.2.1 Shared Location Caches in CMP Systems

The most straightforward approach to adding a location cache to a CMP system involves sharing. Multicore processors commonly share an L2 or L3 cache amongst all of the cores. Similarly, it is possible to create a single location cache capable of being accessed by each of the cores in the system if those cores share a cache at some level. For example, if all four cores share a single L2 cache, these cores can be served by a single location cache. If those four cores instead share a pair of L2 caches, a shared location cache approach would in turn require the use of a simplified MESI protocol along with a pair of location caches in order to remain simplistic and avoid coherency problems.

In the case where the highest-level cache is L2, the location cache operates on every access initiated by every processor core it serves. The source of the access is completely disregarded, and only the transaction's address is taken into consideration. A cache system utilizing a cache configuration with four cores sharing two location caches is shown in Figure 3.5.

Assume Core 0 initiates a memory access. Its L1 cache and shared location cache LCache0 parse the tag and index information and check for matches. If the L1 cache hits, the result of the location cache access is ignored and L1 returns the requested data to the processor. If the L1 cache misses, the result of the location cache access determines how to proceed. If the location cache also hits, the way information stored in the location cache is used to access the L2 cache as if it were direct-mapped. If the location cache hits, a hit in L2 is guaranteed. If the

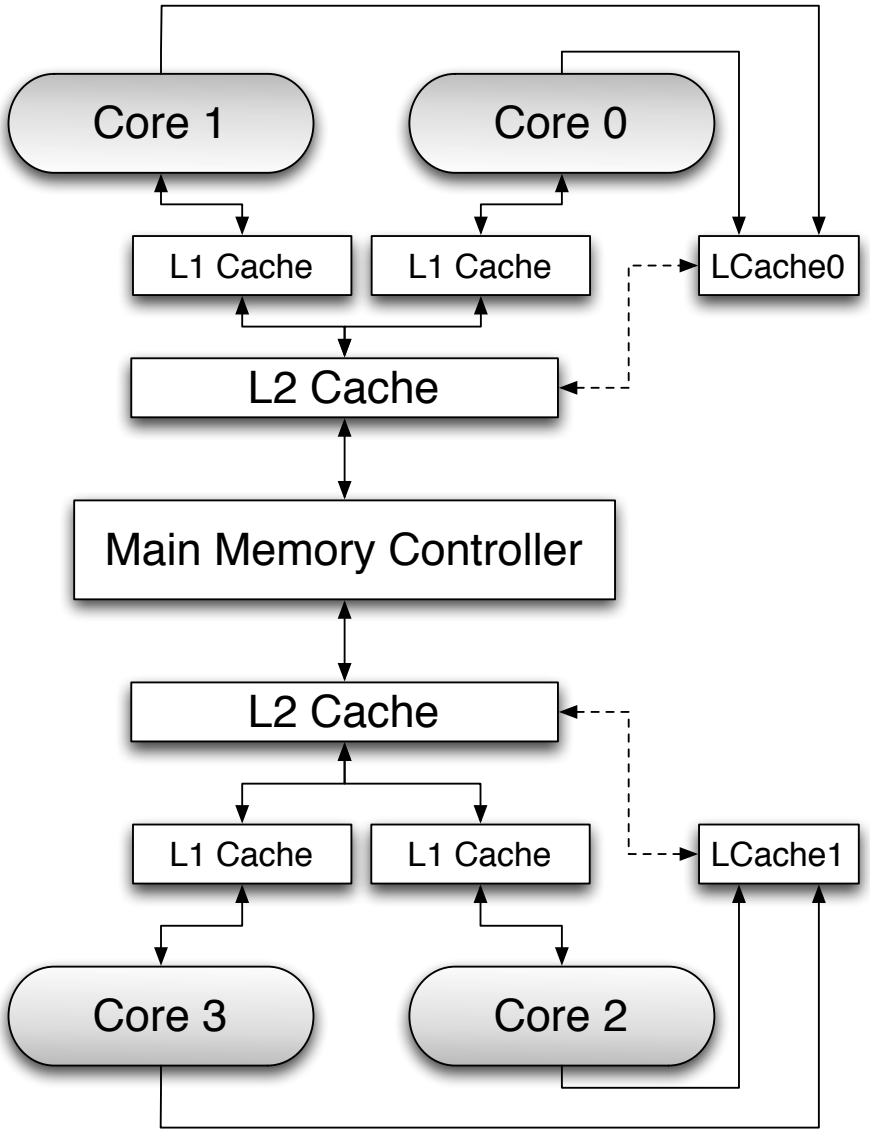


Figure 3.5: CMP Cache System Using a Shared Location Cache.

location cache misses, L2 is accessed in its normal set-associative manner and new way information is provided to the location cache for future use. Note, however, that a miss in the location cache would not necessarily indicate a miss in the L2 cache. Now assume Core 1 attempts to access the same memory address, and it is not found in its L1 cache. The way information for this address was previously stored in the location cache by Core 0, and can now be used to access L2 as a direct-mapped cache.

Let us consider another example. Assume that the L1 caches of both Core 0 and Core 1 have cached the same line of data. Core 0 now performs a write to this address, changing the data. At this point the MESI protocol triggers the L1 of Core 1 to change the line's state from *Shared* to *Invalid* , and the newly-updated line in Core 0's L1 will retain its Shared status. Note that the corresponding line in the location cache does not need to be removed or modified, as it still points to the correct location in L2. If Core 1 now tries to access this address again, it will find that its own copy in L1 is marked *Invalid* . It can now used the shared location cache, which still knows the location of the line in L2, to access L2 as if it were a direct-mapped cache. In this case a location cache can be very useful for programs that require multiple cores accessing and writing to the same memory location.

Other than the simplicity of implementation, the other advantage to this configuration is that it lacks any coherency issues. Since all of the memory transactions sharing the same L2 cache pass through the same location cache, no additional implementation changes are required to keep the location cache and its target cache coherent. When a cache system utilizes a pair of L2 caches, as in the

case mentioned above, as long as the L2 caches remain coherent with each other the location caches wills also remain coherent. If the L2 coherency is handled by the MESI protocol, no modifications to the location caches are required to allow them to support such a configuration. That is, the location caches do not need to be equipped with MESI protocol bits. The coherency issue is completely taken care of by the MESI protocol implemented for the L1 caches.

While this setup is easy to implement, it does have drawbacks. An access initiated by Core 0 is likely to overwrite way information in the location cache that will be used again by Core 1 in the near future. In addition, with multiple processors expecting complete access to the location cache, the location cache will need a read and write port for every processor in the system. Simultaneous accesses to the same line in the location cache will increase latenency, and reduce the efficiency of the location cache itself. Resolving these issues will increase both the complexity and power consumption of the location cache.

Another concern is that of replacement. Four cores running four different programs could create a great degree of churn in the limited number of location cache lines. The hit rate of the location cache would suffer as the unrelated access from all of the cores are constantly replacing each other. While this behavior can be relieved somewhat by increasing the size of the location cache, this unwanted behavior will always exist to some degree.

3.2.2 Private Location Caches in CMP Systems

An L2 cache utilizing a private location cache for each processor is shown in Figure 3.6. When used in a CMP system, the simplistic location cache design is

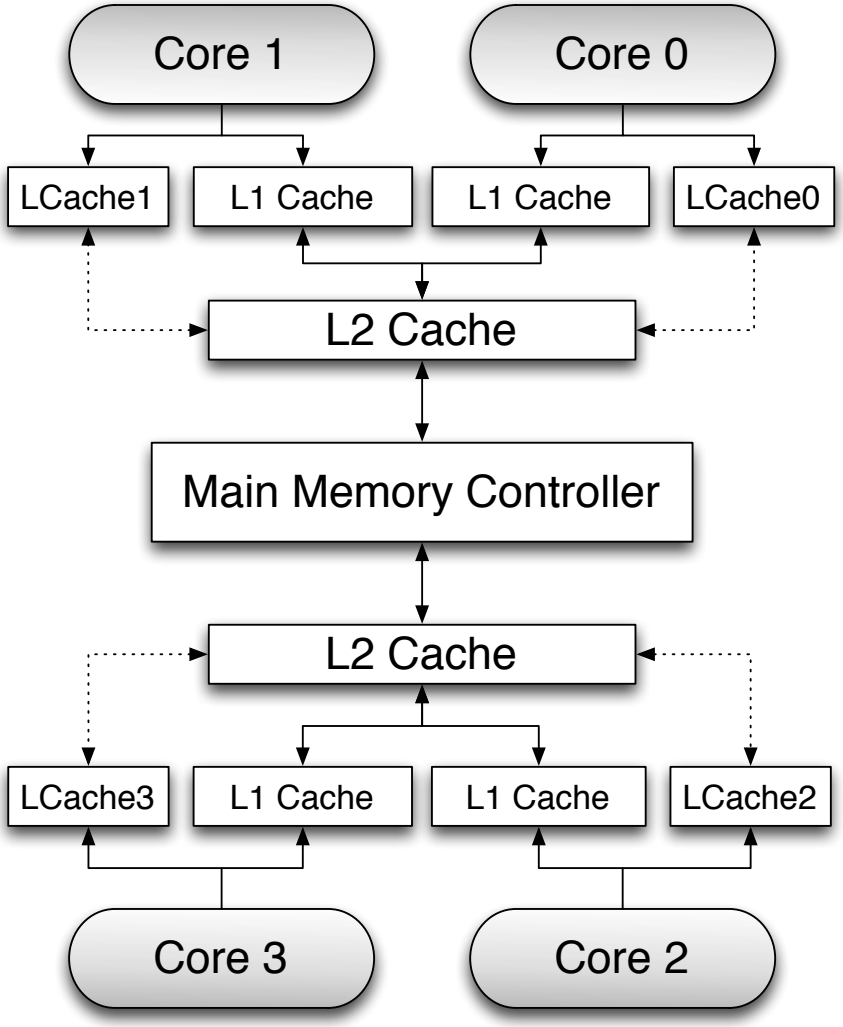


Figure 3.6: CMP Cache System Using Private Location Caches.

prone to incoherency when multiple location caches assist a single target cache. For example, if Core0 writes a value into its L2 cache, the way information is then stored in Core0's location cache (LCache0). Later, if Core1 writes a value into the L2 cache it shares with Core0 and evicts this previous entry from L2, Core0's location cache now points to data that is no longer present. Due to this possibility it was necessary to extend the concept of the location cache presented above for CMP systems.

Previously we introduced the MESI protocol and how it was utilized in our traditional cache system. We propose extending the Location Cache with a simplified version of this protocol. This modification can be performed by adding only a single additional bit to each line in the location cache. This bit will determine whether the location cache is in a *Shared* or *Invalid* state. A line in a location cache is marked *Invalid* if the line it references is no longer present in the target cache or if the line has not yet been written to. In all other cases, the line is marked *Shared*. When a location cache lookup is performed, in order for a location cache hit to occur the following two conditions must be satisfied:

1. The requested line must be present in the location cache
2. The requested line must be marked Shared

This alteration does not come without a cost. In order to perform this operation, additional care needs to be taken when lines are evicted from the target cache. When such an eviction occurs in the target cache, the tag and index portion of this reference is passed to each of the connected location caches. The location caches then check if they contain lines matching the newly-evicted entry. If the

evicted transaction address is not present in the other location caches, no further operation is performed. If the evicted transaction address is present in the other location caches, the lines in the location caches are marked *Invalid*. This will prevent future use of this line, which will be overwritten at the next opportunity.

The operation of the location caches here is a little different from that of the shared case. When the way information is stored for a transaction initiated by Core 0, for example, it is stored in Core 0's private location cache. This location cache, LCache0, cannot be read from or written to by Core 1. This alleviates the problem where each core is constantly overwriting each other's cache information, and results in increased location cache hit rates. However, let us say Core 0 has way information for a transaction stored in its private location cache. Now Core 1 performs an access that ultimately results in the line pointed to by Core 0's location cache being evicted from L2. Core 0's location cache now points to an address that no longer exists in L2. Here our coherency protocol would require L2 to transmit the address of the evicted L2 line to each of its connected location caches. If that address is present in any line in a location cache, it is marked *Invalid* . This ensures our private location caches remain coherent.

Now assume the L1 caches of Core 0 and Core 1 contain the same line of data, and their private location caches have stored the way information for the line's location in L2. Core 0 writes new data into the line, causing the status of the line in Core 0's L1 cache to move from *Shared* to *Modified* , and from *Shared* to *Invalid* in Core 1's L1 cache. However, since the line is not evicted from L2, its way information remains the *Shared* designation in both Core 0's and Core 1's location caches. If Core 1 tries to access the line again, it will find that it is marked

Invalid in its L1 cache, and *Shared* in its location cache. Thus, it can access L2 using the known way information that remained in the location cache.

This extension is powerful in that it allows several location caches to be utilized against a single target cache. Since only two states are added, *Shared* and *Invalid* , only a single additional bit of storage for each line is required in a location cache. Combined with the fact that location caches can be efficient with a very small number of lines [15], very little overhead is created by using this modification. Similar to the case of shared location caches in CMP systems, if the L2 coherency in handled by the MESI protocol, no protocol bits are required in the location caches for L2 data coherency. In summary, the protocol bit added in each location cache line is used for L1 cache coherency; the MESI protocol bits used for L2 will take care of any L2 data coherency issues automatically.

Chapter 4

Low Leakage Cache Design and Location Cache Support

The gated-ground technique was developed to reduce the amount of leakage power consumed in large cache systems [14]. Currently, neither gated-ground caches nor location caches are supported by the widely-used CACTI cache power estimation tool. In this chapter we will discuss the utilization of location caches in combination with a gated-ground cache, as well as the modifications made to CACTI to support such an architecture.

4.1 Support for Gated-Ground Caches

A great deal of leakage power is consumed by the cells in a cache as large as the two 4MB L2 caches in the Intel Xeon E7320. In its default form, CACTI calculates the leakage power in normal mode. Many modern processors utilize the

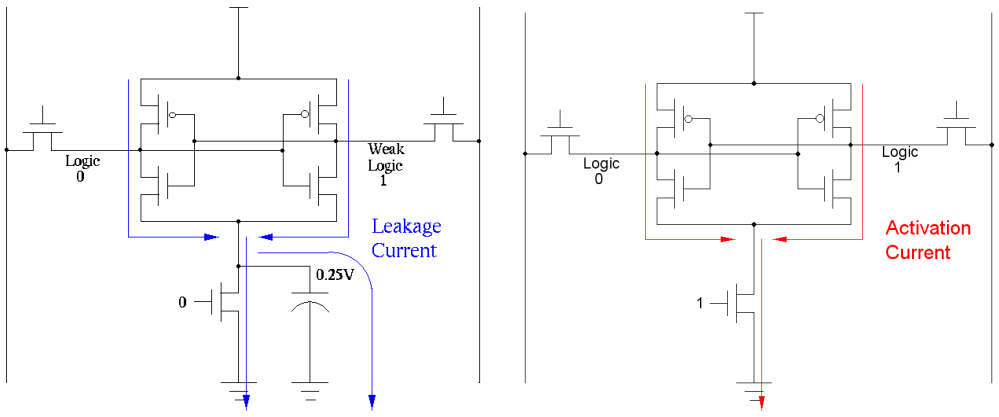
gated-ground technique to reduce the leakage power consumed by these increasingly large caches. The following sections document the changes made to allow CACTI to compute power estimates for these types of caches.

4.1.1 Calculation of Leakage Power

CACTI was modified to allow the computation of gated-ground leakage by utilizing a new scaling factor, $R\_normal\_to\_gated$. Computation of this ratio between normal mode leakage and gated-ground leakage was accomplished using HSpice. A $65nm$ process was utilized to construct and measure the leakage through a single cell operating in the normal mode. In addition, a row of 16 cells was constructed to share a single gated-ground transistor sized at 300λ, , where λ is $32.5nm$ for our $65nm$ process, and the leakage through one of these cells was computed for comparison to the normal mode.

$$R\_normal\_to\_gated = \frac{Gated\_leakage\_power_{HSpice}}{Normal\_leakage\_power_{HSpice}} \tag{4.1}$$

The ratio of the leakage between normal and gated-ground modes, shown in Equation (4.1), was found to be 0.114227. Thus, a cache in gated-ground mode consumes just 11% of the power consumed by the same cache in normal mode. This value is multiplied by the leakage power obtained directly from CACTI to arrive at an approximation of the leakage power consumed by the cache in gated-ground mode. Note that the virtual ground voltage under this experimental set up is $0.25V$, while the normal power supply is $1.1V$.



(a) (b)

Figure 4.1: Relationship between Activation and Leakage energy.

4.1.2 Calculation of Activation Energy

In order to provide an accurate measure of the power used in a gated-ground cache, it is necessary to compute the energy consumed when a cache line is activated. CACTI provides no built-in capability for calculating the activation energy of a gated-ground cache, so a modification had to be devised. In order to ensure that the computed activation energy directly correlated with the rest of our results, we could not simply use the measurement from HSpice. Instead, a method was developed to directly relate the activation energy to a known value that CACTI could already compute.

Activation energy is closely related to leakage energy. In Figure 4.1(a), the leakage current through an SRAM cell in the gated-ground sleeping mode is shown. Figure 4.1(b) shows the same cell as it is being activated. These two figures illustrate that the leakage energy leakage energy and the activation energy consumed by the cell during the wake-up phase are closely related. The activation

energy is nothing more than an increased amount of leakage energy, and it is for this reason that we have chosen to base our calculation of the activation energy on a memory cell's leakage energy.

$$R\_normal\_activation = \frac{E_{wake-up}}{E_{normal\_leakage\_0.3ns}} \tag{4.2}$$

As in the previous section, a single normal mode cell and a set of 16 cells sharing a 300λ gated-ground transistor are examined using HSpice. The wake-up time of the gated-ground array was determined to be approximately 0.3ns. Then the amount of leakage energy that is consumed during this period was computed. These values were used to compute a ratio between the leakage energy and the activation energy during the wake-up period, as shown in Equation (4.2).

$$Activation\_energy = 2.828 \times I_{cell} \times Vdd \times N_{bits\_per\_line} \tag{4.3}$$

The ratio between the activation energy and the normal mode leakage was determined to be approximately 2.828. Using this value, we were able to modify CACTI such that it could compute the activation energy of a line in the gated-ground mode. This was accomplished by multiplying the normal mode leakage calculation by our ratio, $R\_normal\_activation$, resulting in Equation (4.3). The result of this equation is the activation energy of a single line of the cache that is directly correlated to the rest of the CACTI power output.

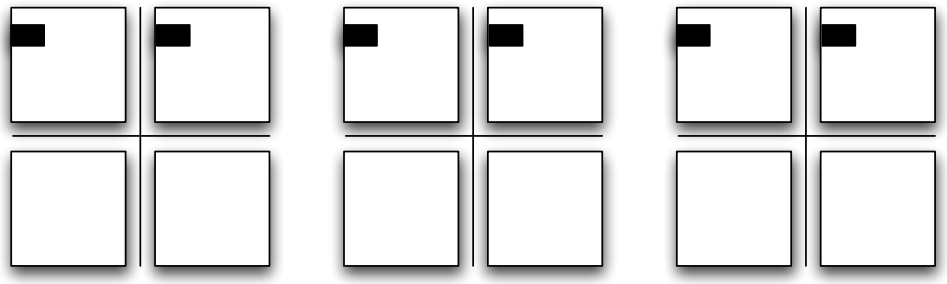


Figure 4.2: Default CACTI Cache Access Scheme.

4.2 Location Cache Support

CACTI provides no built-in support for location caches, so CACTI 5.0 was revised to support such a model. The cache configuration generated by CACTI lends itself well to the inclusion of a location cache. CACTI was modified to allow activation of the cache on a subarray by subarray basis, allowing a greater degree of control over the portions of the cache which will be activated.

CACTI places data in a given line by evenly dispersing the bits among an entire rows of subarrays, as shown in Figure (4.2). The shaded portions of the subarrays represent the cells that make up a single cache line access in an L2 cache, for example. The H-Tree configuration of the cache allows signals to propagate through all cells simultaneously, resulting in an architecture that is resistant to hot spots. However, this configuration is not conducive to power savings when a location cache is connected. Since the data is dispersed through the row of subarrays, each subarray must be activated in all cases.

We propose a change that involves packing the entire cache line into a single subarray, as shown in Figure (4.3). Here an access that used to require bits from

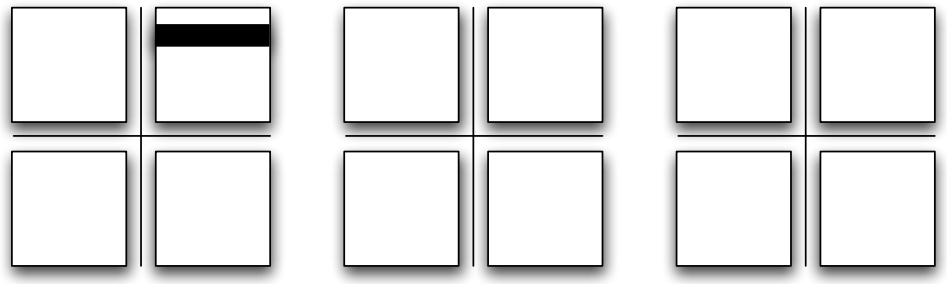


Figure 4.3: Proposed CACTI Cache Access Scheme.

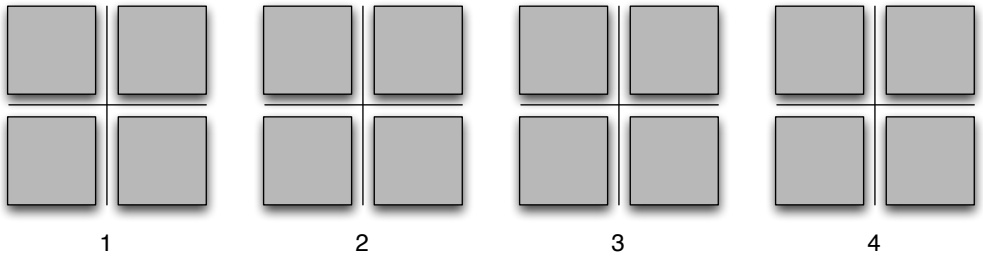


Figure 4.4: Mats Activated When *Ndwl* = 8.

six subarrays now involves data from only a single subarray. While such an implementation in an L1 cache may lead to hot spots in certain areas of the chip, the L2 is accessed with such relative infrequency that hot spots can safely be avoided. In addition, when CACTI is not using bitline multiplexing, such a modification will not increase chip area due to the fact that CACTI places a sense amplifier for each bit line [20].

$$number\_mats\_horizontal\_direction = \frac{Ndwl}{2} \tag{4.4}$$

This modification allows for significant power savings in the subarrays, as well as minor savings in the predecoder blocks. In the CACTI code, many power

calculations rely on the computation of the number of active mats. By default, CACTI calculates the number of active mats as in Equation (4.4). If we assume a cache with an *Ndwl* of 8, 4 mats will be activated, as shown in Figure 4.4.

Our modification involves altering this value, allowing us to activate mats or subarrays individually. $Ndwl$, the number of divisions in the word line, also indicates how many subarrays are configured across a row of the cache. Given that a mat is two subarrays wide, we can determine that the number of mats present along a row of the cache is calculated using Equation (4.4). CACTI assumes that this entire row of mats must be activated during an access.

$$subarrays\_per\_set = \frac{Ndwl}{Nspd} \tag{4.5}$$

$$mats\_per\_set = \frac{Ndwl}{4 \times Nspd} \tag{4.6}$$

$$LC\_hit\_mats = \frac{Ndwl}{4 \times Nspd \times Associativity} \tag{4.7}$$

To allow for simple modification of the CACTI code, we have altered this equation to account for the presence of a location cache. First, in order to allow for a subarray activation scheme, and thus partial mat activation, the equation needs to be capable of producing fractional results. There are four subarrays in a single mat, so each subarray is represented in the following equations by increments of 0.25 mats. Second, the proper number of subarrays required for an access in the case of a location cache hit needed to be determined. Recall that $Nspd$ is the number of sets present on a given line in the cache. Therefore, the number of

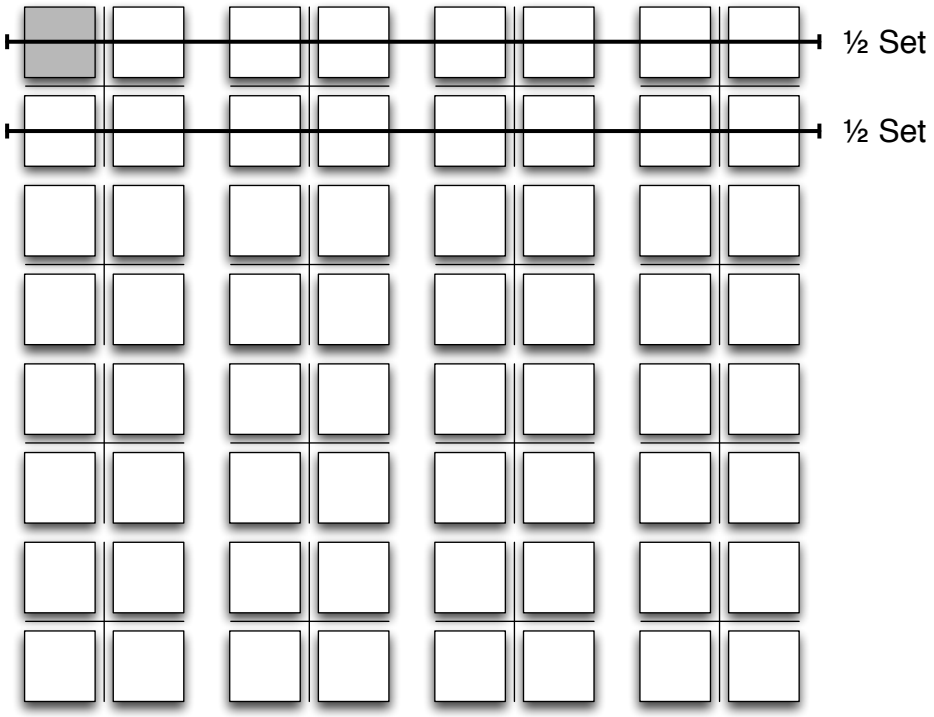


Figure 4.5: Mats Activated When *Ndwl* = 8, *Nspd* = 0.5, and Associativity = 16.

subarrays required to store a single set of data is shown in Equation (4.5). Given that there are four subarrays per mat, the number of mats required to store a single set of data is shown in Equation (4.6).

For example, CACTI determined the optimal configuration of the 16-way set associative 2MB L2 caches used in this work to be $Ndwl = 8$, $Ndbl = 8$, and $Nspd = 0.5$ as shown in Figure 4.5. Applying this configuration to the default CACTI calculation shown in Equation (4.4) results in 4 mats utilized during an access. In Figure 4.5, since $Nspd = 0.5$, the entire set of cache data is distributed into two rows of subarrays. According to Equation (4.5), the set of data is distributed into 16 subarrays ($\frac{8}{0.5}$), which is 4 mats ($\frac{8}{4\times0.5}$) by Equation(4.6).

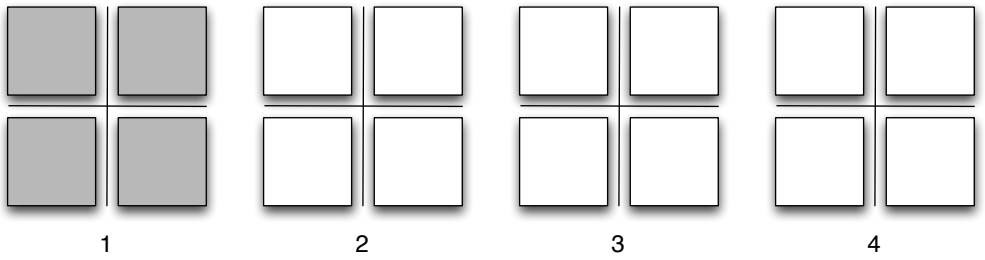


Figure 4.6: Mats With Predecoders Activated When *Ndwl* = 8, *Nspd* = 0.5, and Associativity = 16.

In the case of a location cache hit, only a single way from the set needs to be accessed because the way number for this address in L2 was stored during a prior transaction. Due to the packing structure described above, we can divide the total number of subarrays used during a location cache miss by the associativity of the cache to determine the number of subarrays required to access a single way of data. Incorporating this into Equation (4.6) results in Equation (4.7), the new equation utilized by CACTI to determine the number of subarrays to access upon a location cache hit.

In the case of a location cache hit, Equation (4.7) results in a value of 0.25, implying only a single subarray must be accessed (0.25 mats). The CACTI values affected by this change are shown in Table (4.1). The bulk of the savings can be attributed to the reduction of cells being accessed, which is included in the power computation for the bitlines in CACTI.

$$predecoder\_LC\_hit\_mats = \left\lceil \frac{Ndwl}{4 \times Nspd \times Associativity} \right\rceil \tag{4.8}$$

There are, however, additional minor savings attributable to components of the predecoder block. For the computation of the predecoder components shown

Table 4.1: Cacti Power Variable Modifications

CACTI Power Variable	$number\_mats\_horizontal\_direction$ replacement variable
Bitlines	Bitline_Mats
Bitline Precharge EQ Drivers	Bitline_Mats
Subarray Output Drivers	Bitline_Mats
Row Predecoder Block Drivers	Predecoder_Mats
Bit Mux Predecoder Block Drivers	Predecoder_Mats
Sense Amp Predecoder Block Drivers	Predecoder_Mats
Row Predecoder Blocks	Predecoder_Mats
Bit Mux Predecoder Blocks	Predecoder_Mats
Sense Amp Predecoder Blocks	Predecoder_Mats
Row Decoders	Predecoder_Mats
Bit Mux Decoders	Predecoder_Mats
Sense Amp Decoders	Predecoder_Mats

in Table (4.1), Equation (4.4) is replaced with Equation (4.8). This equation produces a whole number representing the number of mats whose predecoders must be activated. Our previous example where LC_hit_mats was assigned a value of 0.25 implies a $predecoder\_LC\_hit\_mats$ value of 1, as shown in Figure (4.6). This is due to the fact that each mat is equipped with a single predecoder. In order for the address to be decoded and utilized properly, if any number of subarrays within a given mat are accessed then that mat's predecoder must be active.

Chapter 5

Location Cache Power Simulation

In this Chapter we will discuss our use of the Simics system simulator to simulate a location cache implementation for CMP machines. Section 5.1 discusses how Simics was modified to support the simulation of location caches. Section 5.2 discusses the activation policy used throughout the work. In Section 5.3, the process used to run benchmarks in Simics is presented. Finally, Section 5.4 details the extensions made to the simulator to allow for cycle-by-cycle power estimation.

5.1 Simics Location Cache Implementation

The location cache implementation for Simics is based upon the *g-cache* module provided by Virtutech [21]. *G-cache* is a fully-functional cache system implementation with detailed transaction and timing statistics. This implementation allows a variety of caches to be designed and constructed in a modular fashion. The flexibility afforded by this system was used to implement a location cache module that could easily interface with the pre-existing *g-cache* code.

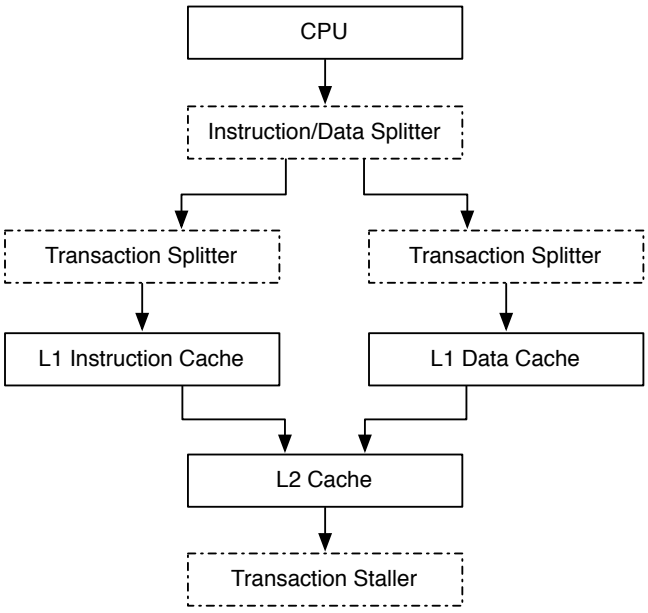


Figure 5.1: Configuration of a Simics Cache System [26].

As a system is simulated, each memory transaction initiated by the CPU is passed along to the *g-cache* modules. The modules are then free to examine and operate on the transaction, or pass it along to the next connected module. When applied in the x86 architecture, several additional modules are required to get correct operation out of the *g-cache* system. The connection of these modules can be viewed in Figure 5.1. First, an *Instruction/Data Splitter* is required. This module determines if a given transaction is operating on data or is part of an instruction fetch, and routes the transaction to the proper cache. This will be utilized in systems with a split L1I / L1D cache, such as the cache configuration utilized by the Xeon E7320. The second additional module is called a *Transaction Splitter*. The x86 architecture allows unaligned memory accesses that cross cache line boundaries [25], so the *Transaction Splitter* separates one of these unaligned

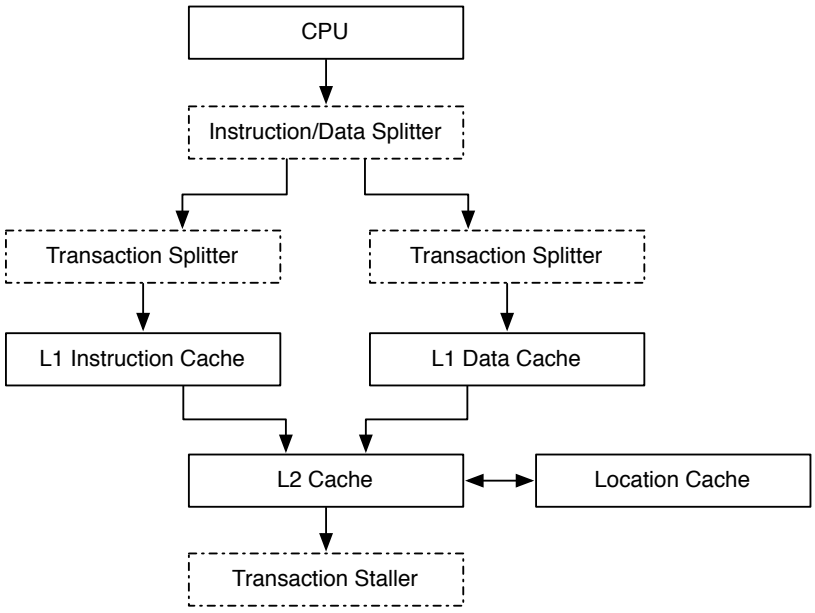


Figure 5.2: Configuration of a Simics Cache System Including a Location Cache.

accesses into two separate aligned accesses. The last additional module is the *Transaction Staller*. The *g-cache* system does not simulate main memory, so when an access would be sent to memory it is instead passed to the *Transaction Staller*. The purpose of this module is to simply simulate the latency of a memory access, so that the timing of the simulation may be preserved.

In order to fully take advantage of the location cache, the *g-cache* modules themselves were modified to utilize the location cache during operation. These modifications are crucial to determine accurate timing and power usage of the system when a location cache is connected. To accomplish this, a new location cache module was added to the *g-cache* system. The new configuration showing the connection of the location cache module to the cache system is shown in Figure 5.2.

Table 5.1: Additional Simics g-cache Attributes

Attribute	Description
$location\_caches$	Provides the link between a target cache and its location cache(s)
$enable\_gated$	Enables gated-ground cache operation
$lines\_per\_subarray$	Maps Simics' logical cache lines to match the physical structure from CACTI
$enable\_subarray\_activation$	Forces all lines in a subarray to be activated upon its access

Due to the structure of Simics' *g-cache* modules, it was necessary to alter the operation of the location cache somewhat to obtain correct operations. In hardware, the location cache operates in parallel with the L1 cache to provide L2 with location information. It is not possible to simulate this behavior from within the *g-cache* code, so a workaround was developed such that the location cache is accessed directly from the L2 module. This does, however, create an issue when tracking power usage as only transactions that make it to the L2 cache module would be accounted for. This issue and its resolution are discussed and addressed in Section 5.4.2.

To accomplish the proposed location cache system using Simics, a set of new attributes (included in the location cache module shown in Figure 5.2) were added to the existing *g-cache* codebase. These additions are shown in Table 5.1. The $lines\_per\_subarray$ attribute, in particular, is essential in that it allows us to provide a link between the cache structure provided by CACTI. With the number of lines present in each subarray, we can then activate lines on a subarray-by-subarray basis, which is done using $enable\_subarray\_activation$. These two attributes are required in order to properly estimate the power utilized in our

subarray-based activation technique.

This setup allows the Simics location cache module to accurately portray the hardware implementation of a location cache. Way information is stored within the module and retrieved for use by the L2 cache during L2 cache access operations, in addition to providing accurate hit ratios and power statistics.

The modular design of the *g-cache* system makes it easy to simulate the system with or without the location cache. The ability to switch back and forth rapidly eases comparison of the two cases.

5.2 Drowsy Cache Activation Policy

Lines in the L2 cache are put into gated-ground mode according to the drowsy cache policy set forth in [13]. To utilize this activation policy on our gated-ground cache, a *Sleep Controller* module was created in Simics. This module gives Simics the ability to simulate the process of putting lines to sleep and activating them.

At the start of a simulation run, the *Sleep Controller* puts the entire cache into gated-ground mode so that it is consuming the least amount of leakage power possible. When cache operations begin, the module activates a single counter to keep track of the number of CPU cycles elapsed since the entire cache was last driven into gated-ground mode. During L2 cache access operations, cache lines are activated on a subarray basis, discussed in detail in Chapter 4. These lines remain active until the next time the cache is globally driven into the gated-ground mode.

A cache's window size is the number of CPU cycles between these global

sleep events. Once the *Sleep Controller*'s counter exceeds the window size, the entire cache is returned to gate-ground mode as soon as the cache becomes idle and the counter is reset. This is important as the cache must not be driven into gated-ground in the middle of an access or the data written or read by the transaction would be corrupted. An 8000 cycle window size was chosen according to the optimum in-order execution window size determined in [11]. In addition, it has been shown that this simple windowing sleep policy can be as effective as more complicated policies in most cases [11].

5.3 Benchmarking

One of the most useful features offered by Simics is its ability to boot a standard operating system, such as a full Linux distribution. The drawback is that this ability complicates the process of gathering statistics on a given benchmark, as it becomes necessary to be able to recognize the start and end of a benchmark from the host machine.

Simics provides built-in utilities for gathering statistics on benchmarks, and this work makes extensive use of Simics' *Magic Instructions*. This feature involves a special instruction that is added to the standard x86 instruction set in the processor model used by Simics. The execution of this instruction, along with a simple integer passed as a parameter via the CPU's EAX register, allows a very rudimentary way for the target machine to communicate directly with the host. In this work we have used the simplistic method of calling this instruction with a parameter of 1 to indicate the beginning of a benchmark, and a 2 to indicate its end.

Python scripts running on the host monitor the target machine for the execution of one of these instructions. When encountered, the script differentiates between the two parameters signaling the beginning and end of benchmark execution, and handles the statistics gathering appropriately.

It is important to note that the benchmarks themselves did not need to be modified to include these extra instructions. Instead, two simple programs were created that consisted of only a single *Magic Instruction*. One program sends the signal to begin gathering statistics, the other to cease gathering statistics. These two programs are called in series with each benchmark by using a simple shell script on the target machine. This allows us to properly gather statistics in a reliable and repeatable fashion, without altering the benchmarks themselves in any way.

5.4 Power Estimation

One of the difficulties in utilizing Simics for this work was that Simics includes no provisions for gathering statistics on power usage. Due to the complexity of our modifications, it would be prohibitive or impossible to derive the proper power equations by hand. Instead, an extension to Simics was created to provide support for basic power statistics. The following sections discuss the creation and use of this extension.

5.4.1 Cache Power Estimation

This Simics *g-cache* module was further modified to allow for the calculation of dynamic read and write energies, as well as leakage power. The energy values

Table 5.2: Simics g-cache Power Variables

Attribute	Description	Energy
$E_{base\_leakage}$	Base leakage energy of the entire cache per CPU cycle when sleeping	3.09113E-10 J
$E_{active\_leakage\_1way}$	Active mode leakage energy per CPU cycle of 1 cache way	3.52848E-14 J
$E_{sleeping\_leakage\_1way}$	Gated mode leakage energy per CPU cycle of 1 cache way	4.02246E-15 J
$E_{dynamic\_read\_allways}$	Dynamic read access on location cache miss (16 ways)	8.11317E-10 J
$E_{dynamic\_write\_allways}$	Dynamic write access on location cache miss (16 ways)	7.8349E-10 J
$E_{dynamic\_read\_1way}$	Dynamic read access on location cache hit	4.96287E-10 J
$E_{dynamic\_write\_1way}$	Dynamic write access on location cache hit	6.46078E-10 J
$E_{activation\_1way}$	Activation energy for 1 cache way	4.57472E-14 J

supported by this modification are shown in Table 5.2, which are provided by CACTI, are shown with example values from the L2 cache with 64 byte lines used in this work. From these values it is possible to build up a working estimation of power consumption within Simics. All values are provided to Simics in Joules. For those values which are usually represented in terms of Watts, such as leakage power, conversion to Joules is performed by multiplying by the known CPU cycle time prior to passing them to the Simics.

$$\begin{aligned} E_{Leakage_{1cycle}} = E_{base\_leakage} - N_{active\_ways} \times E_{sleeping\_leakage\_1way} \\ + N_{active\_ways} \times E_{active\_leakage\_1way} \end{aligned} \tag{5.1}$$

Leakage power is computed on a cycle-by-cycle basis. Due to our modifications, Simics is aware of the number of active ways in each cache at all times. The leakage power consumed during every cycle is then computed by Equation (5.1). In addition, all dynamic energies are included in the power statistics. In

the case of a location cache miss, or in the absence of a location cache, the dynamic values $E_{dynamic\_read\_allways}$ and $E_{dynamic\_write\_allways}$ are used for $E_{dynamic}$. When a transaction results in a hit in the location cache, $E_{dynamic\_read\_1way}$ and $E_{dynamic\_write\_1way}$ are used. If a cache is implemented using the gated-ground technology, activation energy is also accounted for through the utilization of $E_{activation\_1way}$.

$$E_{Total_{1cycle}} = E_{Leakage_{1cycle}} + E_{dynamic} + E_{activation\_1way} \times N_{ways\_activated} \tag{5.2}$$

By utilizing Equation (5.2) on a CPU cycle-by-cycle basis in each *g-cache* module, it is possible for Simics to account for the total energy consumption of the system while benchmarking. Note that $E_{dynamic}$ in Equation (5.2) can be one of the four dynamic readwrite energy attributes shown in Table 5.2 depending on whether the access is a read or write, and a location cache hit or miss. Similarly, the value of N_{active_ways} and $N_{ways\_activated}$ also depend on whether the access is a location cache hit or miss. Dynamic energy for a single access is computed and added to the total energy during the cycle the access begins, regardless of how many cycles the access will actually take to complete. Therefore, if a cache access is not first initiated during this cycle, $E_{dynamic}$ and $N_{lines\_activated}$ are set to zero even though they may still be in progress. $N_{lines\_activated}$ also varies depending on the presence or absence of a location cache, and whether the transaction resulted in a location cache hit or miss.

This scheme was chosen for its simplicity and ease of implementation, but introduces a small amount of error upon ending the simulation. If the simulator

Table 5.3: Simics Location Cache Power Variables

Attribute	Description	Energy
$E_{leakage}$	Leakage energy per CPU cycle	3.85811E-13
$E_{dynamic\_read}$	Dynamic read access	1.23945E-12
$E_{dynamic\_write}$	Dynamic write access on location cache miss	1.55672E-12

is halted after an access has begun, but before it has completed, an amount of error no greater than the larger of $E_{dynamic\_write\_allways}$ and $E_{dynamic\_write\_1way}$ is introduced into the system. Due to the small amount of energy consumed during a single access, and given the sheer number of total accesses over the course of a simulation, the error introduced is negligible.

5.4.2 Location Cache Power Estimation

Estimation of the power consumed by the location caches is performed similarly to that for the *g-cache* modules themselves. Leakage energy is accumulated on a CPU cycle-by-cycle basis. When dynamic accesses to the entire cache system occur, both dynamic and leakage energies are included in the total energy utilized by the cache. The energy values supported by this modification are shown in Table 5.3, which are determined directly from CACTI and are shown with example values from a location cache with 32 entries. The energy consumed during requests for way information from the cache, which occurs for all accesses to L1, are included as $E_{dynamic\_read}$. The energy consumed during the replacement of way information is included as $E_{dynamic\_write}$.

$$E_{Lost} = N_{Lost\_Accesses} \times E_{dynamic\_read} \tag{5.3}$$

$$E_{Location\_Cache} = E_{dynamic} + E_{Leakage_{1cycle}} + E_{Lost} \tag{5.4}$$

Due to restrictions imposed by structure of the Simics *g-cache* modules, accesses to the location cache occur only upon accesses to L2. In order to make the behavior of the simulator consistent with that of the hardware implementation, an additional function was created to allow the location cache to keep track of all memory accesses that never make it to L2. Cache access operations that result in an L1 cache hit never require writing new way information into the location cache, so these accesses therefore consist entirely of dynamic reads. These accesses, along with the leakage and dynamic energy used since the last L2 access, are then added to the location cache's accumulated energy usage, as shown in Equation (5.4).

$N_{Lost\_Accesses}$ is the number of L1 accesses that have occurred since the last L1 miss. Recall that the structure of the *g-cache* modules requires the location cache to only operate on transactions that access L2, but the hardware implementation operates on all L1 accesses. $N_{Lost\_Accesses}$ provides a way to include the power consumed by these accesses, allowing for more accurate calculation of the power consumed by the location cache itself.

Chapter 6

Experimental Results

6.1 Experimental Environment

In this Chapter, both the details of the simulation environment and system architecture are given, and then the results of our experiments are presented. Simulation data such as location cache hit ratio and power savings rates are provided for both shared and private location caches in a CMP processor like the ones currently in use around the world.

6.1.1 System Architecture

The cache system utilized for this work is based on the architecture of the Intel Xeon E7320 processor, shown in Figure (6.1). The processor utilizes a pair of private split L1 instruction and data caches for each core, 64kB in size. The L1 caches are 2-way set associative with a line size of 64 bytes, for a total of 512 lines per cache. Each pair of processors share a 2MB L2 cache. Each L2 is 16-

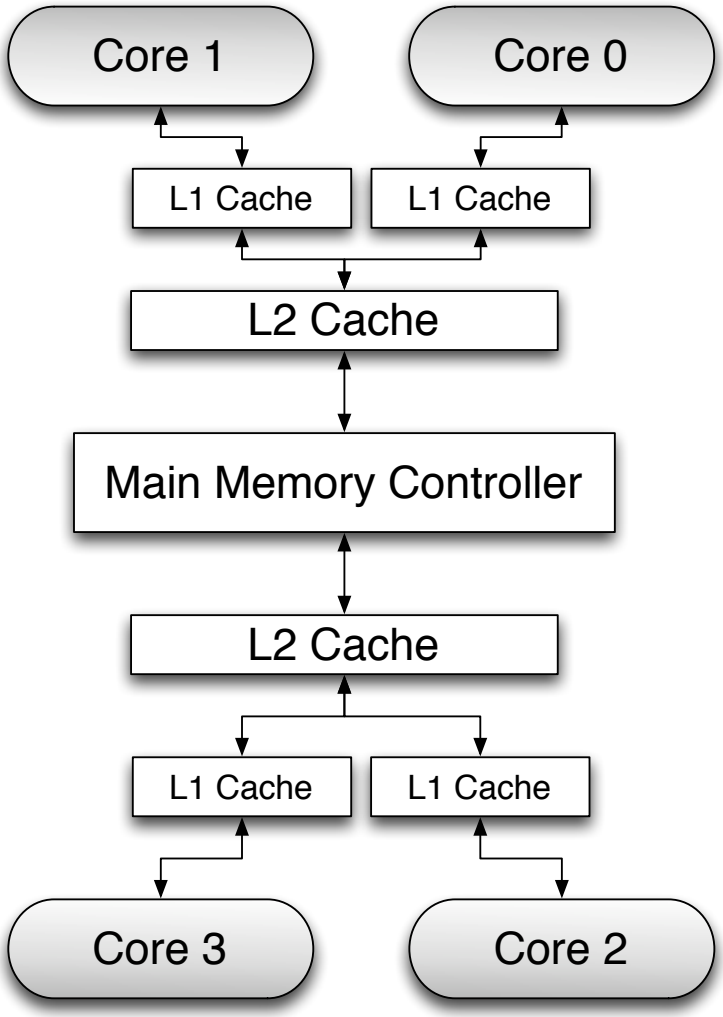


Figure 6.1: Intel Xeon E7320 Cache Configuration.

way set associative with a line size of 64 bytes, for a total of 32,768 lines per cache utilizing the gated-ground technique. The L2 cache's activation policy uses an 8000 cycle window size.

This quad-core processor is essentially equivalent to having a pair of dual-core processors on the same chip. This configuration was chosen to illustrate not only that a location cache architecture is capable of saving power for the currently-popular dual- and quad-core architectures, but that it can also scale as the number of cores placed on a chip inevitably increases.

6.1.2 Simulators

When evaluating the power usage of the cache system, and the location caches themselves, a modified version of CACTI 5.0 was used to evaluate the power consumption of the L1, L2, and location caches. The modifications performed to allow CACTI to more accurately model the behavior of the L2 caches in our architecture were discussed previously in Chapter 4.

The power estimation results provided by CACTI were then integrated into Simics, using the modification discussed in Chapter 5. This integration allowed Simics the ability to provide for cycle-by-cycle power estimation of our cache architecture. Simics' ability to boot a full operating system provided flexibility in the selection of benchmarks, ultimately allowing us to choose any benchmarks capable of being compiled under the Red Hat Linux 7.3 operating system.

6.1.3 Benchmarks

While the use of Simics for benchmarking allows for the selection of a vast array of single-threaded benchmarks, suitable multi-threaded benchmarks are somewhat scarce. For our research we will be using two multi-threaded benchmark suites, SPLASH-2 and ALPBench.

The Stanford Parallel Applications for Shared Memory (SPLASH) were updated in 1995 to include additional benchmarks for use in the evaluation of multiprocessor systems. SPLASH-2 [31] was designed to be portable among many platforms, and allowed for using macros to create different multithreading routines for different operating environments. To compile SPLASH-2 in our Red Hat Linux 7.3 operating environment, we utilized macros created by Bastiaan Stougie [32]. Unfortunately, there were issues compiling and running the Cholesky benchmark on our platform, so it has been excluded from this work. The SPLASH-2 benchmarks utilized in this work, along with a brief description of their purpose, is provided in Table 6.1.

The SPLASH-2 benchmarks were used, despite their age, for a number of reasons. First and foremost is to reduce simulation time. While the entire SPLASH-2 benchmark suite would have executed natively on the hardware modeled in this work in an estimated 7.8 seconds, simulation time using a 2.4GHz dual-core AMD Athlon exceeded 28 hours. To test all 32 configurations required a significant amount of processing time, even for these simple benchmarks. Second, this suite is still regularly used as a benchmark in recent works despite its age, is readily available, and supported compilation for our target operating system.

To temper the use of the older SPLASH-2 suite, we have also chosen a more

Table 6.1: The SPLASH-2 benchmark applications

barnes	3-dimension Particle interaction simulator
fft	Fast Fourier Transform kernel
fmm	2-dimension Particle interaction simulator
lu continuous	Matrix factorization using a 2-dimensional array
lu noncontinuous	Matrix factorization using contiguous blocks of memory
ocean continuous	Simulates large-scale ocean movements using 2-dimensional arrays
ocean noncontinuous	Simulates large-scale ocean movements using 3-dimensional arrays
radiosity	Computes the equilibrium distribution of light in a scene
radix	Radix sort kernel
raytrace	3-dimensional rendering using ray tracing
volrend	3-dimensional volume rendering using ray casting
water n-squared	Calculates forces acting on a water molecule
water spatial	Calculates forces on a water molecule with a more efficient algorithm

Table 6.2: The ALPBench benchmark applications

face_rec	Facial recognition
MPEG	MPEG-2 Encoding and decoding
Sphinx	Speech recognition
Tachyon	3-dimensional ray tracer

Table 6.3: SPLASH-2 Cache System Parameters

Location Cache	With, Without	2
Location Cache Entries	16, 32, 64, 128	4
Location Cache Type	Private, Shared	2
L2 Line Size	64 bytes, 128 bytes	2
L2 Latenency	18 cycles, 10 cycles	2

modern suite to run on a few of the more promising configurations. ALPBench is a suite for multithreaded multimedia applications developed by the University of Illinois at Urbana-Champaign and the Intel Corporation in 2005 [33]. Its benchmarking programs include MPEG-2 encoding/decoding, facial recognition, speech recognition, and raytracing, as shown in Table 6.2. These are all common uses of modern CMP systems, so this suite is ideal for our purposes. Unfortunately, the 183 hour simulation time on the previously-mentioned machine limited our use of this suite to only a few configurations.

6.2 Experimental Results

The SPLASH-2 benchmarking suite was run using the 64 ($2 \times 4 \times 2 \times 2 \times 2$) configurations described briefly in Table 6.3. ALPBench was run in the four ($2 \times 1 \times 1 \times 2 \times 1$) configurations created by fixing the Location Cache Type to Private and the number of Location Cache Entries to 128, as shown in Table

Table 6.4: ALPBench Cache System Parameters

Location Cache	With, Without	2
Location Cache Entries	128	1
Location Cache Type	Private	1
L2 Line Size	64 bytes, 128 bytes	2
L2 Latenency	18 cycles	1

6.4. The results of these runs are described in the sections below. In the following discussions we assume that the L2 access latenency is 18 CPU cycles and the L2 sleep window size is 8000 cycles. However, before the end of this section we also investigate the relationship between power savings rates (by adding a location cache) and the L2 access latenency.

6.2.1 Location Cache Efficiency

$$Lcache_{total\_misses} = Lcache0_{misses} + Lcache1_{misses} + Lcache2_{misses} + Lcache3_{misses} \tag{6.1}$$

$$Lcache_{total\_transactions} = Lcache0_{transactions} + Lcache1_{transactions} + Lcache2_{transactions} + Lcache3_{transactions} \tag{6.2}$$

$$Hit\_Rate = 1 - \frac{Lcache_{total\_misses}}{Lcache_{total\_transactions}} \tag{6.3}$$

Figures 6.2 and 6.3 show the hit rates of the location caches for all possible numbers of location cache entries for the SPLASH-2 and ALPBench benchmarks, respectively. The L2 line size was fixed at 64 bytes, and the private location cache configuration was utilized. For Figure 6.3, the number of location cache

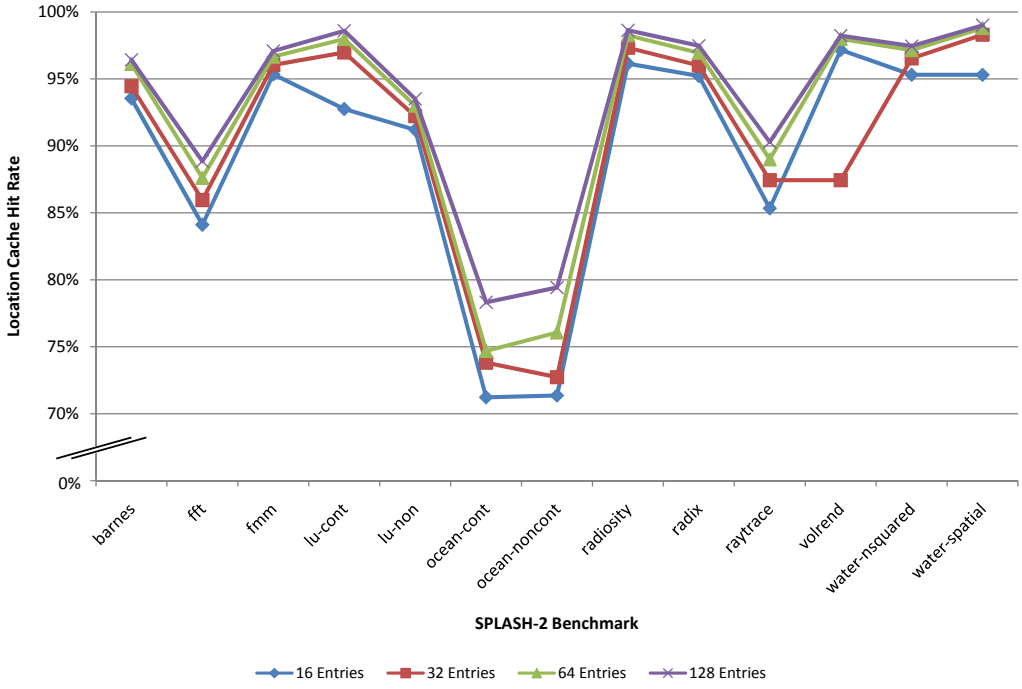


Figure 6.2: Overall hit rate of the connected location caches for SPLASH-2.

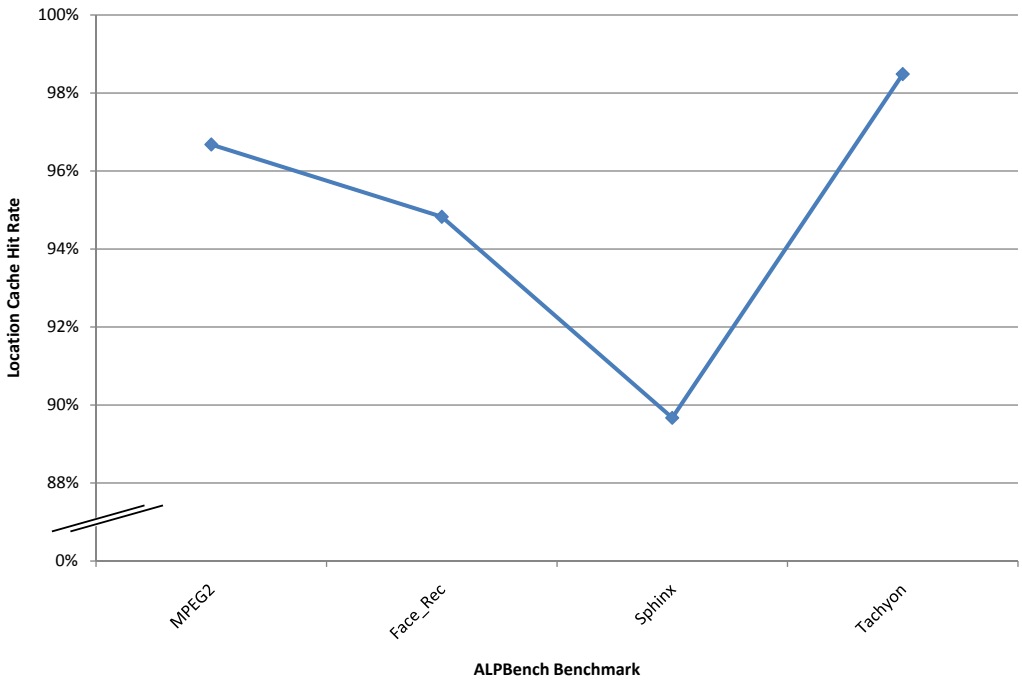


Figure 6.3: Overall hit rate of the connected location caches for ALPBench.

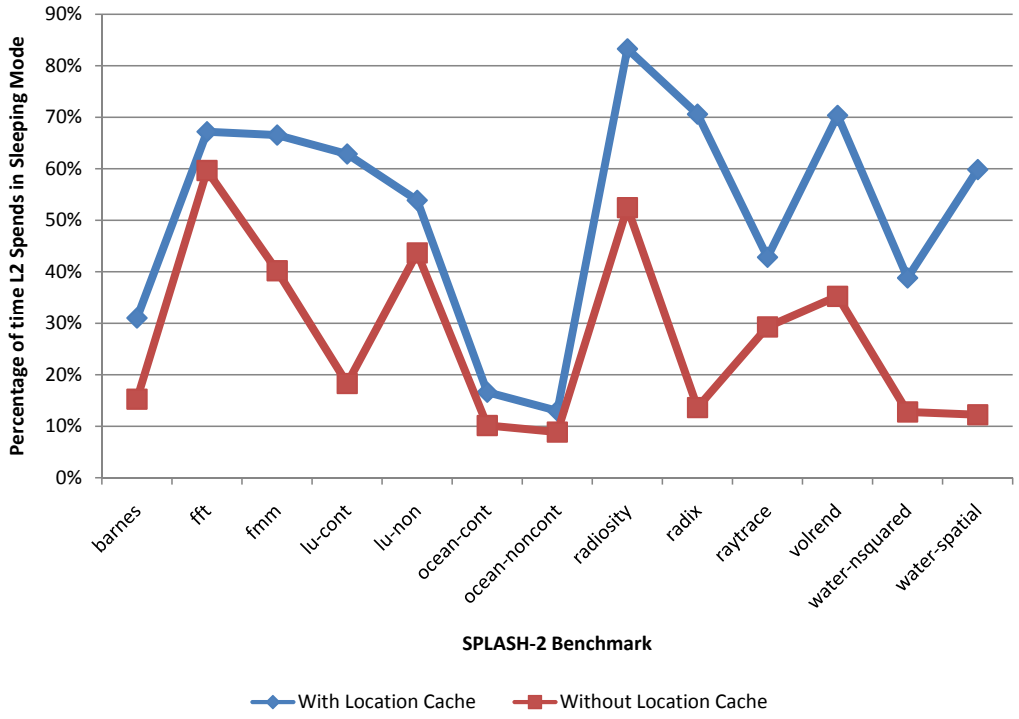


Figure 6.4: Percentage of time L2 spends in gated mode with and without a location cache.

entries is fixed at 128. The location cache hit ratio for our configuration was calculated using Equation (6.3). The location cache hit rates varied depending upon the number of location cache entries, where generally a higher number of entries resulted in a better hit rate. These figures indicate that the location caches are capable of operating efficiently in a CMP environment. In fact, 11 of the 17 benchmarks achieve location cache hit rates of over 95%.

$$Percentage\_Sleeping = 1 - \frac{\sum_{i=1}^{N_{L2\_ways}} T_{gated}(i)}{N_{L2\_ways} \times T_{simulation}} \tag{6.4}$$

These excellent hit rates translate directly to keeping the L2 cache in the gated-

ground mode for greater periods of time, as shown in Figure 6.4 that utilizes the same configuration as that of Figure 6.2. The percentage of time the L2 caches are in the gated-ground state is calculated according to Equation (6.4), where all times are given in CPU cycles. Here $T_{gated}\,(i)$ represents the amount of time in CPU cycles a given way in the L2 cache spent in gated mode over the course of the benchmark. N_{L2_ways} is the total number of ways in the L2 cache, and $T_{simulation}$ is the total simulation time of the benchmark in CPU cycles. When a location cache is not present or misses, the L2 must be accessed set-associatively, and thus requires waking all related lines from gated-ground mode. In the case of a location cache hit, fewer lines need to be activated, allowing a greater portion of the L2 cache to remain in the more efficient gated-ground mode. This advantage is clearly seen in Figure 6.4, where many of the benchmarks spend significantly more time in the gated-ground state. As we will see in the following section, this behavior translates directly to power savings.

6.2.2 Power Savings

When calculating the power savings realized by adding a location cache to an existing cache configuration, it is important to include the additional power utilized by the location caches themselves in the overall power statistics. Due to the small number of entries and small line size, even the frequent accesses to the location cache result in them consuming less than 0.5% of the total power consumed by the cache system. This is significant, as if even the worst case occurrs and a location cache is able to provide no power savings whatsoever, the additional power wasted through the use of the location cache is minimal. Therefore to achieve the

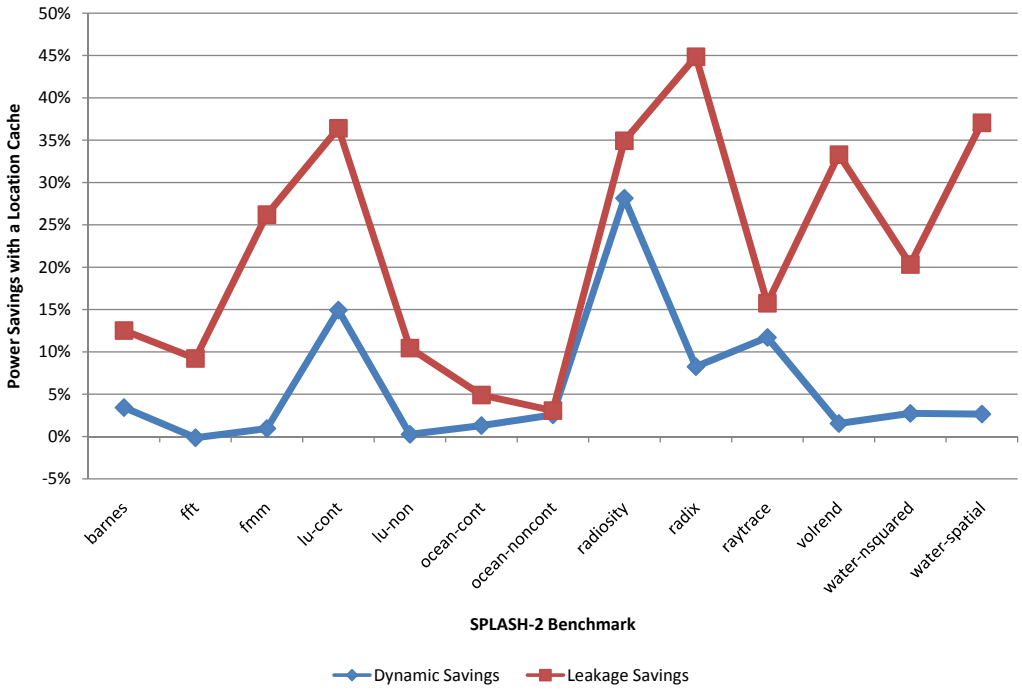


Figure 6.5: Percentage of dynamic and leakage energy saved by adding a location cache.

most accurate results the following statistics include not only power consumed by the L2 caches, but also the location caches and L1 caches. However, we emphasize that the power saving comes from L2 caches whose accesses are aided by the proposed location cache architecture.

Take the Radiosity benchmark from SPLASH-2, for example, with a cache configuration consisting of private location caches with 128 entries each, and an L2 line size of 64 bytes. In this case the L2 caches consume approximately $2.09W$, which is 67.6% of the total power consumed by the cache system. The L1 caches consume about $0.99W$ and the four location caches combined used about $0.01W$, which contribute 32.0% and 0.4% of the total power consumption of the cache system, respectively.

$$P_{no\_lcache} = P_{L1\_caches} + P_{L2\_caches\_no\_lcache} \tag{6.5}$$

$$P_{with\_lcache} = P_{L1\_caches} + P_{location\_caches} + P_{L2\_caches\_with\_lcache} \tag{6.6}$$

$$Power\_Savings\_Rate = \frac{P_{no\_lcache}}{P_{with\_lcache}} \tag{6.7}$$

While the location caches are capable of saving both dynamic and leakage power of L2, it was found that the majority of the power saved by introducing a location cache into a cache system was leakage power of L2. Equation (6.5) represents the total power consumed by a cache system without any location caches connected. Equation (6.6) represents the total power used by a second configuration created by adding location caches to the cache system used in Equation (6.5). The *Power Savings Rate*, or the percentage of power saved by adding location caches to any given configuration, can therefore be calculated using Equation (6.7).

Using an L2 line size of 64 bytes, and private location caches featuring 128 entries each, Figure 6.5 shows that, as a percentage, moving to a private location cache system saves more leakage power than dynamic power. The power savings rates are calculated by comparing the power reduction caused by adding the private location caches to the system. This is a particularly interesting finding, as the amount of chip area dedicated to the cache system is increasing rapidly, along with the leakage power attributable to this increase. Any structure that can

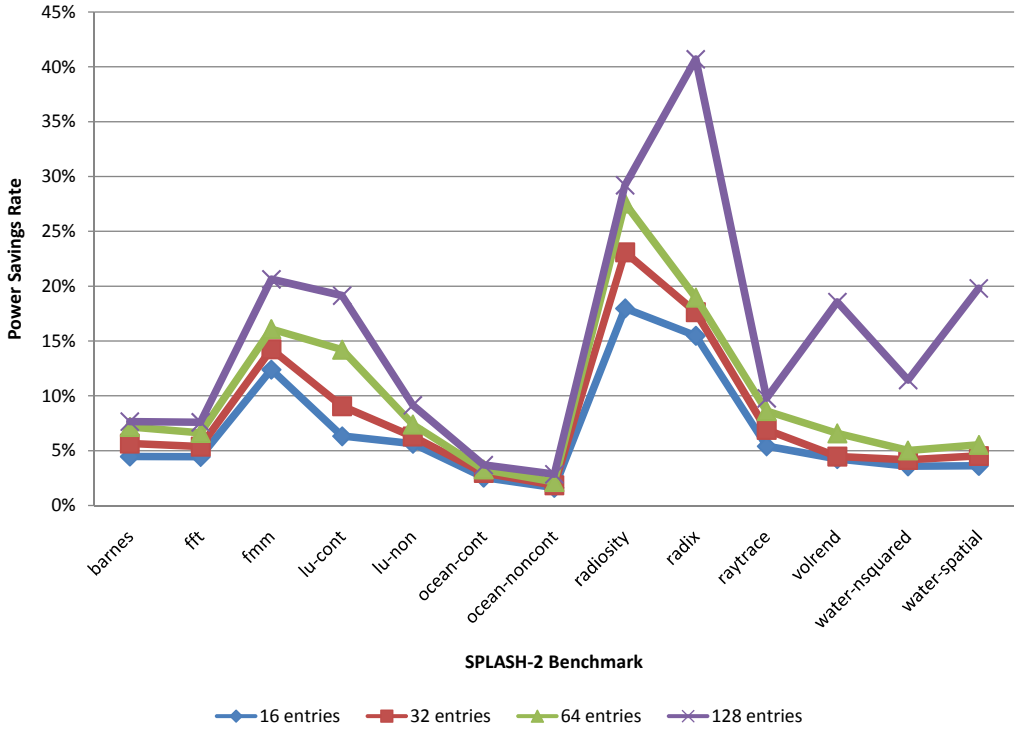


Figure 6.6: Power savings rate for 64 byte lines and 18 cycle L2 latenency using a shared location cache.

significantly reduce leakage power could become valuable for producing future low-power microprocessors.

In Figure 6.6 we can see that the addition of even a shared location cache can save between 2.5% and 40% of the overall power usage, depending on the benchmark and number of location cache entries. Moving to a private location cache system, as shown in Figure 6.7, increases the maximum savings to 42.5% with significantly improved average case performance. It is apparent that even with a very small number of entries, the location cache is able to save a significant amount of power through a variety of benchmarks. It is interesting to find that some of the benchmarks, such as lu-cont, are sensitive to the number of location

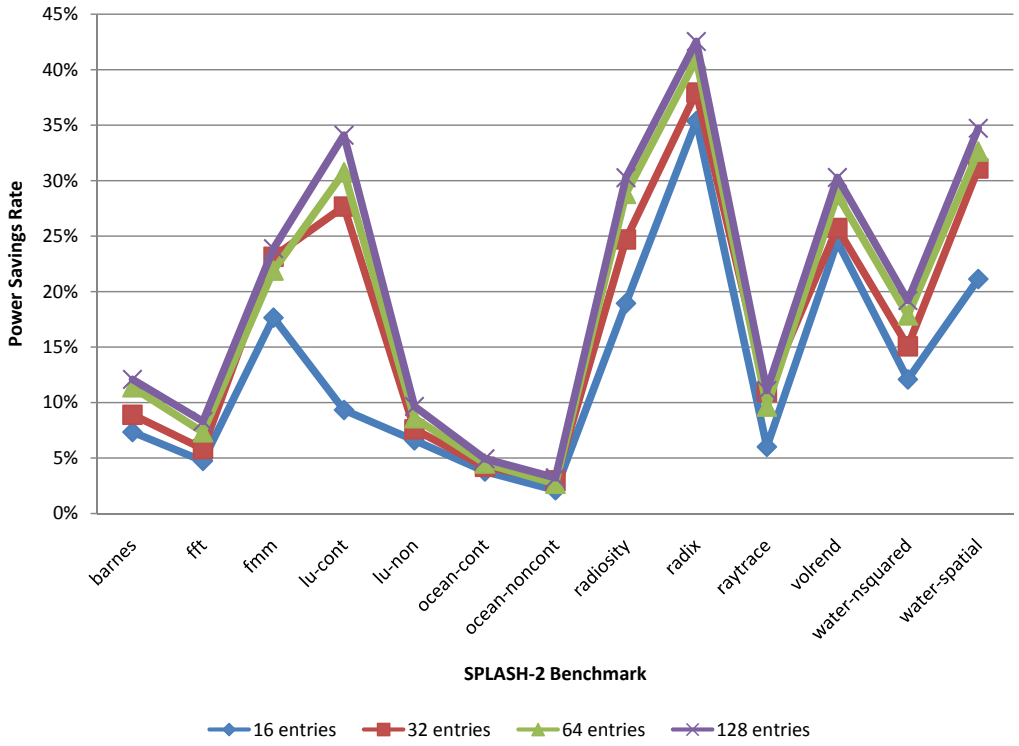


Figure 6.7: Power savings rate for 64 byte lines and 18 cycle L2 latenency using a private location cache.

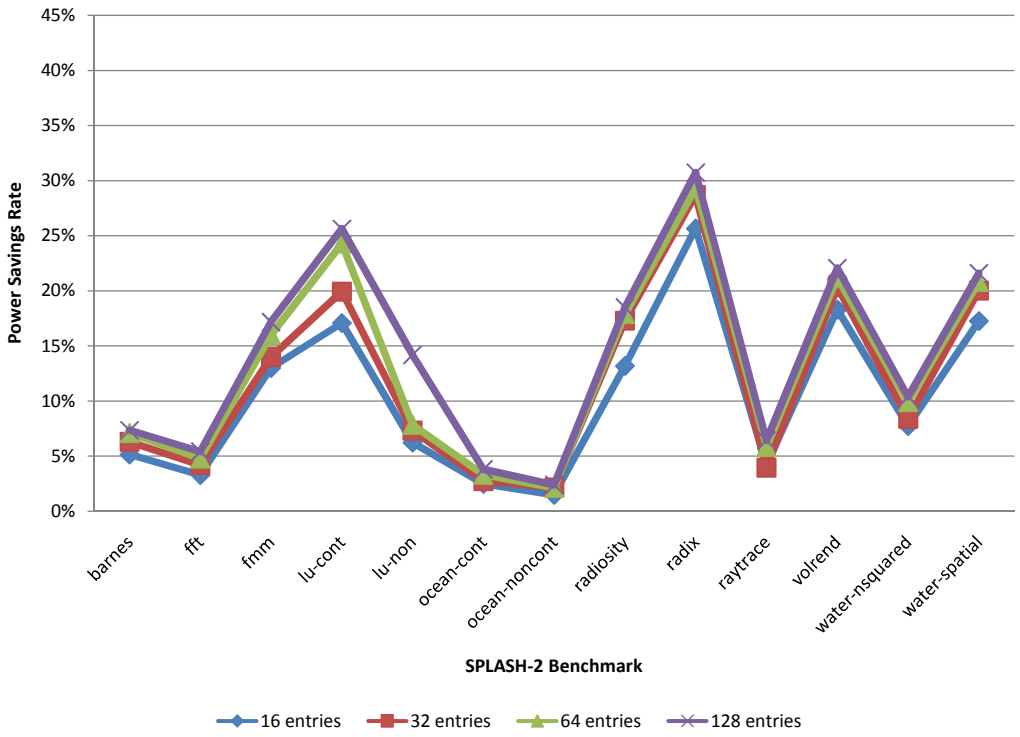


Figure 6.8: Power savings rate for 128 byte lines and 18 cycle L2 latenency.

cache entries, while benchmarks like ocean-cont are not sensitive at all.

Moving from a 64 byte to 128 byte L2 line size decreases the power savings realized by adding a location cache, as can be seen in Figure 6.8. Simply increasing the L2 line size from 64 bytes to 128 bytes drastically reduced the overall power consumption of the cache system. This reduction results in a more efficient cache configuration, leaving less room for the location cache to save additional power. Despite these findings, the location cache is still quite effective in the cases where a 128 byte L2 line size is chosen. This is illustrated in Figures 6.9 (SPLASH-2) and 6.10 (ALPBench), which show the amount of power saved by using a private location cache with both 64 byte and 128 byte L2 lines. While the overall savings

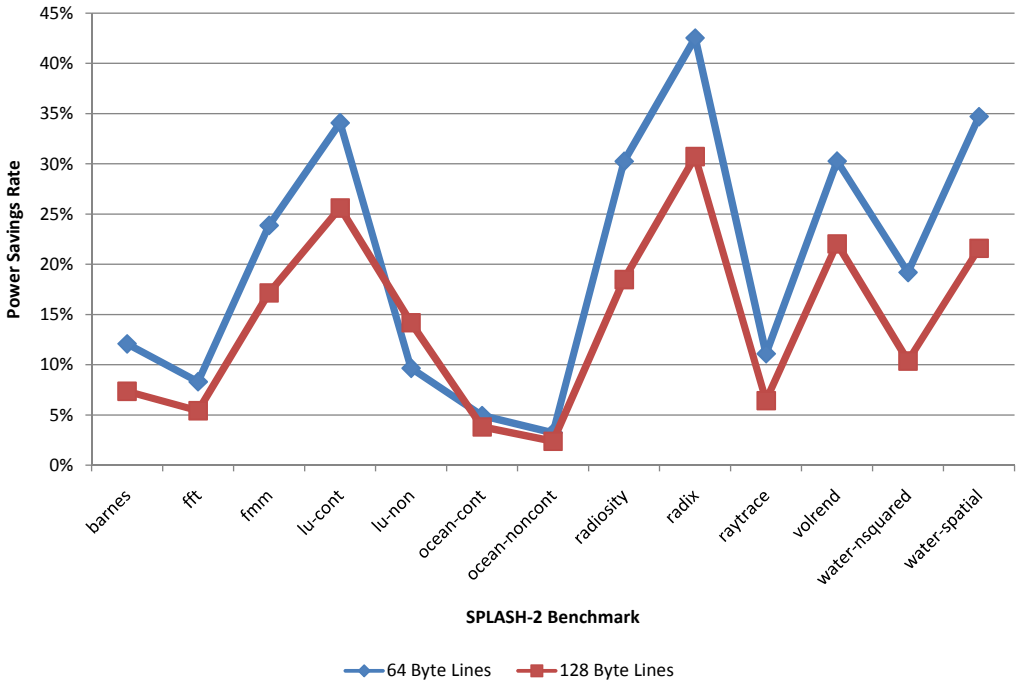


Figure 6.9: Comparison of power savings by location cache using a 64 byte vs. 128 byte L2 line size for SPLASH-2.

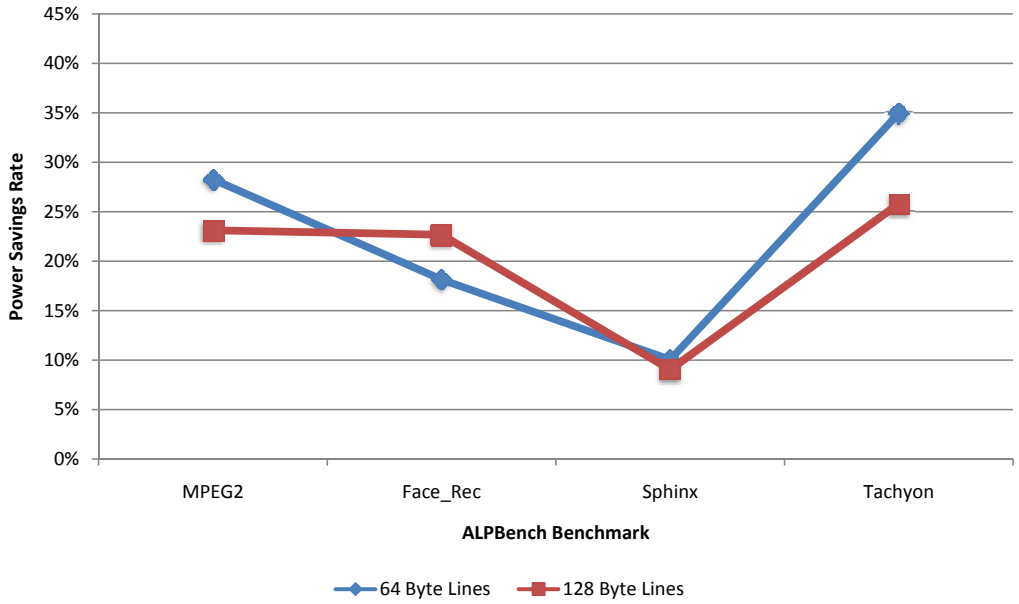


Figure 6.10: Comparison of power savings by location cache of using a 64 byte vs. 128 byte L2 line size for ALPBench.

provided by adding a location cache to the L2 with 64 byte line size was greater, savings were still provided when the L2 was moved to 128 byte lines.

To better illustrate this point, we have plotted the use of both of these techniques in Figure 6.11. The baseline of Figure 6.11 represents a cache with 64 bytes in each L2 line and a 18 cycle L2 latenency. The following three configurations are plotted against the baseline:

1. The baseline with the addition of 128 entry private location caches
2. The baseline, modified to support a 128 byte L2 line size
3. The baseline, modified to support a 128 byte L2 line size along with the addition of private location caches with 128 entries

While each optimization performs well on their own, when combined they

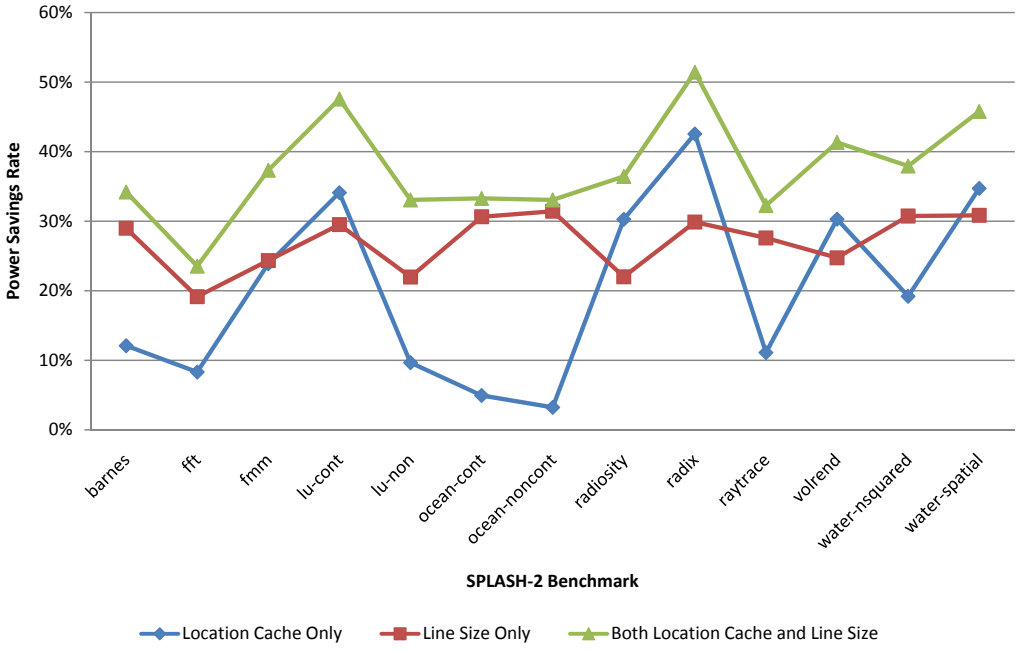


Figure 6.11: Comparison of power savings using longer 128 byte L2 lines and 128 entry location caches.

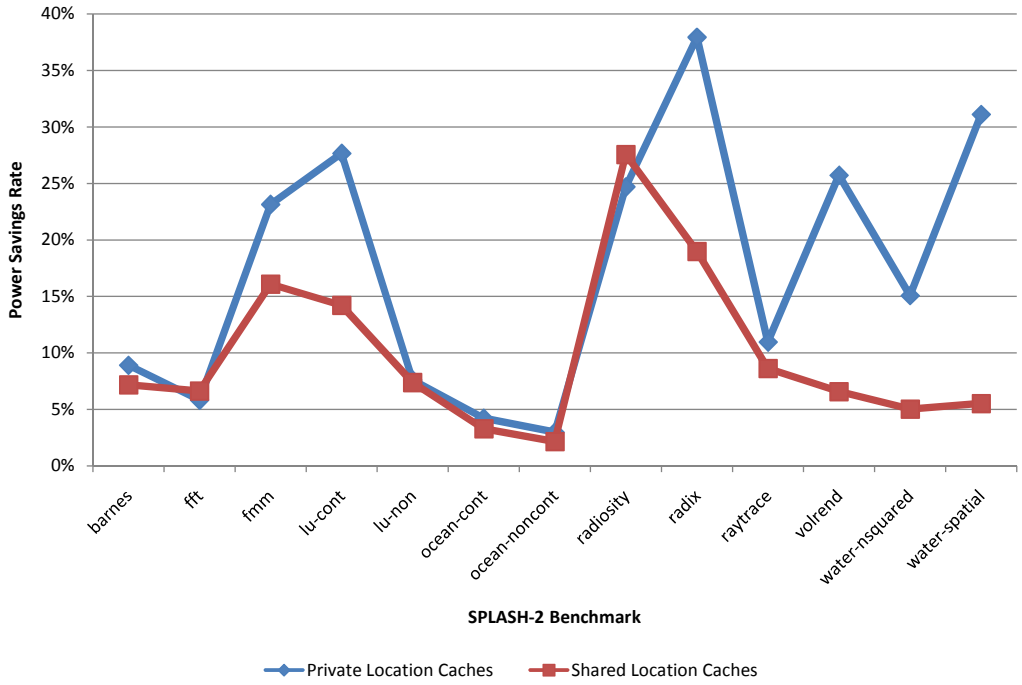


Figure 6.12: Comparison of power savings of the Shared vs. Private location cache implementations.

yield an average savings of about 35% over all benchmarks with a maximum savings of over 50%. This shows that, when paired with an increased L2 line size, private location caches can be a powerful tool in decreasing the power consumption of the increasingly large cache systems of modern processors.

The concept of a shared location cache is promising because it is easy to implement, and provides some energy savings over a cache configuration that lacks a location cache. While the power savings rates for shared and private location caches were previously discussed separately in Figures 6.6 and 6.7, respectively, we will compare them in greater detail here. Unfortunately these benchmarks showed that the configuration with shared location caches exhibited a significant amount of replacement happening as the multiple cores fought over the small

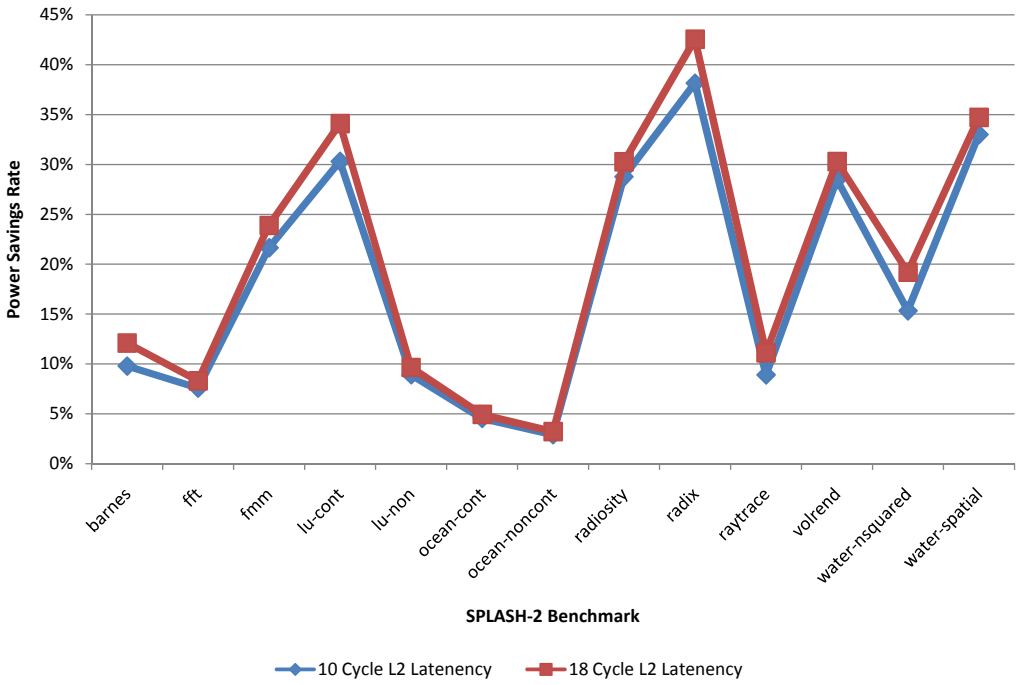


Figure 6.13: Comparison of power savings of using a 10 cycle and 18 cycle L2 latenency.

number of available location cache lines. Dedicating a location cache to each processing core fared better, as shown in Figure 6.12. Here the shared location caches were provided 64 lines each and the private location caches 32 lines each. This was done to keep the total number of location cache lines in each configuration constant at 128 total lines. About half of the benchmarks proved very sensitive to being assigned a private location cache, and two performed slightly better using shared location caches. This solidifies our position that in a CMP system each core must have its own location cache in order to operate at peak efficiency.

One major concern about the location cache design involved the latenency of the target cache. If the sleep window size remained constant, it was thought that a significant decrease in the target cache's latenency could negate any power savings

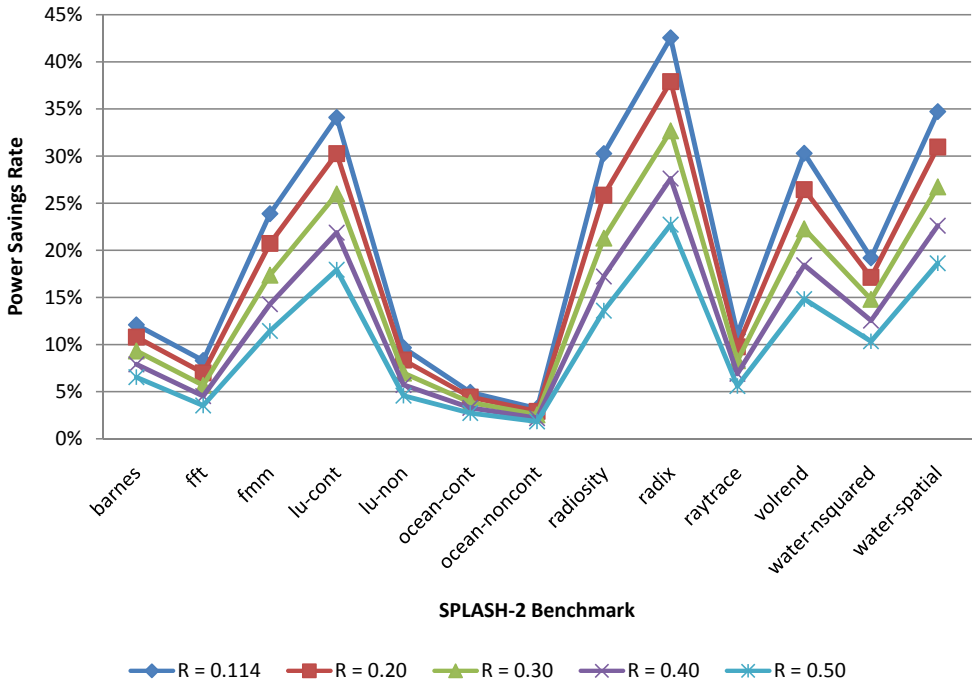


Figure 6.14: The effect of the ratio between gated and normal leakage on power savings.

provided by the location cache. This was found not to be the case, as is shown in Figure 6.13. The figure shows that while altering the latenency of the target cache does have some effect on the performance of the connected location caches, this effect is minimal for even large fluctuations in latenency. This behavior will allow a cache designer some flexibility, and proves a location cache can be a beneficial addition even when paired with a fast target cache.

Another concern was that the value of $R\_normal\_to\_gated$ would have a significant impact on the power savings provided by the location caches. As discussed previously in Section 4.1.1, we calculated the value of $R\_normal\_to\_gated$ to be about 0.114. While this falls in line with previous calculations at other institutions [34], further testing was performed by varying $R\_normal\_to\_gated$

values between our calculated 0.114 and 0.5 to determine its effect on power savings. The results of this experiment are presented in Figure 6.14. The value of $R\_normal\_to\_gated$ does have a significant, although linear, effect on the power savings provided by the location cache. While this was expected, we also found that even when $R\_normal\_to\_gated$ was set to 0.5 that the location cache was still able to save some amount of power in all benchmarks.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this research we have analyzed the power savings realized by utilizing location caches in CMP systems. The working principal of the location cache system is reviewed, and extensions to this principal are proposed to allow location caches to support CMP systems. CACTI 5.0 is modified to grant it support of gated-ground caches, and is then used to provide detailed power consumption estimates for all of the caches in this work. Simics is extended to support cycle-by-cycle power estimation using the power consumption statistics gathered from CACTI. Simics is then used to simulate the SPLASH-2 and ALPBench benchmark suites over a variety of cache configurations. Our goal is to explore the the power savings provided by the location cache over a variety of cache configurations.

We found that the amount of power saved by adding a location cache varies quite significantly depending upon the setup of the tested parameters. The tested

location caches were able to save power over all tested configurations and benchmarks, though they were far more effective at reducing the amount of leakage power than dynamic power. The number of entries in the location cache displayed a surprisingly small effect on the cache's overall power savings, with only about six of the benchmarks showing sensitivity to this parameter. On the other hand, assigning private location caches was a far more effective use of resources. Our simulations show that adding private location caches to the cache system can save between 2% and 43%, depending on the benchmark, of the power utilized by an equivalent cache system without location caches. In addition, the power savings did not appear to be dependant on the latenency of the L2 caches, which indicates that the location caches could be effective in systems with high-speed L2 caches. While the location caches themselves provided less power savings when moving to an 128 byte L2 line size, this combination of techniques provided the most significant power reduction: as much as a 50% savings in some of the benchmarks. This shows that location caches can be a valuable tool for reducing leakage power consumed by a cache system.

7.2 Future Work

Looking towards possible future work, we suggest the following:

1. A significant portion of the power consumed in the large L2 caches simulated in this work was contributed by the interconnects. Currently, a cache system containing location caches offers no interconnect power benefit over a system lacking location caches. To further reduce power consumption

the development of a location cache model that supports *Network on Chip* (NoC) can be explored. Combining location caches with NoCs may provide additional power savings through interconnect power reductions.

2. Operating systems typically treat a CMP as if it were simply a collection of completely unrelated processing cores. Recent work in [35] provides background on making the OS aware of the structure of the CMP cache system, which may provide additional power savings using more effective access techniques. It would be advantageous to learn the effect of these OS techniques on the power savings provide by location caches in CMP systems.

3. This work was limited to simulation based upon the cache configuration of the Xeon E7320. While this provides a look at the performance of location caches in a modern processor, it would be helpful to evaluate their performance over a more diverse set of cache configurations. Such research may identify further optimizations that may be made to the location cache principal. In addition, viewing location cache performance over a variety of cache window sizes may help determine how sensitive location caches are to the low leakage policy used.

4. Finally, recent research suggests that CMPs utilizing 80 or more cores are already on the horizon [1]. While a private location cache scheme has proven itself effective over four cores, it is unlikely that such a configuration will scale to CMPs with so many processor cores. A new location cache model, or extension of the current model, will need to be developed

in order to efficiently serve CMPs with dozens of cores.

Bibliography

[1] S. Vangal et al., "An 80-Tile 1.28TFLOPS Network-on-Chip in 65 nm CMOS" *In Proceedings of IEEE International Solid-State Circuits Conference*, Feb. 12, 2007.

[2] R. Varada, S. Tam, J. Benoit, K. Chou, "SOC Design Challenges in a Multithreaded 65nm Dual Core Xeon MP Processor," *International SOC Conference*, 2007.

[3] S. Rusu, et al., "A 65-nm dual-core multithreaded Xeon processor with 16-MB L3 cache," *IEEE Journal of Solid-State Circuits,* vol. 42, no. 1, pp. 17-25, Jan. 2007.

[4] "Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture," *Proceedings of International Symposium on High Performance Computer Architecture*, Feb. 2005.

[5] J. L. Hennessy, D. A. Patterson and D. Goldberg, *Computer architecture : a quantitative approach*, 3rd ed., Morgan Kaufmann Publishers, San Francisco, 2003.

[6] J. L. Hennessy, D. A. Patternson, *Computer Architecture: A Quantitative Approach*, 4th Edition, Morgan Kaufmann Publishers, San Francisco, 2007.

[7] K. Ghose and M. B. Kamble, Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line segmentation, *Proceedings of International Symposium on Low Power Electronics and Design*, pp. 70 75, 1999.

[8] C. Su and A. Despain, "Cache design tradeoffs for power and performance optimization: A case study," *Proceedings of International Symposium on Low Power Electronics and Design*, pp. 63–68, 1997.

[9] T. Lyon, E. Delano, C. McNairy, and D. Mulla, "Data cache design considerations for the itanium2 processor," *Proceedings of the 2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'02)*, pp. 356 –362, 2002.

[10] A. Hasegawa, I. Kawasaki, K. Yamada, S. Yoshioka, S. Kawasaki, and P. Biswas, "Sh3: high code density, low power," *IEEE Micro*, vol. 15, pp. 11–19, December 1995.

[11] K. Flautner, N. S. Kim, S. Martin, D. Blaauw and T. Mudge, "Drowsy Caches: Simple Techniques for Reducing Leakage Power," *Proceedings of 29th Annual International Symposium on Computer Architecture*, pp. 148-57 , May 2002.

[12] M. Powell, S. H. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar, "Gated-V_{dd}: a circuit technique to reduce leakage in deep-submicron cache memories," *Proceedings of International Symposium on Low-power Electronics and Design*, pp. 90-95, 2000.

[13] N. S. Kim, K. Flautner, D. Blaauw, and T. Mudge, "Circuit and microarchitectural techniques for reducing cache leakage power," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 2, pp. 167-184, Feb. 2004.

[14] A. Agarwal, H. Li, K. Roy, "DRG-Cache: A Data Retention Gated-Ground Cache for Low Power," *Proceedings of the 39th Design Automation Conference*, pp. 473-478, Jun. 2002.

[15] M. Rui, W. Jone, and Y. Hu, "Location cache: a low-power L2 cache system," *ACM/IEEE International Symposium on Low Power Electronics and Design*, pp. 120-125, Aug. 2004.

[16] S.Wilton and N. Jouppi., "An enhanced access and cycle time model for on-chip caches." *Western Research Lab Research Report 93/5*, Jun. 1994.

[17] G. Reinman and N. Jouppi., "CACTI 2.0: An integrated cache timing and power model." *Western Research Lab Research Report 2000/7*, Feb. 2000.

[18] P. Shivakumar and N. Jouppi., "CACTI 3.0: An integrated cache timing, power, and area model." *Western Research Lab Research Report 2001/2*, Aug. 2001.

[19] D. Tarjan, S. Thoziyoor, and N. Jouppi, "CACTI 4.0." *HP Laboratories, Technical Report*, 2006.

[20] S. Thoziyoor, N. Muralimanohar, and N. Jouppi, "CACTI 5.0." *HP Laboratories, Technical Report*, 2007.

[21] P. S. Magnusson et al. "Simics: A Full System Simulation Platform," *IEEE Computer*, vol. 35, no. 2, pp. 5058, Feb. 2002.

[22] Z. Chen, M. Johnson, L. Wei, K. Roy, "Estimation of Standby Leakage Power in CMOS Circuits Considering Accurate Modeling of Transistor Stacks, *International Symposium on Low Power Electronics and Design*, pp. 239-244, 1998.

[23] A. Agarwal, K. Roy, "A Single-Vt Low-Leakage Gated-Ground Cache for Deep Submicron," *IEEE Journal of Solid-State Circuits*, Vol. 38, No. 2, pp. 319-328, February 2003.

[24] B. Calhoun, F. Honore, A. Chandrakasan, "Design Methodology for Fine-Grained Leakage Control in MTCMOS, *International Symposium on Low Power Electronics and Design*, pp. 104-109, 2003.

[25] "Intel Architecture Software Developer's Manual - Volume 1: Basic Architecture," pp. 5-2, 1999.

[26] "Simics User Guide," pp. 210, June 2007.

[27] B. Qi, *Performance Analysis of Location Cache for Low Power Cache Systems*, MS thesis, Dept of Electrcial and Computer Engineering, University of Cincinnati, May, 2007.

[28] B. Calder, D. Grunwald, and J. Emer, "Predictive sequential associative cache," *Proc. of the 2nd IEEE Symposium on High-Performance Computer Architecture (HPCA '96)*, pp. 244–254, 1996.

[29] K. Inoue, T. Ishihara, and K. Murakami, "Way-predicting set-associative cache for high performance and low energy consumption," *Proceedings of International Symposium on Low Power Electronics and Design*, pp. 273–275, 1999.

[30] T. N. Vijaykumar, "Reactive-associative caches," *Proceedings of International Conference on Parallel Architectures and Compiler Techinques (PACT'01)*, pp. 49–61, 2001.

[31] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. *In Proceedings of the 22nd International Symposium on Computer Architecture*, pages 2436, June 1995.

[32] B. Stougie, *Optimization of a Data Race Detector*, MS thesis, Dept of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, Oct., 2003.

[33] M.-L. Li, R. Sasanka, S. V. Adve, Y.-K. Chen, and E. Debes. The ALPBench Benchmark Suite for Complex Multimedia Applications. In *IEEE Int. Symp. on Workload Characterization*, 2005.

[34] D-H. Lee, D-K. Kwak, K-S. Min, "Comparative Study on SRAMs for Suppressing Both Oxide-Tunneling Leakage and Subthreshold Leakage in Sub-70-nm Leakage Dominant VLSIs," *20th Internation Conference on VLSI Design*, 2007.

[35] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn, "Using OS Observations to Improve Performance in Multicore Systems," *Micro IEEE*, Vol. 28, No. 3, pp. 54-66, May-Jun. 2008.