Organizing the Last Line of Defense before Hitting the Memory Wall for CMPs

Chun Liu Anand Sivasubramaniam Mahmut Kandemir

Dept. of Computer Science and Eng.,
The Pennsylvania State University,
University Park, PA 16802.
{chliu,anand,kandemir}@cse.psu.edu

Abstract

The last line of defense in the cache hierarchy before going to off-chip memory is very critical in chip multiprocessors (CMPs) from both the performance and power perspectives. This paper investigates different organizations for this last line of defense (assumed to be L2 in this paper) towards reducing off-chip memory accesses. We evaluate the trade-offs between private L2 and address-interleaved shared L2 designs, noting their individual benefits and drawbacks. The possible imbalance between the L2 demands across the CPUs favors a shared L2 organization, while the interference between these demands can favor a private L2 organization. We propose a new architecture, called **Shared Processor-Based Split L2***, that captures the benefits of these two organizations, while avoiding many of their drawbacks. Using several applications from the SPEC OMP suite and a commercial benchmark, Specjbb, on a complete system simulator, we demonstrate the benefits of this shared processor-based L2 organization. Our results show as much as 42.50% improvement in IPC over the private organization (with 11.52% on the average), and as much as 42.22% improvement over the shared interleaved organization (with 9.76% on the average).*

1. Introduction

The ability to pack billions of transistors on-chip has opened the doors to an important trend in building high-performance computer systems. Rather than throwing all these resources into a single processing core and making this core very complex to design and verify, chip-multiprocessors (CMPs) consisting of several simpler processor cores can offer a more cost-effective and simpler way of exploiting these higher levels of integration. CMPs also offer a higher granularity (thread/process level) at which parallelism in programs can be exploited by compiler/runtime support, rather than leaving it to the hardware to extract the parallelism at the instruction level on a single (larger) multiple-issue core. All these compelling reasons motivate the trends toward CMP architectures, and there is clear evidence of this trend in the several commercial offerings and research projects addressing CMP designs [16, 10, 13, 14, 15, 3].

The advantage of moving to multiple cores within one die reduces off-chip communication costs (both time and power) between the processor cores that are incurred in traditional SMPs (symmetric multiprocessors). At the same time, this tighter integration exerts an even higher pressure on off-chip accesses to the memory system — there are several cores that need to access the memory system, and they may have to possibly contend for the same buses/pins to get there. With inter-processor communication/sharing costs going down in CMPs (compared to SMPs), the latencies and contention for off-chip accesses to the memory system become even more significant.

As always, caches are an elegant solution to this problem and one needs to use larger and smarter caches to control off-chip costs. A considerable space and power budget in CMPs is consequently devoted to the cache hierarchy. Closest to the processing datapaths are the L1 caches that serve the most frequent case of the requests, and where access times are extremely critical. Rather than making L1 caches very large, or providing numerous ports for concurrent access by all cores, it is more important to not increase their access times by doing so. Consequently, L1 caches are kept private to a datapath, and may not be as large. We use the term *private* to imply that a datapath does not have to go across a shared interconnect to get to its assigned unit. Further, data in one of these units can get replicated in unit(s) assigned to others as will be detailed later. Note that L1 caches assigned to each datapath to service its requests, that also service/snoop coherence traffic coming from an interconnect (without loss of generality, we will use a bus as the shared interconnect), are still classified as private in our terminology.

We can keep adding further levels of caches to exploit program locality, until we get to the "last line of defense" before we have to go off-chip. At this level, it is more important to reduce the probability of going off-chip than optimizing its access times. Consequently, we have to provide large capacities for this level of the hierarchy, which raises the following question — *How should we organize this level of the cache hierarchy to provide good performance in a fairly power-efficient manner?* — that this paper sets out to explore. Note that this last line of defense has not only got to reduce the off-chip accesses to memory, but also serves a vital role in facilitating sharing and inter-processor communication.

Without loss of generality, in this paper, we use L2 as the last line of defense, i.e., the last level of the cache hierarchy, before a request needs to go off-chip. We assume that any additional levels of the cache hierarchy between L1 and this last level to be subsumed by the reference stream coming to L2. L2 is the last line of defense in several CMPs [16, 3].

There are two obvious alternatives for organizing the L2 structure for CMPs:

- The first approach is to use a *Private L2* organization

[8]. Here, the L2 cache space is partitioned between the processor cores equally, and each core can access its L2 space without going across the shared bus. Typically, upon an L2 miss, you also consult other L2 units (they snoop on the bus) to find out if they have data and go off-chip otherwise.

- The second approach, which is also the organization in current chip multiprocessors such as the Piranha [3] and Hydra [16], is to use a *Shared L2,* wherein the L2 cache is placed on the other side (not on the processor side) of the bus, i.e., the shared bus is between L1s and L2, and the coherence actions take place between the L1s. Upon missing in the L1s, L2 is looked up and only upon a miss is an off-chip access needed.

There are pros and cons for each approach. In the private L2 case, the cache units are closer to a processor, not requiring bus accesses in the common case, reducing both access latency and bus contention. The downside is that data blocks can get duplicated across the different L2 units, depending on how prevalent data sharing is. This can lessen the overall effectiveness of how many blocks can be maintained on-chip. The other drawback is that physically partitioning these L2s and pre-assigning them to each core can introduce load balancing problems, i.e., one L2 unit may be over-utilized while another is grossly under-utilized. These drawbacks are not a problem in the case of the shared L2, since the L2 structure can be viewed as one big shared unit, where there is no duplication of blocks, and load balancing can be addressed with possible non-uniform distribution of L2 space. The downside for the shared L2 organization is the possible interference between the CPUs in evicting each other's blocks, together with the higher latency (and contention) in getting to the L2 over the shared interconnect upon an L1 miss.

The other organization issue with shared L2 is that it is not very desirable to maintain this structure as one big monolithic unit (note that private L2 automatically keeps this as several equally-sized chunks). The dynamic power consumption per access for a cache structure [18, 5] is largely dependent on its size amongst other factors, and for reasons of thermal cooling and reliability considerations we want to keep the power consumption low for this on-chip structure. The common solution for handling this is to *split* the L2 cache into multiple banks that are individually addressed so that the dynamic power expended for an access becomes much smaller. The addresses can be interleaved between these banks, causing successive blocks to fall in different banks, and we refer to this structure as *Shared Interleaved (SI) L2.*

To our knowledge, there is no prior in-depth investigation of how to organize this last line of defense from the memory wall for CMPs by evaluating the trade-offs between these different approaches. In addition to evaluating these trade-offs using several applications from the SPEC OMP [2] suite and the Specjbb [23] commercial benchmark, on the Simics [21] full system simulator, this paper makes the following contributions:

- We present a new L2 cache organization called *Shared Processor-Based Split* L2 cache as the last line of defense. This structure is similar to a shared interleaved L2 in that there are multiple (smaller) L2 units on the other (opposite side of the processor cores) side of the bus, but is different in that the units for lookup are selected based on the processor cores (henceforth referred to as CPUs) that issued the requests. This allows better utilization of the L2 space to balance the load between CPUs, while still relatively insulating such usage between the CPUs.

- We detail the architectural issues that are important for implementing this new L2 cache organization. Our solution can be implemented using a simple table structure, can work with existing buses, and can easily integrate with existing cache coherence protocols.

- This new solution provides a flexible way of configuring L2 caches based on application workload characteristics, without extensive hardware support. In particular, L2 demands vary (i) across applications, (ii) temporally within an application, and (iii) spatially across the CPUs running an application. Consequently, we may want to allocate different fractions of the L2 storage to different CPUs at different times. In this study, we use a simple profile-driven approach for such allocations, though our mechanisms are general enough to be exploited by a more viable/better allocation strategy. At the same time, one can always resort to equally splitting up the L2 storage between the cores whenever needed.

We demonstrate the benefits of this shared processor-based L2 organization detailing when our mechanisms give the most savings. Our results show as much as 42.50% improvement in IPC over the private organization (with 11.52% on the average), and as much as 42.22% improvement over the traditional shared organization (with 9.76% on the average).

The rest of this paper is organized as follows. The next section presents the CMP architecture under consideration, the existing L2 cache organizations, and the mechanisms needed to implement our shared processor-based split organization. The details of the workloads, our simulation platform, the methodology for the evaluation, and our experimental results are given in Section 3. A discussion of related work is presented in Section 4. Finally, Section 5 summarizes the contributions of this paper and outlines directions for future work.

2. L2 Organizations and Proposed Architecture

2.1. CMP Hardware and L2 Organizations

The CMP under consideration in this paper is of the shared multiprocessor kind, where a certain number of CPUs (of the order of 4-16) share the memory address space. As mentioned earlier, we assume L2 to be the last line of defense before going off-chip, and consequently each CPU has its own private L1 that it can access without going across a shared interconnect. It is also possible that a data block can get duplicated across these private L1s. Several proposed CMP designs from industry and academia already use such private L1-based configurations. We keep the subsequent discussion simple by using a shared bus as the interconnect (though one could use fancier/higher bandwidth interconnects as well). We also use the MOESI [20] protocol (the choice is orthogonal to the focus of this paper) to keep the caches coherent across the CPUs.

For the L2 organization, as mentioned earlier, there are two possibilities. The first option is to place the L2 units on the CPU side of the interconnect (the L2 is also private) as is shown in Figure 1(a). These private L2 units are of equal size, and can directly serve the misses coming from L1 in case it can be satisfied from its local storage. If not, it needs to arbitrate for the bus and place the request on the bus to get back a reply either from another cache or from off-chip memory. The other L2 caches need to snoop on the bus to check if any transaction pertains to them so that they can

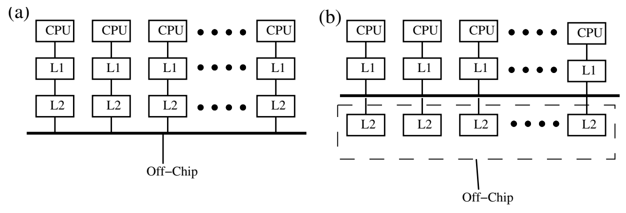


Figure 1. (a) Private L2 Organization. (b) Shared L2 Organization.

take appropriate actions (responses for read/write requests or invalidates for upgrade requests). In this case, the coherence actions take place between the L2 caches, and not all bus transactions may need to permeate to L1 because of filtering effects by L2 (as noted in [20]). The private nature of L2 units can cause a data block to come into multiple L2 units based on the application access/sharing patterns, thereby reducing the aggregate L2 size.

The other option is to move the L2 to the opposite side of the bus, i.e., it becomes a shared unit that is equally accessible across the shared interconnect to all the CPUs with equal cost (see Figure 1(b)). Note that while logically in this case the L2 appears as one big monolithic unit (from the viewpoint of the CPUs), it is typically banked (divided into subunits/splits) to reduce the per-access dynamic power. The data blocks are placed in these banks based on their addresses, e.g., one could either interleave consecutive blocks on successive banks, or one could fill one bank first with consecutive addresses before going to another bank (in our study it does not matter which of these two options is chosen and we simply use the first option in the experimental evaluation). With this shared L2 organization (SI), there is at most one copy of a block within this structure, thereby utilizing the space more effectively (note that banking the L2 into multiple units has the same hit/miss behavior as a single monolithic non-banked structure). The downside of this approach is that upon an L1 miss, the requests need to go on the shared bus to get the data either from another L1, or from L2, or from off-chip memory. In SI, the coherence protocol is performed between the L1 caches.

2.2. Proposed Architecture

As observed, the advantages of private L2 are in reducing on-chip interconnect traffic (latency and contention). The downside is the possible increase in off-chip memory accesses because of reduced aggregate L2 capacity (due to duplication). The other problem may be due to the fact that the load induced on the L2 units may be different across the CPUs (i.e., one may have higher locality while another may exhibit poor locality). Consequently, we expect private L2 to perform better when (i) the sharing is not high across the CPUs (so that fewer duplicates reside across the L2 units) and (ii) the locality behavior is balanced across the CPUs.

The shared L2 organization described above does not have these two drawbacks. Despite the level of sharing across the CPUs, there is at most one copy of a data block that can reside in L2. Further, since all of L2 (and all its banks) are equally usable by any of the CPUs (the banking is done by addresses rather than by the CPUs using them), the imbalance of the locality behavior across the CPUs can be addressed by letting one CPU use up more space compared to another. The downside of the shared L2 organization is that (i) the load on the on-chip interconnect may become higher and (ii) there could be interference between the CPUs when utilizing L2 (one could evict the blocks used by another).

Having looked at the pros and cons of both these approaches, we next present our *Shared Processor-Based Split* L2 cache organization. The main goal here is to try to reduce off-chip accesses since we are at the last line of defense. It may be easier to provision additional hardware for on-chip components, e.g., wider buses, multiple buses, or fancier interconnects, and the more serious issue is to avoid the memory wall, i.e. *avoid going off-chip as much as possible*. This cost is eventually going to be the limiting factor on performance (and perhaps even power) with the ever-growing application datasets, and the widening gap between processor and memory speeds.

Consequently, our solution strategy is to:

- Try to approach the behavior of private L2s when the sharing is not very significant and when the load is more evenly balanced. On the other hand, we would like to approach the behavior of shared L2 for higher sharing behavior and when the load is less balanced, while still insulating one CPU from another.
- Possibly pay a few extra cycles on-chip (perhaps not in the common case) if doing so will reduce the off-chip accesses.

Our shared processor-based split L2 uses the underlying organization of the Shared L2 described above, i.e., the L2 is on the other side of the bus, so that we can accommodate high data sharing whenever needed. However, in order to reduce the interference between the CPUs on this shared structure, our splits are based on the *CPU ids* rather than memory addresses, i.e., each CPUs is assigned one or more units/splits of the L2 structure. A request coming from an L1 miss goes across the shared bus, and checks the split(s) assigned to the corresponding CPU (note that the other L1s may snoop as usual). If the request can be satisfied by the split(s) being accessed — it is to be noted that as in the shared L2 case, there are no duplicates for a block in L2 — then the data is returned and the execution proceeds as usual. However, when the data does not reside in any of these split(s), instead of immediately going off-chip, the other (i.e., the ones that are not assigned to this CPU) splits are consulted to see if the requested block resides on any of those splits. If it does, then the data block is returned to the requesting L1 to continue with the execution. Only when the block is not found in any of the splits is an off-chip memory access needed.

Note that there are three scenarios for an L2 lookup in our proposal:

- *Local Hit:* When the lookup finds the data block in the L2 split(s) assigned to the CPU issuing the request, all the splits assigned to that CPU are accessed in parallel and the performance cost is the cost of looking up any one split (taking say a cycles).
- *Remote Hit:* In this case, the lookup amongst its assigned split(s) fails, but the data block is found in another split that is not assigned to it. Consequently, the cost of servicing the request becomes $2a$ cycles.
- *L2 Miss:* In this case, the data block is not in any of the splits, and requires an off-chip access. The cost in this case will involve the cost of the off-chip access, in addition to the $2a$ cycles.

One may have opted to remove the *Remote Hit* case by looking up all the L2 splits (ignoring the CPU id) in parallel. Though performance efficient, the reason we refrain from doing this is due to the dynamic power issue, i.e., the dynamic power in this case is not going to be any lower than an unbanked monolithic L2 structure, which we discarded in the first place for this reason.

Having looked at the costs of lookup and servicing the requests, we next need to examine how the data blocks get placed in the different splits. Since we perform placement at the time of a miss, we need to consider the following cases:

- *Local Miss* + *Remote Miss*: Since the block is not in L2 (any of its splits), we assume that the CPU issuing the request is the one that will really need it. We randomly pick one of the splits assigned to the requesting CPU, and then place the block within that split based on its address (as is usually the case).
- *Local Miss* + *Remote Hit*: In this case, we assume that the CPU issuing the latest request needs the block more than the CPU currently associated with the split where the block resides (i.e., we expect temporal locality). Consequently, we move the block over from the remote split to one of the splits (chosen randomly) for the requesting processor.

2.3. Hardware Support

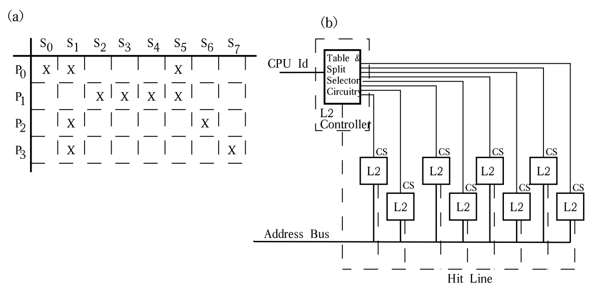


Figure 2. (a) The table structure that shows the processor-L2 split associations. (b) Addressing L2 splits in our proposal.

We present a simple table-based mechanism for implementing the shared processor-based split L2 proposal. We propose to maintain a table in the L2 controller as shown in Figure 2(a) for an architecture with 4 processors and 8 L2 splits. The columns denote the splits of the L2 and the rows denote the CPUs, and an "X" in the table indicates that the corresponding split is assigned to the specified CPU. When an L1 miss is incurred, the bus carries not just the address of the block being referenced, but also the id (which is usually the case, e.g., [22]) of the CPU initiating the request. The L2 controller (see Figure 2(b)) has already been given (downloaded to it in software) this table a priori, and it simply needs to lookup the table based on this CPU id, and can generate the chip selects (CS) for the corresponding L2 splits that need to be looked up. If these lookups fail (which will be sensed by the Hit Line in the figure), then the chip select is sent to the other splits (that were not looked up in the previous try). Only when this lookup fails as well do we need to go off-chip.

We would like to point out that this mechanism can easily fit into many asynchronous bus designs, since the interface to L2 is not very different from the external viewpoint (for the bus or the CPUs). Further, the coherence mechanism that takes place between L1s is not affected with this enhancement (it is not any different from a monolithic or an address-banked L2).

We propose the following simple extensions to the operating system (OS) to work with this hardware: (i) a system call that specifies how the "X"s for the CPUs should be allocated to the application, which can be invoked at any point during its execution, (ii) the operating system updating the table on the L2 controller appropriately during this system call, as well as updating its (shadow) software copy of this table that it maintains for each application, and (iii) the context switch mechanism at each CPU looking up the corresponding row of the OS shadow copy and updating the L2 hardware table accordingly. Note that protection is not really compromised, and if ever the OS wants to disallow a process from being allocated too many splits (or a specific split) the access control can be performed at the time of the system call. In addition, there could be OS issues in figuring out how to partition the L2 space between applications over the course of execution, which is beyond the scope of this paper.

2.4. Exploiting the Proposed Mechanism

There are several benefits that we would like to point out with this implementation:

- There could be more than one "X" in each column of the table, meaning that a split could be shared by different CPUs. Consequently, if we know that some CPUs have a high degree of data sharing, then we could assign them the same splits so that the local hit case is optimized.
- There could be more than one "X" in each row, and we do not necessarily have to assign an equal number of "X"s across the rows. This allows assigning more than one split to a processor, based on its L2 space demand, and also optimizing for a non-uniform (heterogeneous) assignment (i.e., giving different number of splits to different processors) if the L2 locality behavior is different across the behaviors at any one time.
- We can disallow two CPUs interfering with each other, i.e., one evicting the blocks of another, by not giving them an "X" on the same column.
- This table can be *dynamically loaded* to the L2 controller (possibly by memory mapping in some registers) using an operating system call, during the course of execution. This allows the ability to convey application-level information dynamically to the L2 hardware for the best performance-power trade-offs at any instant.

One can always opt for filling in the table to provision an equal allocation of splits to the CPUs — we refer to this as the *Shared Split Uniform (SSU)* case. This is the simplest of the options, where one may not want to analyze high-level workload behavior to further refine the table settings. However, our table-based implementation of the shared processor-based split L2 mechanism allows scope for non-uniform allocation of splits to the CPUs, i.e., different CPUs can get different number of L2 splits, and we refer to this as the *Shared Split Non-Uniform (SSN)* case in this paper. In general, this software controlled table-based mechanism allows us to exploit the following workload characteristics:

- *Intra-Application Heterogeneity:* This is the L2 load imbalance within an application, and can be further classified as:

– *Spatial Heterogeneity:* If the L2 localities of different CPUs are different within a single application, we can opt to allocate non-uniform splits to better meet those needs.

– *Temporal Heterogeneity:* During the course of execution of an application, we can dynamically change the number of splits allocated to a CPU at different points of time, depending on how its dynamic locality behavior changes.

- *Inter-Application Heterogeneity:* Applications can have entirely different L2 behaviors. Whether they are running one after another, or even if they are running at the same time (on different CPUs or when they are time-sliced on the same CPUs), our mechanism provides a way of controlling the L2 split allocation according to their specific demands.

There are several techniques/algorithms that one can employ to fill the table values at possibly every instant of time. Such a detailed evaluation of all possible techniques is beyond the scope of this work. Rather, our objective here is to show the flexibility of our mechanisms, and we illustrate/use a simple *profile-driven approach* to set the table values to benefit from the uniformity/heterogeneity of the L2 locality/sharing behavior.

3. Experiments

3.1. Methodology and Experimental Setup

The design space of L2 configurations for comparison is rather extensive to do full justice in terms of evaluation within this paper. Consequently, in our experiments where we compare the four approaches — Private (P), Shared Interleaved (SI), Shared Split Uniform (SSU) and Shared Split Non-uniform (SSN) — we set some of the parameters as follows. In the Private case, the number of L2 units has to obviously match the number of CPUs, say p. In the experiments for SI, we use p banks as well, with the address based interleaving of blocks. In the SSU case, we use p splits each with the same size as the private case. Note that in the case of SI and SSU, we are not restricted by p, i.e. we could have more (of smaller size) or less (of larger size) banks/splits, while keeping the overall L2 capacity the same as the private case. In our experiments, we simply set the number of splits/banks to p, since it is closest to the private case (in terms of access times). In the SSN case, the splits are themselves all of the same size. The way that we provide different cache capacities for different CPUs is by giving different number of splits to the different CPUs. In order to examine the benefits of such heterogeneous allocations, our experiments use more than p splits. More specifically, our default experiments use a 8 processor configuration with 2 MB overall L2 capacity as in shown in Table 2, and we use 16 splits of 128K each in the SSN case.

There are several techniques/algorithms that one can employ to fill the table values at possibly every instant of time to accommodate the different kinds of application/spatial/temporal heterogeneity explained in the previous section for SSN. Such a detailed evaluation of all possible techniques is beyond the scope of this work. Rather, our objective here is to show the flexibility of our mechanisms, and we use a rather simple scheme. Specifically, we categorize CPUs into three groups based on the load that they impose on L2 — high, medium, and low — and accordingly give them 512K, 256K, and 128K of L2 cache space, respectively. For instance, if a CPU is categorized at some instant as high L2 load, then it would be allocated 512K, i.e., 4 splits of 128K each. The SSN schemes that we consider (for the 2MB L2) are given in Table 1. For example, SSN-152 denotes, that the L2 space is divided into one 512K (i.e., 4 splits of 128K) chunk, five 256K chunks (i.e. each of 2 splits of 128K), and two 128K chunks (i.e., one split each). Note that the total size of the chunks matches the total L2 capacity, which is 2 MB in this case. A CPU, over the course of the execution, can move from one chunk to another. However, we do not vary the number of chunks or the size of the chunks over the course of execution (though our table-based mechanism allows that as well).

SSN Configuration	Allocation Chunks
SSN-152	1*512K, 5*256K, 2*128K
SSN-224	2*512K, 2*256K, 4*128K
SSN-304	3*512K, 4*128K

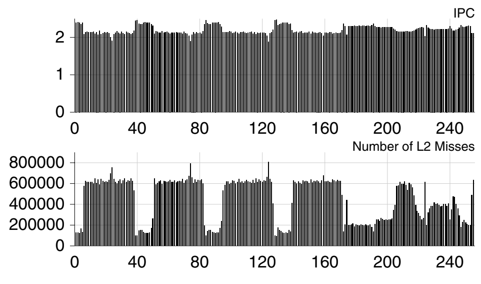
Table 1. SSN configurations studied for the 2MB L2. Note that the splits are themselves of 128K each, an integral number of such splits — called a chunk — are allocated to a CPU.

Parameter	Default Value
Number of Processors	8
L1 Size	8KB
L1 Line Size	32 bytes
L1 Associativity	4-way
L1 Latency	1 cycle
L2 Size (Total Capacity)	2MB
L2 Associativity	4-way
L2 Line Size	64 bytes
L2 Latency	10 cycles
No. of L2 Splits in SI, SSU	8
No. of L2 Splits in SSN	16
Memory Access Latency	120 cycles
Bus Arbitration Delay	5 cycles
Replacement Policy	Strict LRU

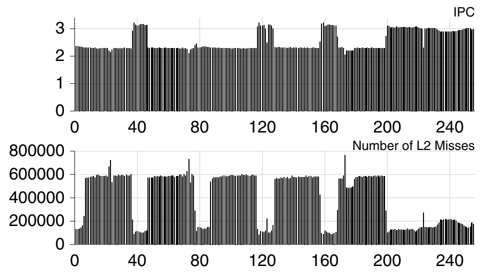
Table 2. Base simulation parameters used in our experiments.

The allocation of chunks to CPUs is performed using a profile-based approach in this study. Specifically, we divide the execution into a certain number of epochs (256 in this case), and for each epoch we sort the CPUs in decreasing order of L2 miss rates, and then allocate the largest chunk to the CPU with the highest miss rate, the second largest chunk to the CPU with the second highest miss rate, and so on. This is a rather simple scheme that can use profile information to make reasonable allocations during the run. It is conceivable that future research can develop much more sophisticated (and dynamic) split allocation algorithms, and our table-based mechanism allows that. To conduct our experiments, we modified Simics [21] (using Solaris 9 as the operating system) and implemented the different L2 organizations. The default configuration parameters are given in Table 2, and these are the values that are used unless explicitly stated/varied in the sensitivity experiments.

In this study, we evaluated nine applications (eight from SPEC OMP [2] and Specjbb [23]). The important characteristics of these applications are given in Table 3. The second column in this table gives the number of L1 misses (averaged over all processors) and the third column the L1 miss rate (averaged over all processors) when the P version is used. The corresponding L2 statistics are given in columns



(a)SpecJBB (the P case)



(b)SpecJBB (the SSU case)

Figure 3. IPC and the number of L2 misses for specjbb. Note that the y-axes are on different scales.

four and five. The last column shows the total number of instructions simulated. All benchmarks have been simulated for 4 billion cycles and use all 8 processors in our base configuration, unless stated otherwise.

While many of the experiments use one application running on Simics where spatial and temporal heterogeneity during its execution may arise, we also conduct experiments with running two applications concurrently to stress application heterogeneity. We use four processors for running each application, i.e., we use space sharing rather than time slicing for multiprogramming.

3.2. Base Results

We first summarize the overall results to illustrate the benefits of shared processor-based split L2, by comparing its IPC results with those for the two other organizations (P and SI) which are the current state-of-the-art (see Table 4). The last two columns in Table 4 give the best IPC value for a given workload across all the L2 configurations considered (and —within the parentheses— the percentage improvement it brings over the private L2 case), as well as the version that gives this best IPC, respectively. These results are given for each application running individually, as well as some multiprogrammed workloads with two applications A and B running concurrently on four different processors each (specified as A+B).

We can make several observations from these results. First, we find that except in two workloads (applu and swim+apsi), the shared L2 organization based on processor-based split (our SSU or SSN proposal) does the best. In workloads such as swim, mgrid and specjbb, with higher L1 miss rates (which in turn exerts a higher pressure on L2), we find dramatic improvements in IPC ranging from 30.9% to 42.5% in some configurations (SSU or SSN). Even if we consider the average across these fourteen workloads, we get an average 11.52% IPC improvement over the private case. Let us examine these results in detail in the following discussion.

We first attempt to show why private L2 does better in the two workloads, and not as well in the rest. There are primarily two factors affecting L2 performance: (i) the degree of sharing which can result in duplication between the private L2 units or cause repeated non-local split references in SSU or SSN, and (ii) the imbalance of the load that is imposed on the L2 units by the different processors. In our experiments, we tracked the number of blocks that were residing in multiple private L2 units at any instant, which is an indication of the level of data sharing. In the case of applu, only around 12% of the blocks are shared at any time and even these are mainly shared between 2 CPUs. Consequently, there is not much duplication for this workload (applu) to affect the private L2 case. In addition, applu does not exhibit much spatial or temporal heterogeneity as will be shown later, making this workload more suitable for private L2. In the case of swim+apsi, there is no sharing across the CPUs running these different applications, and they also have similar L1 miss behavior (shown in Table 3), which is an indication of the load on L2.

On the other hand, in most of the other applications, the data sharing and/or the load imbalance issues make the shared L2 perform better than the private L2 case. For instance, when we consider specjbb, SSU is over 31% better than the private L2, due to both sharing effects (we find around 35% of the blocks are being shared on the average when considering entire execution), as well as the imbalance caused by spatial and temporal heterogeneity (to be shown shortly). While Table 4 summarizes the overall results, we also collected detailed statistics over the course of execution, and present some of them (the IPC and L2 misses) for each epoch in the execution for both the P and SSU cases in Figure 3 for specjbb. One can see that there is a direct correlation between the L2 misses and the corresponding IPC values, and we find that SSU is able to lower the L2 misses, thereby improving the overall IPC. Specifically, while the IPC in the P case never exceeds 2.5, we find IPCs over 3.0 during several epochs in the execution for the SSU case. The less frequent accesses to the off-chip memory with SSU provide this IPC improvement. Although not explicitly quantified in this study, this can also reduce the main memory energy consumption. We will get further into the correlations between application characteristics and L2 behavior later in this section.

Of the two problems — duplication due to sharing and load imbalance across the CPUs — SI mainly addresses the sharing issue, i.e., only one copy of a block resides in all of L2 regardless of how many CPUs access it. While it can accommodate some amount of imbalance/heterogeneity by letting one or more CPUs occupy more L2 space than others, this can also lead to interference between the CPUs (eviction of one by another). We find that SI brings some improvement over the private case in some of the workloads (e.g., art_m, swim, mgrid, swim+mgrid). However, providing a shared L2 alone is not sufficient in many others, or is not necessarily the best option even in these cases. Note that SI represents the current state-of-the-art for the shared L2 organization. On the other hand, when we go to SSU and SSN, in addition to data sharing, we also find the insulation of L2 space of one CPU from another, and the possible spatio-temporal heterogeneity in the workloads can help these schemes over and beyond what SI can provide. In nearly all the cases where SI does better than the private case, SSU or SSN performs even better, and our two schemes do better than the private case even when SI is not a good option.

Benchmark	L1		L2		Number of
	Number of Misses	Miss Rate	Number of Misses	Miss Rate	Instructions (in millions)
ammp	53176353	0.007	2015486	0.062	25,528
art_m	66061168	0.009	25712966	0.507	22,967
galgel	111400050	0.014	10683462	0.127	24,051
swim	261622475	0.111	95881598	0.296	7,761
apsi	378868309	0.117	27170810	0.083	15,713
fma3d	18853410	0.002	6199405	0.239	26,189
mgrid	333243418	0.153	68292105	0.185	10,294
applu	111232574	0.009	26477050	0.168	21,519
specjbb	828522034	0.353	22689664	0.083	9,413

Table 3. The benchmark codes used in this study and their important characteristics for the private L2 case.

Workload	P	SI	SSU	SSN-152	SSN-224	SSN-304	Best IPC	Best Version
ammp	5.967	6.026	6.049	6.020	6.010	6.006	6.049 (1.4%)	SSU
art_m	5.368	5.470	5.529	5.261	5.459	5.555	5.555 (3.5%)	SSN-304
galgel	5.622	5.583	5.618	5.724	5.731	5.727	5.731 (1.9%)	SSN-224
swim	1.821	2.462	2.399	2.488	2.543	2.596	2.596 (42.5%)	SSN-304
apsi	3.687	2.680	3.812	3.703	3.587	3.507	3.812 (3.4%)	SSU
fma3d	6.122	6.131	6.135	6.181	6.171	6.159	6.181 (1.0%)	SSU-152
mgrid	2.406	3.050	3.022	3.086	3.101	3.150	3.150 (30.9%)	SSN-304
applu	5.030	4.894	4.937	4.947	4.949	4.944	4.949 (-1.6%)	P
specjbb	2.200	2.547	2.890	2.606	2.709	2.810	2.890 (31.3%)	SSU
ammp+apsi	4.169	4.104	4.292	5.436	4.803	4.401	5.436 (30.4%)	SSN-152
ammp+fma3d	5.897	5.949	5.977	5.936	5.900	5.895	5.977 (1.4%)	SSN-152
swim+apsi	2.509	1.776	2.018	2.050	2.007	2.006	2.050 (-18.3%)	P
swim+mgrid	2.513	2.851	3.067	2.698	2.470	2.401	3.067 (22.1%)	SSU
Average:							(11.52%)	

Table 4. The IPC results for different workloads and different L2 management strategies.

Our shared processor-based split L2 organizations (both SSU and SSN) are not only able to address the sharing issue (perhaps not as effectively as SI since there may be an additional cost — though not as expensive as going off-chip — to go to a split not allocated to that processor), but are able to reduce the interference between the CPUs, and possibly allocate more/less space to a CPU as needed. This helps reduce L2 misses and the associated off-chip memory access costs. These advantages make these schemes provide much better IPC characteristics compared to either the P or the SI cases as can be observed in Table 4.

The previous results largely depend on application characteristics, both in terms of the degree of sharing and in the heterogeneity of the load that the CPUs exercise on the L2 cache. We found that the latter effect seems to have more consequence on the results presented above. For instance, in mgrid, on the average at any instant, less than 3% of the blocks were shared across the CPUs (i.e., they were present in more than one L2 at any time in the private case). On the other hand, when we move to the shared L2 configurations for this application, we get more than 25% savings compared to the private case. Consequently, in the rest of this discussion, we examine the issue of heterogeneity of L2 load in greater depth, and look at this heterogeneity at an intra-application (finer) and an inter-application (coarser) granularity.

3.2.1. Intra-Application Heterogeneity In order to understand how the application characteristics impose a heterogeneous load on the L2 cache, we next define two important metrics and track their values over the course of the execution for each epoch. We call these metrics the *Spatial Heterogeneity Factor (SHF)* and the *Temporal Heterogeneity Factor (THF),* and they are defined as follows:

$$SHF_{epoch} = \frac{\sigma_{cpu}(L1Misses)}{L1Accesses_{epoch}}$$

$$THF_{cpu} = \frac{\sigma_{epoch}(L1Misses)}{L1Accesses_{cpu}},$$

where SHF_{epoch} and THF_{cpu} correspond to the spatial heterogeneity factor for a single epoch across the CPUs, and the temporal heterogeneity factor for a single CPU across the epochs, respectively. The $\sigma_{cpu}(L1Misses)$ represents the standard deviation of the L1 misses across the CPUs for that epoch, while $\sigma_{epoch}(L1Misses)$ represents the standard deviation of the L1 misses across the epochs for a single CPU. $L1Accesses_{epoch}$ and $L1Accesses_{cpu}$ give the number of L1 accesses within an epoch (across all the processors) and the number of L1 accesses by a processor (across all epochs), respectively. Essentially, these metrics try to capture the standard deviation (heterogeneity) between the CPUs (spatial) or over the epochs (temporal) of the load imposed on the L2 structure (which is the L1 misses). The reason they are weighted by the L1 accesses is because we want to more accurately capture the resulting effect on the overall IPC, i.e., in an application the standard deviation may be high but if the overall accesses/misses are low, then there is not going to be a significant impact on the IPC.

We plot SHF_{epoch} and THF_{cpu} for each epoch and each processor, respectively, in Figures 4 and 5 for the nine individual application executions. We find a direct correlation between the cases where our split shared L2 organizations does better (in Table 3) and the cases where the heterogeneity factors are high in these graphs. Specifically, we

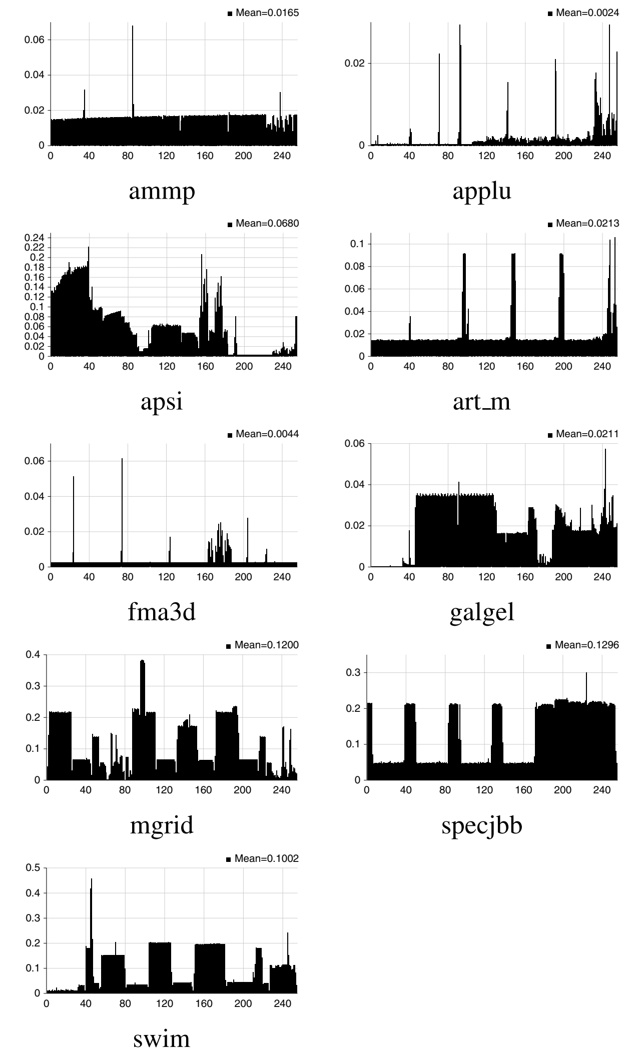


Figure 4. Spatial Heterogeneity Factor (SHF) for each epoch. Note that the y-axes are on different scales.

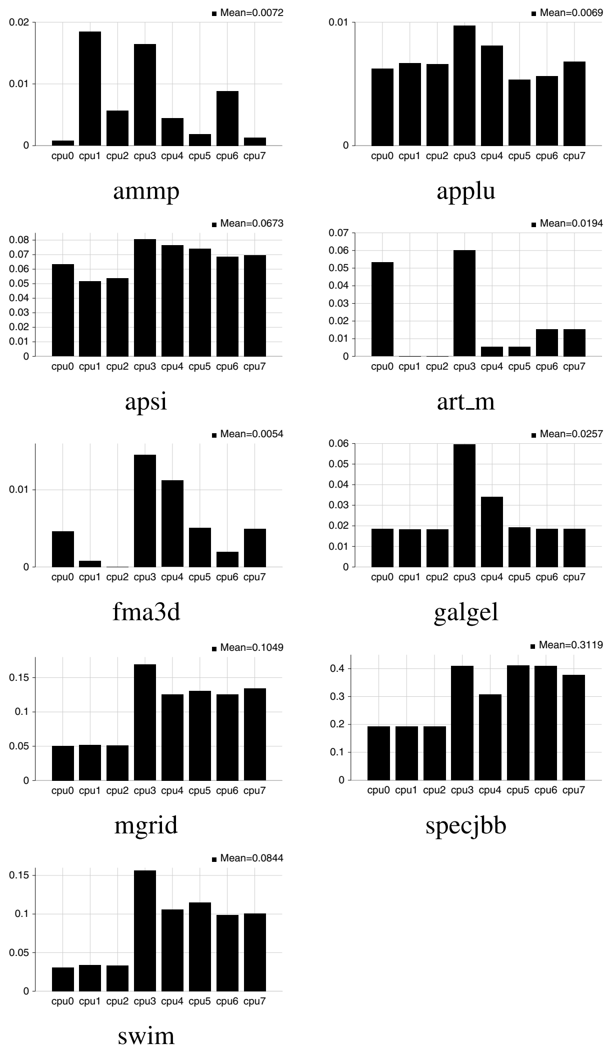


Figure 5. Temporal Heterogeneity Factor (THF) for each CPU. Note that the y-axes are on different scales.

find the heterogeneity — both spatial and temporal — is much higher in specjbb, mgrid, swim and to some extent in apsi, compared to the other five. Note that these are also the applications where we find significant improvements for SSU or SSN compared to the P or SI cases.

When we compare SSU and SSN, we find that though the latter gives slightly better results (in five of our nine individual application executions), the difference between the two schemes is not very significant. There are several reasons for this behavior. First, the allocation of L2 units to the CPUs in SSN is not necessarily the most efficient. For instance, if we consider SSN-152, only when there is one CPU that dominates on the L2 load, with five others in-between the extremes right through the execution, would this be an ideal choice. If the application characteristic does not match this static choice of different chunk sizes, then the performance may not be very good. It should be emphasized that this is a problem of the specific implementation evaluated here, rather than a problem of our table-based mechanism, and future work — using our table-based mechanism — can possibly develop fine-grain dynamic split allocation strategies based on L2 behavior. Second, in addition to misses, the other performance advantage that can come for SSN compared to SSU is in moving more of the remote hits (of SSU) to the local hit side. Because of non-uniform allocations, it is possible that a CPU with a higher L2 load may find more blocks within its allocation rather than in someone else's allocation (which is still a hit but incurs a higher access latency). However, if we look at Table 5 which shows the local and remote hit (and L2 misses) fractions for SSU, we see that the contribution of remote hits is not very high, and it is the effect of the misses that is more important.

3.2.2. Inter-Application Heterogeneity In terms of inter-application heterogeneity, our four workloads in Table 4

Workload	Local Hit	Remote Hit	Miss
ammp	94.2%	2.9%	2.9%
art_m	47.3%	11.1%	41.6%
galgel	80.4%	10.4%	9.2%
swim	69.5%	0.7%	29.8%
apsi	89.7%	4.2%	6.2%
fma3d	76.1%	1.7%	22.2%
mgrid	81.6%	0.3%	18.1%
applu	76.6%	10.0%	13.4%
specjbb	75.1%	17.1%	7.8%
ammp+apsi	91.8%	2.2%	6.0%
ammp+fma3d	90.9%	2.2%	6.9%
swim+apsi	76.8%	8.5%	14.6%
swim+mgrid	79.0%	3.8%	17.2%

Table 5. The breakdown of L2 accesses for SSU.

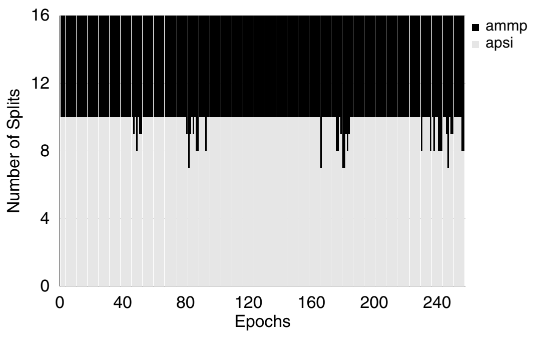


Figure 6. The L2 space allocation for ammp+apsi under SSN-152.

— ammp+apsi, ammp+fma3d, swim+apsi, swim+mgrid — capture different scenarios of L2 load. In ammp+fma3d, the load introduced on L2 (see the miss rates of these two applications in Table 3) by both applications is rather low, thus not showing significant difference across the schemes. In swim+apsi and swim+mgrid, the L2 load by both applications is rather high (and balanced), with the balance being higher in the former (see the heterogeneity graphs for apsi and mgrid, where the latter shows higher heterogeneity) making the private case a fairly good choice. Still, the load across applications is more or less balanced, thus making the schemes again comparable. On the other hand, when we consider ammp+apsi, we have the first with a rather low load, and the second with a rather high load. With an unequal allocation to this space-shared multiprogrammed workload, we can give different amounts of cache space to these individual applications so that we can get the best overall IPC. For instance, with SSN-152, we can give 1.25MB (of the total 2MB L2) to apsi and the other 0.75MB to ammp, though this partitioning can change from epoch to epoch based on the dynamics of the execution (see Figure 6 to see how L2 space is allocated to the two applications for the duration of execution which more or less tracks this 5:3 proportion). Consequently, this can provide a lower miss rate for apsi without really affecting the miss rate of ammp, to provide a better overall IPC value.

3.3. Sensitivity Analysis

We have also studied the impact of L1 and L2 sizes, memory access cost, and the number of CPUs. The reader is refered to [11] for detailed results, and the overall benefits of our approach are still quite significant across the different configurations.

4. Related Work

There has been a considerable amount of previous research in designing memory hierarchies and cache coherence protocols for SMP (multi-chip multiprocessors) systems. It is well beyond the scope of this paper to cover all such related work, and the reader is referred to [20] for an in-depth treatment of the different contributions to this area. At the same time, prior studies have also looked at the capacity/organization issues for L3 shared caches [7] and the characteristics of the workloads affecting memory hierarchy design [19, 4, 24], again in the context of SMP systems. On the other hand, our work is targetting single chip multiprocessors, where an off-chip access to the memory hierarachy can incur a much higher cost than exchanging information on the shared interconnect between the cores on a CMP. Consequently, it becomes more important to reduce off-chip accesses, rather than save a few cycles within the chip. There has been no prior in-depth comparison of the pros and cons of private vs. shared organizations for the on-chip last line of defense to the memory wall for CMPs.

An advantage of our implementation of the Shared Processor-based Split Cache design, is the adaptability/morphability to application characteristics, i.e., one can possibly give the right core, the right L2 space at the right time, changing the allocation whenever needed. Prior work has recognized the importance of morphable/malleable/adaptive caches for different purposes. One body of work [28, 1] looks at adjusting cache sizes and/or other parameters, primarily in the context of uniprocessors, in order to save dynamic/leakage power. The other kinds of work [17] on this topic dynamically adjust the cache for performance benefits. In the design automation area, there have been efforts to design application-specific memory hierarchies based on software-managed components [6]. But none of these have looked at the benefits of malleability of L2 organization for CMPs.

Another related work on the topic of adjusting cache space to application characteristics is that by [26, 25], wherein they partition the cache space between multiple processes/activities executing on one processor inorder to reduce their interference between the time slices. Cache space usage across multiple threads has also been studied in the context of SMT processors [27].

The popularity of CMPs is clearly evident from the different commercial developments and research projects [**?**, 16, 10, 9, 13, 14, 15, 3] on this topic. The issues concern the design of the interconnect, and the design of the datapath for effective on-chip parallelism. To our knowledge, this is the first work that has examined different L2 organizations, and proposed a new one, in order to reduce off-chip memory accesses.

5. Concluding Remarks

The low latency of accessing the cache hierarchy and exchanging information between the processors is a distinct advantage with chip multiprocessors. With such deep levels of integration, it becomes extremely critical to reduce off-chip accesses, that can have important performance and power ramifications in their deployment. Consequently, this paper has examined the organization for the last level of the on-chip cache hierarchy (called the last line of defense) before an access goes off-chip, where it can incur a high latency and significant dynamic power.

Two well-known principles for organizing this level include the private and shared organization, but each has its relative advantages and drawbacks. In the private case, the CPUs are relatively insulated from the load they impose on this level, and do not need to traverse a shared interconnect to get to this level in the common case. On the other hand, the advantage of the shared organization is that it can allow one CPU to eat into the cache space of another whenever needed to allow more heterogeneous allocation of this cache space. However, such a capability can also become very detrimental in causing interference between the address streams generated by the different CPUs, thereby incurring additional misses.

Recognizing these trade-offs, we have proposed a new last line of defense organization wherein we use the underlying advantages of the shared structure (to balance the load), and at the same time provide a way of insulating the diverse requirements across the cores whenever needed. The basic idea is to have multiple cache units, and have them allocated on a CPU id basis (rather than on an address basis as is done in typically banked caches). Each lookup from a CPU looks first at its set of units, but can subsequently lookup other units as well before going off-chip (thus maintaining the shared view). However, upon a miss, the request can allocate the block only into one of its units, thereby reducing the interference.

We have proposed a flexible mechanism for implementing this shared processor-based split organization, that allows software to configure the splits spatially (between the CPUs) and temporally (can vary over time). With CMPs possibly targeting a diverse range of workloads, from commercial high-end workloads, to scientific and embedded applications, it becomes important to allow this flexibility for dynamic adaptation. At the same time, this mechanism can integrate easily with existing interconnects and coherence mechanisms. This organization is also fairly power efficient (though not evaluated quantitatively in this paper), since we found that the number of units referenced upon each access is comparable to that of the private and address-based shared mechanisms, and in fact reduces off-chip accesses.

Our results found that even when we use a single split for each CPU we are doing better than the private (P) or the shared address-based interleaved (SI) organizations, since it is able to better balance the load, and reduces the interference. In this paper, we have not delved into methods for allocating the splits to the CPUs at runtime, and this is part of our future work. Still, preliminary results with non-uniform allocations between the CPUs shows benefits in multiprogrammed workloads, where the loads can be quite diverse as was shown here, and even in a single program execution sometimes, if the load imbalance is high.

In our ongoing work, we are examining techniques for allocating the splits to the CPUs temporally over the execution, compiler support for determining allocation units, and the power consumption of these organizations.

Acknowledgements: This work has been funded in part by NSF grants 0103583, 0130143 and Career Award 0093082.

References

[1] D. H. Albonesi. Selective cache ways: On-Demand Cache Resource Allocation. In *International Symposium on Microarchitecture*, pages 248–, 1999.

[2] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady. SPEC OMP: A New Benchmark Suite for Measuring Parallel Computer Performance. In *Proc. WOMBAT,* July 2001.

[3] L. A. Barroso et. al. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In Proc. *International Symposium on Computer Architecture,* Vancouver, Canada, June 12–14 2000.

[4] L. A. Barroso, K. Gharachorloo, A. Nowatzyk, and B. Verghese. Impact of Chip-Level Integration on Performance of OLTP Workloads. In Proc. *the Sixth International Symposium on High-Performance Computer Architecture,* January 2000.

[5] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In Proc. *the 27th International Symposium on Computer Architecture,* June, 2000.

[6] F. Catthoor, S. Wuytack, E. D. Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom Memory Management Methodology – Exploration of Memory Organization for Embedded Multimedia System Design.* Kluwer Academic Publishers, 1998.

[7] M. Dubois, J. Jeong, S. Razeghia, M. Rouhaniz, and A. Nanda. Evaluation of Shared Cache Architectures for TPC-H. In Proc. *the Fifth Workshop on Computer Architecture Evaluation using Commercial Workloads,* Cambridge, Massachusetts, Feb 2002.

[8] K. M. Jackson, K. N. Langston. IBM S/390 Storage Hierarchy G5 and G6 Performance Considerations. 43(5/6):847, 1999.

[9] I. Kadayif, M. Kandemir, and U. Sezer. An Integer Linear Programming Based Approach for Parallelizing Applications in On-Chip Multiprocessors. In Proc. *Design Automation Conference,* New Orleans, LA, June 2002.

[10] V. Krishnan and J. Torrellas. A Chip Multiprocessor Architecture with Speculative Multi-threading. *IEEE Transactions on Computers, Special Issue on Multi-threaded Architecture,* September 1999.

[11] C. Liu, A. Sivasubramaniam, and M. Kandemir. Organizing the Last Line of Defense before Hitting the Memory Wall for CMPs. Penn State University *Tech Report CSE-03-019*, 2003.

[12] L. Benini and G. De Micheli. System-level Power Optimization: Techniques and Tools. *TODAES* 5(2): 115-192 (2000).

[13] MAJC-5200. http://sun.com/microelectronics/MAJC/5200wp.html

[14] MP98: A Mobile Processor. http://www.labs.nec.co.jp/MP98/

[15] B. A. Nayfeh, L. Hammond, and K. Olukotun. Evaluating Alternatives for a Multiprocessor Microprocessor. In Proc. *the 23rd Intl. Symp. on Computer Architecture,* pp. 66–77, Philadelphia, PA, 1996.

[16] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The Case for a Single Chip Multiprocessor. In Proc. *the 7th Intl Conference on Architectural Support for Programming Languages and Operating Systems,* ACM Press, New York, 1996, pp. 2–11.

[17] P. Ranganathan, S. V. Adve, and N. P. Jouppi. Reconfigurable Caches and Their Application to Media Processing. In *Proc. ISCA*, pages 214–224, 2000.

[18] G. Reinman and N. P. Jouppi. CACTI 2.0: An Integrated Cache Timing and Power Model. Compaq, WRL, *Research Report 2000/7,* February 2000.

[19] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, and A. Gupta. The Impact of Architectural Trends on Operating System Performance. In *Proc. 15th ACM Symposium on Operating System Principles,* Colorado, December 1995.

[20] J. P. Singh and D. Culler. *Parallel Computer Architecture: A Hardware-Software Approach,* Morgan-Kaufmann, 1998.

[21] Simics. http://www.simics.com/

[22] SPARC UPA System Bus. Sun Microsystems. http://www.sun.com/oem/products/manuals/802-7835.pdf

[23] Specjbb2000 Java Business Benchmark. http://www.specbench.org/osg/jbb2000/

[24] R. Stets, K. Gharachorloo, and L. Barroso. A Detailed Comparison of Two Transaction Processing Workloads. In Proc. *the 5th Annual Workshop on Workload Characterization,* November 2002.

[25] G. Suh, S. Devadas, and L. Rudolph. Dynamic Cache Partitioning for Simultaneous Multithreading Systems. In *Proc. IASTED PDCS,* August 2001.

[26] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic Partitioning of Shared Cache Memory. *Journal of Supercomputing*, 2002.

[27] D.M. Tullsen, S.J. Eggers, and H.M Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. *Proc. of Intl. Symp. on Comp. Arch.*, pages 392-403, 1995.

[28] S.-H. Yang, M. D. Powell, B. Falsafi, K. Roy, and T. N. Vijaykumar. An Integrated Circuit/Architecture Approach to Reducing Leakage in Deep-Submicron High-Performance I-Caches. In *Proc. HPCA*, pages 147–158, 2001.