StimulusCache: Boosting Performance of Chip Multiprocessors with Excess Cache

Hyunjin Lee, Sangyeun Cho, and Bruce R. Childers
Dept. of Computer Science, Univ. of Pittsburgh
{abraham,cho,childers}@cs.pitt.edu

Abstract

Technology advances continuously shrink on-chip devices. Consequently, the number of cores in a single chip multiprocessor (CMP) is expected to grow in coming years. Unfortunately, with smaller device size and greater integration, chip yield degrades significantly. Guaranteeing that all chip components function correctly leads to an unrealistically low yield. Chip vendors have adopted a design strategy to market partially functioning processor chips to combat this problem. The two major components in a multicore chip are compute cores and on-chip memory such as L2 cache. From the viewpoint of the chip yield, the compute cores have a much lower yield than the on-chip memory due to their logic complexity and well-established memory yield enhancing techniques. Therefore, future CMPs are expected to have more available on-chip memories than working cores. This paper introduces a novel on-chip memory utilization scheme called StimulusCache, which decouples the L2 caches of faulty compute cores and employs them to assist applications on other working cores. Our extensive experimental evaluation demonstrates that StimulusCache significantly improves the performance of both single-threaded and multithreaded workloads.

1. Introduction

Continuous device scaling causes more frequent hard faults to occur in processor chips at manufacturing time [3, 24]. Two major sources of faults are physical defects and process variations. First, physical defects can cause a short or open, which makes a circuit unusable. While technology advances improve the defect density of semiconductor materials and the manufacturing environment, with ever smaller feature sizes, the critical defect size continues to shrink. Accordingly, physical defects remain a serious threat to achieving profitable chip yield. Second, process variations can cause mismatches in device coupling, unevenly degraded circuit speeds, and higher power consumption, increasing the probability of circuit malfunction at nominal conditions.

To improve chip yield, processor vendors have recently adopted "core disabling" for chip multiprocessors (CMPs) [1, 19, 25, 26]. Using programmable fuses and registers, this approach disables faulty cores and enables only functional ones. As long as there are enough sound cores, this technique produces many partially operating CMPs, which would otherwise be discarded without core disabling. For instance, IBM's Cell processor reportedly has a yield of only 10% to 20% with eight synergistic processor elements (SPEs). However, by disabling one (faulty) SPE, the yield jumps to nearly 40% [26]. AMD sells tri-core chips [1], which is a byproduct of a quad-core chip with a faulty core. The NVIDIA GeForce 8800 has three product derivatives with 128, 112, and 96 cores [19]. The GeForce chips with small core counts are believed to be partially disabled chips of the same 128-core design; all the designs have the same transistor count. Lastly, the Sun UltraSPARC T1 has three different core counts: four, six, and eight cores [25].

Inside a chip, logic and memory have very different yield characteristics. For the same physical defect size and process variation effect, memory may be more vulnerable than logic due to small transistor size. However, various fault handling schemes have been successfully deployed to significantly improve the yield of memory, including parity, ECC, and row/column redundancy [13]. Furthermore, a few cache blocks may be "disabled" or "remapped" to opportunistically cover faults and improve yield without affecting chip functionality [4, 12, 15, 16]. In fact, ITRS reports that the primary issue for memory yield is to protect the support logic, not the memory cells [13].

Traditional core disabling techniques take a core and its associated memory (e.g., private L2 cache) offline without consideration for whether the core or its memory failed. Thus, a failed core causes its associated memory to be unavailable, although the memory may be functional. For example, AMD's Phenom X3 processor disables one core along with its 512KB L2 cache [1]. Due to the large asymmetry in the yield of logic and memory, however, such a coarse-grained disabling scheme will likely waste much memory capacity in the future. We hypothesize that the performance of a CMP will be significantly improved if the sound cache memories associated with faulty cores are utilized by other sound cores. To explore such a design approach, this paper proposes *StimulusCache*, a novel architecture that utilizes unemployed "excess" L2 caches. These excess caches come from disabled cores where the cache is functional.

We answer two main technical questions for the proposed

This work was supported in part by NSF grants CCF-0811295, CCF-0811352, CCF-0702236, and CCF-0952273.

StimulusCache approach. First, what is the system requirement (including system software and microarchitecture) to enable StimulusCache? Second, what are the desirable excess cache management strategies under various workloads? The ideas and results we present in this paper are also applicable to chips without excess caches. For example, under a low system load, advanced CMPs dynamically put some cores into a deep sleep mode to save energy [11]. In such a scenario, the cache capacity of the sleeping cores could be borrowed by other active cores. We make the following contributions in this paper:

- **A new yield model for processor components.** We develop a "decoupled" yield model to accurately calculate the yield of various processor components having both logic and memory cell arrays. Based on component yield modeling, we perform an availability study for compute cores and low-level cache memory with current and future technology parameters to show that there will likely be more functional caches available than cores in future CMPs (Section 2). StimulusCache aims to effectively utilize these excess caches.
- **Architectural support for StimulusCache.** We develop the necessary architectural support to enable StimulusCache multicore architectures (Section 3). We find that the added datapath and control overhead to the cache controllers is small for 8 and 32 core designs.
- **Strategies to utilize excess caches.** We explore and study novel policies to utilize the available excess caches (Section 4). We find that organizing excess caches as a non-inclusive shared victim L3 cache is very effective. We also find it beneficial to monitor the cache usage of individual threads and limit certain threads from using the excess caches if they cannot effectively use the extra capacity.
- **An evaluation of StimulusCache.** We perform a comprehensive evaluation of our proposed architecture and excess cache policies to assess the benefit of StimulusCache, which is compared with the latest private cache partitioning technique, DSR [21] (Section 5). We examine a wide range of workloads using an 8-core and a 32-core CMP configurations. StimulusCache is shown to consistently boost the performance of all programs (by up to 45%) with no performance penalty.

2. Decoupled Yield Model for Cores and Caches

2.1. Baseline yield model and parameters

Chip yield is generally dictated by defect density D_0, area A, and clustering factor α. We use a negative binomial yield model from the ITRS report [13], where the yield of the chip die (Y_{Die}) is:

$$Y_{Die} = Y_M \times Y_S \times \left(\frac{1}{1 + AD_0/\alpha}\right)^{\alpha} \qquad (1)$$

In the above, Y_M is the material intrinsic yield, which we fix to 1 and do not consider in this work. Y_S is the systematic yield, which is generally assumed to be 90% for logic and 95% for memory [13]. α is a cluster parameter and assumed to be 2 as in the ITRS report. Although technologies with smaller feature sizes are more vulnerable to defects, ITRS targets the same D_0 for upcoming technologies when matured, due to process technology advances.

To compute a realistic yield with equation (1) in the remainder of this paper, we derive D_0 from the published yield of the IBM Cell processor chip, which is 20% [26].[1] For accurate calculation, we differentiate the logic portion whose geometric structure is irregular from the memory cell array that has a regular structure in each functional block. While the memory cell array may be more vulnerable to defects and process variability, it is well-protected with robust fault masking techniques, such as redundancy and ECC [13, 16, 20].

We use CACTI version 5.3 [30] to obtain the area of the memory cell array in a memory-oriented function block. From CACTI and die photo analysis, we determined that the memory cell array of the PPE and the SPEs account for about 8% and 14% of the total chip area, repectively.[2] Based on the above analysis, we determine the total memory cell array area is 22% of the chip area ($175mm^2$ in 65nm technology). We can derive D_0 with equation (1) using the total non-memory chip area. We calculated D_0 to be $0.0181/mm^2$.

2.2. Decoupled yield model

Given multiple functional blocks in a chip and their individual yields (Y_{block}), the chip yield can be computed as [7]:

$$Y_{Die} = \prod_{i=1}^{N} Y_{block_i} \qquad (2)$$

It is clear that the yield of a vulnerable functional block can be a significant potential threat to the overall yield. Therefore, it becomes imperative to evaluate each functional block's yield separately to prioritize and guide design tuning activities, e.g., implementing isolation points and employing functional block salvaging techniques. To accurately evaluate the yield of individual functional blocks, as suggested in the previous subsection, we propose to define their yield in terms of the logic yield and the memory cell array yield as follows:

$$Y_{block_i} = Y_{logic_i} \times Y_{memory_i} \qquad (3)$$

[1] In Sperling [26] the yield for the Cell processor was vaguely given as 10%–20%. While a lower yield makes an even stronger case for StimulusCache, we conservatively use the highest yield estimate (20%).

[2] CACTI reports that in a 512KB L2 cache (Cell processor's PowerPC element has a 512KB L2 cache) the memory cell array accounts for about 78% of the total cache area. We measure the L2 cache area of PPE from the die photo and use 78% of that to the memory cell array area. The cell area of the local memory in SPEs is directly measured using the die photo.

Functional blocks	Total area (mm^2)	Logic area (mm^2)	Cell array area (mm^2)	Yield
FEC	2.775	2.425	0.350	95.74%
IEC	0.798	0.798	—	98.57%
FPC	1.776	1.776	—	96.86%
MEC	1.897	1.634	0.263	97.10%
BIU	2.094	2.094	—	96.31%
Processing	9.340	8.727	0.613	84.85%
L2 Cache	5.318	1.117	4.201	98.01%

Table 1. Estimated functional block yields of ATOM processor. (FEC: Front End Cluster, IEC: Integer Execution Cluster, FPC: Floating Point Cluster, MED: Memory Execution Cluster, BIU: Bus Interface Unit)

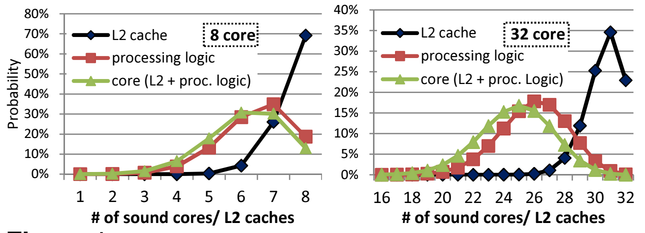


Figure 1. Yield of L2 cache, processing logic, and core (L2 cache + processing logic) for 8-core (left) and 32-core (right) CMPs.

Using D_0 derived in Section 2.1, we estimate the expected yield for the key functional blocks of the ATOM processor [10] using our decoupled yield analysis approach.[3] Table 1 depicts the area and yield for each functional block. The processing block has the five logic-dominant functional blocks (FEC, IEC, FPC, MEC, and BIU). The L2 cache is a memory-dominant functional block. Although FEC and MEC are logic-dominant functional blocks, they have 32KB and 24KB 8-T (i.e., eight transistors compose one cell) L1 cache. To accurately estimate the functional block yield of FEC and MEC, the 8-T cell array's 30% area overhead over a conventional 6-T cell array is faithfully modeled.

Figure 1 depicts the yield for 8-core and 32-core CMPs using an ATOM-like core [10] as a building block.[4] It separately shows core and L2 cache yield along with the traditional "combined" yield, computed with the decoupled yield model. For the 8-core case in Figure 1(a), less than 13% of the chips have eight sound cores and caches. It is clearly shown that this low yield is caused by the poor yield of the compute cores. In contrast, the cache memory has a much higher yield; in 70% of the produced dies, all eight cache memories are functional. As the core count increases, the discrepancy between the number of sound cores and L2 caches widens. Figure 1(b) shows the 32-core case, where 83% of the chips have at least 30 sound L2 caches while only 5% of dies have 30 sound cores or more.

[3]For various process generations, the initial defect density and the trend of defect density improvement ("yield learning") are very similar [31]. Thus, we can use the derived defect density for 45nm technology without loss of generality.

[4]We assume that the yields of chip I/O blocks and other supporting blocks (e.g., PLL) are 100% for simpler and intuitive analysis. Typically, such blocks employ large geometries, which dramatically decreases the effective defect density. Moreover, we assume that the L2 cache's cell array is salvaged by redundancy and cache block disabling [20]. We employ Monte Carlo simulation [16] to calculate the cell array yield when such salvaging techniques are used. With 5% row redundancy and disabling of up to 8 lines, the cell array yield is 99.82%.

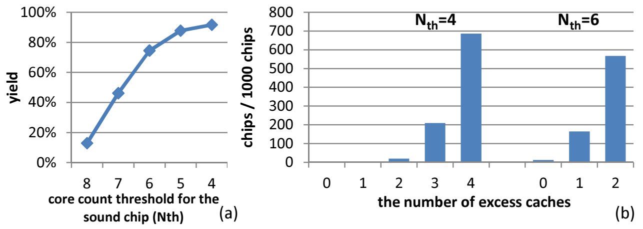


Figure 2. (a) Yield with varying core count thresholds (N_{th}) for the 8-core CMP. (b) The number of chips (out of 1,000 chips) with different numbers of excess caches when four cores enabled (left) and six cores enabled (right).

With core disabling, chip yield can be greatly improved. Figure 2(a) depicts the yield improvement due to core disabling for the 8-core case. We define the criteria for a "good die" based on the core count threshold, N_{th}—i.e., does the chip have at least N_{th} healthy cores? When $N_{th} = 4$, the yield is 91% whereas the raw yield ($N_{th} = 8$) is just 13%. Figure 2(b)(left) shows the available excess caches in 1,000 good dies when $N_{th} = 4$ and 4 cores are enabled (i.e., we have two product configurations: 8 cores or 4 cores with excess caches). It is shown that more than 68% of the 4-core chips have four excess caches. Figure 2(b)(right) plots the available excess caches when $N_{th} = 6$ and 6 cores are enabled. 57% of all 6-core chips have two excess caches. These results demonstrate that there will be plenty of excess caches from the loss of faulty cores in future CMPs. Once tapped, these unemployed, virtually free cache resources can be used to improve the performance of CMP systems.

3. Overview of StimulusCache

Given the high likelihood of available excess caches, one would naturally want to utilize them to improve system performance. A naïve strategy could simply allocate excess caches to cores that run cache capacity-hungry applications. Adding more capacity to specific cores creates *virtual L2 caches* which have more capacity than other caches. However, with diverse workloads on multiple virtual machines (VMs), deriving a good excess cache allocation can become complex. For example, the user might pursue the best performance, while, in another case, the user may want to guarantee QoS and fairness of specific applications. To achieve these potential goals, we propose a hardware/software cooperative design approach. In this section, we illustrate the proposed StimulusCache framework by discussing its hardware support, software support, and an extended example.

3.1. Hardware design support

Shared and private caches are two common L2 cache designs. There are also many hybrid (or adaptive) schemes [5,

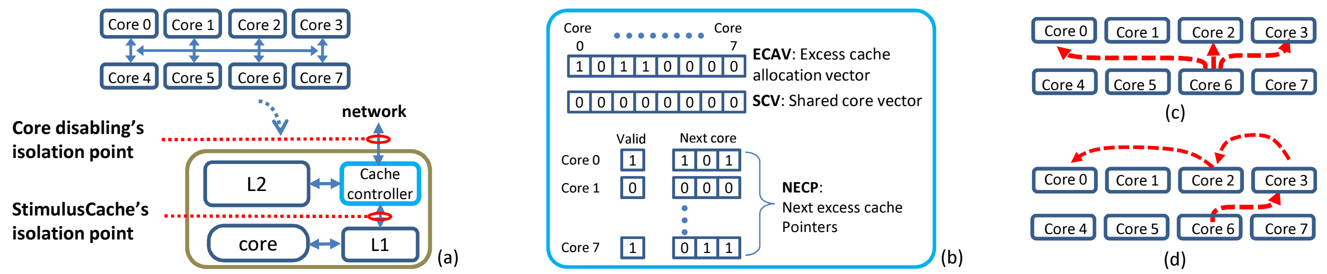

Core 0
Core 1
Core 2
Core 3
Core 4
Core 5
Core 6
Core 7
network
Core disabling's isolation point
StimulusCache's isolation point
L2
Cache controller
core
L1
(a)
ECAV: Excess cache allocation vector
SCV: Shared core vector
Valid
Next core
NECP: Next excess cache Pointers
(b)
(c)
(d)

Figure 3. (a) Fault isolation point comparison: core disabling and StimulusCache. (b) New data structure in StimulusCache's cache controller. ECAV shows which excess caches have been allocated to this functional core. SCV lists the cores that use this excess cache. NECP shows the next level excess cache to search on a miss. (c) Parallel search using ECAV. (d) Serial search using NECP.

21, 22]. A private L2 cache design has several benefits over a shared L2 cache design: fast access latency, simple design, resource/performance isolation, and less network traffic overhead. Such a private design typically has poor utilization. However, the extra cache capacity from available excess caches can mitigate this problem. Thus, our initial StimulusCache design is based on the private L2 cache architecture like IBM Power6 [9] and AMD Phenom [1].

Figure 3(a) shows the fault isolation point of a conventional core disabling technique and StimulusCache in an 8-core CMP that has a private L2 cache per core. Wherever faults occur in the processing logic, conventional core disabling takes offline the whole core including its private L2 cache. Thus, the fault isolation point is the core's network interface. StimulusCache aggressively pushes the isolation point beyond the L2 cache controller. Consequently, StimulusCache can salvage the L2 cache as long as the L2 cache and cache controller are fault-free.

In StimulusCache, each core should be able to access excess caches without any limitation. We introduce a set of hardware data structures in the cache controllers, as shown in Figure 3(b), to provide flexible accessibility to excess cache. The excess cache allocation vector (ECAV) shows which caches should be examined to find requested data on a local L2 miss. Using ECAV, multiple excess caches can be accessed in parallel as shown in Figure 3(c). The Shared Core Vector (SCV) is to assist cache coherence and will be discussed in detail below. Lastly, Next Excess Cache Pointers (NECP) enable fine-grained excess cache management. Each pointer points to the next memory entity to be accessed, being another excess cache or main memory. NECP form access chains of excess caches for individual cores as shown in Figure 3(d). With ECAV and NECP, StimulusCache supports both parallel and sequential search of the excess caches. Parallel access is faster while sequential access has less network traffic and power consumption. The best choice could be determined by the overall system management goal; for example, performance or power optimization. Additionally, each tag has the origin core ID of the cache block. Overall, StimulusCache's memory overheads are: $\lceil \log_2 N \rceil$ bits per block (core ID) and $(3N + N \log_2 N)$ bits per core (ECAV, SCV, and NECP) for an N-core CMP. The overheads correspond to 0.55% and 0.92% of a 512KB L2 cache for an 8-core and a 32-core CMP, respectively.

Although excess caches can be used to improve performance, static allocation for entire program execution may not exploit the full potential of excess cache because programs have different phases with varying memory demand. To support program phase adaptability, excess caches should be dynamically allocated to cores based on performance monitoring. StimulusCache's advantage for dynamic allocation is its inherent performance monitoring capability at cache bank granularity. For example, data flow-in, access, hit and miss counts, which are already implemented in CMPs [2], can be measured and used to fully utilize the potential of excess caches.

Coherence management in StimulusCache is similar to a private L2 cache. For moderate scales (up to 8 cores), broadcast is used for cache coherence. For large scale (greater than 8 cores), a directory-based scheme is used [5,17]. However, to utilize excess caches, the coherence protocol has to be changed. An excess cache can be shared by multiple cores, or it can be exclusively allocated to a specific core. To manage cache coherency, the cache controller has SCV as shown in Figure 3(b). The SCV for a faulty core lists the functional cores that utilize the excess cache of the faulty core. When L1 data invalidation occurs, the SCV identifies the cores that need to receive an invalidation message. For functional cores, SCV entries are empty because their local L2 caches are not shared.

3.2. Software support

An excess cache is a shared resource among multiple cores; system software (e.g., the OS or VMM) has to decide how to allocate the available excess caches. The system software would assign an excess cache to a core in a way that meets the application needs by properly setting the values of ECAV, SCV, and NECP in the cache controllers.

Depending on the resource utilization policy, the system software decides whether an excess cache is exclusively allocated to a core. Exclusive allocation guarantees performance isolation of each core. However, if there is no infor-

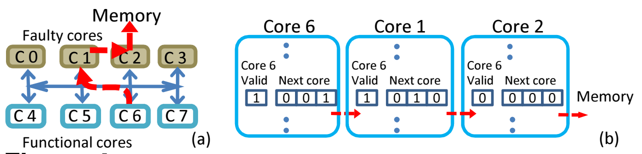


Figure 4. Excess cache allocation example. (a) Excess caches from four faulty cores (core 0–3). (b) NECP in core 6, 1, and 2. An excess cache access chain for core 6 is shown. A zero valid bit indicates that the excess cache is the last one in the chain before the main memory. The access sequence is, therefore, core 6 (working core)⇒core 1 (excess cache)⇒core 2 (excess cache)⇒memory.

mation about memory demands, a fixed exclusive allocation is somewhat arbitrary. In that case, evenly allocating available excess caches to all sound cores is a reasonable choice. If there is not enough excess cache for all available cores, the excess caches are allocated to some cores, and the OS can schedule memory-intensive workloads to the cores with excess cache. Shared allocation can exploit the full potential of excess cache usage. However, the excess caches could be unfairly shared if some cores are running cache capacity thrashing programs.

3.3. An extended example

Figure 4 gives an example that shows how excess caches can be allocated. In this example, cores 0 to 3 are faulty cores, and thus, they provide excess caches. Cores 4 to 7 are functional. Core 6 has been allocated two excess caches from core 1 and core 2. The excess cache from core 1 has higher priority (it is at the first of an access chain). For accessing excess cache to examine data (i.e., a data read), core 6 can search the excess caches of core 1 and 2 simultaneously in parallel or sequentially as shown.

When data is written to the excess cache (e.g., a data eviction from the local L2 cache of core 6), the destination cache of the data has to be determined. Figure 4(a) shows an excess cache access chain. In this example, core 6's L2 eviction data goes to the excess cache in core 1, identified with the NECP (in core 6). If the data should be written to the next cache in the chain, it goes to the excess cache in core 2 based on the NECP in core 1. Figure 4(b) shows how the NECPs in the cache controllers are used to build the excess cache access chain for Figure 4(a).

Figure 5 shows example scenarios of how cache coherence is done in StimulusCache. We show the inclusive L2 cache as examples because the exclusive L2 cache requires no special management in terms of coherency. The excess caches for core along with the core's private L2 cache, create *virtual L2 domain*. Each core has valid *inclusiveness* if the data in L1 cache has the same copy in the virtual L2 domain. Therefore, if exclusive L2 data is migrated to an excess cache, an L1 data invalidation is not needed. As shown in Figure 5(a), only one copy of valid data is kept in either the L2 cache or the excess cache. Figure 5(b) shows a different scenario where two cores have the same data (i.e., each has a replica of the data) which is shared by cache-to-cache transfer. If one core should evict this shared data, the data is not migrated to the excess cache. Instead, it is simply evicted as there is a valid copy in P2's L2 cache, satisfying the L2-L1 inclusiveness requirement. Figure 5(c) shows another scenario. In this case, if exclusive data in L2 is migrated to the excess cache, no L1 invalidation is needed because there is only one L1 data. Finally, Figure 5(d) depicts data migration that incurs L1 invalidation. To maintain L2-L1 inclusiveness, if the data in the excess cache is migrated to P2's L2 cache, then P1's L1 data should be invalidated. The proposed hardware support provides sufficient information to achieve coherency with excess caches.

4. Excess Cache Utilization Policies

Based on the hardware and software support in Section 3, this section presents three policies to exploit excess caches.

4.1. Static private: Static partition, Private cache

This scheme exclusively allocates the available excess caches to individual cores: only one core can use a particular excess cache as assigned by the system software. Figure 6(a) shows two examples of a static allocation of excess caches to cores. If the workload on multiple cores have similar memory demands, the available excess caches can be uniformly assigned to cores (symmetric case). A server workload or a well-balanced multithreaded workload are good examples of this case. However, if a workload has particularly high memory demands, then more excess caches can be assigned to a specific core for the workload. This configuration naturally generates an asymmetric CMP as shown in Figure 6(a).

In effect, the static private scheme expands a core's L2 cache associativity to $(K + 1)N$ using K excess caches that are N-way associative. Figure 6(b) shows this property. When data is found in a local L2 cache, the local L2 cache provides the data. If the data is not found in the local L2 cache (L2 cache miss), the assigned excess caches are searched to find the data. Because the same index bits are used during the search through multiple caches, each set's associativity is effectively increased. Figure 6(c) shows the two cases where data propagation from/to excess caches is needed. As a block would gradually move to the LRU position with the introduction of a new cache block to the same set, a block evicted from the local cache is entered into the next excess cache in the access chain.

4.2. Static sharing: Static allocation, Shared cache

Workloads may not have memory demand that matches cache bank granularity. For example, one workload may need half of the available excess cache capacity while another workload may need a little more capacity than one excess cache. With the static private scheme, some cores may waste excess cache capacity while other cores could use more. In this case, more performance could be extracted if

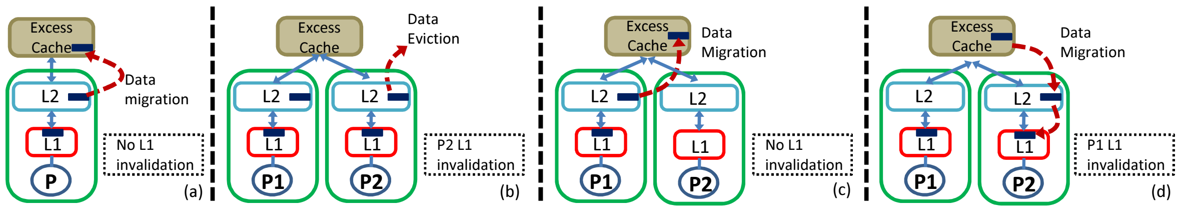


Figure 5. Coherency examples. (a) No L1 invalidation for data migration from L2 to exclusive excess cache. (b) L1 invalidation for data eviction. Data migration does not occur because P2 has the same data. (c) No L1 invalidation for data migration from L2 to shared excess cache. If no other core has the same data like (b), no L1 invalidation is needed because only P1 has valid L1 data. (d) L1 invalidation for data migration. If P2 migrates the data from the shared excess cache to the local L2 cache, P1's L1 data should be invalidated.

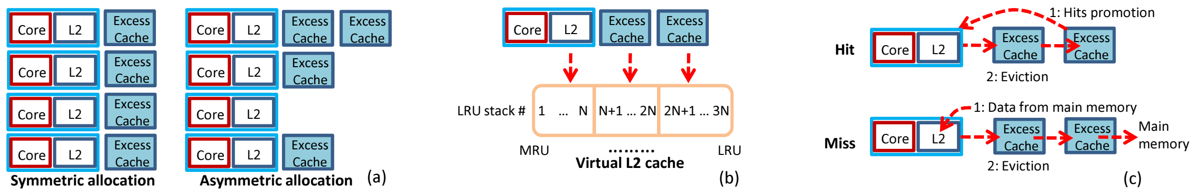


Figure 6. Static private scheme. (a) Two example allocations, symmetric and asymmetric. (b) $3N$-way virtual L2 cache with two N-way excess caches. (c) Data propagation in an excess cache chain during excess cache hit and miss. On a hit, (1) hit data is promoted from the hit excess cache to the local L2 cache; and (2) a block may be replaced from the local cache and propagated to the head of the excess cache chain, and so on. The propagation of a block may extend to the excess cache that previously hit the most, as it has space from promoting a hit block. On a miss, (1) data from the main memory is brought to the local L2; (2) a replaced block causes a cascading propagation from the local L2 cache through the excess cache chain; and (3) a block from the tail of the excess cache chain may move to main memory.

the available excess caches are shared between workloads to fully exploit the available excess cache capacity. The static sharing scheme uses the available excess caches as a shared resource for multiple cores as shown in Figure 7(a). The basic operation of the static sharing scheme is similar to the static private scheme except that the excess caches are accessible to *all* assigned cores. If applications on the cores have balanced memory demands, this scheme can maximize the total throughput. The excess caches can also be allocated "unevenly" to an application with a high demand. If other applications secure large benefits from not sharing with specific applications (i.e., due to interference), such an uneven allocation may prove desirable. Figure 7(b) shows an example in which core 3 has limited access to the excess cache. Core 0 can access two excess caches while core 3 can access only one excess cache. Core ID in the tag memory and the corresponding NECP in the cache controller are used to determine the next destination of the data block.

The static sharing scheme can be particularly effective for shared-memory multithreaded workloads because shared data do not have to be replicated in the excess caches (unlike the static private scheme). Furthermore, "balanced" multithreaded workloads typically have similar memory demands from multiple threads. In this case, the excess caches can be effectively shared by multiple threads in one application. If the initialization thread of a multithreaded workload heavily uses memory, then the static sharing scheme will work like the static private scheme because no other threads usually need cache capacity in the initialization phase.

4.3. Dynamic sharing: Dynamic partition, Shared cache

Static sharing has two potential limitations. It does not adapt to workload phase behavior, nor does it prevent wasteful usage of the excess cache capacity by an application that thrashes. While "capacity thrashing" applications do not benefit from excess caches, they can limit other applications' potential benefits. To overcome these limitations, we propose a dynamic sharing scheme where cache capacity demands from cores are continuously monitored and excess caches are allocated dynamically to maximize their utility.

Figure 8 illustrates how the dynamic sharing scheme operates. We employ "cache monitoring sets" (Figure 8(a)) that collect two key pieces of information, *flow-in counts* and *hit counts*. The counters at the monitoring sets count cache flow-ins and cache hits continuously during a "monitoring period" and are reset as the period expires and a new period starts. At the end of each monitoring period, a new excess cache allocation to use in the next period is determined based on the information collected during the current monitoring period (Figure 8(b)). We empirically find that 1M cycle period is good enough to determine excess cache allocation. The monitoring sets are accessed by all participating cores, while other non-monitoring sets are accessed by only the allocated cores. We find that having one monitoring set for every 32 sets works reasonably well.

To flexibly control excess cache allocation to individual cores, each core keeps an excess cache allocation counter. Figure 8(c) shows how these counters are set based on the ratio of flow-in and hit counts. We have four excess cache

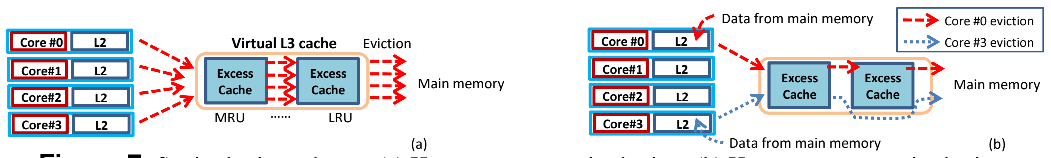


Figure 7. Static sharing scheme. (a) Homogeneous static sharing. (b) Heterogeneous static sharing.

allocation actions: *decrease*, *no action*, *increase*, or *maximize*. When a burst access occurs from a core ($\frac{\text{hits}}{\text{flow-ins}} > B_{th}$), all excess caches are allocated to the core to quickly adapt to the demanding application phase. This is the maximize action. The number of allocated excess caches to a core is decreased when its hit count is zero ($\frac{\text{hits}}{\text{flow-ins}} = 0$). The rationale for this case is that if the core has many data flow-ins, but most data sweep through the excess caches without producing hits, the core should not use the excess caches. A core gets one more excess cache if it proves to benefit from excess caches ($M_{th} < \frac{\text{hits}}{\text{flow-ins}} < B_{th}$); otherwise ($\frac{\text{hits}}{\text{flow-ins}} < M_{th}$) the core will keep what it has. We heuristically determine 12.5% for B_{th}, 3% for M_{th} in our evaluation.

5. Evaluation

5.1. Experimental setup

We evaluate the proposed StimulusCache policies with a detailed trace-driven CMP architecture simulator [6]. The parameters of the processor we model are given in Table 2. We select representative machine configurations to simulate: For an 8-core CMP, we simulated processor configurations with 4 functional cores and 1, 2, 3, or 4 excess caches. For a 32-core CMP, we simulated processors with 16 functional cores and 4, 8, 12, or 16 excess caches.

We choose twelve benchmarks from SPEC CPU2006 [27], four benchmarks from SPLASH-2 [32], and SPECjbb 2005 [27]. Our benchmark selection from SPEC CPU2006 is based on working set size [8]; we picked a range of working set sizes to comprehensively evaluate the proposed policies under various scenarios. For workload preparation, we analyzed L2 cache accesses for the whole execution period of each benchmark with the reference input. Based on the analysis, we extracted each benchmark's representative excess cache interval, which includes the program's main functionality but skips its initialization phases. To evaluate a multiprogrammed workload, we use various combinations of the single-threaded benchmarks. Tables 3(a) and (b) show the characteristics of the benchmarks selected and the multiprogrammed workloads. We simulate 10B cycles for single-threaded and multiprogrammed workloads. Other workloads (SPLASH-2 and SPECjbb 2005) are simulated for their whole execution.

5.2. Results

Single-threaded applications

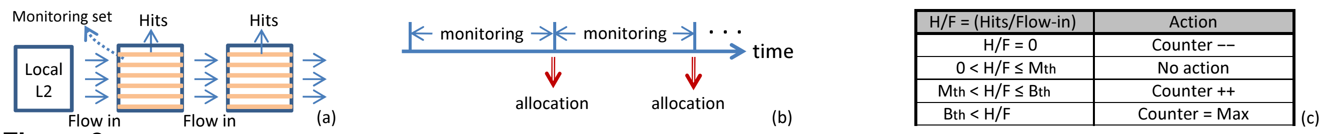
The static private scheme is used for the single-threaded programs and all available excess cache is allocate to the program. Figure 9(a) shows the performance improvement of single-threaded applications with excess caches. Five programs (`hmmer`, `h264ref`, `bzip2`, `astar`, and `soplex`) show more than 20% performance improvement while seven others had less improvement. Four heavy workloads (`gcc`, `mcf`, `milc`, and `GemsFDTD`) had almost no performance benefit from using excess caches. The different performance behavior can be interpreted from cache miss counts and cache miss reductions, shown in Figure 9(b) and (c), respectively. First, the four light workloads (`hmmer`, `h264ref`, `gamess`, and `gromacs`) have significant performance gains with excess caches because more cache capacity reduces a large portion of misses (42%–91%). However, their absolute miss counts are relatively small. In the case of `gamess`, the performance improvement was quite limited because it had almost no misses even without excess cache. Second, moderate integer workloads (`bzip2` and `astar`) have a pronounced benefit with excess cache due to their high absolute miss counts (4.4 and 11.9 per 1K instructions) and a good miss reduction of 44% and 55% each. Third, `soplex` sees a sizable performance gain with at least three excess caches. Figure 9(c) depicts the large miss reduction of `soplex` with four excess caches. It has a miss rate knee at around 2MB cache size (one local cache and three excess cache). Fourth, the heavy workloads (`gcc`, `mcf`, `milc`, and `GemsFDTD`) and one moderate workload (`sphinx3`) have little performance gain. The negligible miss reductions with excess cache explain this result. Our results clearly show that the static private scheme is in general very effective for improving individual program performance; we saw sizable performance gains and no performance degradation. However, there are programs that do not benefit from excess caches at all.

Core's pipeline	Intel's ATOM-like two-issue in-order pipeline with 16 stages at 2GHz
Branch predictor	Hybrid branch predictor (4K-entry gshare, 4K-entry per-address w/ 4K-entry selector), 6 cycle mis-prediction penalty
HW prefetch	Four stream prefetchers per core, 16 cache block prefetch distance, 2 prefetch degree; implementation follows [29]
On-chip network	Crossbar for 8-core CMP and 2D mesh for 32-core CMP at half the core's clock frequency
On-chip caches	32KB L1 I-/D- caches with a 1-cycle latency; 512KB unified L2 cache with a 10-cycle latency; all caches use LRU replacement and have 64B block size
Memory latency	300 cycles

Table 2. Baseline CMP configuration.

Multiprogrammed and multithreaded workloads

Static private scheme. Figure 10(a) shows the performance improvement of multiprogrammed and multithreaded work-



H/F = (Hits/Flow-in)	Action
H/F = 0	Counter ––
0 < H/F ≤ Mth	No action
Mth < H/F ≤ Bth	Counter ++
Bth < H/F	Counter = Max

Figure 8. Operation of the dynamic sharing scheme. (a) Excess caches have "monitoring sets" that track data flow-in and cache hit counts for each core. (b) The monitoring activity and excess cache allocation are done in accordance with a "monitoring period." When each monitoring period expires, the excess cache allocation to apply during the following monitoring period is determined. (c) Excess cache allocation counter calculation. It is done every excess cache allocation time. To provide large cache capacity for highly reused data quickly, the counter is set to the maximum value when high data reuse is detected.

Suite	Characteristics (working set)	Benchmarks (working set size in 1B instructions)

SPEC CPU2006 INT	light
	moderate
	heavy
		464.h264ref (5.5MB); 456.hmmer (2MB);
		473.astar (26MB); 401.bzip2 (24.4MB);
		429.mcf (680.8MB); 403.gcc (73MB);

SPEC CPU2006 FP	light
	moderate
	heavy
		435.gromacs (8.6MB); 416.gamess (1.3MB);
		450.soplex.2 (27.2MB); 483.sphinx3 (10.6MB);
		459.GemsFDTD (800MB); 433.milc (230.8MB);

SPLASH-2	Multithreaded	fmm; ocean; lu; cholesky;
SPECjbb 2005	Server workload	100,000 transactions;

Combination	Applications
LLLL	464.h264ref, 456.hmmer, 435.gromacs, 416.gamess
LLMM1	464.h264ref, 435.gromacs, 473.astar, 483.sphinx3
LLMM2	464.h264ref, 435.gromacs, 401.bzip2, 450.soplex
LLMM3	456.hmmer, 416.gamess, 473.astar, 483.sphinx3
LLMM4	456.hmmer, 416.gamess, 401.bzip2, 450.soplex
LLHH1	464.h264ref, 435.gromacs, 429.mcf, 459.GemsFDTS
LLHH2	464.h264ref, 435.gromacs, 403.gcc, 433.milc
LLHH3	456.hmmer, 416.gamess, 429.mcf, 459.GemsFDTS
LLHH4	456.hmmer, 416.gamess, 403.gcc, 433.milc
MMMM	473.astar, 401.bzip2, 450.soplex. 483.sphinx3
MMHH1	473.astar, 483.sphinx3, 429.mcf, 459.GemsFDTS
MMHH2	473.astar, 483.sphinx3, 403.gcc, 433.milc
MMHH3	401.bzip2, 450.soplex, 429.mcf, 459.GemsFDTS
MMHH4	401.bzip2, 450.soplex, 403.gcc, 433.milc
HHHH	429.mcf, 403.gcc, 459.GemsFDTS, 433.milc

Table 3. Benchmark selection (left) and Multiprogrammed workloads (right).

loads with the static private scheme. LLMM3 had the largest improvement of 17%. The performance improvement of individual applications in the multiprogrammed workloads are depicted in Figure 10(b). When there is a large difference between the improvements of individual programs in a workload, the workload's overall performance improvement is limited by the application with the smallest individual gain. As shown in the previous subsection, there are programs that do not benefit at all from the use of excess caches.

For the multithreaded workloads, the static private scheme brought a large performance improvement for lu (45%) and server (42%). Other benchmarks had a 10% to 15% performance improvement. lu has a miss rate knee just after a total 512KB cache size. Therefore, adding one excess cache to each core has a great performance benefit. server has a high L2 cache miss rate of over 40% and lends itself to a large improvement given more cache capacity with excess caches. The multithreaded workloads we examined have symmetric behaviors (threads have similar cache demands) and all of them benefit from more cache capacity using the static private scheme.

Static sharing scheme. The multiprogrammed and multithreaded workloads can benefit from excess caches by sharing the extra capacity from the excess cache. Figure 11(a) shows the performance improvement from employing a different number of excess caches with the static sharing scheme. The performance improvements of individual programs are shown in Figure 11(b). This figure presents the result when four excess caches were used.

For intuitive discussion, we categorize the multiprogrammed workloads into four groups. First, workloads in group 1 obtain significantly more benefits from the static sharing scheme than the static private scheme. They have at least two light programs and no heavy programs. Therefore, the programs in these workloads share excess cache capacity in a "fair" manner without thrashing. Second, workloads in group 2 exhibit limited relative performance benefit with the static sharing scheme compared to the static private scheme. In fact, the performance of LLHH1 and LLHH3 become worse with cache sharing. Performance degradation can be caused by the heavy programs that use up the entire excess cache capacity, sacrificing the performance improvement opportunities of co-scheduled, light programs. Third, workloads in group 3 show sizable performance gains from cache sharing because astar greatly benefits from more cache capacity. Figure 11(b) shows that astar has 135% performance improvement regardless of other co-scheduled programs. Fourth, workloads in group 4 have very small performance improvement from excess cache sharing. Clearly, simply sharing cache capacity without considering the program mix does not result in a performance improvement.

Multithreaded and server workloads have nearly identical performance improvement as the static private scheme. This result suggests that these workloads can readily exploit the given excess cache capacity with the simple static private and static sharing schemes because the threads have balanced cache capacity demands.

Dynamic sharing scheme. The dynamic sharing scheme has the potential to overcome the deficiency of the static sharing scheme, which does not avoid the destructive competition in some co-scheduled programs. Figure 12(a) shows the overall performance gain using the dynamic sharing scheme. As the dynamic sharing scheme is suited to situations when co-scheduled programs aggressively compete

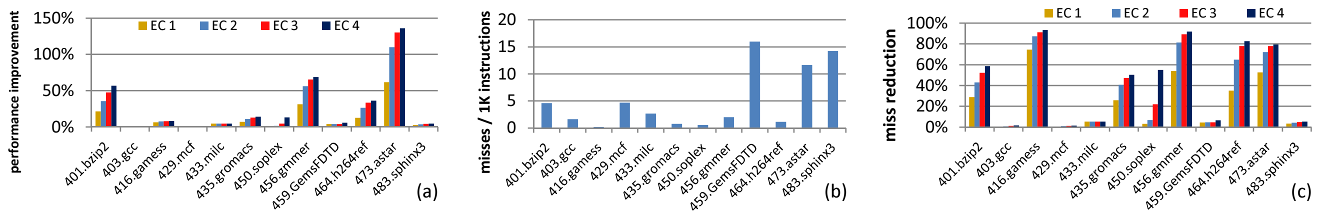


Figure 9. (a) Performance improvement of single-threaded applications. (b) Misses per 1,000 instructions. (c) Miss reduction.

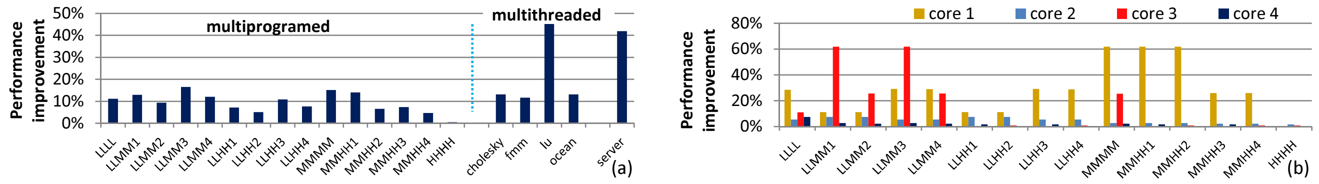


Figure 10. (a) Performance improvement with the static private scheme. (b) Performance improvement of individual programs.

with one another, our presentation focuses on only the multiprogrammed workloads.

The benefit of dynamic sharing is significant when there are heavy programs, especially for group 2 workloads in Figure 12(a) and Figure 11(a). Moreover, the relative benefit is pronounced with a smaller number of excess caches. For example, workloads with a variety of memory demands (e.g., `LLHH1`–`LLHH4` and `MMHH3`–`MMHH4`) gain large benefits from the dynamic sharing scheme with only one excess cache. Figure 12(b) presents the relative benefit of the dynamic sharing scheme to the static sharing scheme when four excess caches are given. The result shows that group 1 workloads have little additional performance gain because the static sharing scheme already achieves high performance in the absence of cache trashing programs. However, `LLMM2` and `LLMM4` still show measurable additional performance gain with the dynamic sharing scheme. Second, group 2 workloads have the highest additional performance improvement with dynamic sharing. All four workloads have at least one program which achieves an additional performance improvement of 5% or more (5.5%, 5.1%, 16.8%, and 16.2%). On the other hand, some programs actually suffer performance degradation because the dynamic sharing scheme strictly limited their use of excess cache capacity. However, the performance degradation is very limited—the largest performance degradation observed was only 0.6% (`milc` of `LLHH4`). Third, group 3 workloads show only a small additional performance gain as the large performance potential of adding more cache capacity has been already achieved with the static sharing scheme. However, there were noticeable additional gains for `MMHH1` and `MMHH2` which have heavy programs. Fourth, `bzip2` in group 4 has a large additional performance gain with the dynamic sharing scheme. The other programs in this group get negligible benefit.

The results demonstrate the capability of the proposed dynamic sharing scheme in StimulusCache; it can robustly improve the throughput of multiprogrammed workloads without unduly penalizing individual programs. The dynamic and adaptive control of excess cache resources allocation among competing co-scheduled programs is shown to be critical to get the most from the available excess caches.

Comparing StimulusCache with Dynamic Spill-Receive

To put StimulusCache in perspective, we compare it with a recently proposed dynamic spill-receive (DSR) scheme [21] which effectively utilizes multiple private caches among co-scheduled programs. Cooperative caching (CC) [5] and DSR are two representative private L2 cache schemes, which could be used to merge excess caches. We chose to compare StimulusCache with DSR because it has better performance than CC for many workloads [21].

Figure 13(a) presents the performance improvement with StimulusCache's three policies and DSR, given four excess caches. Overall, StimulusCache's dynamic sharing and static sharing schemes achieve substantially better performance than DSR. DSR shows the least performance improvement for quite a few workloads (`LLLL`, `LLMM3`, `LLMM4`, `LLHH4`, `MMHH2`, and `HHHH`). Only two workloads (`LLHH1` and `MMHH3`) have better performance improvement with DSR. Figure 13(b)–(d) show individual performance improvement in selected workloads. It is shown that programs like `hmmer`, `bzip2` (in `LLMM4`) and `astar` perform significantly better with StimulusCache than DSR. On the other hand, `soplex` in `MMHH3` performed better with DSR. Even in this workload, the three other programs in `MMHH3` perform better with StimulusCache.

DSR's relatively poor performance comes partly from the fact that it does not differentiate excess caches from other local L2 caches. Excess caches are strictly remote caches and are not directly associated with a particular core. Hence, an excess cache should be a "receiver" in the context of DSR. However, DSR's spiller-receiver assignment decision for each cache is skewed as there are no local cache hits or misses for the excess caches, and surprisingly, the excess caches become a "spiller" from time to time, which blocks their effective use as additional cache capacity. Furthermore, unlike the excess cache chain of the dynamic shar-

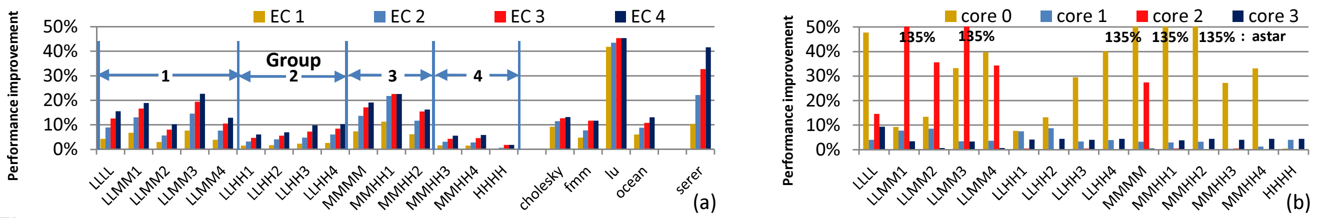


Figure 11. (a) Performance improvement with the static sharing scheme. Workloads are grouped into: Group 1: "Large gain," Group 2: "Limited gain due to heavy applications," Group 3: "Large gain due to `astar`," and Group 4: "Small gain." (b) Performance improvement of individual programs with four excess caches. `astar` consistently shows a high gain of 135%.

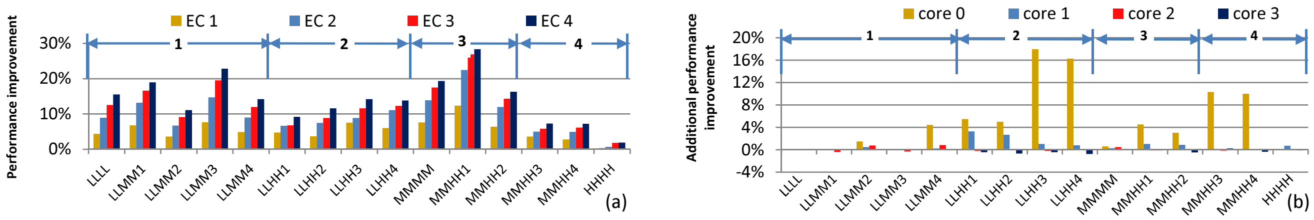


Figure 12. (a) Performance improvement with the dynamic sharing scheme. (b) Additional performance gain with the dynamic sharing scheme compared with the static sharing scheme with four excess caches. Grouping follows Figure 7(a).

ing scheme, a miss in one-level receiver caches in DSR is a global miss. DSR provides a much shallower LRU depth than StimulusCache. Therefore, even if we designate excess caches as receivers in DSR, it does not perform as well as the dynamic sharing scheme of StimulusCache.

Network traffic

Excess caches may introduce additional network traffic due to staggered cache access to multiple excess caches and downward block propagation. A single local cache miss can cause N data propagations from the local cache to the main memory with N excess caches. An excess cache hit generates K block propagations if the K'th excess cache had a hit. Our experiments revealed that StimulusCache does not increase the network traffic significantly. The average on-chip network bandwidth usage per core was measured to be 155.1MB/s (`cholesky`) to 517.3MB/s (`MMHH1`) without excess cache. With excess cache, the bandwidth usage was 187.5MB/s (`fmm`) to 873.7MB/s (`MMHH1`), well below the provided network bandwidth capacity of 8GB/s per core. The increase was 101.7MB/s on average and up to 423.2MB/s (`LLMM3`). The reduced execution times with StimulusCache also push up the network bandwidth usage.

Excess cache latency

In this section, we study how sensitive program performance is to excess cache access latency. Long excess cache latencies may result from slower on-chip networks, network contention, or non-uniform distances between the program location and the excess cache locations. Figure 14 shows the performance improvement of various workloads with excess caches having varied latencies. While the performance improvement decreases with an increase in latency, the overall performance improvement remains significant, even with the longest latency of 50 cycles.

The performance impact due to long excess cache latencies is limited because accesses hit more frequently in the local L2 cache and in the first few excess caches. The extent of the impact varies from workload to workload depending on how frequently an access has to travel further down the excess cache chain. Figure 14(b) and (c) further show that the performance impact varies from program to program within a single multiprogrammed workload. For example, in `LLMM4`, `hmmer` and `bzip2` were measurably affected by the increased excess cache latency. The other two programs in the workload were not. This result is intuitive because the programs that get more benefit from the excess caches could be impacted more from the increased latency.

32-core CMP

Finally, we experimented with a futuristic 32-core CMP configuration, where 16 cores run programs and there are 4–16 excess caches. We use the static sharing scheme for multithreaded and server workloads and the dynamic sharing scheme for multiprogrammed workloads. We use a multiprogrammed workload that encompasses all twelve SPEC2006 benchmarks listed in Table 3(a). Additionally, a second copy of the four heavy workloads are run on cores 13 to 16 to ensure that all 16 functional cores are kept busy. Figure 15(a) shows the overall performance improvement with excess cache.

We make two observations. First, the dynamic sharing scheme for the multiprogrammed workload works well for this large-scale CMP. Using 16 excess caches, the dynamic sharing scheme yields a performance improvement of 11% whereas the static sharing scheme's improvement is only 5%. The superiority of the dynamic sharing scheme is more clearly revealed in Figure 15(b) and (c). Second, the multi-threaded and server workloads also have large performance

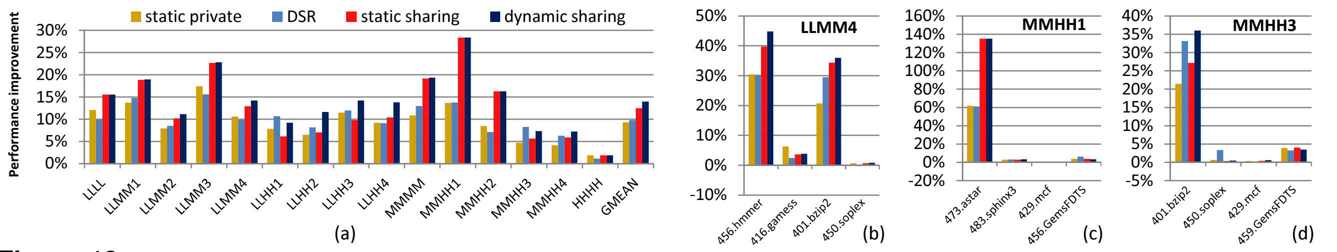


Figure 13. (a) Performance improvement with three StimulusCache policies and Dynamic Spill-Receive (DSR). (b)–(d) Performance improvement of individual programs in three example workloads: (b) `LLMM4` (DSR < static private < static sharing < dynamic sharing); (c) `MMHH1` (static private = DSR << static sharing = dynamic sharing); and (d) `MMHH3` (static private<static sharing<dynamic sharing<DSR).

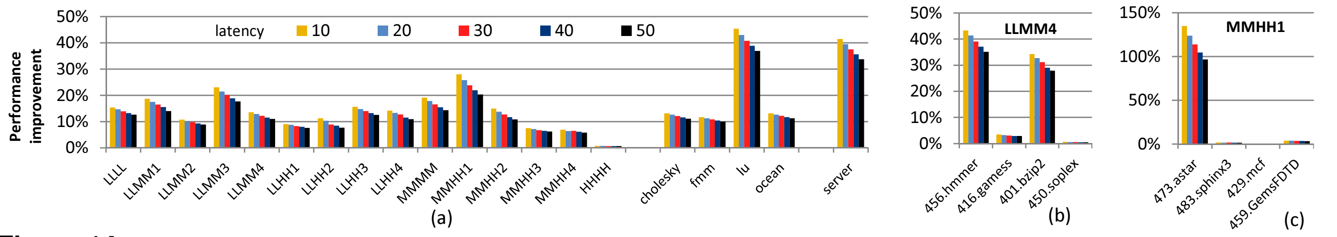


Figure 14. (a) Performance improvement when excess cache latency is varied. (b)–(c) Individual program's performance improvement for (b) `LLMM4` and (c) `MMHH1`.

improvements. `lu` and `server` have 54.2% and 90.5% improvement with only four excess caches, respectively. The performance improvement of `server` is as high as 104.4% with 16 excess caches. This result underscores the importance of on-chip memory optimization for memory-intensive workloads with large footprints.

6. Related Work

To the best of our knowledge, our work is the first to motivate and explore salvaging unemployed L2 caches in CMPs. In this section, we summarize two groups of related work: core salvaging and banked L2 cache management.

Core salvaging aims to revive a faulty processor core with help from hardware redundancy or software intervention. Joseph [14] proposed a VMM-based core salvaging technique. By decoupling physical processor cores from software-visible logical processors, fault management is done solely in the VMM without applications involvement. The main mechanisms for core salvaging are migration and emulation. A thread that is not adequately supported by a core (due to faults) can be transparently migrated to another core. Alternatively, lost hardware features due to faults (e.g., floating-point multiplier) can be emulated by software through a trap mechanism. This work evaluates how such strategies affect a salvaged core's performance. Powell et al. [20] also examine similar core salvaging techniques and demonstrate that the large thread migration penalty is amortized if a faulty resource is rarely used. Furthermore, they suggest an "asymmetric redundancy" scheme to mitigate the impact of losing frequently used resources. For instance, a simple bimodal branch predictor can augment a more complex main branch predictor. After a rigorous examination of core salvaging, they showed that the technique covers at most 21% of the core area. Detouring [18] is an all-software solution. Similar to the previously proposed emulation technique [14], it translates instructions which use faulty functional blocks into simpler instructions that do not need them. Although Detouring's reported coverage is 42.5% of the processor core's functional blocks, it uses binary translation and is subject to significant performance degradation.

If the faulty cores can be salvaged perfectly, it would obviate the need for core disabling and StimulusCache. However, given that the existing core salvaging techniques only (theoretically) cover a small portion of the core area and they incur area and performance overheads, we believe that core disabling will remain a dominant yield enhancement strategy. Note also that the proposed StimulusCache techniques can be opportunistically used when processor cores are put into deep sleep and their L2 caches become idle.

StimulusCache's dynamic sharing policy is related to the CMP cache partitioning. Suh et al. [28] proposed a way partitioning technique with an efficient monitoring mechanism and the notion of marginal gain. Qureshi and Patt [23] proposed the UMON sampling mechanism and lookahead partitioning to handle workloads with non-convex miss rate curves. Qureshi [21] extended UMON to enable private L2 caches to spill and receive cache blocks between a pair of L2 caches. Chang and Sohi [5] proposed Cooperative Caching (CC) and allow capacity sharing among private caches. With a central directory that has all cores' L1 and L2 tag contents, they migrate an evicted block to minimize the number of off-chip accesses. Our excess cache management approach

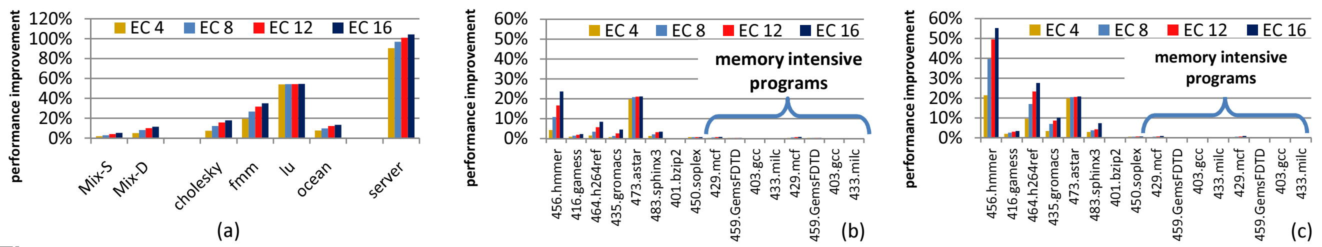


Figure 15. Performance improvement of 32-core CMPs with excess caches. (a) Overall throughput improvement. (b)–(c) Performance improvement of individual programs (b) with the static sharing scheme, and (c) with the dynamic sharing scheme.

is different from the previous work in that we control cache capacity sharing at bank granularity, and accordingly, the related overhead is small. Our mechanism also flexibly controls an individual core's cache access path.

7. Conclusion

Future CMPs are expected to have many processor cores and cache resources. Given higher integration and smaller device sizes, maintaining chip yield above a profitable level remains a challenge. As a result, various on-chip resource isolation strategies will gain increasing importance. This paper proposes StimulusCache where we decouple private L2 caches from their cores and salvage unemployed L2 caches when the corresponding cores become unavailable due to hardware faults. We explore how available excess caches can be used and develop effective excess cache utilization policies. For single-threaded programs, StimulusCache offers a sizable benefit by reducing up to 91% of L2 misses and increasing program performance by up to 131%. We find that our unique logical chaining of excess caches exposes an opportunity to control the usage of the shared excess caches among multiple co-scheduled programs.

References

[1] AMD Phenom Processors. `http://www.amd.com`.

[2] AMD. "BIOS and Kernel Developer's Guide for AMD Athlon64 and AMD Opteron Processors," `http://support.amd.com/us/Processor_TechDocs/26094.pdf`.

[3] S. Borkar. "Microarchitecture and Design Challenges for Gigascale Integration," keynote speech at MICRO, Dec. 2004.

[4] F. A. Bower et al. "Tolerating Hard Faults in Microprocessor Array Structure," *Proc. DSN*, Jul. 2004.

[5] J. Chang and G. S. Sohi. "Cooperative Caching for Chip Multiprocessors," *Proc. ISCA*, 2006.

[6] S. Cho, S. Demetriades, S. Evans, L. Jin, H. Lee, K. Lee, and M. Moeng. "TPTS: A Novel Framework for Very Fast Manycore Processor Architecture Simulation," *Proc. ICPP*, Sep. 2008.

[7] S. M. Domer et al. "Model for Yield and Manufacturing Prediction on VLSI Designs for Advanced Technologies, Mixed Circuitry, and Memory," *IEEE JSSC*, 30(3):286–294, Mar. 1995.

[8] D. Gove. "CPU2006 Working Set Size," *ACM SUGARS Computer Architecture News*, 35(1):90–96, Mar. 2007.

[9] IBM "IBM System p570 with new POWER6 processor increases bandwidth and capacity," *IBM United States Hardware Announcement*, pp. 107–288, May 2007.

[10] Intel ATOM Processors. `http://www.intel.com/technology/atom/`.

[11] Intel Corp. "Intel Microarchitecture, Codenamed Nehalem," technology brief, `http://www.intel.com/technology/architecture-silicon/next-gen/`.

[12] Intel Corp. "Mainframe reliability on industry-standard servers: Intel Itanium-based servers are changing the economics of mission-critical computing," white paper, `http://download.intel.com/products/processor/itanium/RAS_WPaper_Final_1207.pdf`, 2007.

[13] ITRS. *ITRS 2007 Edition Yield Enhancement*, 2007.

[14] R. Joseph. "Exploring Salvage Techniques for Multi-core Architectures," In *Workshop High Performance Computing Reliability Issues (HPCRI)*, Feb. 2005.

[15] H. Lee, S. Cho, and B. Childers. "Performance of Graceful Degradation for Cache Faults," *Proc. ISVLSI*, May 2007.

[16] H. Lee, S. Cho, and B. Childers. "Exploring the Interplay of Yield, Area, and Performance in Processor Caches," *Proc. ICCD*, Oct. 2007.

[17] M. R. Marty and M. D. Hill. "Virtual Hierarchies to Support Server Consolidation," *Proc. ISCA*, Jun. 2007.

[18] A. Meixner and D. J. Sorin. "Detouring: Translating Software to Circumvent Hard Faults in Simple Cores," *Proc. DSN*, Jun. 2008.

[19] NVIDIA GeForce 8800 GPU. `http://www.nvidia.com`.

[20] M. D. Powell et al. "Architectural Core Salvaging in a Multi-Core Processor for Hard-Error Tolerance," *Proc. ISCA*, Jun. 2009.

[21] M. K. Qureshi. "Adaptive Spill-Receive for Robust High-Performance Caching in CMPs," *Proc. HPCA*, Feb. 2009.

[22] M. K. Qureshi et al. "Adaptive Insertion Policies for High-Performance Caching," *Proc. ISCA*, Jun. 2007.

[23] M. K. Qureshi and Y. N. Patt. "Utility-Based Partitioning of Shared Caches," *Proc. MICRO*, Dec. 2006.

[24] SEMATECH. *Critical Reliability Challenges for ITRS*, *Technology Transfer #03024377A-TR*, Mar. 2003.

[25] S. Shankland. "Sun begins Sparc phase of server overhaul," `http://news.zdnet.com/2100-9584_22-145900.html`.

[26] E. Sperling. "Turn Down the Heat ... Please—Interview with Tom Reeves of IBM," *EDN*, July 2006.

[27] Standard Performance Evaluation Corporation. `http://www.specbench.org`.

[28] G. E. Suh et al. "A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning," *Proc. HPCA*, Feb. 2002.

[29] J. Tendler et al. "POWER4 system microarchitecture," *IBM Techical White Paper*, Oct. 2001.

[30] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. "CACTI 5.1 Technical report," HP Laboratories, Palo Alto, 2008.

[31] C. Webb. "45nm Design for manufacturing," Intel Technology Journal, 12(3):121–129, Nov. 2008.

[32] S. C. Woo et al. "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proc. ISCA*, July 1995.