Appears in Proceedings of the 18th International Symposium on High Performance Computer Architecture (HPCA-18), February, 2012.

Decoupled Dynamic Cache Segmentation

Samira M. Khan, Zhe Wang and Daniel A. Jiménez
Department of Computer Science
The University of Texas at San Antonio
{skhan,zhew,dj}@cs.utsa.edu

Abstract

The least recently used (LRU) replacement policy performs poorly in the last-level cache (LLC) because temporal locality of memory accesses is filtered by first and second level caches. We propose a cache segmentation technique that dynamically adapts to cache access patterns by predicting the best number of not-yet-referenced and already-referenced blocks in the cache. This technique is independent from the LRU policy so it can work with less expensive replacement policies. It can automatically detect when to bypass blocks to the CPU with no extra overhead. In a 2MB LLC single-core processor with a memory intensive subset of SPEC CPU 2006 benchmarks, it outperforms LRU replacement on average by 5.2% *with not-recently-used (NRU) replacement and on average by* 2.2% *with random replacement. The technique also complements existing shared cache partitioning techniques. Our evaluation with 10 multi-programmed workloads shows that this technique improves performance of an 8MB LLC four-core system on average by* 12%*, with a random replacement policy requiring only half the space of the LRU policy.*

1. Introduction

Modern processors have large on-chip caches to mitigate off-chip memory latency. The least recently used (LRU) replacement policy represents the cache blocks in a set as a recency stack where the most recently used (MRU) block is at the top of the stack. This policy picks the LRU block as the replacement candidate as blocks recently used are more likely to be accessed in the near future. However, in a three-level cache hierarchy, this temporal locality is filtered by the L1 and L2 caches. Thus, LRU performs poorly in the last-level cache (LLC) [23, 7, 16]. Recent proposals change the insertion policy, placing incoming blocks in different stack positions to adapt to workload characteristics [23, 7, 28]. These proposals are resistant to *scans* and to *thrashing*. A scan is a burst of accesses to data that are never reused before being evicted. Thrashing workloads have a working set size greater than the cache size, causing useful cache blocks to be evicted. Though these proposals take care of scanning and thrashing workloads, they depend on the recency levels the LRU policy and insert blocks adaptively in either recent or non-recent positions. Thus, these techniques cannot be applied to policies with no inherent levels, for example random replacement. In this paper we propose a decoupled technique that is scan and thrash resistant and can be used with any replacement policy. Recent insertion-based proposals require low overhead but cannot detect blocks that will not be reused, i.e. zero-reuse blocks. Replacement policies that can *bypass* zero-reuse blocks [14, 10], i.e., pass them along to the CPU without placing them in the cache, require significant hardware overhead. Our proposed technique is not only scan and thrash resistant, but can automatically detect zero-reuse blocks irrespective of the base replacement policy without significant overhead.

We propose segmenting cache sets into referenced and non-referenced blocks, i.e. blocks that have been referenced at least once since being placed in the cache and blocks that have not yet been referenced since their initial access. This technique attempts to maintain the best number of non-referenced and referenced blocks depending on the workload characteristics. We propose a scalable low-overhead *segment predictor* based on sampling that can predict the best segmentation at runtime. It adjusts the segmentation based on cache access patterns. It can resist scans and thrashing by evicting non-referenced blocks. This technique is completely decoupled from the replacement policy. Dynamic Cache Segmentation(DCS) decides whether a victim should be selected from non-referenced blocks or referenced blocks. Any replacement policy can be used to choose the victim from that specific list. The technique can improve performance with inexpensive policies like Not Recently Used (NRU) and random. This scan and thrash resistant technique requires far less space than LRU and can still detect zero-reuse blocks. Dynamic segmentation with automated bypass can improve performance with random replacement, requiring only half the space overhead of

LRU. It can also be used for multi-core workloads with a conjunction of any cache partitioning techniques. The contributions of this paper are:

- We propose a dynamic cache segmentation technique that attempts to keep the best number of non-referenced and referenced blocks in cache sets. We propose a sampling-based scalable low-overhead technique to predict the best segmentation.
- We show that the segmentation technique can be decoupled from the replacement policy and can work with inexpensive policies like NRU and random. It also automatically detects bypassing opportunities without any extra overhead irrespective of the replacement policy being used.
- We show that cache segmentation complements current shared cache partitioning techniques. Dynamic cache partitioning even with a default random policy can outperform LRU using half the space overhead.

In a single-core processor with 2MB LLC, DCS outperforms LRU policy on average by 5.2% with NRU replacement and on average 2.2% with random replacement, with a memory intensive subset of SPEC CPU 2006 benchmarks. Evaluation with multi-programmed workloads shows that the technique improves the performance of a 8MB LLC in a four-core system on average by 12% with random replacement requiring half the space of LRU.

2. Motivation

Recent work has shown that last-level caches (LLCs) perform poorly with the LRU replacement policy [23, 7, 16]. Upper level caches filter out the temporal locality, destroying the property upon which LRU relies, that most recently used block will be used again in the near future. LRU performs poorly in the LLC when the working set is larger than the cache size (thrashing) or bursts of non-temporal references evict the active working set (scanning).

2.1. Addressing Workload Behavior

Recent proposals addressing these effects include changing the insertion policy [23, 7] so that thrashing and scanning workloads can perform well in the LLC. Dynamic Insertion Policy (DIP) [23] avoids thrashing by attempting to keep a portion of the workload in the cache by inserting blocks responsible for thrashing into the LRU position. Thus, some part of the working set always remains in the cache and thrashing blocks can not pollute the whole cache. DIP uses LRU for all other non-thrashing workloads, so it performs poorly for workloads where frequent scans discard the working set in the LLC. RRIP [7] is an insertion policy with NRU replacement policy that is both scan and thrash resistant. It inserts thrashing blocks at the end of the NRU stack and other blocks near the end of the NRU stack. Thus RRIP adapts to both thrash resistant DIP and scan resistant LFU. The performance of RRIP depends on the number of NRU levels as non-scanning blocks must be re-referenced before they are evicted.

Technique	Thrash resistant	Scan resistant	Detect bypass	Decoupled	Overhead
DIP	yes	no	no	no	low
RRIP	yes	yes	no	no	low
DCS	yes	yes	yes	yes	low

Table 2: Comparison among techniques

2.2. A Decoupled Flexible Policy

Clearly, it would be best to have a policy that is resistant to both scanning and thrashing but also works with mixed access patterns. Both DIP and RRIP are dependent on a policy that divides the blocks of a set in levels of recency (LRU and NRU) to insert blocks into a specific position. We propose a mutable technique that is decoupled from the choice of replacement policy and works with all access patterns. Additionally, the technique can detect when to bypass blocks without extra overhead. Most blocks brought into the LLC are never used again. Recent work has proposed techniques where zero-reuse blocks are identified and evicted earlier [14, 13, 12]. However they require significant extra overhead to identify blocks that are never reused. We propose a mutable scan and thrash resistant technique with automated bypass that works with any replacement policy without significant overhead.

2.3. Segmenting Cache Sets with Prediction

We propose to segment cache sets into two parts: the *referenced list* and *non-referenced list*. Blocks that have been referenced again after being brought into the cache belong to the referenced list while blocks that have not been reused are in the non-referenced list. A *segment predictor* predicts the best segmentation for the two lists. It predicts the best segment size for the non-referenced list; the resulting size for the reference list is simply the size of the non-referenced list subtracted from the total number of ways. This technique attempts to keep the best number of referenced and non-referenced blocks by choosing replacement victims from the list that has more blocks than the predicted best number of blocks. This technique can address thrashing and scanning blocks by keeping the non-referenced list as a singleton block. It also chooses the best

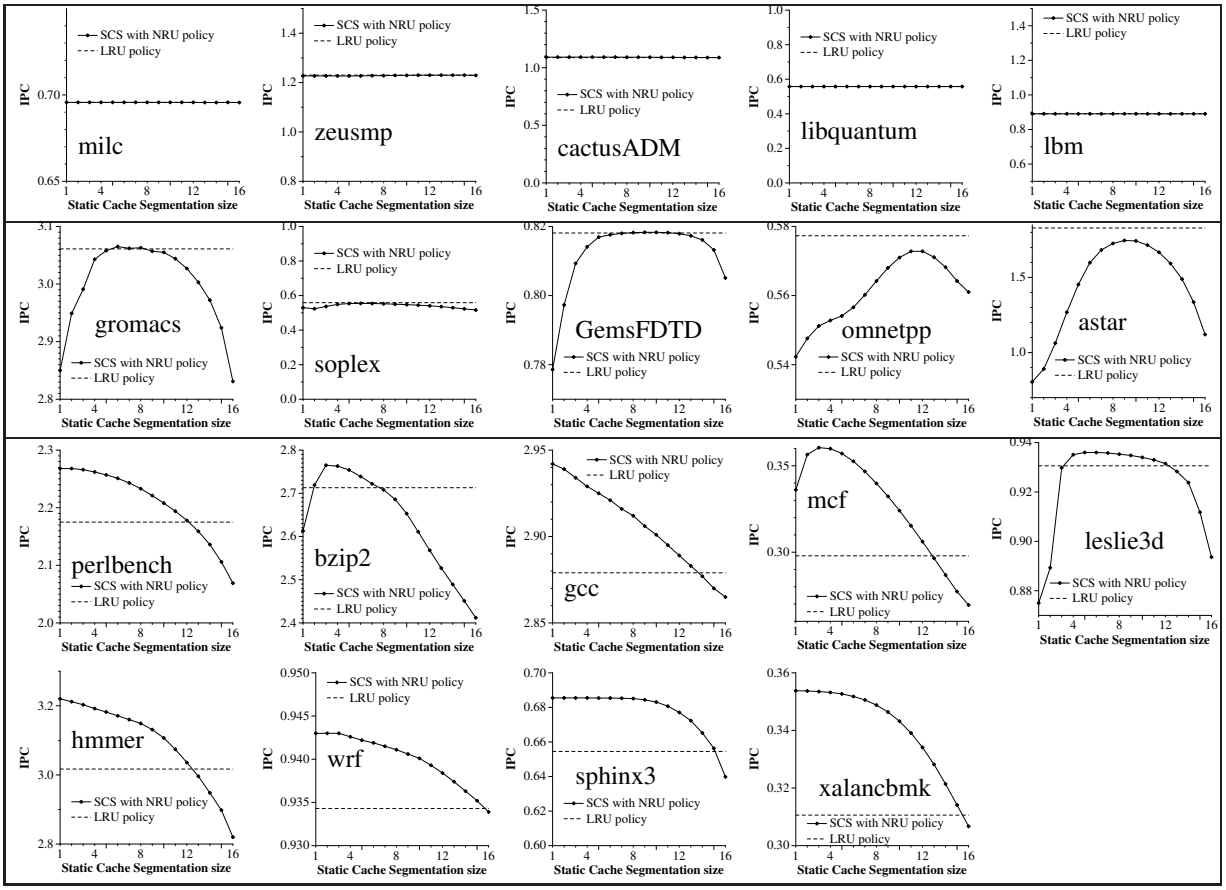

Table 1: Effect of static segmentation. Row 1 shows the benchmarks with no effect of segmentation. Row 2 shows LRU friendly benchmarks and row 3 shows benchmarks which are effected by segmentation.

segmentation size for mixed access patterns. Table 1 shows the instructions-per-cycle (IPC) with static segment sizes for the SPEC CPU 2006 benchmarks. We show the IPC when each benchmark is run with static segement size 1 to 16. The maximum IPC is achieved at different segment size for different benchmarks. For scanning and thrashing workloads, the best static segment size of the non-referenced list is one, where mixed access pattern workloads perform best with variable segment sizes. For example, perl performs best with segment size 1 but bzip performs best with segement size 4. This clearly shows that we need an adpative mechnism to determine the best segment size at runtime and unlike previous policies [23, 7], staticially determining the insertion position is not enough to achieve the best possible performance.

Our technique selects the list from which the victim should be chosen. Any replacement policy can be used within a list, even random replacement. Thus, the replacement policy is decoupled from the segmentation technique. Another advantage is that segmentation automatically detects when to bypass lines. When the predicted best segmentation size is one, blocks arriving to the LLC cannot be referenced. So segmentation can identify a zero-reuse block without any extra overhead.

3. Related Work

This section describes related work. There have been many replacement policy proposals for both disk caches and CPU caches. Prior work has proposed different versions of the LRU replacement policy. Segmented LRU [9] was proposed for disk caches. It augments each cache block with a reference bit dividing the traditional LRU stack of cache blocks into two logical sub-lists, the referenced list and the non-referenced list. The replacement policy chooses the LRU block from the non-referenced list. If all cache blocks are in the referenced list then it selects the global LRU cache block from among all the blocks in the set. This policy performs poorly for LRU friendly workloads as the stale blocks in the referenced list are rarely evicted.

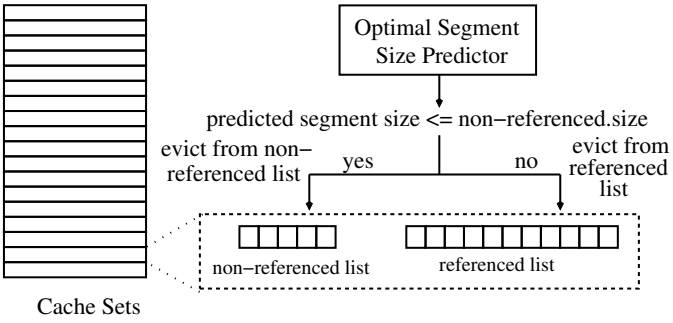

Figure 1: Logical view of the replacement scheme

There are other variations of the LRU replacement policy, such as the LRU-K policy [21]. This policy takes into account the k^{th} access from the last access to determine the victim. The Least Frequently Used (LFU) policy chooses the victim considering access frequency rather than the recency [17]. LFU performs poorly with workloads that have temporal locality, thus much work has been done to develop hybrid policies that take into account both recency and frequency [20, 2, 8]. Adaptive tuning policies like ARC and CAR [20, 2] use segmentation to choose which list to evict from. However, they require a large overhead to determine the best segmentation. ARC requires tracking 2c lines for a cache with size c. Pseudo-LIFO [4] chooses the victim from a fill stack rather than a recency stack. This policy requires extra overhead to track the fill positions of the blocks. Other work also proposes to divide the LRU blocks into different partitions and replace from different portions of the partitions [8, 19]. They all maintain a static ordering while deciding which partition should be used to choose the victim. They cannot choose the partition adaptively.

Recent work propose changing the insertion policy to adapt to different cache access patterns [23, 7]. Dynamic Insertion Policy (DIP) places incoming lines at the end of the LRU stack [23]. RRIP attempts to insert lines near the end of the recency stack [7]. Both of them takes into account two types of workloads. DIP can handle LRU friendly and thrashing workloads. RRIP can handle thrashing and scanning workloads.

Another area of research predicts the last touch of the blocks [16, 18]. Dead block predictors can detect when a block is accessed for the last time and evict them to make room for useful lines [14, 13]. However, dead block prediction requires significant extra overhead for the predictor.

4. Dynamic Segmented Cache

Dynamic cache segmentation (DCS) partitions cache sets into non-referenced blocks and referenced blocks. It attempts to keep the best partition for all sets. If one of the lists has more blocks than it is suppose to, the next victim is chosen from that list. Figure 1 shows the logical view of the segmented cache. The segment predictor predicts the best partition size for a given workload. When a block must be replaced from a set, either list can be chosen depending on the current partition of the set. DCS needs only one bit per cache block to differentiate referenced blocks from the non-referenced blocks.In this section we describe in detail the idea of DCS as well as several versions of the policy based on different underlying replacement policies.

4.1. Predicting the Best Segmentation

Here we discuss how the segment predictor predicts the best segment size. We use set-dueling with sampling to determine the best segment size [23]. Set-dueling samples a few "leader" sets from the cache that always implement one particular policy chosen from the possible policies. Counters are used to keep track of which policy yields the fewest misses and all of the other cache sets, i.e. "follower" sets, follow the best policy. We consider two designs: one uses some sampled sets of partial tags kept externally and the other uses sampled sets in the cache. These randomly chosen sample sets imitate a decision tree used to predict the best segmentation. We consider five segments sizes for the non-referenced list to choose from. They are segment size 1, 4, 8, 12 and 16 for a 16-way set-associative cache. Considering more segment sizes does not significantly improve performance. In the segment predictor, two segment sizes set-duel at each level. The winner of one level determines the next competitor segment size for the next level. Figure 2

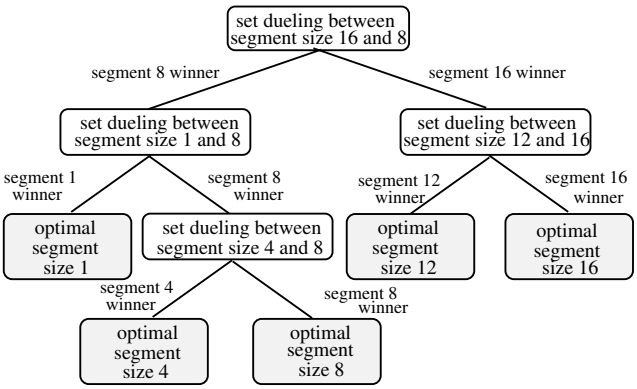

Figure 2: Decision Tree Analysis to find the best non-referenced segment size

shows each leaf representing the final best size. Each node in the tree determines the next segment size to be considered. For example, if between segment sizes 8 and 16, segment size 8 is winning, that means that having at most 8

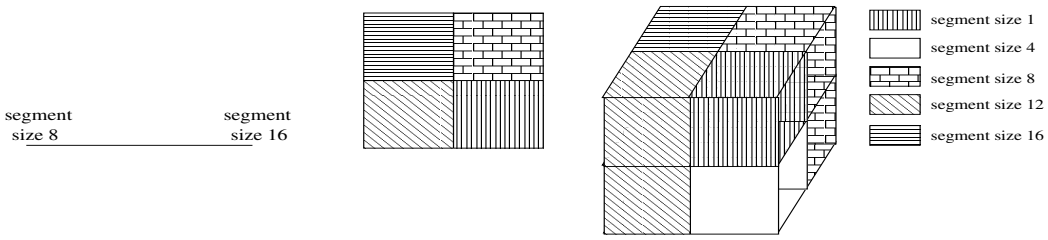

Figure 3: 3D model of the Optimal Segment Predictor

blocks in the non-referenced list will yield the best performance. In order to determine which partition between 1 and 8 is better, the next level duels between segment sizes 1 and 8. This way each node selects the next segment size until it reaches a leaf. We use three kinds of leader sets. Two leader sets are static and always uses segment sizes 8 and 16, ensuring that the predictor will respond to workload phase changes. Depending on which leader set is winning, the adaptive leader picks the segment size following the decision tree. This way the technique can consider variable segment sizes using only three types of leader sets. We can visualize DCS as a three-dimensional decision making engine while set-dueling is only one-dimensional as depicted in Figure 3. Set-dueling can make only one of the two decisions. In a two-dimensional decision engine, we can choose one of four outcomes. We have added another dimension so that we can choose one of eight decisions. However, we find that choosing one of five outcomes is sufficient. This technique is different from multi-set-dueling [19] where each decision has its own leader sets and they simultaneously set-duel at each level in groups of two. Multi-set-dueling is not scalable as the number of leader sets has to grow linearly with the number of outcomes being considered.

4.2. Decoupled from Replacement Policy

Partitioning the cache sets into non-referenced and referenced list enables decoupling the replacement policy from the segmentation. DCS only decides which list should be used for choosing replacement victims depending on the current and best list sizes. Each list is free to use any replacement policy to choose the replacement victim. DCS can be used with simple and light replacement policies like Not Recently Used (NRU) and random replacement. In Section 7 we show that DCS with simple NRU and random policies outperforms the more costly LRU policy.

Dynamic Segmentation with NRU Policy The NRU policy approximates the recency stack based LRU policy but using only one bit, called the NRU bit, per cache block. The NRU recency stack has only two levels, so each level can have more than one block. By contrast, the LRU policy is organized such that each block will reside in a distinct level. In NRU, when a new line is placed in the cache, its NRU bit is set to zero moving the cache block to the top of the recency stack. On a cache miss a block whose NRU bit is set to one is selected as the victim. Since there can be many blocks in the lower recency level, the NRU victim search starts from a fixed cache way. If there is no block with its NRU bit set to one, then all of the NRU bits are set to one. We adapt DCS to use the NRU policy for each list to select the victim. Each cache block needs one bit to identify referenced and non-referenced lines and one bit to track the NRU policy within the lists.

Dynamic Segmentation with Random Policy The random replacement policy selects a victim pseudo-randomly from a cache set. It has no information about temporal locality, so it tends to perform poorly for LRU friendly workloads. It also has no information about thrashing/scanning workloads so it performs poorly for them as well. Nevertheless, it is the simplest replacement policy with no cache block overhead. We show that DCS works well even with random replacement. The segment predictor picks the list that from which the victim should be chosen and a block is randomly chosen from that list. This provides the random policy some information about the workload access pattern. In Section 7 we show that DCS with random replacement can outperform the LRU policy. Per block overhead in this case is just one bit that differentiates non-referenced and referenced lines.

Ignoring Segmentation Figure 1 shows that there are some workloads that will always perform better with LRU than with any segmentation size. In these cases we can choose to ignore the segmentation. We can implement a version of DCS that incorporates non-segmented policies like NRU/LRU. After the predictor chooses the best segmentation, that segmentation set-duels with the non-segmented policy. If the winner of this final stage is non-segmented policy, then the segment predictor concludes that the workload is LRU-friendly and the replacement decision ignores the segmentation. In the case of NRU policy, it picks one of the non-recently used blocks from the whole set.

4.3. Automated Bypass

The LLC frequently has a large fraction of zero-reuse blocks [14, 10]. Bypassing these blocks to the core can significantly improve LLC efficiency. Bypassing allows the LLC to save space for other useful blocks in the cache by not placing a suspected zero-reuse block in the cache. One advantage of DCS is that it can inherently detect zero-reuse blocks without any extra overhead. A large number of zero-reuse blocks are referenced when there are thrashing/scanning workloads. DCS can detect these workloads and identify the phases where it can safely bypass an incoming block. When the current partition size is one, DCS can safely bypass incoming blocks. The policy keeps one incoming block in the LLC after every 32 misses to prevent the working set from becoming stale.

4.4. Ensuring Thrashing Resistant

DCS does not enforce the partition size in cache sets. Rather, it attempts to gradually converge to the partition size predicted by the segment predictor. When bursts of lines are placed in the cache they all start from the non-referenced list. This initialization can cause the number of lines to far exceed the predicted best size. Figure 4 shows the run time

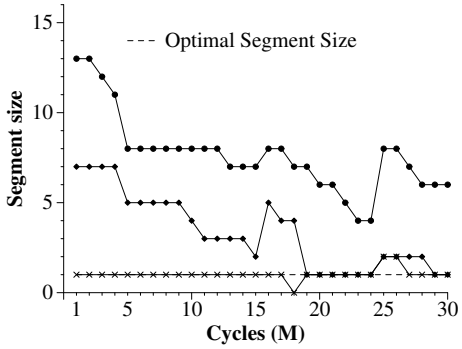

Figure 4: Runtime non-referenced segment size for three representative sets from 483.xalancbmk

non-referenced segment size of three representative sets for the benchmark `483.xalancbmk` chosen randomly. The segment predictor predicts the best non-referenced list size to be one. Some sets quickly converge to the best size while some converge slowly. However, there are also a large number of sets where bursts of accesses always keep the current partition far more than the predicted best size of one. We solve this problem by inserting blocks in the not-recent part of the not-referenced list. This ensures that thrashing blocks are discarded from the cache, while maintaining part of the working set in the cache.

4.5. Mutable Policy

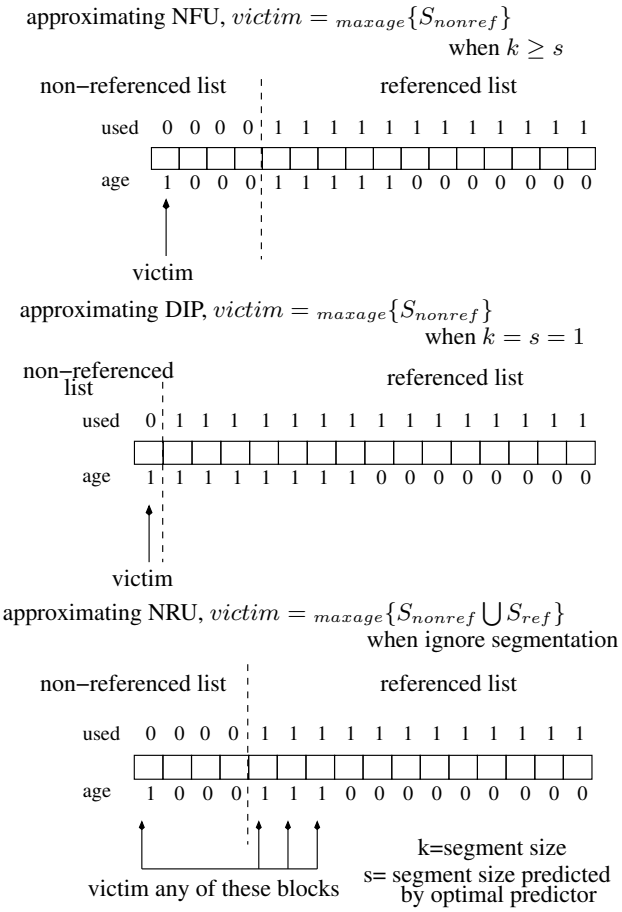

Figure 5: Three cases of Dynamic segmentation

Depending on the underlying replacement policies allowed by implementation constraints, DCS can mutate among LRU, LFU, DIP and in between. Using NRU as the baseline policy it mutates between NRU, which approximates LRU, Not Frequently Used (NFU), which approximates LFU, DIP (with underlying NRU policy), and in between. Let S is the set of the blocks in the lists. $S_{nonref} = \{a_i\}_{i=1}^{k}$ and $S_{ref} = \{a_i\}_{i=1}^{assoc-k}$. If k is the size of non-referenced list and s is the size predicted by the optimal predictor, according to our policy

$$\text{victim} = \begin{cases} {}_{maxage}\{S_{nonref}\}, \text{if } k \geq s \\ {}_{maxage}\{S_{ref}\}, \text{if } k < s \\ {}_{maxage}\{S_{ref} \bigcup S_{nonref}\}, \text{ignore segmentation} \end{cases}$$

The first case in Figure 5 shows DCS approximating to LFU. In this case blocks in the non-referenced list have been accessed just once. However all the other blocks in the referenced list have been accessed more than once. If there are

no blocks in the non-referenced list, then one block from the referenced list is evicted. In this case that block is a Not Frequently Used block in the whole set. Since we implement a dynamic insertion policy when the segment predictor predicts the non-referenced segment size to be one, DCS falls back to DIP [23] with default replacement policy NRU. With DIP, blocks are inserted at the end of the LRU list. With DCS, when best non-referenced segment size is predicted to be one, incoming blocks are inserted in the NRU position of the non-referenced list. This ensures that thrashing blocks are evicted as soon as possible. This is depicted in the 2nd case in the Figure 5. In the last case, the replacement policy ignores the segmentation. In this case since the underlying policy is one-bit NRU, it replaces the Not Recently Used block irrespective of which segment it resides in. Depending on the underlying policy used in the referenced and non-referenced lists, DCS can fall back to other policies as well. For example when FIFO policy is used in the non-referenced list it becomes a 2Q policy [8].

5. DCS with Shared Cache Partitioning

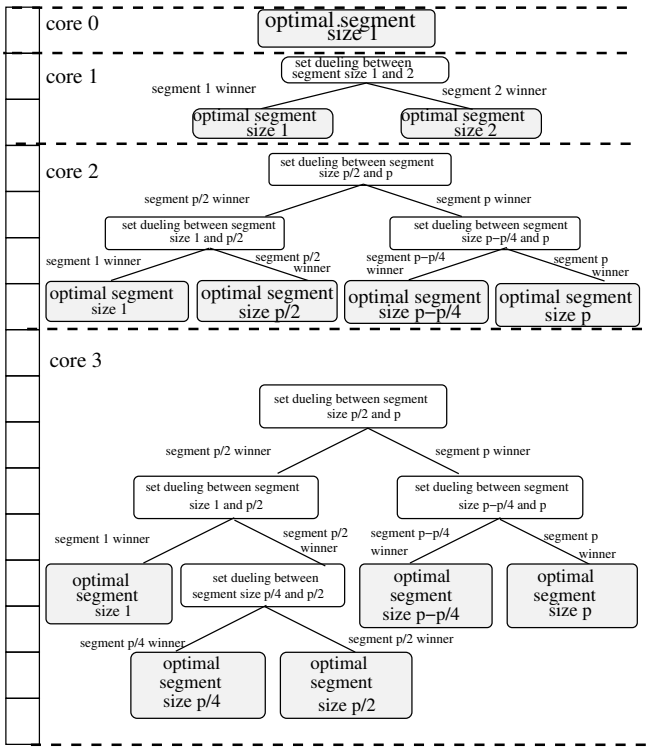

Figure 6: Dynamic Segmentation in each partition of UCP

DCS partitions cache sets into referenced and non-referenced lists. This technique can partition shared cache sets depending on the mixed access pattern from all the workloads. However, it does not take into account the different behavior of different programs or threads. This can be solved by using DCS in conjunction with any adaptive way-partitioning technique [25, 26, 27]. Our shared-cache-aware DCS technique segments the ways belonging to a specific thread assigned by a cache partitioning technique. Thus, DCS complements way-partitioning. Since cache segmentation is decoupled from replacement policy, segmenting the partitioned cache can work with random replacement. Partitioning the partitions essentially limits the number of replacement candidates such that even choosing a candidate at random is often sufficient. In Figure 6 we show how DCS determines the segment size from each UCP partition [25]. If the partition size is p, the segment size is chosen from p, $p - p/4$, $p/2$, $p/4$ and 1.

6. Experimental Methodology

This section outlines the experimental methodology used in this study.

We use a memory-intensive subset of 19 SPEC CPU 2006 benchmarks chosen with a methodology outlined below. We have grouped these 19 benchmarks according to their response to static segmentation shown at table 1. Benchmarks like *433.milc* experience no effect under segmentation. Some of them are very LRU-friendly. Table 4 shows the benchmark grouping. We use a modified ver-

Insensitive to segmentation	*433.milc 434.zeusmp 436.cactusADM 462.libquantum 470.lbm*
LRU friendly	*435.gromacs 450.soplex 459.GemsFDTD 471.omnetpp 473.astar*
Sensitive to segmentation	*400.perlbench 401.bzip2 403.gcc 429.mcf 437.leslie3d 456.hmmer 481.wrf 482.sphinx3 483.xalancbmk*

Table 4: Benchmarks grouping

sion of CMP$im, a memory-system simulator [5]. The version we used was provided with the JILP Cache Replacement Championship [1]. It models an out-of-order 4-wide 8-stage pipeline with a 128-entry instruction window. This infrastructure enables collecting instructions-per-cycle figures as well as misses per kilo-instruction. The experiments model a 16-way set-associative last-level cache to remain consistent with other previous work [15, 18, 23, 24]. The microarchitectural parameters closely model Intel Core i7 (Nehalem) with the following parameters: L1 data cache: 32KB 8-way associative, L2 unified cache: 256KB 8-way L3: 2MB/core. Each benchmark is compiled for the x86_64 instruction set. The programs are compiled with the GCC 4.1.0 compilers for C, C++, and FORTRAN. We use SimPoint [22] to identify a single one billion instruction characteristic interval (i.e.*simpoint*) of each benchmark. Each

Name	MPKI (LRU)	MPKI (MIN)	IPC (LRU)	FFWD	Name	MPKI (LRU)	MPKI (MIN)	IPC (LRU)	FFWD
astar	2.275	2.062	1.829	185B	bzip2	0.836	0.589	2.713	368B
cactusADM	13.529	13.348	1.088	81B	gcc	0.640	0.524	2.879	64B
GemsFDTD	13.208	10.846	0.818	1060B	gromacs	0.357	0.336	3.061	1B
hmmer	1.032	0.609	3.017	942B	lbm	25.189	20.803	0.891	13B
leslie3d	7.231	5.898	0.931	176B	libquantum	23.729	22.64	0.558	2666B
mcf	56.755	45.061	0.298	370B	milc	15.624	15.392	0.696	272B
omnetpp	13.594	10.470	0.577	477B	perlbench	0.789	0.628	2.175	541B
soplex	25.242	16.848	0.559	382B	sphinx3	11.586	8.519	0.655	3195B
wrf	5.040	4.434	0.934	2694B	xalancbmk	18.288	10.885	0.311	178B
zeusmp	4.567	3.956	1.230	405B					

Table 3: SPEC CPU 2006 benchmarks with LLC cache misses per 1000 instructions for LRU and optimal (MIN), instructions-per-cycle for LRU for a 2MB cache, and number of instructions fast-forwarded to reach the simpoint (B = billions).

benchmark is run with the first `ref` input provided by `runspec`.

6.1. Single-Thread Workloads

For single-core experiments, the infrastructure simulates one billion instructions. We simulate a 2MB LLC for the single-thread workloads. In keeping with the methodology of recent cache papers [15, 16, 24, 23, 18, 11, 14, 6, 7], we choose a memory-intensive subset of the benchmarks. We use the following criterion: a benchmark is included in the subset if the number of misses in the LLC decreases by at least 1% when using the optimal [3] replacement and bypass policy instead of LRU. Table 3 shows memory intensive SPEC CPU 2006 benchmarks with the baseline LLC misses per 1000 instructions (MPKI), optimal MPKI, baseline instructions-per-cycle (IPC), and the number of instructions fast-forwarded (FFWD) to reach the interval given by SimPoint.

6.2. Multi-Core Workloads

Table 5 shows ten mixes of SPEC CPU 2006 simpoints chosen four at a time with a variety of memory behaviors characterized in the table by cache sensitivity curves. We use these mixes for quad-core simulations. Each benchmark runs simultaneously with the others, restarting after 250 million instructions, until all of the benchmarks have executed at least one billion instructions. For the multi-core workloads, we report the weighted speedup normalized to LRU. That is, for each thread i sharing the 8MB cache, we compute IPC_i. Then we find SingleIPC_i as the IPC of the same program running in isolation with an 8MB cache with LRU replacement. Then we compute the weighted IPC as $\sum \text{IPC}_i/\text{SingleIPC}_i$. We then normalize this weighted IPC with the weighted IPC using the LRU replacement policy.

7. Results

In this section we present results and analysis of decoupled dynamic cache segmentation (DCS) by itself and with the optimizations described in Section 4.

7.1. Effect of Dynamic Segmentation

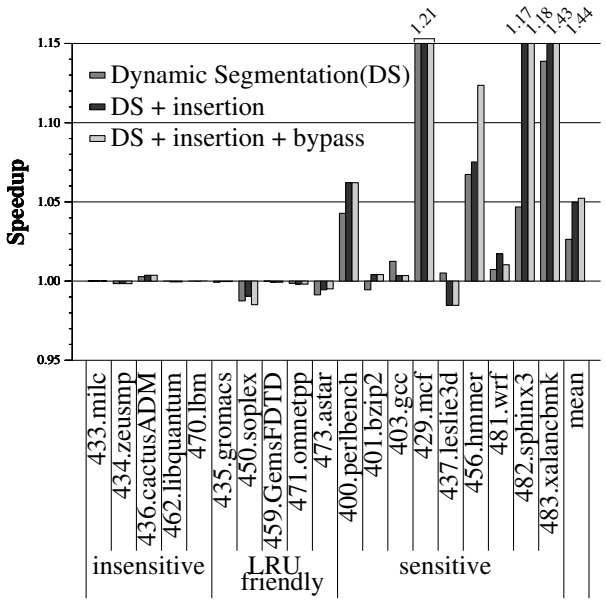

Figure 7: Enforced thrash resistance and automated bypassing

In this section we report how DCS performs in presence of enforced partitioning and automated bypassing. Figure 7 shows the speedup of DCS compared to LRU replacement. DCS uses NRU as the base line replacement policy. The x-axis represents a memory intensive subset of SPEC CPU 2006 benchmarks grouped according to their behavior with respect to static segmentation as illustrated in Table 4. DCS alone achieves a geometric mean 2.4% speedup and re-

Work -load Name	Benchmarks	Type	Cache Sensi- tivity Curve
mix1	*mcf hmmer libquantum omnetpp*	ssif	
mix2	*gobmk soplex libquantum lbm*	sfii	
mix3	*zeusmp leslie3d libquantum xalancbmk*	isis	
mix4	*gamess cactusADM soplex libquantum*	iifi	
mix5	*bzip2 gamess mcf sphinx3*	siss	
mix6	*gcc calculix libquantum sphinx3*	siis	
mix7	*perlbench milc hmmer lbm*	sisi	
mix8	*bzip2 gcc gobmk lbm*	sssi	
mix9	*gamess mcf tonto xalancbmk*	isss	
mix10	*milc namd sphinx3 xalancbmk*	ifss	

Table 5: Multi-core workload mixes with cache sensitivity curves, LLC misses per 1000 instructions (MPKI) on the y-axis for LLC sizes 128KB through 32MB on the x-axis. s=sensitive, i=insensitive and f=LRU friendly

duces misses by 4.3% on average. By inserting the non-referenced lines in the non-recent part of the NRU stack we enforce thrash resistance, improving performance by a geometric mean of 4.9% over LRU and reducing misses by 6.7%. Turning on the automated bypassing yields an average speedup of 5.2% and reduces misses by 7.8%. In general, we can see that DCS can improve performance up to 44% for the sensitive benchmarks. However, it hurts performance for LRU-friendly workloads when the default policy used is NRU. The insensitive benchmarks show no or very minimal effect with DCS.

7.2. Effect of Segment Predictor

We randomly choose 16 sampled sets for each of the leader sets. The counter size is 11 bits for a 2MB LLC. The in-cache configuration uses the original cache sets to predict the best segmentation. The out-of-cache configuration uses 15-bit partial tags represented externally to the cache to reproduce the sampling set behavior. The in-cache configuration improves performance by 4.9% and reduces misses by 7% on average over LRU replacement. The out-of-cache configuration improves performance by 5.2% and reduces misses by 7.8% on average over LRU policy. The out-of-cache configuration has the advantage that it avoids cache sets that use the wrong segmentation in the leader sets. This is why LRU-friendly workloads perform worse with the in-cache predictor than the out-of-cache predictor. However, the out-of-cache predictor is limited by the accesses of the original cache. So if the winning non-referenced segment size is one, the predictor decides which segmentation is better given that the LLC access pattern is the dynamic segmentation policy. The in-cache predictor decides which segmentation is best given that LLC access pattern is following the NRU policy. This is why benchmarks *437.leslie* and *481.wrf* perform poorly in the out-of-cache predictor configuration.

7.3. Comparison with Other Policies

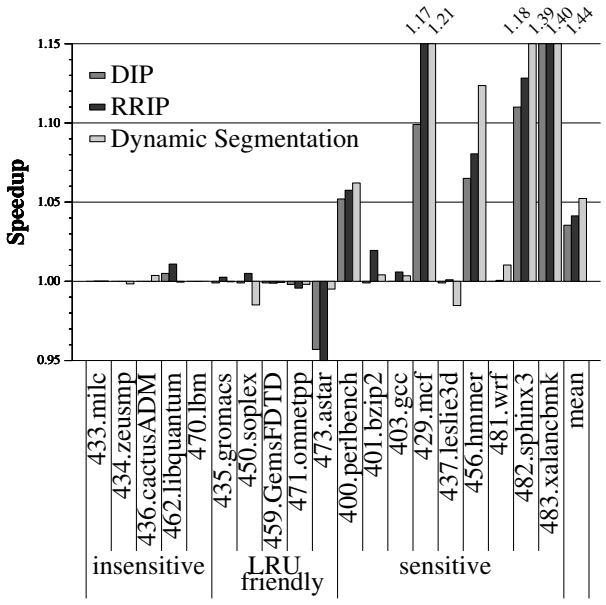

Figure 8: Comparison with other policies

Figure 8 compares DCS with other recent scan and thrash resistant policies. We compare DCS technique with DIP [23] that uses LRU as the baseline policy. We also compare DCS with RRIP [7] that uses two-bit NRU as the baseline policy. DCS uses a one-bit NRU policy. All policies use the same number of leader sets. Figure 8 shows that, compared to LRU replacement, DIP and RRIP achieve geometric mean 3.1% and 4.1% speedups. DCS outperforms both of these techniques, achieving a 5.2% geometric mean speedup.

In Figure 9 we show that DCS with random policy outperforms LRU replacement. Random replacement by itself slows down performance on average by 1%. However, DCS with random replacement improves performance by

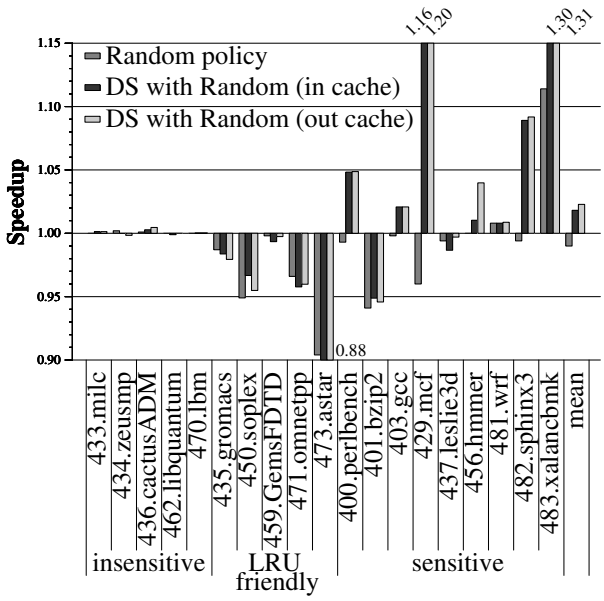

Figure 9: Dynamic Segmentation with Random Policy

	DCS NRU (in)	DCS NRU (out)	DCS Rand (out)
ref bit, per line	1bit	1bit	1bit
repl state, per line	1 bit	1 bit	0 bit
set type, per set	2 bits	2 bits	2 bits
psel counters	23 bits	23 bits	23 bits
cache overhead ((perline×ways +per set)× sets+psel counters)	8.5KB	8.5KB	4.5KB
sampler set entry (partial tag +ref bit+valid bit+repl state)	0 bit	18 bits	17 bits
number of sampler set	0	16	16
total sampler overhead (4 types ×sampler sets×sampler ways)	0	2.25KB	2.12KB
total	8.5KB	10.75KB	6.62KB
% increase in LLC Data Area	0.41%	0.52%	0.32%

Table 6: Space overhead in a 2MB 16 way LLC

1.8% and 2.2% respectively for the in-cache and out-of-cache predictors. DCS coupled with random replacement improves performance up to 33% over LRU.

7.4. Space Overhead

We compare the space overhead of our dynamic cache segmentation (DCS) with LRU, DIP and RRIP in Table 7. LRU and DIP use four bits per cache block for the LRU stack in a 2MB 16 way LLC. RRIP uses two NRU bits per cache block. DCS also uses two bits per cache block. However, it uses one of the bits for segmentation and the other bit for a one-bit NRU policy. For DIP and RRIP, the extra overhead is the bits required to differentiate the leader sets from the follower sets. Both of them need only one counter to set-duel between two policies. DCS with the in-cache predictor uses only 12 bits more space than RRIP. These 12 bits are two extra counters (1 bit and 11 bits) used to choose among five non-referenced segment sizes. DCS with the out-of-cache predictor uses 15-bit partial tags to obtain the best segmentation. The out-of-cache segment predictor has 15 bits of partial tag, one bit for segmentation and one bit for NRU. The out-of-cache predictor uses 2.12 KB. DCS with either in-cache and out-of-cache predictor uses far less space than the LRU policy.

	LRU	DIP	RRIP	DCS NRU (in)	DCS NRU (out)	DCS Rand (out)
per line	4b	4b	2b	2b	2b	1b
extra	0B	257.3B	513.3B	514.8B	2.25KB	2.12KB
total	16KB	16.25KB	8.5KB	8.5KB	10.75KB	6.62KB

Table 7: Comparing space overhead with other policies

7.5. Dynamic Segment Size

In Figure 10, we show the predicted best non-referenced segment size for each of the benchmark. Insensitive benchmarks use a non-referenced segment size of eight most of the time. LRU-friendly benchmarks choose to ignore the segmentation. However, segmentation-sensitive

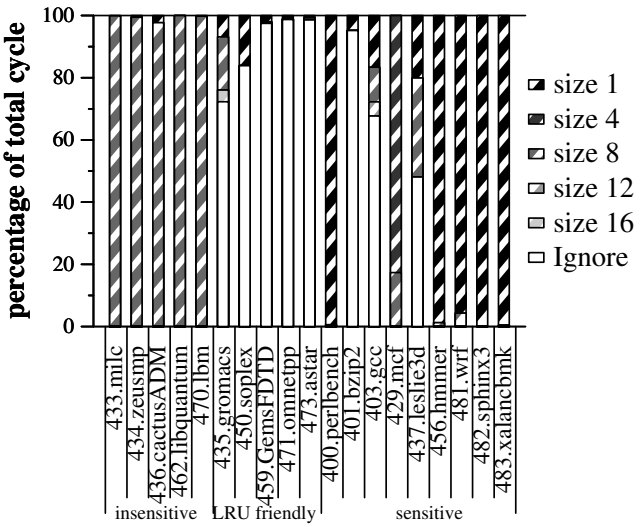

Figure 10: Runtime predicted best non-referenced segment size

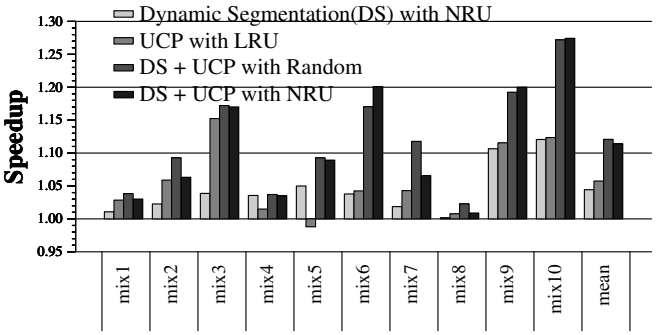

Figure 12: Complementing cache partitioning technique

benchmarks choose different segmentations in various phases. Benchmarks like *400.perlbench*, *482.sphinx3* and *483.xalancbmk* are thrashing workloads and always choose segment size one. *403.gcc*, *429.mcf* and *437.leslie3d* clearly go through different phases and choose different segmentations at runtime.

7.6. Cache Sensitivity

Figure 11 shows DCS performance with several LLC sizes: 1MB, 2MB, 4MB and 8MB. All of the cache configurations are 16-way associative. Our technique with the NRU policy outperforms LRU replacement on average by 5.2-8.7 for various cache sizes.

7.7. DCS with Shared Cache Partitioning

In this section we show that our technique complements current way-partitioning techniques. Figure 12 compares cache segmentation with utility-based cache partitioning (UCP) [25] on multi-core workloads with a 8MB LLC. DCS is not aware of thread-specific characteristics. It adapts to the mixed access pattern from all threads and predicts the best segmentation size for that mixture. Figure 12 shows that it achieves 4.4% normalized weighted speedup compared to LRU replacement. We implement a combined version of DCS and UCP in which UCP predicts the best partitioning for each thread and DCS predicts the best segmentation for each thread. UCP achieves 5.4% weighted speedup over LRU. Complementing UCP with DCS achieves 11.4% speedup using an underlying NRU policy. It also achieves 12% speedup with an underlying random replacement. The space overhead for UCP is 68.75KB. However, our technique with random as the base replacement policy achieves 4.5% performance improvement over UCP with less than half of the space requirement. The space overhead of our technique is 45.25KB with NRU and 28.75KB with random replacement, which is respectively 0.55% and 0.35% of the 8MB LLC capacity. Our technique with NRU and random replacement performs similarly. Segmentation decreases the number of candidates in each set. Victims can only be selected from the specific ways belonging to the specific list of that thread. This essentially makes NRU and random perform very similarly. We have used 32 dedicated sets for UCP and 16 dedicated sets for DCS. We have also compared our technique with Thread Aware RRIP (TA-RRIP) which is not a cache partitioning technique [7]. TA-RRIP outperforms LRU on average by 10.2%. We outperform TA-RRIP even with an underlying random-policy.

8. Conclusion

Many previous studies have proposed a variety of mechanisms to improve LLC performance by adapting to cache access pattern by insertion, zero-reuse prediction and bypass. In this paper we have introduced a single segmentation technique that inherently provides mutable cache management and bypass. We propose a decoupled technique that is scan and thrash resistant even with random replacement policy. Our technique complements existing cache partitioning techniques by segmenting the dynamic partition of each thread on the runtime.

This paper investigates dynamic cache segmentation in the context of single and multi-programmed workloads. In future work we plan to take into account both shared and private data in multi-threaded workloads. Shared data should have an adaptive segment while private data of competing threads should be segmented per thread just like the multi-programmed workloads.

References

[1] A. R. Alameldeen, A. Jaleel, M. Qureshi, and J. Emer. 1st JILP workshop on computer architecture competitions cache replacement championship. http://www.jilp.org/jwac-1/.

[2] S. Bansal and D. S. Modha. Car: Clock with adaptive replacement. In *in Proceedings of the USENIX Conference on File and Storage Technologies (FAST*, pages 187–200, 2004.

[3] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[4] M. Chaudhuri. Pseudo-lifo: the foundation of a new family of replacement policies for last-level caches. In *Proceedings of the 42nd International Symposium on Microarchitecture*, MICRO, pages 401–412, NY, USA, 2009.

[5] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob. CMP$im: A pin-based on-the-fly single/multi-core cache simulator. In *Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)*, June 2008.

[6] A. Jaleel, W. Hasenplaugh, M. K. Qureshi, J. Sebot, S. S. Jr., and J. Emer. Adaptive insertion policies for managing shared caches. In *Proceedings of the 2008 International Conference on Parallel Architectures and Compiler Techniques (PACT)*, September 2008.

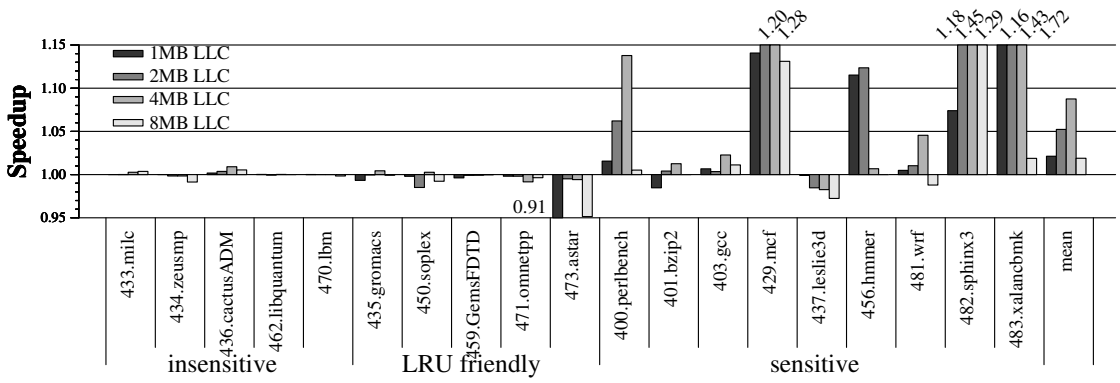

Figure 11: Cache size sensitivity of Dynamic Segmentation

[7] A. Jaleel, K. Theobald, S. S. Jr., and J. Emer. High performance cache replacement using re-reference interval prediction. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA)*, June 2010.

[8] T. Johnson and D. Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In *VLDB*, pages 439–450, 1994.

[9] R. Karedla, J. S. Love, and B. G. Wherry. Caching strategies to improve disk system performance. *Computer*, 27:38–46, March 1994.

[10] G. Keramidas, P. Petoumenos, and S. Kaxiras. Cache replacement based on reuse-distance prediction. In *In Proceedings of the 25th International Conference on Computer Design (ICCD)*, pages 245–250, 2007.

[11] G. Keramidas, P. Petoumenos, and S. Kaxiras. Cache replacement based on reuse-distance prediction. In *ICCD*, pages 245–250, 2007.

[12] S. M. Khan, D. A. Jiménez, B. Falsafi, and D. Burger. Using dead blocks as a virtual victim cache. September 2010.

[13] S. M. Khan, Y. Tian, and D. A. Jiménez. Sampling dead block prediction for last-level caches. In *Proceedings of the International Symposium on Microarchitecture*, MICRO, pages 175–186, USA, 2010.

[14] M. Kharbutli and Y. Solihin. Counter-based cache replacement and bypassing algorithms. *IEEE Transactions on Computers*, 57(4):433–447, 2008.

[15] A.-C. Lai and B. Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. In *International Symposium on Computer Architecture*, 2000.

[16] A.-C. Lai, C. Fide, and B. Falsafi. Dead-block prediction & dead-block correlating prefetchers. *SIGARCH Comput. Archit. News*, 29(2):144–154, 2001.

[17] D. Lee, J. Choi, J. hun Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies. In *In Proceedings of the Conference on Measurement and Modeling of Computer Systems*, pages 134–143, 2001.

[18] H. Liu, M. Ferdman, J. Huh, and D. Burger. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *Proceedings of the International Symposium on Microarchitecture*, pages 222–233, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[19] G. H. Loh. Extending the effectiveness of 3d-stacked dram caches with an adaptive multi-queue policy. In *Proceedings of the 42nd International Symposium on Microarchitecture*, pages 201–212, New York, NY, USA, 2009.

[20] N. Megiddo and D. Modha. Arc: A self-tuning, low overhead replacement cache. In *In Proceedings of the Conference on File and Storage Technologies (FAST)*, 2003.

[21] E. J. O'neil, P. E. O'Neil, G. Weikum, and E. Zurich. The lru-k page replacement algorithm for database disk buffering. pages 297–306, 1993.

[22] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder. Using simpoint for accurate and efficient simulation. *SIGMETRICS Perform. Eval. Rev.*, 31(1):318–319, 2003.

[23] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. S. Jr., and J. S. Emer. Adaptive insertion policies for high performance caching. In *34th International Symposium on Computer Architecture (ISCA),*, 2007.

[24] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A case for mlp-aware cache replacement. In *Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 167–178, Washington, DC, USA, 2006.

[25] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th International Symposium on Microarchitecture*, pages 423–432, Washington, DC, USA, 2006.

[26] S. Srikantaiah, M. T. Kandemir, and M. J. Irwin. Adaptive set pinning: managing shared caches in chip multiprocessors. In *ASPLOS*, pages 135–144, 2008.

[27] G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, HPCA, pages 117–, Washington, DC, USA, 2002.

[28] Y. Xie and G. H. Loh. Pipp: promotion/insertion pseudo-partitioning of multi-core shared caches. In *International Symposium on Computer Architecture*, pages 174–183, 2009.