# Enhancing L2 Organization for CMPs with a Center Cell

Chun Liu Anand Sivasubramaniam Mahmut Kandemir Mary Jane Irwin
Dept. of Computer Science and Eng.,
The Pennsylvania State University,
University Park, PA 16802.
{chliu,anand,kandemir,mji}@cse.psu.edu

## Abstract

*Chip multiprocessors (CMPs) are becoming a popular way of exploiting ever-increasing number of on-chip transistors. At the same time, the location of data on the chip can play a critical role in the performance of these CMPs because of the growing on-chip storage capacities and the relative cost of wire delays. It is important to locate the data at the right place at the right time in the on-chip cache hierarchy. This paper presents a novel L2 cache organization for CMPs with these goals in mind.*

*We first study the data sharing characteristics of a wide spectrum of multi-threaded applications and show that, while there are a considerable number of L2 accesses to shared data, the volume of this data is relatively low. Consequently, it is important to keep this shared data fairly close to all processor cores for both performance and power reasons. Motivated by this observation, we propose a small Center Cell cache residing in the middle of the processor cores which provides fast access to its contents. We demonstrate that this cache organization can considerably lower the number of block migrations between the L2 portions that are closer to each core, thus providing better performance and power.*

## 1 Introduction

With deeper levels of integration, the relatively high cost of going to off-chip memory is forcing chip designers to provision large on-chip cache structures. For instance, the IBM Power5 [9] has 1.9 MB L2 cache, and Intel's Itanium2 [18] is projected to have 24MB of on-chip L3 cache. With such large caches, the uniformity of cache access latency, which computer architects have traditionally assumed, breaks down. The signals have to traverse longer wires [17], making it more expensive to access some parts of the cache compared to the others. Exposing this non-uniformity of cache accesses to the micro-architecture is becoming increasingly important, so that one can spatially place the data items into these structures based on application access patterns. This has led to recent investigations into optimizing data placement for these large Non Uniform Cache Architectures (NUCA) [19].

At the same time, Chip-Multiprocessors (CMPs) are becoming a popular and cost-effective technique for exploiting the growing transistors on a single die. We already have several dual core commercial offerings in the market from Intel, AMD and IBM, and 8 cores from Sun's Niagara project[15]. These multi-core architectures demand even more on-chip cache real estate to sustain their processing needs, and the need for larger on-chip L2/L3 (in this paper, without loss of generality we assume that the L2 cache is the last level of the hierarchy before going off-chip) is expected to become more acute. One cannot escape the non-uniformity in accessing different portions of this space. The non-uniformity is not just in accessing the different portions of the cache from a single core, but also in the access times to a single location from the different cores. This makes the design of the L2 cache (where do we allocate the cache space on chip? how should the cores and cache space be relatively located?), and its usage (where should a data item be placed? how do we move data items across this space spatially and temporally?) extremely critical in CMPs.

The issue of data placement to locate frequently accessed items in the cache cells with lower access times has been explored in [19]. More recently, there have been proposals to exploit NUCA for CMPs. Chishti et al. [8] extend their earlier idea [7] by replicating tags to facilitate look up. However, there needs to be a protocol to maintain coherence between these replicas, and the global wires running these protocols are also going to be long. Beckmann and Wood [5] propose a layout for cache space organization where different portions are closer to different cores, and suggest schemes for migrating the blocks between these cores. While this scheme does not replicate the blocks (and does not require coherence maintenance at the L2 level), the center portion which is expected to eventually contain the shared blocks (across the cores), is equally far from all the cores. As our results will show, shared data accesses constitute a substantial number of L1 misses, suggesting we might want to keep the shared portion equally close to all the cores (rather than equally far from all the cores).

Our approach on the other hand comes from studying the characteristics of twenty two multi-threaded applications (including NAS parallel benchmarks, SpecOMP [1], and commercial workloads such as Apache and SPECJbb [20]). A detailed characterization of the L2 access patterns of these applications reveals that a substantial number of L2 accesses are to shared blocks, making it important to place them equally close to all the cores. Further, while the number of accesses is substantial, the number of blocks themselves that are widely shared is rather low. These results suggest that we can provide a relatively small L2 space (which we refer to as the *Center Cell*) - sized at 64KB - at the center, with the processing cores around this region. The rest of the L2 space, though shared, is partitioned among the cores and is intended to mainly hold privately-accessed data items. We present details on this shared L2 organization for a four-core CMP, together with statistics on the access times to different portions of the L2 from each core. There is a wide design space for exploitation of this organization based on where to place data items, how to search for their presence, how to migrate them based on access patterns (to exploit non-uniformity in access latencies), and what to replace. After pointing out the design choices, this paper then conducts a detailed evaluation of the proposed architecture using the twenty two applications to show the performance and power benefits of our approach.

The rest of this paper is organized as follows. Section 2 studies the applications and characterizes their L2 access behavior. In Section 3, we present details of our Center Cell based shared L2 organization, together with the design choices for exploiting this architecture. An evaluation of this architecture is conducted in Section 4. Section 5 discusses the related work on non-uniform cache architectures. Finally, Section 6 summarizes the contributions of this paper and identifies directions for future work.

| Application | L2 accesses per thousand instructions | Simulated cycles per core with perfect L2 (in millions) |
|---|---|---|
| bt.a | 56.22 | 4,806 |
| cg.a | 127.21 | 2,167 |
| ep.a | 13.30 | 17,300 |
| ft.a | 134.88 | 1,865 |
| is.a | 177.45 | 1,168 |
| lu.a | 54.08 | 4,939 |
| luhp.a | 58.98 | 4,600 |
| mg.a | 89.44 | 2,976 |
| sp.A | 51.19 | 5,060 |
| ua.a | 27.25 | 9,303 |
| 310.wupwise | 8.85 | 20,030 |
| 312.swim | 103.61 | 2,084 |
| 314.mgrid | 64.18 | 3,617 |
| 316.applu | 15.33 | 28,075 |
| 318.galgel | 11.87 | 25,335 |
| 320.equake | 5.41 | 65,537 |
| 324.apsi | 25.10 | 4,672 |
| 328.fma3d | 63.08 | 4,205 |
| 330.art | 4.15 | 6,322 |
| 332.ammp | 3.23 | 34,515 |
| SPECJbb | 45.23 | 5,241 |
| Apache | 41.3 | 1,296 |
| **Average:** | 54.73 | 11,596 |

**Table 1. Our applications and important statistics.**

## 2 Motivation

The first consideration in our search for a suitable L2 organization is finding a layout, based on application characteristics. Specifically, we would like to understand the sharing behavior of multithreaded applications at the L2 level, to figure out how much data is really (and actively) shared.

### 2.1 Workloads

We use a large number of diverse multithreaded benchmarks for this study:

- *Scientific Applications*: We use both NAS Parallel Benchmarks (NPB 3.2) [2] and SpecOMP [10]. The NAS Parallel Benchmarks, derived from computational fluid dynamics (CFD) applications, are designed to evaluate the performance of large parallel computers. The benchmarks include five kernels and three pseudo-applications, and we use the Class A benchmarks in our study. SpecOMP has been designed by SPEC to evaluate shared memory multiprocessor (SMP) system performance for OpenMP, covering eleven applications, and we use the Class M benchmarks for our experiments.

- *Commercial Applications*: We use two server workloads including SPECJbb [20] and Apache. SPECJbb evaluates the performance of server side Java by emulating a three-tier client/server system (with an emphasis on the middle tier). It measures the performance of the CPU, memory system, and the system scalability by exercising the Java Virtual Machine (JVM), Just-In-Time (JIT) compiler and some aspects of the operating system. Apache is a widely used open-source multithreaded HTTP web server. In this work, we use Apache 4.0 for Sparcs and the SURGE web server benchmark [3] as the client to initiate the requests.

All these benchmarks were run on the Simics [16] complete system simulator with the SPARC target. Since our primary interest is in the L2 accesses, we collected the L1 miss traces and used them in all our experiments in the interest of simulation time. We collected the L2 cache access traces for all the benchmarks using a simulated system with 4 cores, each with a private 16KB L1 cache. The L1 caches are kept coherent using the MESI protocol.

For all the NAS Parallel benchmarks and SpecOMP benchmarks, we marked the initialization phase of the benchmarks at the source code level. Trace collection was started 100 million instructions after the end of the initialization phase to warm up the caches. We collected 400 million subsequent L1 misses for our studies, which was roughly 2.2 GB in the compressed form for each application. For the SPECjbb benchmark, since we have no precise control to demarcate the initialization phase, we warmed up the cache for the first 15 billion instructions, which covers the test, initialization, and terminal ramp-up phases. We then collected 400 million L1 misses (during timing measurement) for our experiments. In the case of Apache, we started the trace collection after the SURGE [3] client has successfully retrieved 600 web pages, beyond which point the utilization of our CPUs reached almost 100 percent.

The collected trace for each application contains the initiator of the memory request, the physical and virtual addresses, the time elapsed (in cycles) since the last access from that core, the size of the memory request, whether it is a kernel or user access, instruction or data, read or write, etc. The number of L2 accesses per thousand instructions for each benchmark along with the number of simulated cycles are given in Table 1.

### 2.2 Characterization

We examine the spatial and temporal L2 sharing behavior of these applications using the above traces. In the following results, we assume the L2 cache has infinite capacity since we are interested in the application characteristics.

Figure 1 (a) illustrates the spatial sharing characteristics of these benchmarks. The Y-axis plots the cumulative fraction of L2 blocks, and the points ”s2”, ”s3”, and ”s4” on the X-axis correspond to the fraction of total blocks in L2 that are shared by 2, 3, and 4 cores respectively. Note that since L2 is of infinite capacity, this sharing is across the entire run of the experiment. The point marked as ”private” in Figure 1 (a) corresponds to the fraction of L2 blocks that are ever accessed by only one core.

Although there are some individual differences across these benchmarks, the general trend (the exceptions are 314.mgrid, lu.a, and luhp.a) we observe is that the percentage of blocks that are shared is a much smaller fraction of the blocks that are privately accessed by a core. In fact, on the average, the percentage of blocks ever shared by two or more cores constitute only 36% of the total L2 blocks across these twenty two benchmarks.

While the above observation may suggest that optimizations for sharing are not important, we note that the percentage of accesses to shared blocks paints a completely different picture, as illustrated in Figure 1 (b). In fact, shared accesses dominate the L2 accesses in many of the benchmarks, averaging 78% of the total L2 accesses across our twenty two benchmarks. This makes it essential to optimize the latency for shared data accesses in L2.

It is possible that, despite this magnitude of shared block accesses, the accesses from the different cores are temporally separated such that they could still be treated as virtually private accesses to a core. In Figure 2, we plot the temporal characteristics of accesses to shared blocks. In this graph, we use the notion of a *running distance*. We define the running distance as the number of accesses by a core to a L2 block before it is accessed by another core. For instance, D0 denotes that after a core accesses a block, the next access to the same block is from another core. D1 denotes that there is one more access from the same core, before another core accesses it. Larger the running distance, the higher the temporal locality from the same core, making it appear more private than shared. As can be gleaned from Figure 2, fewer than 15% of the running distances ever cross 4. Most running distances are lower than 3, with single touch accesses (before another core touches the block) constituting a significant fraction (over 30%) in many benchmarks. That is, there is a high degree of active sharing in these benchmarks.

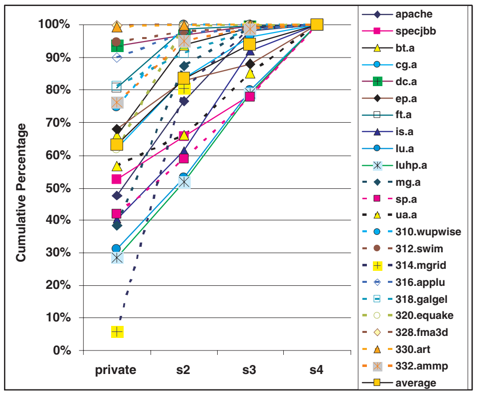
All these characteristics point out that latency of accesses to shared blocks in L2 is very critical. It is not just there are high number of accesses to such data, but the accesses from different cores are temporally interleaved (i.e., the blocks are actively shared). However, the leverage that we may have towards optimizing accesses for such data is the observation that the number of blocks that fall in this category is relatively low. These observations motivate our L2 organization described in the next section.

## 3 Center-Cell based L2 Organization and Data Lookup

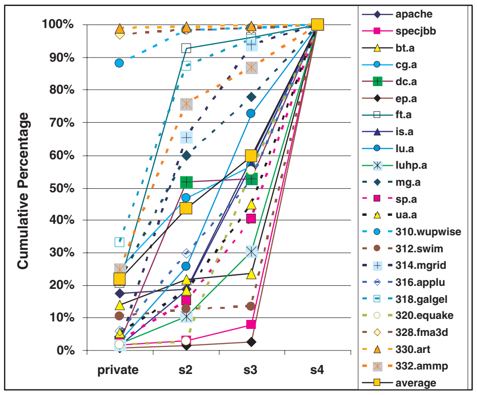
The results from the previous section show that the number of L2 blocks shared among processors is not very high, whereas the number of accesses to these shared blocks is high. Therefore, while the L2 space allocated to shared blocks can be small, its position within L2 is critical[1]. In particular, since some of these blocks are widely shared across processors (see Figure 1), it may not be a good option to place them close to only a single processor or into a location which is far from all processors (as in the case of [5]).

Based on these observations, we propose a new L2 organization depicted in Figure 3. In this organization, the L2

[1]Note that our L2 is shared, and therefore, a shared block can be anywhere in the L2. However, we want to place actively shared blocks into a certain cell (center cell) for improving performance.



(a) Cumulative percentage of L2 blocks



(b) Cumulative percentage of accesses to L2 blocks (private or shared by 2,3,4)

**Figure 1. Percentage of L2 blocks and accesses. "s2", "s3" and "s4" on X-axis indicate that the number of cores sharing/accessing the blocks is 2, 3 and 4 respectively. Each graph is drawn as a cumulative percentage starting from the "Private" case.**

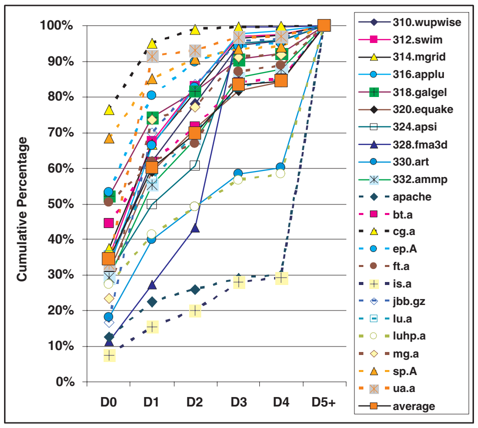


**Figure 2. Running distance distribution of the shared blocks in L2. X axis shows the running distance, and the Y axis shows the cumulative percentage of occurrences of that running distance.**

space is divided into multiple cells of equal size, and one cell (of 64KB), referred to as the *center cell*, is reserved for the shared blocks. The statistics presented in the previous section suggest that this small capacity for the center cell should be sufficient to capture most of the actively shared blocks. Note that this shared cell is placed in the middle to enable fast access from every processor core in the architecture. The remaining cells are distributed in the L2 space as illustrated in Figure 4 from the perspective of a single processor. As far as the access latencies are concerned, the cells form three non-overlapping *rings* around a processor core. The first ring, Ring-0, is the fastest one. It contains 18 cells, with access latencies ranging from 4 cycles to 12 cycles. The second ring, Ring-1, accommodates 25 cells, with a maximum latency of 20 cycles. Finally, the outermost ring, Ring-2, includes the remaining 21 cells, with a maximum access latency of 32 cycles. A cell residing in any of these rings is referred to as the *preferred cell* for that core; i.e., all these 64 cells are the preferred cells for that core, whereas the remaining cells in the cache are termed as the *non-preferred* cells. The goal in this design is to cluster the privately-accessed data of each core into its preferred cells to reduce access latency. Clearly, we want the most frequently used blocks to reside in Ring-0 cells. Note that such a design also enables the use of clustered tag [7], if desired, to obtain a summary of the L2 tags to quickly determine if a cache line exists in the preferred cells or not. Consequently, both access latency and dynamic power consumption can be cut down.

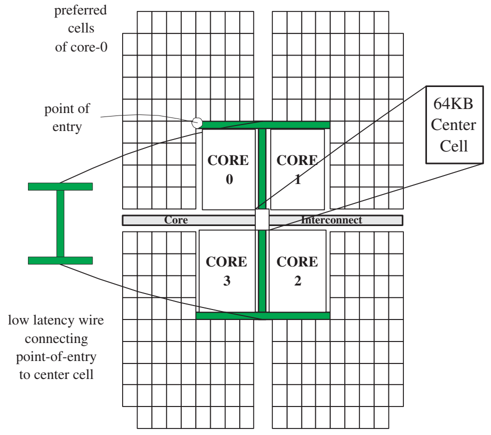

preferred cells of core-0
point of entry
64KB Center Cell
CORE 0
CORE 1
Core
Interconnect
CORE 3
CORE 2
low latency wire connecting point-of-entry to center cell

**Figure 3. Four-core 16MB L2 chip layout.**

An important question at this point is how fast an access to the center cell can be. There are two primary factors that influence the access latency of a cell in this organization: its capacity and its distance from the processor that accesses it. In this case, the capacity does not present a problem since it is really small (64KB). As far as the distance is concerned, we assume the availability of a low latency long wire [4], which is typically used for fast interconnects. We use additional set of low latency wires, shown as the big "I" in Figure 3, to connect the center cell to the point of entry of the preferred cells. The results we obtained from using the Berkeley Predictive Technology Model (BPTM) [6] to calculate the wire delay for 3200um global wire at 25nm technology are shown in Table 2. Our calculations indicate that the access latency for the center cell is 5 cycles at 10 GHZ. Overall, the center-cell based L2 organization shown in Figure 3 provides very fast access to shared blocks (considering

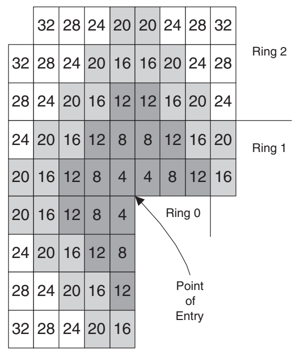


**Figure 4. Layout of the preferred cells and access latencies as seen from a core's perspective.**

the fact that the fastest preferred cell has a latency of 4 cycles as shown in Figure 4). As a result, we do not need to migrate back and forth a shared block frequently between different regions of L2, as in the case of prior work [5].

There are three critical issues that need to be addressed in the context of this center cell based L2 organization: (1) Placement: Where should a new (incoming) L2 block be placed in the L2 space?; (2) Migration/Eviction: When/how should a block be migrated within L2 based on variations in interprocessor data access and sharing patterns? Also, what happens to a block that needs to be replaced (evicted) from a cell?; and (3) Search: How should we search for a block in L2? The rest of this section discusses these issues in detail. Our goal in studying these questions is to understand the ways in which the access non-uniformity in this L2 organization can be exploited.

## 3.1 Placement

A placement scheme decides where to store a new block coming from off-chip memory. While one can place it into any randomly-selected cell in L2, a better option would be something like a *first-touch* policy, where the block is placed into one of the fastest cells (4 cycle latency) in the preferred region of the core that brings it into L2. Since our early evaluation showed that the first touch policy is clearly superior to the random placement policy, we use the former in all our experiments.

## 3.2 Migration and Eviction

In this center cell based L2 architecture, a block can migrate from one cell to another in two ways: *access triggered* and *eviction triggered*. When a block is requested by a processor, we may want to migrate it to a different cell than its current one to better exploit non-uniform access latencies in the future accesses. For example, while the block is in a preferred cell of a core, an access to it by another core moves it to the center cell[2]. It is possible at this point that the center cell is full, and consequently, a block needs to be evicted from it to create space for the new block. Our approach evicts the victim block to one of the lowest latency preferred cells (i.e., 4 cycle cells) of the core that used the block in question last time. This is an example of eviction triggered

[2]Another type of access triggered migration is to move a block from a high latency cell to a low latency cell upon an request.

| Material | Technology | W | S | Length | T | H | Dielectric | Delay |
|---|---|---|---|---|---|---|---|---|
| Cu | 25nm | 0.35um | 0.35um | 3200um | 1.2um | 0.15um | 2.0 | 0.06ns (1 cycle) |

**Table 2. Important parameters for a 3200um low-latency wire at 25nm.**

migration. In our architecture, an eviction triggered migration can cause subsequent evictions. For example, a block evicted from the center cell into a preferred cell of 4 cycles can cause another block from that cell to be evicted to a cell of 8 cycles, and this in turn, can cause yet another block from the 8 cycle cell to be evicted to a cell of 12 cycles, and so on. Figure 5 illustrates this cascade evictions. Note that, while, as explained above, these evictions can be triggered by a block evicted from the center cell, they can also occur when a new block coming to L2 displaces a block that resides in the 4 cycle cell (i.e., as a result of placement), or when a block coming from a high latency cell displaces a block from a low latency cell.

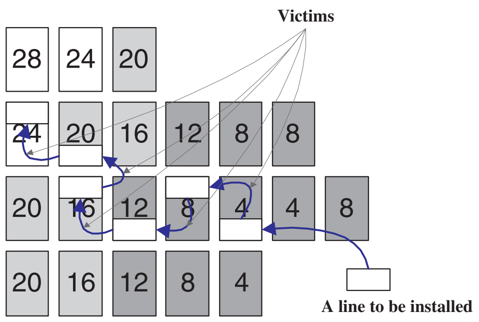


**Figure 5. Cascade evictions of L2 blocks.**

## 3.3 Search

In our L2 organization, data lookup can be performed in different ways, depending on how tags and data are organized, each exhibiting a different trade-off between latency and power consumption. In this work, we study two different schemes which we call *CT (Clustered Tag)* and *DT (Distributed Tag)*. Each scheme has its own tag/data organization. In the *DT* scheme, tag and data are stored close to each other in the same cell. While it is possible to lookup the requested data by probing the entire cache, this would normally be very costly. A better option would be to first search the data in nearby cells (i.e., the ones with low access latency) and then expand the search to other cells upon failure. This should work fine in principle due to locality of data references. Based on this observation, in the *DT* scheme, the requesting core probes its own preferred cells (as will be discussed shortly in detail) and the center cell first. In case of failure, the request is forwarded, by the center cell, to the preferred cells of the other cores. The reason that we employ the center cell for forwarding the request is to avoid a potential race condition. It could so happen that two cores start requesting the same block which sits in a third core's preferred cell. In such a case, if the center cell does not keep track of all the outstanding requests to other preferred cells, the first request will bring the block to the center cell, while the second request will miss in the third core's preferred cell and will subsequently issue a request to the off-chip memory. To prevent this from happening, the center cell in our implementation maintains an *outstanding request queue* (only for the requests directed to other preferred cells) so that the second request in question could be suppressed if it hits in the outstanding request queue of the center cell. In the rest of this paper, we refer to this search strategy as *CC.DT*, which means center cell based organization which uses distributed tags.

The behavior of this search scheme can be affected by tuning the order in which different cells are looked up. Specifically, we implemented three different variants of *CC.DT*:

- Aggressive Search will start probing the non-preferred cells immediately if the requested block is missing in its Ring-0 (and in the center cell). This illustrated in Figure 6(a).
- Moderate Search will start probing the non-preferred cells only when the requested block is missing in its Ring-1. This is shown in Figure 6(b).
- Conservative Search, shown in Figure 6(c), will start probing the non-preferred cells only when the requested block is missing in its Ring-3.

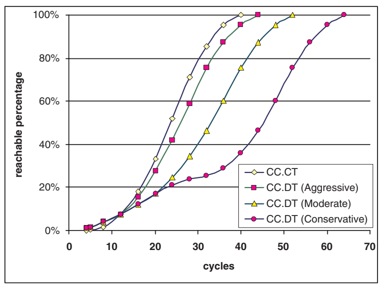
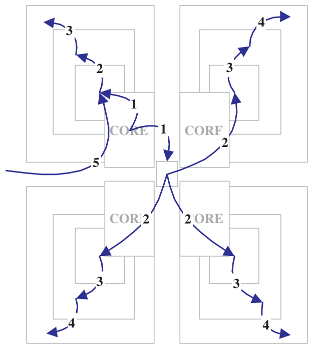


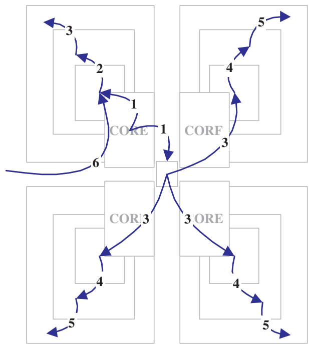
**Figure 7. Fraction of L2 cache that can be accessed under a given number of cycles.**

There is a tradeoff between performance (lookup delay) and power consumption among these three probing (search) schemes. The aggressive search scheme is clearly preferred from the performance angle. On the other hand, one can also expect its power consumption to be very high. The conservative search scheme represents the other extreme where latency can be very high (as accesses to preferred cells and non-preferred cells are serialized). However, its power consumption will be low in cases the requested block is caught in Ring-1 of the requester. The moderate search strikes a balance between these two extremes.

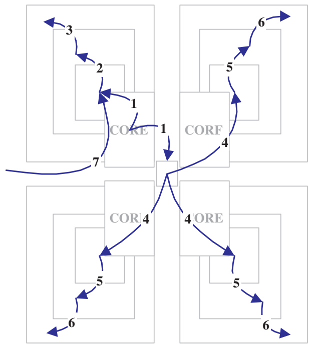
In the *Clustered Tag* scheme (denoted as *CC.CT* in our discussion), tags are clustered together and stored in places close to the core, similar to [7]. When a request comes to a preferred/non-preferred region, we first search the tags to determine the data cell in which the requested block resides, which can be accomplished in 4 cycles, based on our layout. Once this cell is located, we directly access it for retrieving the block. In our implementation of *CC.CT*, however, we still use the concurrent tag-data search for the center cell.



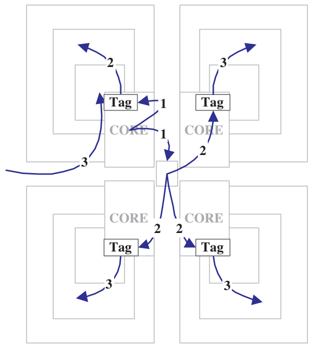
(a) *CC.DT* with Aggressive search
1. Probe both Ring-0 and center cell
2. If miss in both, probe preferred Ring-1
and all three non-preferred Ring-0
3. If still missing, probe preferred Ring-2
and all three non-preferred Ring-1
4. Probe all three non-preferred Ring-2 if needed
5. Bring data in through memory interface



(b) *CC.DT* with Moderate search
1. Probe both Ring-0 and center cell
2. If miss in both, probe preferred Ring-1
3. If still missing, probe preferred Ring-2
and all three non-preferred Ring-0
4. Probe all three non-preferred Ring-1 if needed
5. Probe all three non-preferred Ring-2 if needed
6. Bring data in through memory interface



(c) *CC.CT* with Conservative search
1. Probe both Ring-0 and center cell
2. If miss in both, probe preferred Ring-1
3. If still missing, probe preferred Ring-2
4. Probe all three non-preferred Ring-0
5. Probe all three non-preferred Ring-1
6. Probe all three non-preferred Ring-2
7. Bring data in through memory interface



(d) *CC.CT*
1. Probe the tags of preferred cells and center cell
2. If miss, probe non-preferred tags
3. If miss in non-preferred cells, bring data through
memory interface (note: access could happen in any
of the rings, thus the corresponding latency value varies)

**Figure 6. The search (probing) strategies employed by *CC.CT* and by different variants of *CC.DT***

This is because (1) this cell is small and (2) its latency affects the performance of all four cores. Consequently, in *CC.CT*, when a core misses in L1, it first probes the center cell and the tags of its preferred cells. If the request hits in the center cell, only the center cell will be accessed. If the tags of the preferred cells report a hit, the core needs to send its request only to the cell that has the block. This approach, depicted in Figure 6(d), tends to reduce unnecessary search in other cells and can be beneficial from the dynamic energy viewpoint.

Figure 7 shows the reachable cache percentage (i.e., the fraction of L2 cache that can be accessed) with a given number of cycles. As we can see from the figure, among the *CC.DT* schemes, more aggressive search options can cover larger portion of the cache under the same latency value, compared to the less aggressive search options. For example, given 30 cycles, the aggressive, moderate, and conservative search schemes can cover 60%, 40% and 25% of L2 cache, respectively. On the other hand, *CC.CT* performs better than all the *CC.DT* variants beyond 12 cycles. But, up to 12 cycles, *CC.CT* covers much less cache capacity than all the *CC.DT* schemes due to its serial tag-data access.

| Parameter | Value |
|---|---|
| Instruction Set | Sparc V9 |
| CPU Cores | 4 In-Order Single Issue cores |
| Core Speed | 10GHZ |
| L1 I & D Caches | 16KB, 4-way, 3 cycles |
| Center Cell | 64KB, 4-way, 3 cycles<br>4 outstanding requests |
| L2 Cell | 64KB, 4-way, 3 cycles |
| Total L2 capacity | 16MB |
| L1,L2 cache block size | 64 Bytes |
| Memory latency | 200 cycles |

**Table 3. Simulation parameters.**

## 4 Results

### 4.1 Experimental Parameters

The simulated system has 16MB+64KB shared L2 cache with each core having 4MB *preferred* cells, and the *center cell* is 64KB. As discussed before, the access latency to the center cell from any of the cores is 5 cycles. The access times to the preferred cells have already been explained (see Figure 4). The rest of the simulation parameters are given in Table 3.

The schemes that we are experimentally comparing are summarized in Table 4. We have already described our

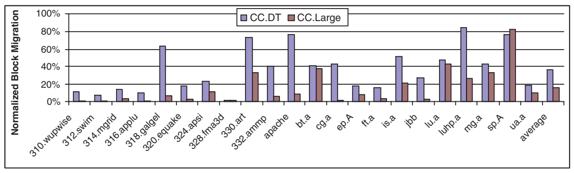


**Figure 8. Normalized cache block migration with *CC.DT* and *CC.Large* with respect to *M.DT*.**

*CC.CT* and *CC.DT* schemes where we use a 64KB center cell, with the difference being whether the tags are distributed or clustered. In the *M.CT* architecture, we do not have a center cell, i.e., L2 has only preferred regions; each is 4MB with their access latencies given in Figure 4. The migration scheme automatically migrates the block to the core accessing the block upon each request. Consequently, under *M.CT*, a shared block can keep shuttling between the preferred regions of the different cores. While the search strategy is to serially probe tag and data in the case of *M.CT*, the search is done in parallel for *M.DT*. The replacement mechanism for *M.CT/M.DT* works the same way as *CC.CT/CC.DT*, except that there is no center cell to be considered. Note that, like *CC.DT*, *M.DT* can also have different search options such as Aggressive, Moderate, and Conservative.

Finally, in order to demonstrate that it is important for the center cell to be equally close to all the cores, rather than equally distant from them (as in [5]), we also introduce one more scheme called *CC.Large* where the center cell is much larger (4MB versus our default 64KB) with a longer access latency (15 cycles from each core). In this scheme, the preferred L2 regions have 3 MB for each core, with their maximum latency now reduced from 32 to 24 cycles.

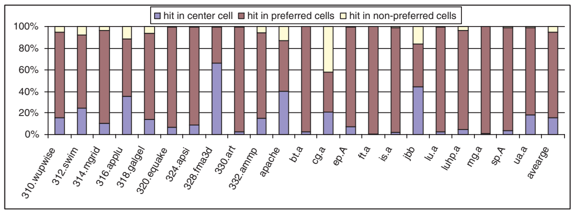
## 4.2 Reduction of Cache Block Migration

Our first set of results is intended to show that our approach can considerably reduce the number of block migrations between the different parts of L2, since that can impact both performance and power. Figure 8 shows the number of migrations of *CC.DT* normalized with respect to *M.DT*. As can be seen, by adding a center cell, we are able to drastically cut down the shuttling of blocks between the different L2 regions. When a block in *CC.DT* is migrated to the center cell, it remains there even when another core accesses it, while in *M.DT*, the blocks would frequently move from one preferred region to another. On the average, the center cell saves about 63% of the block migrations. Note that we achieve these savings with just a 64K center cell, and we do not need a large center cell (as in *CC.Large*) to obtain such drastic reductions (as is also illustrated in the same figure).

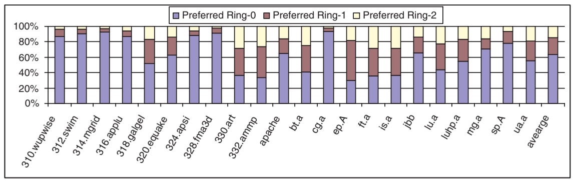
## 4.3 Profile of L2 Hits

Having shown that the center cell substantially reduces the number of block migrations within L2, we now illustrate how effective it can be toward satisfying the requests from each core by examining the behavior of CC more closely (note that the hit behavior of *CC.DT* and *CC.CT* are the same). Figure 9 (a) shows the breakdown of L2 hits in terms of (i) hits in the preferred cells of the requesting cores, (ii) hits in the center cell, and (iii) hits in the non-preferred cells. We can observe the effectiveness of our scheme from this bar-chart by noting that the number of hits in the non-preferred cells is fairly low (less than 6% on average). The center cell is able to satisfy a considerable number of requests in many of the applications. Specifically six of the applications - 312.swim, 316.applu, 328.fma3d, apache, jbb and cg.a - have 20% or more hits in the center cell. Note that these were also the applications which showed a low running distance in Section 2. The average hit ratio in the center cell is around 15.6% across all applications.

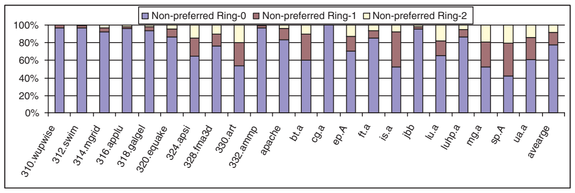
We wish to point out that in addition to the hit ratio (in the center cell), the positional information of the hits in the preferred and non-preferred regions are equally important since the access latencies are quite dependent on the location of the cell that satisfies the requests. Figures 9 (b) and (c) give the breakdown of the hits in the preferred and non-preferred L2 cache regions, respectively, in terms of which ring satisfies a request. From the figure for the preferred regions, we see that we are exploiting the NUCA property by meeting more of the requests from Ring-0 which has lower access latency than the other two rings. The figure for the non-preferred accesses gives us additional interesting insights. We note that the hits, here again, are coming from the lower number rings (particularly Ring-0 which satisfies around 78% of the non-preferred hits on the average). This is an indication that the requests from different cores are more temporally close to each other (i.e., more active sharing), which is again another motivation for the architecture proposed in this paper.



(a) Breakdown of L2 hits.



(b) Breakdown of L2 hits in preferred regions.



(c) Breakdown of L2 hits in non-preferred regions.

**Figure 9. L2 hit distribution.**

## 4.4 L2 Hit Latency

Having shown the profile of where the data is located on an L2 hit, we now show the actual L2 access latency upon a hit in Figure 10. This experiment accounts for the latency of accesses to the different parts of L2 as explained earlier. In nineteen out of the twenty two applications, *CC.DT* (with

|  | M.CT | CC.CT | M.DT | CC.DT | CC.Large |
|---|---|---|---|---|---|
| Preferred size / core | 4MB | 4MB | 4MB | 4MB | 3MB |
| Latencies for preferred cells | tag: 4 cycles + 4, 8, 12, 16, 20, 24, 28, 32 | tag: 4 cycles + 4, 8, 12, 16, 20, 24, 28, 32 | 4, 8, 12, 16, 20, 24, 28, 32 | 4, 8, 12, 16, 20, 24, 28, 32 | 4, 8, 12, 16, 20, 24 |
| Center cell size | None | 64KB | None | 64KB | 4MB |
| Center cell latency | None | 5 cycles | None | 5 cycles | 15 cycles |
| Minimum access latency | 8 cycles | 8 cycles | 4 cycles | 4 cycles | 4 cycles |
| Maximum access latency | 40 cycles | 41 cycles | 64 cycles | 65 cycles | 49 cycles |

**Table 4. Schemes we compare in our experiments. Table shows the sizes of the center cell and preferred L2 regions of each core. It also shows the latency to these L2 portions from each core.**

the *aggressive* search option) has the lowest hit latency. In general, CC does better than M, because of the presence of the center cell which is equally accessible by all the cores. In the M schemes, the high sharing behavior is causing the blocks to migrate frequently between the different regions, as was observed earlier, leading to a higher access latency overall (i.e., a core may frequently need to go to a non-preferred region to get its blocks). On the other hand, in CC, a core can locate the data in the center cell rather than having to go to a non-preferred region each time a core misses in its preferred region.

CC is also a better option than *CC.Large* since (i) in the case of a hit in the center cell, the latter incurs a much higher access latency than the former (15 cycles versus 5 cycles) because of its much larger size, (ii) as noted earlier in our characterization studies, we only need a small center cell to hold much of the shared working set, making the hit rates in the center cell of CC and *CC.Large* not very different, and (iii) in *CC.Large* we are cutting down the size of the preferred regions in the expectation of increasing the hit rates of the center cell.

Between the *CC.DT* (which uses aggressive searches) and *CC.CT* search techniques, the latter always incurs an additional tag look up latency (4 cycles) before the data is accessed, while the former does both in parallel for a given L2 cell. Since our statistics provided earlier show that most of the accesses hit in Ring-0, the tag accesses in *CC.CT* get in the critical path making it slower. The exceptions are cg.a, and to a lesser extent, SPECJbb. As observed earlier in Figure 9 (a), cg.a has the most non-preferred hits. In *CC.CT*, having all the tags clustered together helps us detect preferred region misses faster, to initiate the search in the non-preferred regions earlier (where it is found in Ring-0). Overall, the average L2 hit latencies for *M.CT*, *M.DT*, *CC.CT* and *CC.DT* under the aggressive search option are 15.3, 12.9, 14.2 and 11.2 cycles, respectively. That is *CC.DT* achieves 9% saving over *M.DT* on average.

Having shown the aggressive search results for *CC.DT*, we now examine the L2 hit latencies for the *moderate* and *conservative* search schemes, and we present the results in Figure 11. The results for *CC.DT* under the aggressive search option are reproduced here for the ease of comparison. As expected, the aggressive scheme does better since the search ends quicker for the hits in the non-preferred regions. This benefit would be more significant for applications which have higher hits in non-preferred regions as is the case of cg.a and SPECJbb. On the average, the results with the different variants of *CC.DT* are within 8% of each other, and thus one may simply want to employ the moderate or conservative search schemes in the interest of lowering interconnect traffic and reducing power.

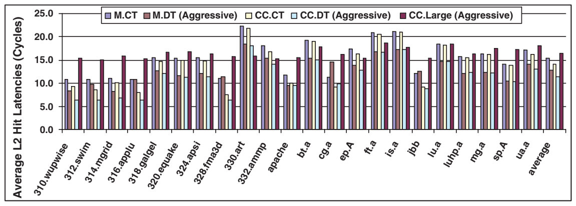


**Figure 10. Average L2 hit latencies with the different schemes.**

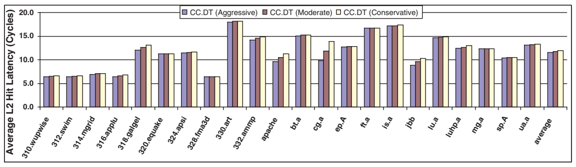
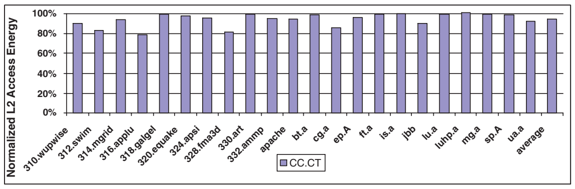


**Figure 11. Average L2 hit latencies for *CC.DT* under the different search options.**

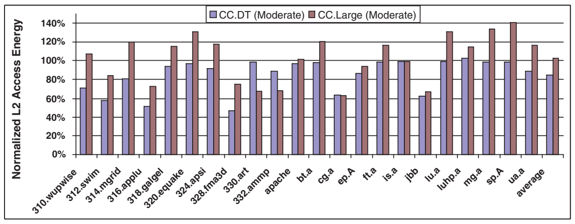
## 4.5 Average Access Energy

Figure 12 shows the average energy consumption number per L2 access. To calculate the data and tag access power numbers, we used CACTI [22]. This is shown for our proposed CC schemes (*CC.CT*, and *CC.DT* with the moderate search option), and compared with the corresponding schemes without a center cell (*M.CT* and *M.DT* with moderate), and the scheme which uses a large center cell (*CC.Large* with moderate). We present the results in two graphs for ease of comparison. In Figure 12 (a), we compare *M.CT* and *CC.CT*, and in (b), we compare *M.DT*, *CC.DT*, and *CC.Large*.

Similar to the latency results, we find that the large center cell (*CC.Large*) does poorly in terms of the energy consumption as well. As noted earlier, shared data access constitute a substantial portion of the L2 accesses, and by making the center cell large, we are incurring much more energy per access. This is another reason in favor of opting for a much smaller center cell as in our proposal. When we look at Figure 12 (a), we see that *CC.CT* saves 6% energy on average than *M.CT*. Similarly, from Figure 12 (b), one can observe that *CC.DT* saves 15% energy than *M.DT*. The results show that having a small center cell in the organization is important from power perspective as well.



(a) *CC.CT* normalized with respect to *M.CT*.



(b) *CC.DT* and *CC.Large* normalized with respect to *M.DT*.

**Figure 12. Normalized average energy consumption per L2 access.**

### 4.6 Execution time

We finally present the overall execution cycles for the different schemes across our benchmarks in Figure 13. Since our point here is to illustrate the benefit of our center cell architecture, we present the reduction in execution time for the different schemes with respect to that for the *M.CT* scheme (which frequently shuttles the shared blocks among the preferred regions of each core).

As we can see from these results, in most of the applications the *CC.DT* scheme does the best, providing 2.1% execution time improvement over *M.CT* on the average. In some applications, such as ft.a, it provides as much as 6.3% performance improvement. In the few cases where *CC.DT* does not fare as well, as in cg.a and SPECJbb where the hits in the non-preferred regions are high (see Figure 9), we note that *CC.CT* does the best. As noted earlier, in addition to the non-preferred region hits, *CC.CT* benefits from the hits in the lower latency center cell as well compared to *M.CT*. Finally, the choice of a small center cell is again reiterated by the poor performance of *CC.Large*, as in Apache, where the hits in the higher latency center cell lead to worse execution time.

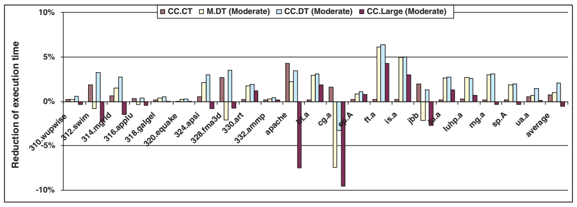


**Figure 13. Reduction in execution time with respect to *M.CT*.**

## 5 Related Work

In this section, we discuss the prior work on L2 cache organizations with non-uniform access latency. [4] proposed to use a thicker metal for mitigating wire delays, which is a growing problem in circuit design. Kim et al. [13] proposed a non-uniform cache architecture. Their proposal is based on the observation that growing wire delays will force significant changes in the physical layout of large caches. They also evaluated two techniques, called the aggressive broadcast and the energy-friendly multicast search, for locating a block in their architecture. Chishti et al. [7] proposed NuRapid, a non-uniform latency cache organization, that decouples tag placement from data placement. It clusters tags and puts them into places close to the core, and this helps improve both performance and power consumption. Our work is different from these studies since we target a CMP, whereas [13] and [7] focus on single processor based architectures.

Several recent papers proposed schemes that extend the non-uniform latency cache idea to CMP environments. For example, Chishti et al. [8] extended NuRAPID [7] to the multi-core processor paradigm. Their approach is based on replication to keep the frequently accessed shared blocks close to processor cores that use them. Zhang and Asanovic [23] discussed an approach called the victim replication. This approach keeps copies of local primary cache victims within the local L2 cache slices. While both these schemes ([8] and [23]) are effective in reducing access latency to shared data, they require coherence maintenance. Specifically, replicating data within L2 requires a complex coherency control protocol which itself can be costly from both performance and power perspectives. In comparison, the approach proposed in this paper does not replicate data blocks; instead, we reduce latency and power consumption in accessing shared blocks by placing them into a small center cell, which is very close to all the processors. Therefore, from an implementation viewpoint, our approach is simpler.

Beckmann and Wood [5] proposed a block-migration scheme to extend the DNUCA for the CMPs, but their protocol is complex and the presented results rely on an oracle-based search, whose practical implementation is not elaborated in the paper. One of the problems associated with this approach is that it puts the shared data into a place within L2, which is equally far from the processors that share it. Each access to such a block tries to bring the block closer to the processor that requests it. However, as a result, heavily shared blocks tend to cluster in the middle regions which are far from the processors. This in turn increases the number of migrations and causes extra latency and power consumption. In fact, one can view this scheme as a variant of ours with a very large center cell *CC.Large*. In contrast, we keep the center cell very small and close to all the processors, and this cuts the number of block migrations significantly, as demonstrated by our experimental results. The reduction in block migrations also helps reduce power consumption.

Huh et al. [11] studied how to partition the NUCA L2 to reduce the interconnect traffic, thus improve the performance and power consumption. Similar to [5], their approach keeps the shared data far from the requesting cores, if both cores are on the opposite side of the die. Iyer [12], Kim et al. [14] and Suh et al. [21] studied the sharing fairness of L2 cache for CMPs to prevent one thread from polluting the cache so that the overall throughput could be improved.

## 6 Concluding Remarks

As chip multiprocessors (CMPs) grow in popularity, it becomes very critical to effectively manage the large on-chip cache space due to wire delays. Making sure that the right data is at the right place at the right time can ensure timely access to the data so that the processing cores are not unduly delayed.

The two main themes of prior work in exploiting non-uniformity of L2 cache latencies for CMPs have primarily tried to use replication (tag and/or data) [8, 23], and explicit migration/staging of the data based on access patterns [5]. With replication, we need to explicitly maintain consistency and, in our work, we have avoided the additional complexity of dealing with replicas. Our proposal is more in line with the latter theme of migrating/moving the data to keep access times low for each core.

This paper has presented the idea of using a small, low-latency center cell based L2 organization that is equally close to all the cores. This cell can hold shared data blocks, which constitute a substantial number of L2 misses though the number of such blocks is itself quite low. Using a large number of diverse multi-threaded benchmarks, we have shown that this architecture is a better alternative, in terms of L2 access latency, access energy, and execution time, than a scheme (without center cell) that simply migrates blocks back and forth between the regions closer to each core. It is also a better alternative having a large center cell (*CC.Large*) and perhaps staging the data between different parts of this large cell (as is done in [5]) since our results show that there is high active sharing (for the few) blocks. Placing such blocks in a small center cell and not migrating them back and forth can give both performance and power savings.

There are several interesting directions for future work related to discretionarily placing data in the center cell, bringing blocks directly to center cell based on prior history, and compiler support for exploiting the center cell based L2 organization. In addition, we are also working on expanding our cache organization to higher number of cores.

## References

[1] V. Aslot, M. J. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady. Specomp: A new benchmark suite for measuring parallel computer performance. In *Proceedings of the International Workshop on OpenMP Applications and Tools*, pages 1–10, 2001.

[2] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarkssummary and preliminary results. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 158–165, 1991.

[3] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 151–160, 1998.

[4] B. M. Beckmann and D. A. Wood. TLC: Transmission Line Caches. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 43, 2003.

[5] B. M. Beckmann and D. A. Wood. Managing wire delay in large chip-multiprocessor caches. In *Proceedings of the 37th annual International Symposium on Microarchitecture*, pages 319–330, 2004.

[6] Y. Cao, T. Sato, D. Sylvester, M. Orshansky, and C. Hu. New paradigm of predictive mosfet and interconnect modeling for early circuit design. In *Proc. of IEEE Custom Integrated Circuit Conference*, pages 201–204, 2000.

[7] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Distance associativity for high-performance energy-efficient non-uniform cache architectures. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 55, 2003.

[8] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Optimizing replication, communication, and capacity allocation in cmps. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA'05)*, 2005.

[9] J. Clabes, J. Friedrich, M. Sweet, J. DiLullo, S. Chu, D. Plass, J. Dawson, P. Muench, L. Powell, M. Floyd, B. Sinharoy, M. Lee, M. Goulet, J. Wagoner, N. Schwartz, S. Runyon, G. Gorman, P. Restle, R. Kalla, J. McGill, and S. Dodson. Design and implementation of the power5&#8482; microprocessor. In *Proceedings of the 41st annual conference on Design automation*, pages 670–672, 2004.

[10] L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, 1998.

[11] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler. A nuca substrate for flexible cmp cache sharing. In *Proceedings of the 19th annual International Conference on Supercomputing*, 2005.

[12] R. Iyer. CQoS: a framework for enabling QoS in shared caches of CMP platforms. In *Proceedings of the 18th annual international conference on Supercomputing*, pages 257–266, 2004.

[13] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 211–222, 2002.

[14] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 111–122, 2004.

[15] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[16] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, , and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, 2002.

[17] D. Matzke. Will physical scalability sabotage performance gains? *Computer*, 30(9):37–39, 1997.

[18] C. McNairy and D. Soltis. Itanium 2 processor microarchitecture. *IEEE Micro*, 23(02):44–55, 2003.

[19] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, N. Ranganathan, D. Burger, S. W. Keckler, R. G. McDonald, and C. R. Moore. Trips: A polymorphous architecture for exploiting ilp, tlp, and dlp. *ACM Trans. Archit. Code Optim.*, 1(1):62–93, 2004.

[20] Specjbb2000 java business benchmark. standard performance evaluation corporation (spec), fairfax, va, 1998. available at http://www.spec.org/osg/jbb2000/.

[21] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *J. Supercomput.*, 28(1):7–26, 2004.

[22] S. Wilton and N. Jouppi. Cacti: An enhanced cache access and cycle time model, 1996.

[23] M. Zhang and K. Asanovic. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *32nd International Symposium on Computer Architecture (ISCA'05)*, 2005.