Sanguthevar Rajasekaran, Lance Fiondella, Reda A. Ammar, Mohamed F. Ahmed

Multi-Core Technologies: Architectures, Algorithms, & Applications

Contents

Chapter 1

FSB: A Flexible Set Balancing Strategy for Last Level Caches

Mohammad Hammoud
Postdoctoral Research Associate
School of Computer Science (SCS)
Carnegie Mellon University Qatar

Sangyeun Cho
Associate Professor
Department of Computer Science
University of Pittsburgh

Rami Melhem
Professor
Department of Computer Science
University of Pittsburgh

This paper describes Flexible Set Balancing (FSB), a practical strategy for providing high-performance caching. Our work is motivated by large asymme-

try in cache sets' usages. FSB extends the lifetime of cache lines via retaining some fraction of the working set at underutilized sets to satisfy far-flung reuses. FSB promotes a very flexible sharing among cache sets, referred to as many-from-many sharing, providing significant reduction in interference misses. Simulation results using a full-system simulator which models a 16-way tiled chip multiprocessor platform demonstrate that FSB achieves an average miss rate reduction of 36.6% on multithreading and multiprogramming benchmarks from SPEC2006, PARSEC, and SPLASH-2 suites. This translates into an average execution time improvement of 13%. Furthermore, evaluations manifested the outperformance of FSB over some recent proposals including DSBC [27] and V-WAY [25].

1.1 Introduction

Processor and memory speeds are increasing at about 60% and 10% per year, respectively [15]. Besides, after the emergence of chip multiprocessors (CMPs) as a main stream architecture of choice, the off-chip bandwidth is expected to grow at a much slower rate than the number of processor cores on a CMP chip [7]. These factors together substantially increase the capacity pressure on the on-chip memory hierarchy, and, in particular, the last level cache (LLC). Intelligent design and management of LLC continue, accordingly, to be very essential to bridge the increasing speed and bandwidth gaps between processor and memory.

In this work we observe that more than two thirds of cache lines placed in an LLC logically shared by 16 CMP cores remain unused between placement and eviction. As such, these lines don't contribute to good utilization of the silicon estate devoted to the caches. One reason for this phenomenon is that cache lines might be re-referenced at distances greater than the cache associativity [23]. The problem is magnified on CMPs that share caches as on-chip lifetimes of cache lines can become shorter due to the increasing interferences between co-scheduled threads/processes. Cache performance can be improved by retaining some fraction of the working set long enough to provide cache hits on future reuses [23, 19].

Computer programs exhibit a non-uniform distribution of memory accesses across different cache sets [27, 25]. Fig. 1.1 demonstrates this fact by showing the number of misses experienced by cache sets at different physically distributed, logically shared L2 banks on a 16-way tiled CMP for two benchmarks, SpecJBB and MIX3[1]. Only the sets that exhibit the maximum and the minimum misses are shown. Clearly, some sets suffer from large local miss

[1]Description of the adopted CMP platform, the experimental parameters, and the benchmark programs can be found in Section 1.4.1.

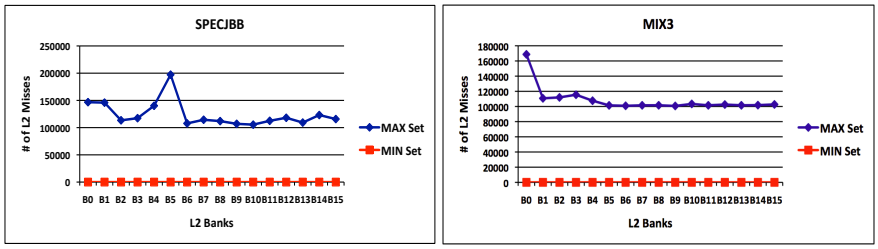

FIGURE 1.1: **Number of misses experienced by two cache sets at different L2 banks for SPECJBB and MIX3 (MAX Set = the set that experiences the maximum misses and MIN Set = the set that experiences the minimum misses).**

ratios while some others remain underutilized. Our work extends the lifetime of cache lines by exploiting this phenomenon via *flexibly* retaining cache lines evicted from highly pressured sets at underutilized ones.

Recently, Rolán *et al.* [27] proposed Dynamic Set Balancing Cache (DSBC) to mitigate the large asymmetry in cache sets' usages. DSBC suggests associating every two cache sets, making the capacity of an underutilized set available for a stressed one. When a cache line is evicted from a set whose working set seems not to fit in, the line can be stored at another set whose working set seems to fit in. They refer to the former and latter sets as *source* and *destination* sets, respectively. DSBC associates source and destination sets after a request made by a source set. The source and destination sets remain associated as long as the destination set hosts at least one line from the source set. Upon the eviction of the last line retained by the source set at the destination set, the association between the two sets is broken and the destination set (assuming still underutilized) can be subsequently shared by another source set.

There are inherent drawbacks with DSBC. Once an association is established between two sets, the source, S, is not allowed to retain blocks at any other set but the destination set, D. If the program phase changes after association and both S and D become stressed, they will compete on only D's capacity (i.e., retention is unidirectional), potentially causing significant thrashing at D. Besides, S is not permitted to request more capacity although many other underutilized sets might be available. Lastly, if D becomes underutilized, it is not allowed to share its space by more than one source set though it might be capable of doing so and many other sources might be in need of its available space.

We refer to the capacity sharing provided by DSBC as *one-from-one* sharing. We propose Flexible Set Balancing (FSB) in which a highly pressured set is allowed to retain its lines at many underutilized sets. We refer to this sharing as *many-from-one* sharing because the capacity of many sets can be shared by a single set. Furthermore, we allow many pressured sets to retain their lines

at a single underutilized set. This sharing is referred to as *one-from-many* sharing because the capacity of a single set can be shared by multiple sets. Consequently, FSB offers a very flexible (*many-from-many*) capacity sharing among cache sets. FSB adapts to phase changes in programs and doesn't solely associate any set with any other set (i.e., makes a set a sole owner of another set). As long as there is a space available at any set, any stressed set can immediately leverage that space.

The major contributions of our work are as follows:

- We propose Flexible Set Balancing (FSB), a caching strategy that extends the lifetime of cache lines via exploiting the phenomenon of workload imbalance among cache sets. FSB suggests a very flexible sharing between cache sets, referred to as *many-from-many* sharing, seeking to minimize interference misses and maximize system performance.
- We evaluate our work on a full system simulator which models a 16-way tiled CMP and find that FSB reduces interference cache misses of a baseline shared last level cache by an average of 36.6%. This translates into an average execution time improvement of 13%.
- We employ the two recent and closely related works, DSBC [27] and V-WAY [25], on a CMP platform and compare them against FSB. On average, FSB provides 27.2% and 29.2% miss reductions against DSBC and V-WAY, respectively.

The rest of the paper is organized as follows. A motivational study and background are given in Section 1.2. We detail FSB in Section 1.3. We evaluate FSB and some related designs in Section 1.4. Section 1.5 summarizes prior work and we conclude in Section 1.6.

1.2 Motivation and Background

1.2.1 Baseline Architecture

Our proposed scheme doesn't impose any limitation on employing any caching architecture. Without loss of generality, we assume a 16-way tiled CMP platform. Economic, manufacturing, and physical design considerations suggest tiled architectures (e.g., Tilera's Tile64 and Intel's Teraflops Research chip) which co-locate distributed cores with distributed cache banks in tiles communicating via a network on-chip (NoC) [14]. Each tile encompasses a core, private L1 caches (I/D), and an L2 cache bank. We assume block interleaved logically shared L2 cache banks. An in-cache directory coherence MESI-based protocol is employed [6, 13, 35]. Hence, each L2 cache line is associated with a bit vector indicating which cores had cached copies of that line in their L1 private caches. Lastly, we assume an LRU replacement policy.

1.2.2 A Caching Problem

To mitigate the high off-chip data access latency, the microprocessor industry has incorporated techniques such as deep cache hierarchies, large associative last level caches (LLC), and sophisticated data prefetchers. But even with these techniques, a significant number of cache lines still miss in LLC [3]. Evaluations of 10 benchmarks from Spec2006, PARSEC, and Splash-2 (Section 1.4.1 describes the benchmark programs) manifested that more than two thirds of the cache lines are never reused before getting evicted. A similar observation appeared in [23]. These cache lines are referred to as *zero reuse lines*.

Many reasons cause the occurrence of zero reuse lines at LLC. First, memory references exhibit locality and are not evenly distributed across cache sets. This skew reduces the effectiveness of a cache and results in storing a considerable number of lines that are less likely to be re-referenced before replacement [22]. Second, the access stream visible to LLC is filtered through the higher level(s) caches on the memory hierarchy. Third, some cache lines reveal no temporal locality. Fourth, many cache lines exhibit far-flung reuses. That is, an evicted block might probably be used many times in the future, although not in the near future [8]. Fifth, the advent of CMPs exacerbates the problem due to interferences among co-scheduled threads/processes on an underlying shared LLC. Recent research work on CMP cache management has recognized the importance of the shared CMP design [29, 13, 18, 35, 10]. Furthermore, many of today's multi-core processors, the Intel Core^{TM}2 Duo processor family [26], Sun Niagara [21], and IBM Power5 [28], feature shared caches. We conducted a quantification study on different kinds of misses (i.e., compulsory, intra-processor, and inter-processor) on our adopted CMP model and found that 69.5% of misses are inter-processor (i.e., lines are replaced at earlier times by different processors).

1.2.3 Dynamic Set Balancing Cache and Inherent Shortcomings

Cache sets' usages are typically asymmetric [27, 25]. An intuitive solution to the zero reuse lines phenomenon is to extend the lifetime of some cache lines long enough so that at least a portion of these lines can provide cache hits on future reuses. Dynamic Set Balancing Cache (DSBC) [27] extends the lifetime of some cache lines by exploiting the asymmetry in cache sets' usages. Specifically, lines are shifted from sets with high local miss rates to sets with low local miss rates where they can be found later. Once a set reaches a saturation level (set's miss rate hits a maximum value of $2K - 1$ where K is the associativity of the cache) it requests a free (not associated yet) underutilized set. If such a set is found, both sets, the highly pressured one (or the *source*) and the underutilized one (or the *destination*), are associated.

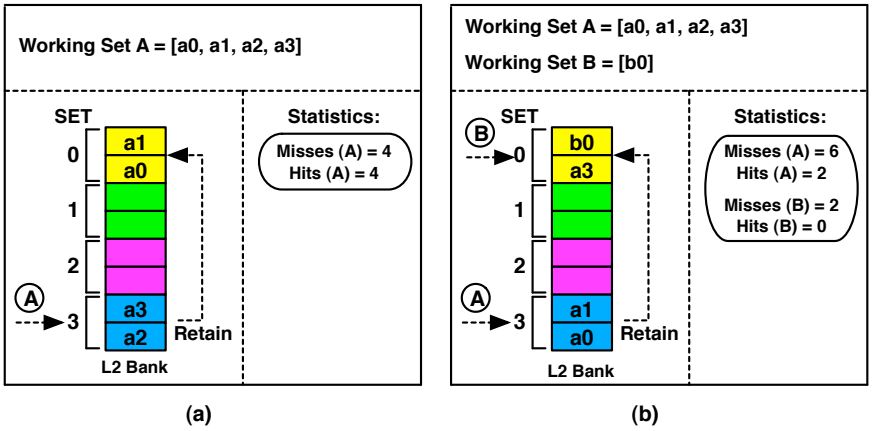

FIGURE 1.2: **DSBC in operation. (a) A maps originally to set 3. The program executes A's references in the order of A, A. DSBC is able to save much of A's interference misses. (b) A and B map originally to sets 3 and 0, respectively. The program executes A's and B's references in the order of A, B, A, B. DSBC is incapable of adapting to the phase change in the program.**

As long as the two sets are associated, the source is allowed to retain its lines at the destination but not the reverse (i.e., unidirectional retention).

DSBC maintains a table with one entry per set called the Association Table (AT). AT stores in the i-th entry AT(i).index which corresponds to the index of the set associated with set i. Besides, AT stores a source/destination (s/d) bit (AT(i).s/d) that indicates whether the set is associated or not. Each AT entry can have three different values. First, if a cache set is not associated, its corresponding AT entry stores the set's index and s/d = 0. Second, if a set is a source set, its corresponding AT entry stores the destination index and s/d = 1. Lastly, if a set is a destination set, AT stores the source index and s/d = 0. When a certain request misses at a source set, the destination set is looked up for either a secondary hit or a definitive miss.

DSBC has a number of shortcomings. First, once a destination set, D, is designated, it will continue receiving retained lines from a source set, S, until the association is broken. This overlooks the fact that D's pressure progressively increases while receiving more lines from S. Nevertheless, after association a new program phase can start where S might remain pressured (and still associated with D) and D becomes highly pressured (due to receiving lines not only from S but further from a new large working set which maps now to it). As a result, S and D render competing on only D's resources causing significant thrashing. We illustrate this problem in an example.

Consider a 2-way set associative cache shown in Fig. 1.2. For simplicity we represent a cache by a linear array consisting of only 4 sets. Assume first

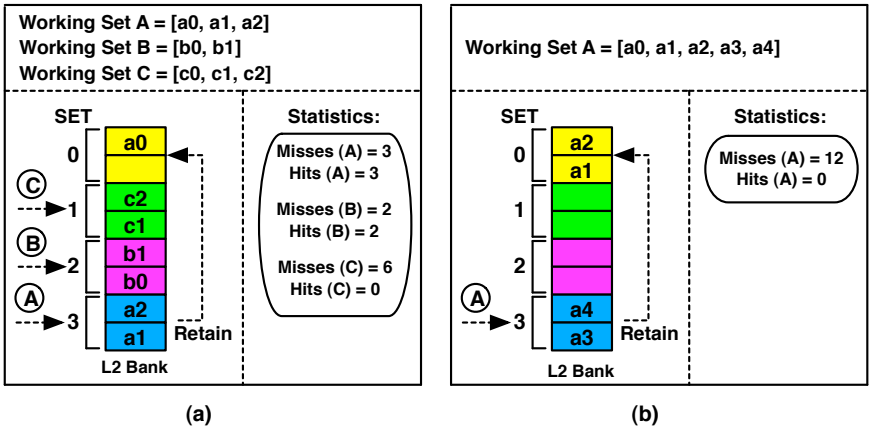

FIGURE 1.3: **DSBC in operation. (a) The program executes A's, B's, and C's references in the order of A, B, C, A, B, C. DSBC doesn't allow one-from-many sharing. (b) The program executes A's references twice. DSBC doesn't allow many-from-one sharing.**

(Fig. 1.2(a)) that a working set A with reference pattern $[a0, a1, a2, a3]$ maps to set 3 and has been observed twice by a program. The sequence of references of A can't co-reside in set 3. Accordingly, DSBC selects an underutilized set, say set 0, in the cache and displaces the evicted blocks from set 3 to set 0. Fig. 1.2(a) shows the final residences of lines in the cache after the completion of the program. A's resultant misses and hits are, consequently, 4 and 4, respectively (the cache is assumed to be initially empty). If the traditional caching strategy is to be followed, 4 more misses will be incurred.

In Fig. 1.2(b), presumably at a different phase in the program, a new working set B with reference pattern $[b0]$ is considered and assumed to map to set 0. As in Fig. 1.2(a), working set A still maps originally to set 3 and acts as a source set associated with set 0 as a destination set. We assume that the program executes A's and B's references in the order of A, B, A, B. The figure shows the final residences of lines after the completion of the program. A's resultant misses and hits are, consequently, 6 and 2, respectively. B, on the other hand, experienced 2 misses and got no hits. Note that DSBC didn't even attempt to break the association between sets 0 and 3 during the program's execution because there was always at least one retained block at set 0. *If DSBC would rather adapt to the phase change in the program, during the first execution of B's references, the evicted blocks from set 0 (i.e., $a0$) can be retained at another underutilized set (say set 1) so that in the second execution of A's and B's references no misses will be incurred.*

We refer to the sharing policy employed by DSBC among cache sets as *one-from-one* sharing. That is, a destination set is shared by only a single

source set. Fig. 1.3(a) shows three working sets A, B, and C with reference patterns $[a0, a1, a2]$, $[b0, b1]$, and $[c0, c1, c2]$, respectively. We assume that A, B, and C map originally to sets 3, 2, and 1, respectively. The figure demonstrates two issuances of A's, B's, and C's reference patterns in the order of A, B, C, A, B, C. A's lines can't all co-reside in set 3 and DSBC selects set 0 as a destination set for set 3. Also, C's lines can't all co-exist in set 1. However, DSBC doesn't select any destination set for set 1 because no set that is both underutilized and not associated yet is found. As a result, C's references will experience *zero* hits during their two issuances (with this cache topology C is said to experience far-flung reuses). The cache depicts the final residences of all the cache lines after the completion of the program. The misses and hits counts of A are 3 and 3, respectively. On the other hand, B's references miss twice and hit twice. Lastly, C's references miss 6 times and get no hits. *If DSBC would allow set 0 to be shared by both sets, 3 and 1, C's misses will be avoided when issued on the second time. We refer to this kind of flexible sharing as* ***one-from-many*** *sharing. That is, a single destination set can be shared by multiple source sets.*

Finally, as a consequence of the adopted one-from-one sharing strategy, DSBC doesn't allow a source set S to retain blocks in more than one destination set D. As such, if the working set that maps to S is large enough that both S and D are incapable of providing enough capacity as required, many conflict misses can be incurred. Fig. 1.3(b) assumes a working set A with reference pattern $[a0, a1, a2, a3, a4]$ that maps to set 3. The program issues A's references twice. DSBC selects an underutilized set; say set 0, where evicted lines from set 3 can be retained. The cache in the figure depicts the final residences of A's lines after the completion of the program. The final misses and hits counts are 12 and 0, respectively (assuming that the cache was initially empty). In this case, DSBC didn't provide any benefit for A. *If DSBC would allow more than one destination set to be shared by set 3; A's misses will be avoided when issued on the second time. We refer to this kind of flexible sharing as* ***many-from-one*** *sharing. That is, many destination sets can be shared by a single source set.*

1.2.4 Our Solution

We propose Flexible Set Balancing (FSB), a caching strategy that adapts to phase changes in programs and allows *many-from-many* sharing among cache sets. The difference in this work compared to DSBC are two key insights: (1) retention should be efficiently and dynamically allowed at any point during the program's execution in any direction seeking for spare space to effectively minimize interference misses, (2) one-from-many and many-from-one (i.e., many-from-many) sharing should be allowed among cache sets for high flexibility. We demonstrate our solution with an example.

Fig. 1.4(a) demonstrates the same example shown in Fig. 1.2(b) but with FSB being incorporated instead of DSBC. Again, A and B are assumed to map

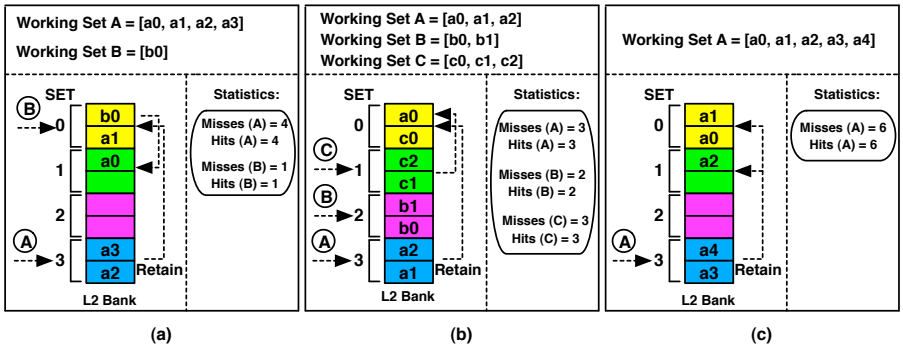

FIGURE 1.4: **Our solution. (a) The program executes A's, and B's references in the order of A, B, A, B. We adapt to the phase change in the program. (b) The program executes A's, B's, and C's references in the order of A, B, C, A, B, C. We allow one-from-many sharing. (c) The program executes A's references twice. We allow many-from-one sharing.**

to sets 3 and 0, respectively. The program executes A's and B's references in the order of A, B, A, B. In the first issuance of A's references, FSB selects set 0 as a destination set for set 3. Afterwards, when B is issued, line $a0$, which has been already retained at set 0, is evicted again. FSB doesn't discard $a0$ but yet retain it again at a new underutilized set, say set 1. In the second issuance of the working sets, A's and B's references hit on all their cache lines. As such, misses and hits outcomes become 4 and 4 for A, and 1 and 1 for B. Therefore, FSB saves 3 misses as compared to DSBC. The figure shows the final residences of all the cache lines after the program's completion. Clearly, this example illustrates FSB's capability to adapt to phase changes.

Fig. 1.4(b) shows the same example illustrated in Fig. 1.3(a). Again, we assume that A, B, and C map to sets 3, 2, and 1, respectively and that the program observes A's, B's, and C's references in the order of A, B, C, A, B, C. FSB allows set 0 to be shared by many source sets. As such, in the first iteration of the working sets when lines of C can't all co-reside in set 1, FSB retains $c0$ at set 0 (the current least pressured set). In the second iteration, the references of A, B, and C hit on all their cache lines. Misses and hits outcomes become, accordingly, 3 and 3 for A, 2 and 2 for B, and 3 and 3 for C. As compared to DSBC, FSB saves the three misses incurred by C in Fig. 1.3(a). The cache array in the figure displays the final residences of all the lines after the program's completion. Clearly, this example demonstrates FSB's efficiency in reducing conflict misses with one-from-many sharing.

Lastly, Fig. 1.4(c) illustrates the same example demonstrated in Fig. 1.3(b). Again, we assume that A maps to set 3 and that the program issues A's sequence of references twice. FSB allows many sets to be shared by a source set. As such, in the first issuance of A, FSB selects an underutilized

set, say set 0, and retains $a1$ and $a0$ at, then selects another underutilized set, say set 1, and retains $a2$ at. In the second issuance, all references of A hit in the cache. FSB, consequently, saves the 6 misses incurred by DSBC in Fig. 1.3(b). Clearly, this example magnifies the potential of FSB in reducing interference misses by employing many-from-one sharing among cache sets.

1.3 Flexible Set Balancing (FSB)

Flexible Set Balancing (FSB) regulates cache allocation by flexibly retaining a fraction of a working set at underutilized cache sets to minimize interference misses and maximize system performance. FSB is extensible and practical in that it can be employed on single-core as well as multi-core architectures. FSB is oriented towards last level caches (in our case L2). FSB requires three main capabilities: (1) deciding upon source and destination sets, (2) retaining working sets of source sets at destination sets in a many-from-many sharing fashion, and (3) locating retained blocks on destination sets when requested. We next describe each capability in turn and close with an analysis on FSB's hardware storage, area, energy, and latency requirements.

1.3.1 Retention Limits

FSB is a pressure-aware strategy where lines evicted from highly pressured sets (source sets) are retained at low pressured sets (destination sets). The pressure at a cache set can be measured in terms of cache misses or hits. In this work we adopt cache misses as a pressure function but provide in Section 1.4 a study on a variety of pressure functions. The pressure information can be recorded at an array embedded within the L2 controller of a cache bank. Each cache set corresponds to an entry in the pressure array and the indexes of the cache sets are used to index the array. Each time a miss occurs at a certain set, the array can be updated accordingly (by incrementing the corresponding array slot). In order to allow the array to accurately represent pressures at sets, after every time interval, we keep only part of the pressure values (e.g., 0.25 of values by shifting each value 2 bits to the right). That permits FSB to adapt to undergoing phase changes in programs. The collected pressures can be utilized to guide the retention process.

Clearly, the set that corresponds to the maximum value in the pressure array is the most highly pressured set. In contrast, the lowest pressured set is the one that corresponds to the minimum value in the array. In this work we define two limits, the low pressure limit (LPL) and the high pressure limit (HPL), to allow a *range* of highly pressured sets to retain their blocks at a range of low pressured sets. A range can encompass one set or many. When the pressure of a set is below LPL, the set is deemed to be within the limit

of the destination sets and can receive lines from *any* source set. In contrast, when the pressure of a set is above HPL, the set is considered to be within the limit of source sets and is permitted, accordingly, to retain its lines at *multiple* destinations sets. Clearly, this allows many-from-many sharing among cache sets. LPL and HPL are defined in equations (1) and (2). The range of source and destination sets can be expanded or contracted by altering α. The max and min parameters are the maximum and minimum pressures on the pressure array.

$$LowPressureLimit(LPL) = min + (\alpha \times (max \text{ - } min)) \quad \textbf{(1)}$$

$$HighPressureLimit(HPL) = max - (\alpha \times (max \text{ - } min)) \quad \textbf{(2)}$$

1.3.2 Retention Policy

FSB maintains a small retention table (RT) per each L2 bank. Each cache set has a corresponding RT entry. As such, the number of entries in RT equals the number of cache sets in the L2 bank. RT can store in the i-th entry *many* RT(i).index values, each pointing to a destination set with a different index. In Section 1.4, we empirically show that four RT(i).index pointers are enough to attain an efficient FSB. RT(i).index pointers can be used by FSB to locate retained blocks upon future reuses (more on this shortly).

When an LRU line, L, is evicted from a set i, our retention policy proceeds as follows:

1. We look up i's corresponding pressure value in the pressure array, generate minimum (MIN) and maximum (MAX) values, and calculate HPL and LPL.
2. If i's pressure is greater than HPL, i becomes a source set and L is deemed eligible for retention. Otherwise, L is discarded.
3. In parallel, RT(i) entry is looked up. If L is eligible for retention and RT(i) entry has no pointers to destination sets, we check if MIN is less than LPL. If satisfied, we retain L at the cache set corresponding to MIN and create an equivalent RT(i).index pointer. Otherwise, we discard L.
4. If RT(i) entry, on the other hand, has pointers (or at least one pointer), we use these pointers to index the pressure array, generate the minimum value out of the indexed values, and compare it against LPL. If satisfied, we retain L at the corresponding cache set and no RT(i).index pointer is created. Otherwise, we check if an invalid RT(i).index exists.
5. If an invalid RT(i).index is found and MIN satisfies LPL, we retain L at the corresponding set and create an equivalent RT(i).index pointer. Otherwise, we discard L.

Note that upon retention, we insert L as the most recently used (MRU) line in the selected destination set. The LRU line evicted at the destination set, to make room for L, is discarded simply because the destination set doesn't satisfy HPL. As such, FSB avoids ripple effects.

The LRU evicted line, L, at the source set can be either *native* or *retained*.

If L is native, FSB simply proceeds with our previously suggested retention policy. Otherwise, we check if L is *active*. We define L to be active if at least one core on the CMP platform had cached a copy of L (in its L1). This can be easily determined from L's associated directory bit vector. We assume that an active L is currently in use by the caching core(s) and, accordingly, attempt to retain it again. If L is retained and not active, we assume that it has been kept long enough in the cache without providing a cache hit, and, as such, avoid retaining it over again (although it is eligible for retention).

The pressure array is updated not only at a miss/hit but further when retaining a line at a destination set. When a destination set receives a retained line, its corresponding pressure value is incremented. This is critical so as to reflect the progressive increasing pressure on a destination set each time it receives a retained line. This makes FSB very flexible and attentive as it allows selecting a different destination set once the pressure of the current destination set surpasses LPL.

Retaining cache lines at destination sets requires extending lines' tags. This is due to the fact that a cache line must have a one-to-one correspondence with a unique address. For instance, assume a line E is retained at a destination set S and that S has a line F which has an identical tag field as E. E and F addresses are, in fact, only distinct because they differ in their index fields. Now E and F co-reside at S and thus become indistinguishable. Nevertheless, this suggests a simple solution. That is, augmenting each line's tag with the index field. Finally, upon discarding a retained line, R, from S we match R's augmented index j with the augmented indexes of S's resident lines. A "no match" outcome means that R is the last retained line at S from the source set j. Consequently, we index RT(j) entry and invalidate the RT(j).index pointer that points to S.

To that end, we note that the retention process is activated in parallel with the resolution of a definitive miss. As such, the latency required to retain a cache block becomes completely hidden as resolving an L2 miss usually takes hundreds of cycles. However, in principle, FSB's retention policy would require hardware to compute MAX, MIN, HPL, and LPL, which may appear expensive. Smart implementation strategies exist. We need not, for instance, compute upon every eviction the exact MAX and MIN values. The L2 misses are usually infrequent and the MAX and MIN values don't henceforth vary much upon a single L2 miss. As an approximation, we can compute MAX and MIN after a reasonable amount of misses (e.g., 1K L2 misses) and perform partial comparisons incrementally. In another design, one could employ comparators combined with multiplexors in a tree structure as adopted in [27]. Lastly, α can be simply set to 0.25 (i.e., power of 2) (corroborated by the sensitivity study presented in Section 1.4.4). With this setting, a multiplier is not needed to compute HPL and LPL but a simple shifter. Section 1.3.4 describes FSB's storage, area, energy, and latency requirements.

Component	Bits Per Entry	K Entries	KB Per Tile
RT Entry	10	2	2.5
Augmented Bits Per an L2 Line	9	8	9.2
Total KBytes			11.7
% Increase of On-Chip Cache Capacity			2%

TABLE 1.1: **FSB storage overhead.**

1.3.3 Lookup Policy

Upon a request to a cache line, L, the cache starts always looking up the set i that L's index designates. RT(i) entry is also looked up concurrently. If a hit occurs at set i, the request is satisfied and the pressure array is updated (only if the pressure function involves hits). If, on the other hand, a miss occurs at set i, the cache sets identified by the RT(i).index pointers (if any) are *serially* looked up until either a secondary hit is acquired or a definitive miss is proclaimed. Sets' lookups are serialized in order to keep FSB simple, avoid port contention, and reduce power dissipation[2]. Section 1.4 demonstrates that such a serial policy doesn't hurt performance because the gain from hits on retained lines exceedingly offsets the loss from sequential lookups. Upon a secondary hit, the request is satisfied and the pressure array is updated (only if the pressure function involves hits). If a definitive miss is asserted, the pressure array is updated at slot i (if the pressure function involves misses). On a definitive miss, the retention policy is triggered and, in parallel, the requested cache line is fetched from the main memory and inserted in set i.

FSB doesn't swap retained lines upon hits to return them to their original sets for several reasons. First, this simplifies management. Second, FSB is oriented towards last level caches; once a hit is obtained on a retained line, the line is moved to the upper cache where successive accesses can find it. Third, swapping is undesirable because it requires four accesses to the tag-store, consumes energy, and increases port contention [25].

1.3.4 FSB Cost

FSB comes at a little storage, area, latency, and energy overheads. In this work we assume a 32 KB 2-way associative I/D L1 caches and a 512KB 16-way associative L2 bank (512 cache sets) per each CMP tile. Section 1.4 shows that 4 pointers per each RT entry are enough for an effectively performing FSB. Each RT pointer requires 10 bits (1 valid bit and 9 bits to index the 512 L2 sets). Table 1.1 shows that $\sim$2% storage overhead is required by FSB.

To model area and energy we use CACTI v5.3 [16]. We assume a 45nm technology. Table 1.2 demonstrates the area and energy per access required for

[2]Prior research has made use of serialization to increase flexibility and improve performance in large caches [9, 12]. Existing processors have also adopted serialization for looking up tag and data arrays seeking to reduce power dissipation [11, 32].

TECHNOLOGY	BASELINE Energy	FSB Energy	BASELINE Area	FSB Area
45nm	1.23nJ	1.26nJ	5.36mm^2	5.47mm^2

TABLE 1.2: **Baseline and FSB required energy and area in a 512KB/16-way/64B/LRU L2 bank.**

COMPONENT	PARAMETER
Cache Line Size	64 B
L1 I/D-Cache Size/Associativity	32KB/2way
L1 Hit Latency	1 cycle
L1 Replacement Policy	LRU
L2 Cache Size/Associativity	512KB per L2 bank or 8MB aggregate/16way
L2 Bank Access Penalty	12 cycles
L2 Replacement Policy	LRU
Latency Per NoC Hop	3 cycles
Memory Latency	320 cycles

TABLE 1.3: **System parameters**

both a baseline L2 bank and an L2 bank with FSB being incorporated. The TR table, in addition, requires 0.14 mm^2 and 0.015 nJ area and energy per access, respectively. Note that the energy savings due to reducing off-chip accesses is not considered. Such savings are expected, in fact, to counterbalance our calculated energy overhead and further provide advantages as chip crossings are one of the greediest energy consumers [17]. Finally, and due to augmenting lines' tags by indexes, FSB incurs a negligible increase in latency (only 0.02 ns) per each L2 bank access.

NAME	INPUT
SPECJbb	Java HotSpot (TM) server VM v 1.5, 4 warehouses
Bodytrack	4 frames and 1K particles (16 threads)
Fluidanimate	5 frames and 300K particles (16 threads)
Swaptions	64 swaptions and 20K simulations (16 threads)
Barnes	64K particles (16 threads)
Lu	2048×2048 matrix (16 threads)
MIX1	Hmmer (reference) (16 copies)
MIX2	Sphinx (reference) (16 copies)
MIX3	Barnes, Ocean(1026×1026 grid), Radix (3M Int), Lu, Milc (ref), Mcf (ref), Bzip2 (ref), and Hmmer (2 threads/copies each)
MIX4	Barnes, FFT (4M complex numbers), Lu, and Radix (4 threads each)

TABLE 1.4: **Benchmark programs**

1.4 Quantitative Evaluation

1.4.1 Methodology

We present our results based on detailed full-system simulation using Virtutech's Simics 3.0.29 [31]. We use our own CMP cache modules fully developed in-house. We implement the XY-routing algorithm and accurately model congestion for both coherence and data messages. A tiled CMP architecture comprised of 16 UltraSPARC-III Cu processors is simulated running with Solaris 10 OS. Each processor uses an in-order core model with an issue width of 2. The tiles are organized as a 4×4 grid connected by a 2D mesh NoC. Each tile encompasses a switch, 32KB I/D L1 caches, and a 512KB L2 cache bank. A distributed MESI-based directory protocol is employed. After every 20 million instructions, we keep only 0.25 of the pressure values (see Section 1.3.1). Table 1.3 shows our configuration's experimental parameters.

We use a mixture of multithreaded and multiprogramming workloads to study FSB and related designs. For multithreaded workloads we use the commercial benchmark SpecJBB [30], five shared memory programs from the SPLASH-2 suite [33] (Ocean, Barnes, Lu, Radix, and FFT), and three applications from the PARSEC suite [4] (Bodytrack, Fluidanimate, and Swaptions). We composed multiprogramming workloads using the considered SPLASH-2 benchmarks and five other applications from SPEC2006 [30] (Hmmer, Sphinx, Milc, Mcf, and Bzip2). Table 1.4 shows the data sets and other important features of the simulated workloads. Lastly, the programs are fast forwarded to get past of their initialization phases. After various warm-up periods, each SPLASH-2 and PARSEC benchmark is run until the completion of its main loop, and each of SpecJBB, MIX1, MIX2, MIX3, and MIX4 is run for 8 billion user instructions.

1.4.2 Comparing FSB against Shared Baseline

Let us first compare FSB against the baseline shared (S) scheme. Fig. 1.5 (a) shows the L2 miss rates of S and four FSB configurations normalized to S. We denote FSB with retention tables (RT) storing 1, 2, 4, and 8 RT(i).index pointers per each entry i as FSB-1, FSB-2, FSB-4, and FSB-8, respectively. Furthermore, we assume a low pressure limit (LPL) and a high pressure limit (HPL) each with $\alpha = 0.2$. Section 1.4.4 offers a sensitivity study on different α values. We adopt cache misses as a pressure function but Section 1.4.3 provides a study on a variety of other functions. The figure demonstrates that as the number of pointers per an RT entry increases, FSB achieves higher L2 miss rate reductions. This behavior is apparent on all the examined benchmark programs. FSB centers around the flexible many-from-many sharing policy. More pointers indicate more exploitation to the many-from-many sharing strategy

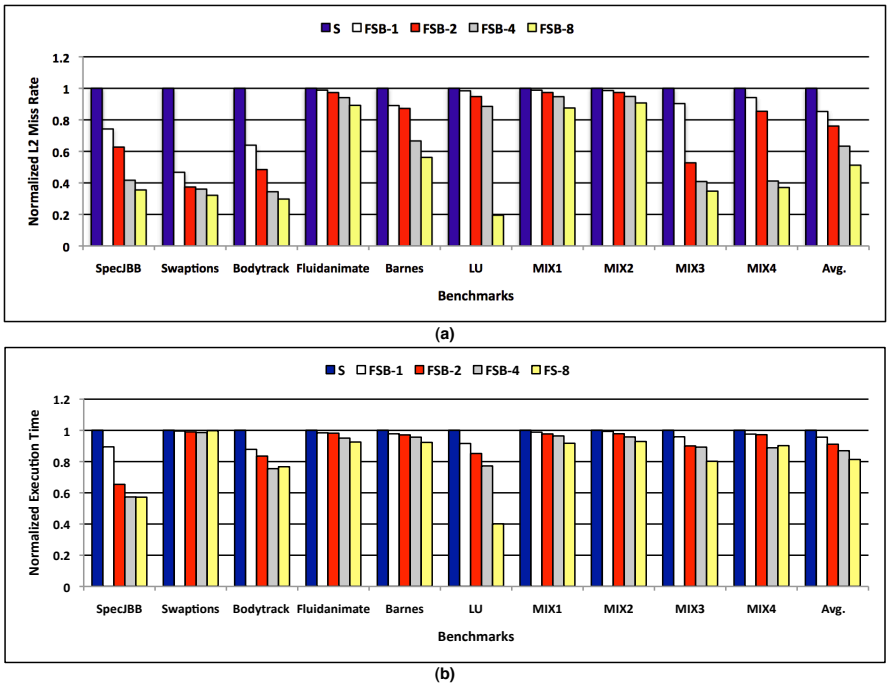

FIGURE 1.5: **L2 miss rates and execution times of the baseline shared scheme (S), FSB-1, FSB-2, FSB-4, and FSB-8 (all normalized to S).**

and, consequently, more alleviation to the imbalance across sets. On average, FSB-1, FSB-2, FSB-4, and FSB-8 accomplish average miss rate reductions of 14.6%, 23.9%, 36.6%, and 48.7%, respectively.

FSB strategy adopts a serial lookup policy (see Section 1.3.3 for more details). Upon a miss on the original set i, RT(i).index pointers (if any) are utilized to serially index and lookup corresponding L2 cache sets. Only the tag-stores are looked up until either a secondary hit is obtained or a definitive miss is asserted. Each tag-store access takes less than 0.68 ns, estimated by CACTI v5.3 [16] assuming a 45nm technology. This incurs a higher latency per each L2 access that misses at the original set. As such, although more RT(i).index pointers result in more L2 miss rate reductions, a latency cost is to be paid. Fig. 1.5 presents the execution times of S, FSB-1, FSB-2, FSB-4, and FSB-8 normalized to S. A main observation is that as we proceed through FSB configurations (FSB-1 to FSB-8), the performance of each application monotonically improves until FSB-8 is knocked. Under FSB-8 the case changes and programs are split into three categories: (1) no benefit is accomplished (e.g., SpecJBB), (2) a benefit is achieved (e.g., Fluidanimate, Barnes, Lu, MIX1, MIX2, and MIX3), and (3) a degradation is observed versus FSB-4 (e.g., Swaptions, Bodytrack, and MIX4).

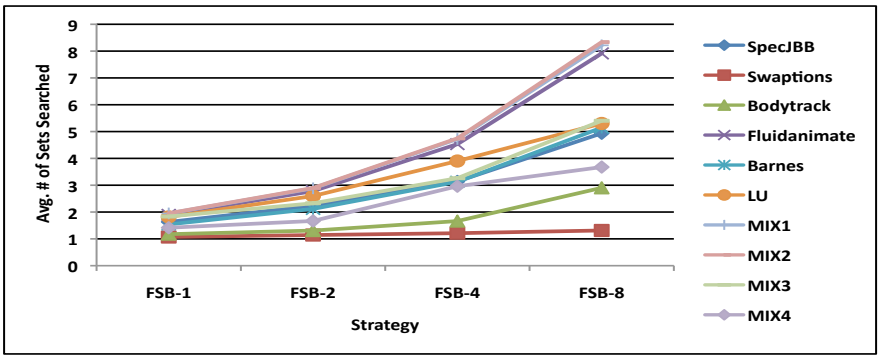

FIGURE 1.6: **The average number of L2 cache sets searched under FSB-1, FSB-2, FSB-4, and FSB-8.**

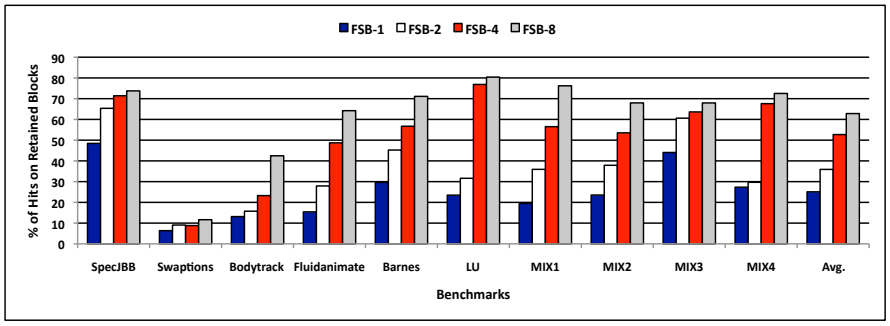

FIGURE 1.7: **The percentage of hits on retained cache lines under FSB-1, FSB-2, FSB-4, and FSB-8.**

Two factors define the eligibility of applications for accomplishing higher or lower performance when switching in between FSB's configurations: (1) the gain, G, from miss rate reduction and (2) the loss, L, from increased access latency. Let $\triangle_i$ be defined as $G - L$ for FSB-i. When $\triangle_8$ exceeds $\triangle_4$, the performance of the application improves by switching from FSB-4 to FSB-8, otherwise, it degrades. Swaptions, Bodytrack, and MIX4 achieve miss rate reductions of 6.1%, 7%, and 7%, respectively after increasing RT pointers from 4 to 8. In fact, under FSB-8, these three applications reduce the L2 miss rates the least as compared to the other examined programs (see Fig. 1.5 (a)). Clearly, $\triangle_4$ of each of Swaptions, Bodytrack, and MIX4 overpasses $\triangle_8$, thus they degrade under FSB-8 in comparison to FSB-4. FSB-1, FSB-2, FSB-4, and FSB-8 outperform S by averages of 4.3%, 8.8%, 13%, and 18.6%, respectively. Although FSB-8, on average, surpasses the remaining FSB's configurations, we consider FSB-4 more desirable for two main reasons. First, FSB-4 doesn't

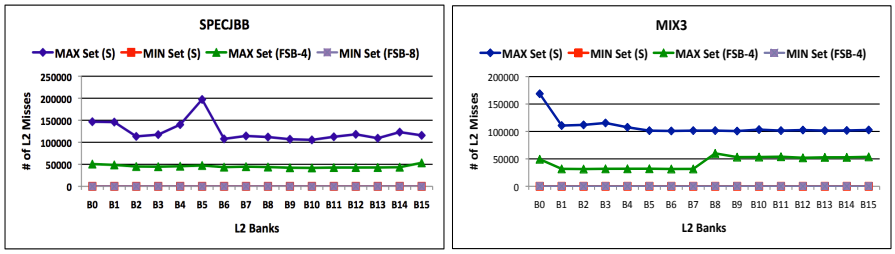

FIGURE 1.8: **The number of L2 misses experienced by cache sets at different L2 banks for SpecJBB and MIX3 programs under the baseline shared scheme (S) and FSB-4. Only the sets that exhibit the maximum (MAX Set) and the minimum (Min Set) misses are shown.**

observe any degradation in performance for any application when compared against the preceding configurations. Second, FSB-4 offers a better tradeoff between hardware complexity, power dissipation, and performance.

To that end, Fig. 1.6 depicts the average number of L2 cache sets searched for all the applications under FSB-1, FSB-2, FSB-4, and FSB-8. Furthermore, Fig. 1.7 displays the percentage of hits on retained cache lines for each program. In fact, the latter figure explores FSB's efficiency in satisfying far-flung reuses after retaining some fraction of the working set at underutilized sets. *With FSB-4, more than half of hits are satisfied by retained lines.* On average, the percentage of hits on retained lines provided by FSB-1, FSB-2, FSB-4, and FSB-8 are 25%, 35.8%, 52.6%, and 62.8%, respectively. Finally, Fig. 1.8 explores FSB's effectiveness in mitigating non-uniformity across sets by showing the number of misses experienced by cache sets at different L2 banks for two benchmarks, a multithreading one (i.e., SpecJBB) and a multiprogramming one (i.e., MIX3). We present only the sets that exhibit the maximum and the minimum misses for the baseline shared, S, and FSB-4.

1.4.3 Sensitivity to Different Pressure Functions

In the previous section we utilized cache misses as a pressure function. We tested other functions that can be used to measure pressures at cache sets. Fig. 1.9 plots the results for only three functions F1, F2, and F3 which denote functions with *misses* only, *hits* only, and *spatial hits*, respectively. We assume a low pressure limit (LPL) and a high pressure limit (HPL) each with $\alpha = 0.2$. The spatial hits function simply updates the pressure array with different values upon hits depending on lines' frames. That is, upon a hit on a line, L_{mru}, which exists at the MRU position, the function increments the bucket that corresponds to L_{mru}'s set by 1. However, upon a hit on a line, $L_{mru} - 1$, next to L_{mru}, the function increments the corresponding bucket by 2, and so on. The idea stems from the fact that a single highly contended line

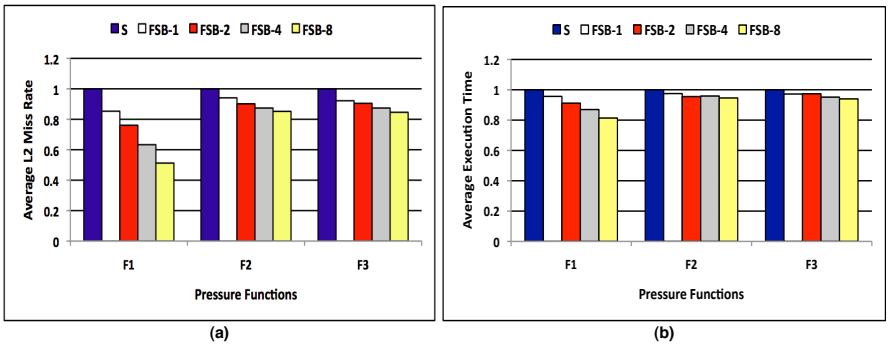

FIGURE 1.9: **Average L2 miss rates and execution times of all the benchmark programs under the baseline shared scheme (S), FSB-1, FSB-2, FSB-4, and FSB-8 (all normalized to S) (F1, F2, and F3 are pressure functions that involve misses, hits, and spatial hits, respectively).**

(say a lock) could result in a very high hit count at a particular set when, in fact, the pressure of lines competing for that set is very low. As depicted in Fig. 1.9 (b), on average, F2 produces performance improvements of 2.4%, 4.4%, 4.1%, and 5.4% for FSB-1, FSB-2, FSB-4, and FSB-8 over the baseline shared (S) scheme, respectively. F3, on the other hand, offers average performance improvements of 2.7%, 2.6%, 4.8%, and 6% for FSB-1, FSB-2, FSB-4, and FSB-8 over S, respectively. Lastly, F1 surpasses both, F2 and F3, and provides average performance improvements of 4.3%, 8.8%, 13%, and 18.6% for FSB-1, FSB-2, FSB-4, and FSB-8 versus S, respectively. For the examined benchmarks, we conclude that cache misses is preferable among the tested functions to represent pressures at cache sets. More comprehensive functions can be considered in a future work.

1.4.4 Sensitivity to LPL and HPL

So far, we have been using $\alpha = 0.2$ for the low and the high pressure limits, LPL and HPL. As Section 1.3.1 describes, by altering α, the range of source and destination sets can be expanded or contracted. We tested FSB-1, FSB-2, FSB-4, and FSB-8 with two more α values, particularly 0.1 and 0.3 for both LPL and HPL. Fig. 1.10 shows the results. RL1, RL2, and RL3 denote the retention limits (i.e., LPL and HPL) with α values of 0.1, 0.2, and 0.3, respectively. As demonstrated in Fig. 1.10(a), on average, RL1 provides L2 miss rate reductions of 14.4%, 21.3%, 35%, and 48.3% for FSB-1, FSB-2, FSB-4, and FSB-8 against the baseline shared (S) scheme, respectively. RL2, on the other hand, offers a little more enhancement and produces 14.6%, 23.9%, 39.4%, and 48.7% L2 miss rate reductions for FSB-1, FSB-2, FSB-4, and FSB-8 versus S, respectively. Finally, RL3 achieves 15.2%, 24.3%, 36.2%,

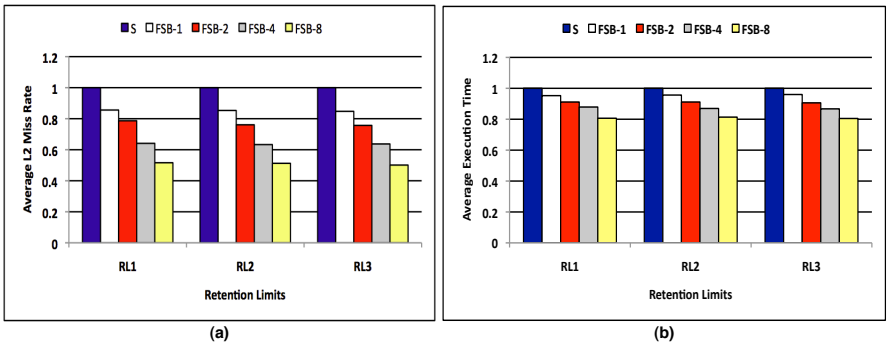

FIGURE 1.10: **Average L2 miss rates and execution times of all the benchmark programs under the baseline shared scheme (S), FSB-1, FSB-2, FSB-4, and FSB-8 (all normalized to S) (RL1, RL2, and RL3 are the Retention Limits- HPL and LPL- with $\alpha = 0.1$, $\alpha = 0.2$, and $\alpha = 0.3$, respectively).**

and 49.8% miss rate reductions for FSB-1, FSB-2, FSB-4, and FSB-8 over S, respectively. Fig. 1.10 (b) depicts the performance outcome. For the simulated benchmarks, we conclude that FSB shows low sensitivity to the examined α values.

1.4.5 Impact of Increasing Cache Size and Associativity

We can improve cache performance not only by efficient cache management but also via increasing cache size and associativity. We note that increasing cache associativity is not equivalent to FSB. First, larger associativity results in fewer sets, which don't help much if the conflict on the sets varies widely. Second, increasing cache associativity equates to merging sets in an indiscriminate way [27]. That is, which sets to merge is not an option. FSB, however, attempts to controllably and selectively increase the associativity of the sets that experience extensive conflicts without decreasing the number of sets (effectively decreasing associativity for underutilized sets).

In this section, we consider only FSB-4 (see Section 1.4.2 for a discussion on FSB's configurations). FSB-4 requires 11.5KB storage overhead per tile (see Table 1.1). To justify FSB-4's incurred overhead, we optimistically augment each cache set of the baseline shared scheme, S, with two more ways. In total, this adds to each L2 bank a 64KB more capacity. We refer to this configuration as S(2W). Moreover, we examine S with a double sized cache (i.e., 1MB instead of 512KB). We denote this latter configuration by S(D). Fig. 1.11 shows the L2 miss rates of S, S(2W), S(D), and FSB-4 normalized to S. The figure demonstrates that doubling the size of the cache results in a greater miss reduction than increasing associativity by two ways. Nonetheless,

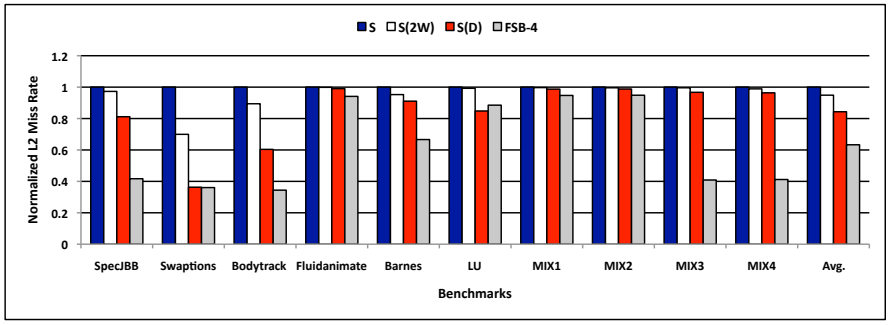

FIGURE 1.11: **L2 miss rates of the baseline shared scheme (S), S with two more ways added (S(2W)), S with double sized cache (S(D)), and FSB-4 (all normalized to S).**

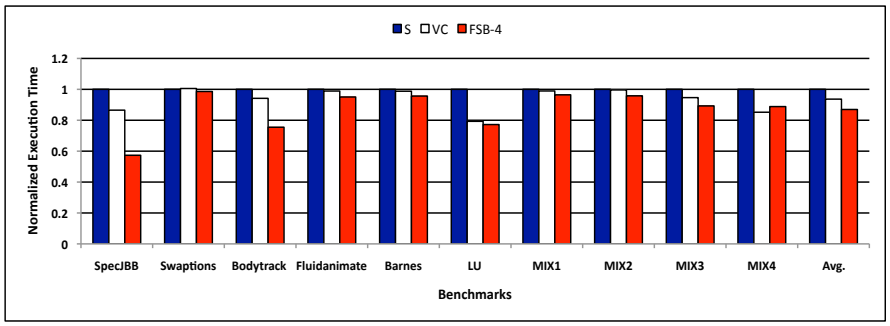

FIGURE 1.12: **Execution times of the baseline shared scheme (S), victim cache (VC), and FSB-4 (all normalized to S).**

FSB-4 surpasses S(D) for all the examined programs except Lu. On average, S(2W), S(D) and FSB-4 achieve L2 miss rate reductions of 5.1%, 15.6%, and 36.6%, respectively. We conclude that FSB-4 is quite attractive as with small design and storage overhead it provides more than 2× miss rate reduction over S(D) which incurs 88.8% increase in the on-chip cache capacity.

1.4.6 FSB versus Victim Caching

In this section we compare FSB against victim cache (VC) [20]. Again, we contrast only against FSB-4. VC effectively extends the associativity of hot sets in the cache to reduce conflict misses. For a fair comparison, we consider a fully associative 16KB VC per tile to approximately match the storage overhead incurred by FSB-4. We, furthermore, optimistically assume only a 6 cycle access time to VC after each miss on an L2 bank. Fig. 1.12 depicts the

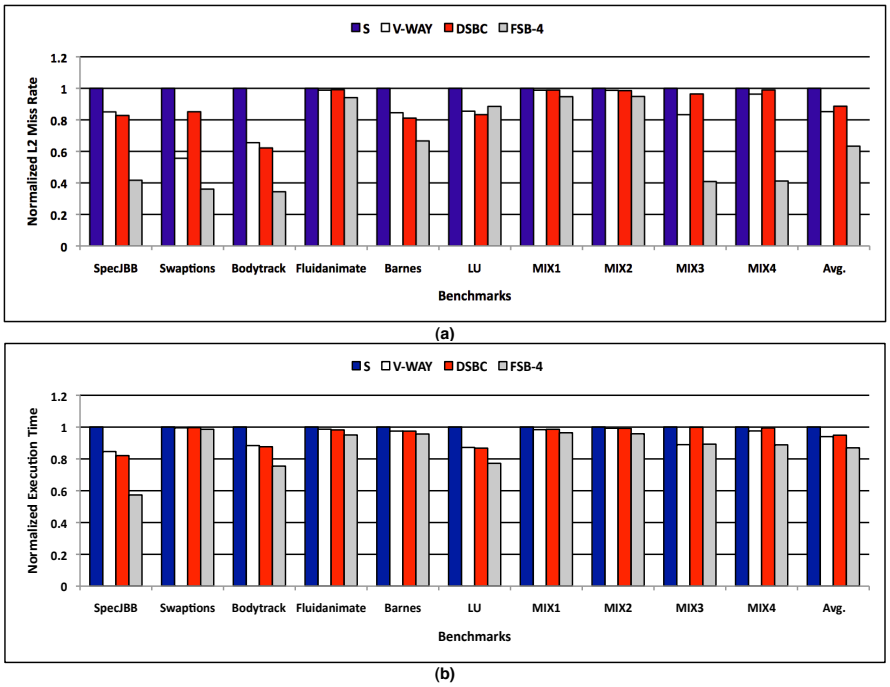

FIGURE 1.13: **L2 miss rates and execution times of the baseline shared scheme (S), variable-way set associative cache (V-WAY), dynamic set balancing cache (DSBC), and FSB-4 (all normalized to S).**

execution times of S, VC, and FSB-4 normalized to S. VC outperforms S by an average of 6.3%. In contrast, FSB-4 improves upon S and VC by averages of 13% and 7.2%, respectively.

1.4.7 FSB versus DSBC and V-WAY

In addition to comparing with victim caching, we compare FSB against the closely related dynamic set balancing cache (DSBC) [27] and variable-way set associative cache (V-WAY) [25] designs. Similar to FSB, both DSBC and V-WAY are directly extensible to CMPs. Section 1.2.3 details DSBC. V-WAY addresses the problem of workload imbalance among sets via varying the associativity of a cache by increasing the number of tag-store entries relative to the number of data lines. The tag and data stores are decoupled. The data-store is structured as one large piece and a *global* frequency based replacement policy, referred to as *Reuse Replacement* is employed in order to achieve better replacements. In reverse, the tag-store keeps a conventional set granular (local) replacement strategy (e.g., LRU).

For the reuse replacement policy, V-WAY associates each data line in the

cache with a *reuse counter*. A reuse count is defined as the number of L2 accesses to a cache line after its initial fill. Upon replacement, a line with a reuse counter equals to zero is replaced. To decide upon the number of bits required for reuse counters, we conducted a study to scrutinize the distribution of reuse counts for all evicted L2 cache lines from our benchmark programs. We observed that 99% of L2 cache lines are reused three or fewer times. Consequently, we choose to use two-bit saturating reuse counters.

Fig. 1.13(a) depicts the L2 miss rates of S, V-WAY, DSBC, and FSB-4 normalized to S. On average, V-WAY and DSBC achieve miss rate reductions of 14.7% and 11.3%, respectively. FSB-4 surpasses V-WAY and DSBC by averages of 27.2% and 29.2%, respectively. Fig. 1.13(b) shows the execution time results. V-WAY and DSBC outperform S by averages of 5.9% and 5%, respectively. FSB-4, however, improves upon V-WAY and DSBC by averages of 7.8% and 8.8%, respectively.

1.5 Related Work

Much work has been done to effectively minimize conflict misses in conventional cache designs. It is, in fact, quite impossible to do justice to this large body of work in this short article. As such, we briefly describe the proposals that are most relevant to FSB. The closest proposals to FSB are the dynamic set balancing cache (DSBC) [27] and the variable-way set associative cache (V-WAY) [25] designs. Sections 1.2.3 and 1.4.7 describe DSBC and V-WAY in detail and Section 1.4.7 compares against them.

Many proposals suggest alternative indexing functions to achieve a more uniform distribution of memory accesses. Predictive Sequential Associative-Cache [5], Column Associative Cache [2], and Hash-Rehash [1] are proposed in the context of direct-mapped caches. They provide the capability of mapping a cache line at an alternative pre-determined (using different hash functions) cache frame in order to provide performance similar to that of 2-way caches (schemes referred to as skewed caches). Rolán *et al.* [27] suggested a skewed set associative cache, denoted as static set balancing cache (SSBC), and found it impractical. Consequently, they proposed DSBC as a superior scheme. Our work simply promotes FSB over DSBC.

Adaptive Group-Associative Cache (AGAC) [22] identifies underutilized cache frames and attempts to utilize them to approximate a global LRU policy while maintaining the fast access in direct-mapped caches. Indirect Index Cache (IIC) [12] suggests a fully associative, software managed secondary cache system. IIC employs a generational replacement policy run by software. Simulation results in [25] manifested the outperformance of V-WAY versus AGAC. Besides, [25] shows that the miss reduction provided by V-WAY is

comparable to that of IIC. In Section 1.4.7 we demonstrate the outperformance of FSB against V-WAY.

Utility Based Cache Partitioning (UCP) [24] partitions at a way-granularity the last level shared cache among concurrently running applications depending on how much each application is likely to benefit from the cache (i.e., utility) rather than the application's demand for the cache. Dynamic Insertion Policy (DIP) [23] makes a key observation that a large number of cache lines become dead on arrival. Thus, a Bimodal Insertion Policy (BIP) is proposed to insert incoming lines frequently in the LRU positions and infrequently (with a low probability) in the MRU positions. Lines inserted at the LRU positions are only promoted to the MRU positions upon hits while residing in the LRU positions. For LRU-friendly workloads (i.e., favoring MRU insertions), however, the changes to the insertion policy might become detrimental to cache performance. As such, a *Set Dueling* mechanism is proposed to select among BIP and LRU depending on which policy incurs fewer misses. Simulation results in [27] demonstrated the outperformance of DSBC versus DIP. As shown in Section 1.4.7, FSB surpasses DSBC .

DIP uses a single policy (LRU or BIP) for *all* the concurrently running applications. A subsequent proposal, namely Thread-Aware Dynamic Insertion Policy (TADIP) [19], extends DIP to use a single policy for *each* application. Promotion/Insertion Pseudo-Partitioning (PIPP) [34] combines dynamic insertion and probabilistic promotion policies to provide the benefits of cache partitioning, adaptive insertion, and capacity stealing all with a single mechanism. Adaptive Set Pinning (ASP) [29] associates processors to cache sets and solely grants them permissions to evict blocks from their sets on cache misses. Therefore, references that may potentially cause inter-processor misses are no more allowed to interfere with each other even if they index to the same set.

Pseudo-Last-In-First-Out (Pseudo-LIFO) [8] proposes a family of replacement policies that manages each cache set as a fill stack. The replacement activities are restrained within a set to the upper part of the fill stack as much as possible. The lower part of the fill stack is left undistributed to extend the lifetime of the resident blocks. Among three members of the Pseudo-LIFO family, namely dead block prediction LIFO (dbpLIFO), probabilistic escape LIFO (peLIFO), and probabilistic counter LIFO (pcounter-LIFO), peLIFO is central. peLIFO synergistically learns the probabilities of experiencing hits beyond each of the fill stack positions and a set of highly preferred eviction positions is then deduced (based on this probability function) in the upper part of the fill stack.

Finally, Scavenger [3] partitions the total storage budget at the last level cache (LLC) into a conventional cache and a novel victim file (VF). Block addresses missing at the LLC are prioritized based on the number of times they have been observed in the LLC miss stream. If a block is evicted from the conventional part of the cache and indicates a high priority (i.e., frequently missed in the recent past), it gets stored in the VF.

1.6 Conclusions and Future Work

Memory accesses are not evenly distributed across cache sets. Such a skew in sets' usages reduces the effectiveness of the conventional cache designs and cache lines become less likely to be re-referenced before eviction. We propose Flexible Set Balancing (FSB), a strategy that exploits the demand imbalance across sets to retain cache lines evicted from highly pressured sets at underutilized sets so as to satisfy far-flung reuses. FSB adapts to phase changes in programs and promotes a very flexible sharing among cache sets. An underutilized set is allowed to share its space by any stressed set during any point in a program's execution, a policy that we refer to as *one-from-many* sharing. Besides, many sets are allowed to share their capacities with a highly utilized set, a policy that we refer to as *many-from-one* sharing. FSB incurs a little storage, area, and energy overheads. FSB achieves an average miss rate reduction of 36.5% versus a shared CMP cache organization for the tested benchmarks. This produces an average execution time improvement of 13%. Furthermore, evaluations manifested the outperformance of FSB over some relevant designs including DSBC [27] and V-WAY [25].

FSB is extensible and practical in that it can be applied to single-core as well as multi-core architectures. In this work we evaluated FSB on a 16-way tiled CMP platform. FSB retains lines only at a bank granularity (intra-tile retention). When an L2 bank can't absorb anymore the working set of a running program, the lines selected for replacements are simply discarded. In fact, in the meantime other L2 banks might indicate the presence of some underutilized sets. As such, one may benefit from retaining lines across L2 banks (inter-tile retention), rather than only within a single L2 bank, so as to satisfy even more far-flung reuses. Exploring the promise of such a strategy is set as a main future direction.

Bibliography

[1] A. Agarwal, J. Hennessy, and M. Horowitz. "Cache performance of operating systems and multiprogramming," *In ACM Transactions on Computer Systems, 6*, Nov 1988.

[2] A. Agarwal and S. D. Pudar. "Column-associative caches: A technique for reducing the miss rate of direct-mapped caches," *ISCA*, May 1993.

[3] A. Basu, N. Kirman, M. Chaudhuri, and J. F. Martínez. "Scavenger: A New Last Level Cache Architecture with Global Block Priority," *MICRO*, 2007.

[4] C. M. Bienia, S. Kumar, J. P. Singh, and K. Li. "The PARSEC Benchmark Suite: Characterization and Architectural Implications," *PACT*, Oct. 2008.

[5] B. Calder, D. Grunwald, and J. S. Emer. "Predictive sequential associative cache," *HPCA*, Feb. 1996.

[6] L. Censier and P. Feautrier. "A New Solution to Coherence Problems in Multicache Systems," *IEEE Trans. Comput. C-27 (12): 1112- 1118*, Dec. 1978.

[7] J. Chang. "Cooperative Caching for Chip Multiprocessors," *PhD thesis, University of Wisconsin-Madison*, 2007.

[8] M. Chaudhuri. "Pseudo-LIFO: The Foundation of a New Family of Replacement Policies for Last-level Caches," *MICRO*, Dec. 2009.

[9] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. "Distance associativity for high-performance energy-efficient non-uniform cache architectures," *MICRO*, 2003.

[10] S. Cho and L. Jin "Managing Distributed Shared L2 Caches through OS-Level Page Allocation," *MICRO*, Dec 2006.

[11] Digital Equipment Corporation, Hudson, MA. "Digital Semiconductor 21164 AlphaMicroprocessor Product Brief," *Technical Document EC-QP97D-TE*, Mar. 1997.

[12] E. G. Hallnor and S. K. Reinhardt. "A fully associative software managed cache design," *ISCA*, 2000.

[13] M. Hammoud, S. Cho, and R. Melhem. "ACM: An Efficient Approach for Managing Shared Caches in Chip Multiprocessors ," *HiPEAC*, Jan. 2009.

[14] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. "Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches," *ISCA*, June 2009.

[15] S. Harris. "Synergistic Caching in Single-Chip Multiprocessors," *PhD thesis, Stanford University*, 2005.

[16] HP Labs. "http://www.hpl.hp.com/research/cacti/"

[17] H. Huang, K. G. Shin, C. Lefurgy, and T. Keller. "Improving Energy Efficiency by Making DRAM Less Randomly Accessed," *ISLPED*, August 2005.

[18] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler. "A NUCA Substrate for Flexible CMP Cache Sharing," *ICS*, June 2005.

[19] A. Jaleel, W. Hasenplaugh, M. K. Qureshi, J. Sebot, S. Steely Jr., and J. Emer. "Adaptive Insertion Policies for Managing Shared Caches," *PACT* , 2008.

[20] N. P. Jouppi. " Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," *ISCA*, 1990.

[21] P. Kongetira, K. Aingaran, and K. Olukotun. "Niagara: A 32-Way Multithreaded Sparc Processor," *IEEE Micro*, March-April 2005.

[22] J. Peir, Y. Lee, and W. Hsu. "Capturing Dynamic Memory Reference Behavior with Adaptive Cache Topology," *ASPLOS*, 1998.

[23] M. K. Qureshi, A. Jaleel, Y. N. Patt, and S. C. Steely Jr.. "Adaptive Insertion Policies for High Performance Caching," *ISCA*, June 2007.

[24] M. K. Qureshi and Y. N. Patt. "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance," *MICRO*, Dec. 2006.

[25] M. K. Qureshi, D. Thompson, and Y. N. Patt. "The V-WAY Cache: Demand-Based Associativity via Global Replacement," *ISCA*, June 2005.

[26] Research at Intel. "Introducing the 45nm Next-Generation Intel Core^{TM} Microarchitecture," *White Paper.*,

[27] D. Rolán, B. B. Fraguela, and R. Doallo "Adaptive line placement with the set balancing cache," *MICRO*, Dec. 2009.

[28] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. "POWER5 System Microarchitecture," *IBM J. Res. & Dev.*, July. 2005.

[29] S. Srikantaiah, M. Kandemir, and M. J. Irwin. "Adaptive Set Pinning: Managing Shared Caches in Chip Multiprocessors," *ASPLOS*, March 2008.

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

[31] Virtutech AB. Simics Full System Simulator "http://www.simics.com/"

[32] D. Weiss, J. J. Wuu, and V. Chin. "The on-chip 3-mb subarray-based third-level cache on an itanium microprocessor," *In IEEE journal of solid state circuits*, Nov. 2002.

[33] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. "The SPLASH-2 Programs: Characterization and Methodological Considerations," *ISCA*, 1995.

[34] Y. Xie and G. H. Loh. "PIPP: Promotion/Insertion Pseudo-Partitioning of Multi-core Shared Caches," *ISCA*, June 2009.

[35] M. Zhang and K. Asanović. "Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors," *ISCA*, June 2005.