Adaptive Insertion Policies for Managing Shared Caches

Aamer Jaleel[†] William Hasenplaugh[†] Moinuddin Qureshi[§] Julien Sebot[‡] Simon Steely Jr.[†] Joel Emer[†]

[†]Intel Corporation, VSSAD
Hudson, MA
{aamer.jaleel, william.c.hasenplaugh, simon.c.steely.jr, joel.emer} @intel.com

[§]IBM T. J. Watson Research Center
Yorktown Heights, NY
mkquresh@us.ibm.com

[‡]Intel Israel Design Center
Haifa, Israel
julien.sebot@intel.com

ABSTRACT

Chip Multiprocessors (CMPs) allow different applications to concurrently execute on a single chip. When applications with differing demands for memory compete for a shared cache, the conventional LRU replacement policy can significantly degrade cache performance when the aggregate working set size is greater than the shared cache. In such cases, shared cache performance can be significantly improved by preserving the entire working set of applications that can co-exist in the cache and preserving some portion of the working set of the remaining applications.

This paper investigates the use of adaptive insertion policies to manage shared caches. We show that directly extending the recently proposed dynamic insertion policy (DIP) is inadequate for shared caches since DIP is unaware of the characteristics of individual applications. We propose Thread-Aware Dynamic Insertion Policy (TADIP) that can take into account the memory requirements of each of the concurrently executing applications. Our evaluation with multi-programmed workloads for 2-core, 4-core, 8-core, and 16-core CMPs show that a TADIP-managed shared cache improves overall throughput by as much as 94%, 64%, 26%, and 16% respectively (on average 14%, 18%, 15%, and 17%) over the baseline LRU policy. The performance benefit of TADIP is 2.6x compared to DIP and 1.3x compared to the recently proposed Utility-based Cache Partitioning (UCP) scheme. We also show that a TADIP-managed shared cache provides performance benefits similar to doubling the size of an LRU-managed cache. Furthermore, TADIP requires a total storage overhead of less than two bytes per core, does not require changes to the existing cache structure, and performs similar to LRU for LRU friendly workloads.

Categories and Subject Descriptors B.3.2 [**Design Styles**]: Cache memories, C.1.4 [**Parallel architectures]**

General Terms Design, Performance.

Keywords Shared Cache, Cache Partitioning, Set Dueling, Replacement

§ Moinuddin Qureshi contributed to this work prior to joining IBM Research.

1. INTRODUCTION

High-performance processors typically contain multiple cores on a single chip which allows them to execute multiple applications (or threads) concurrently. As multi-core processors become pervasive, a key design issue facing processor architects is organizing and managing the on-chip last-level cache (LLC). Since shared caches enable more flexible and dynamic allocation of cache resources, recent processors such as Intel's Core Duo [1], IBM's Power 5 [6] and Sun's Niagara [8] have opted for a shared last-level cache (LLC). As the number of cores on chip increases, the contention caused by applications sharing the LLC increases as well. Thus, performance of such systems is heavily influenced by how efficiently the shared cache is managed. The commonly used LRU replacement policy implicitly allocates cache resources to competing applications based on the rate of demand. As a result, it often allocates cache resources to applications that do not benefit from the cache[18][13]. Shared cache performance can be significantly improved from a cache management scheme that allocates cache resources to applications based on benefit rather than rate of demand.

This study focuses on dynamic management of a shared cache among competing applications. There are four goals we seek from such a management scheme: High performance, robustness, scalability and low design overhead. It should have high-performance for a given metric of performance. Since future processors are expected to have a large number of cores, the variety of competing applications in a workload mix is expected to be high. So the proposed cache management policy must not degrade performance (significantly) of workload mixes where the baseline LRU policy works well. Furthermore, since the number of concurrently executing applications is expected to increase with the number of cores, the proposed mechanism must be scalable to a large number of cores. And, finally, from an implementation point of view, the mechanism must have low overhead and avoid extra storage structures so that the area, power, verification, testing and design overheads are minimized. This paper seeks to design such a high-performance, robust, scalable, and negligible hardware overhead mechanism to manage shared caches.

A recent study [12] showed that dynamically changing the insertion policy can provide high-performance cache management for private caches at negligible hardware and design overhead. The proposed Dynamic Insertion Policy (DIP) [12] consists of two component policies: the Bimodal Insertion Policy (BIP) and the traditional LRU policy. BIP is a thrashing-resistant

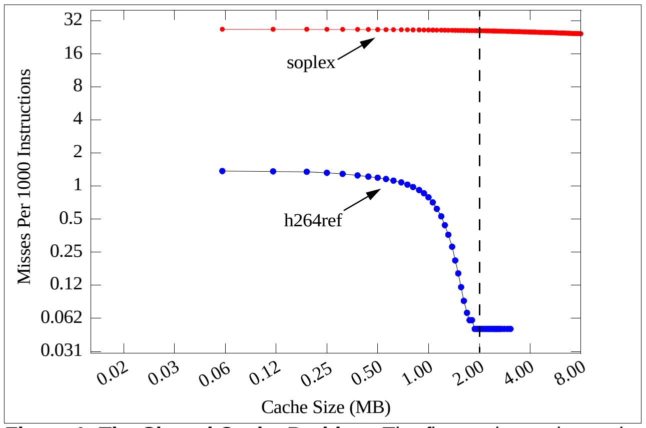

Figure 1: The Shared Cache Problem. The figure shows the cache sensitivity (under LRU) of two SPEC CPU2006 workloads. When both these workloads execute concurrently and share a 2MB cache, *soplex*, a streaming application, interferes with *h264ref*. Cache performance can be improved by reducing the interference.

policy that inserts majority of the incoming lines in the LRU position and few in the MRU position. DIP dynamically chooses between BIP and LRU using Set Dueling Monitors (SDMs). An SDM estimates the misses for any given policy by dedicating a few sets (32 sets) of the cache to always follow that policy. DIP uses the winning policy of the two SDMs (LRU and BIP) for the remaining sets of the cache.

In this paper, we analyze extensions of adaptive insertion for managing shared caches. We show that directly extending DIP to shared caches outperforms the baseline LRU policy but leaves significant room for improvement. Since the direct extension of DIP to shared caches does not differentiate between the behavior of competing applications, DIP is incapable of distinguishing between applications that benefit from the cache and those that do not. We explain this with the following example where two SPEC CPU2006 applications, *h264ref* and *soplex*, share a 2 MB LLC. Figure 1 shows the miss rate curves (under the LRU replacement policy) for both the competing applications when each of them is run separately. The figure shows that *h264ref* continues to benefit from additional cache space till approximately 2MB while *soplex* has little benefit from the cache. When both applications compete for the cache under the LRU policy, LRU can allocate a significant fraction of the cache resources to *soplex*. Cache performance can be improved if more cache space is given to *h264ref* and minimal to *soplex*. This can be achieved by filtering out *soplex* using BIP and retaining *h264ref* using the conventional LRU policy. However, the direct extension of DIP for shared caches uses a single policy for both applications.

To address this shortcoming, we propose the *Thread-Aware Dynamic Insertion Policy (TADIP)*. For a given reference stream, adaptive insertion tries to dynamically choose between two policies: LRU and BIP. Therefore, TADIP needs to make a binary decision (say LRU=0 and BIP=1) for each application executing on a CMP core. When there are N concurrently executing applications[1] competing for a shared cache, the search space of TADIP is an N-bit binary string. TADIP tries to find the best performing binary string out of the 2^N possible strings. When N is small we can use Set Dueling [12] to do a runtime comparison of all possible binary strings and select the best performing one. For example, when N=2 we can dedicate one SDM to each of the four possible insertion policies (00, 01, 10, 11) and use the best insertion policy out of the four for the remaining sets of the cache. However, the number of SDMs increases exponentially with the number of concurrently executing applications making this brute-force approach impractical.

We propose two scalable approaches to avoid the exponential increase in the number of SDMs. These approaches exploit the fact that some of the bits in the best-performing insertion string can be determined independently. For example, applications that do not benefit from the cache can be filtered out by always using BIP. As a result, the exponential search space can be reduced to a linear search space by individually learning the best insertion decision for each application. The first approach, *TADIP-Isolated (TADIP-I)*, learns the insertion policy for each application independently assuming that all other applications are using the LRU policy. The problem with this approach is that the insertion decision of one application may be dependent on the insertion decisions of other concurrently executing applications. The second approach, *TADIP-Feedback (TADIP-F)* improves upon TADIP-I by learning the insertion policy for each application based on the currently best performing insertion policy for the remaining applications.

We evaluate the proposed policies on 2-core, 4-core, 8-core and 16-core systems each sharing a last-level cache of 2MB, 4MB, 8MB, and 16MB respectively. Our results show that a TADIP-managed shared cache improves the average throughput over an LRU-managed shared cache by 14%, 18%, 15%, and 17% for the 2-core, 4-core, 8-core, and 16-core systems respectively. We show that this performance improvement is 2.6x compared to that of DIP and is similar to the performance obtained by doubling the size of the baseline LRU-managed cache. Furthermore, TADIP also outperforms the previously proposed dynamic cache partitioning scheme UCP [13] by 1.3x while avoiding the associated hardware overhead (2KB per core), changes to the existing cache structure, and complexity of UCP. TADIP requires less than two bytes of storage per core, is similar to LRU for LRU friendly workloads, and is scalable to a large number of cores (16 in our evaluation).

2. MOTIVATION

Modern CMPs typically use shared last-level caches to accommodate applications with differing memory requirements. Application performance on such CMPs is governed by how well the shared cache is managed. When multiple applications compete for a shared cache, more cache resources should be allocated to applications that benefit from the cache and fewer to those that do not [18][13]. However, the commonly used LRU policy for managing shared caches is unable to determine whether or not an application benefits from the cache. Instead, the

1. Without loss of generality, we assume one hardware thread per core. Therefore, the maximum number of concurrently executing applications in the system is equal to the total number of cores

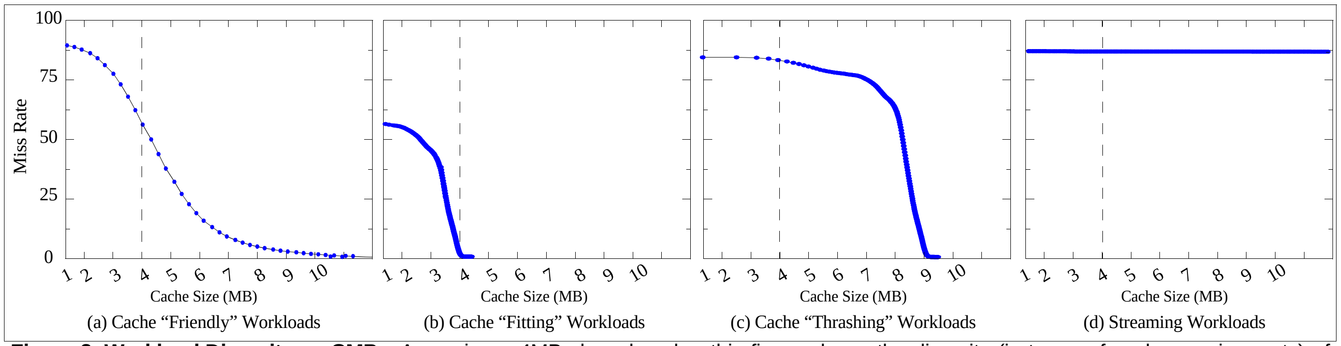

Figure 2: Workload Diversity on CMPs. Assuming a 4MB shared cache, this figure shows the diversity (in terms of cache requirements) of applications that can compete for the shared cache.

LRU policy treats all application misses uniformly and allocates cache resources based on their rate of demand. As a result, applications that have no benefit from cache receive cache resources that could otherwise have been used by applications that benefit from the cache. This suggests the need for a cache management scheme that judiciously allocates cache resources to applications based on benefit rather than rate of demand.

Figure 2 illustrates cache miss rate as a function of cache size for the different types of applications that can share a cache. Assuming that these applications run on a CMP with a 4MB shared cache, we can divide the applications into four categories:

- "Cache Friendly" Applications: These applications benefit from cache and continue to do so with additional cache space. When these applications share the cache with similar applications, the LRU policy works well.
- "Cache Fitting" Applications: These applications require the bulk of the shared cache to perform well. When executing concurrently with other applications, these applications can cause thrashing under the LRU policy when the cache size given to them is less than their working set size. When such applications execute concurrently with cache friendly applications, it is better to reduce the cache resources allocated to them when they are thrashing.
- "Cache Thrashing" Applications: These applications have a working set size that is greater than the size of the available shared cache. These applications cause cache thrashing under the baseline LRU policy but may provide better cache performance using some other replacement policy. Under the baseline LRU policy these applications have no benefit from the cache. When such applications execute concurrently with "cache friendly" or "cache fitting" applications, it is better to reduce the cache resources allocated to them since they have no benefit from the available cache.
- "Streaming" Applications: These applications have extremely large working set sizes and poor cache reuse. They cause thrashing under any replacement policy. When such applications execute concurrently with any other application, it is better to minimize cache resources allocated to them since they do not benefit from the cache.

A recurring theme for improving shared cache performance is to reduce cache contention by reducing cache resources allocated to applications that have little benefit from the cache. Doing so allows concurrently executing applications that benefit from the cache to have better performance. Another recurring theme is that concurrently executing applications can cause cache thrashing. A recent study [12] showed that cache thrashing can be significantly reduced by retaining some fraction of the working set in the cache. This was achieved by modifying the cache insertion policy, i.e. the recency position where the incoming line is placed. They proposed the Bimodal Insertion Policy (BIP) which inserts most incoming lines in the LRU position and very few in the MRU position. BIP provides high performance for thrashing workloads while responding to changes in the working set of the application. A runtime mechanism, DIP [12], chose between LRU and BIP depending on the workload.

Besides avoiding thrashing, BIP can also be used to reduce the amount of cache resources allocated to an application. This is because BIP significantly shortens the lifetime of the lines that are inserted in the LRU position. For two concurrently executing applications A and B, if application A always uses the LRU policy and application B always uses BIP, then application A naturally receives more cache resources because the cache lines of A tend to live longer in the cache than the cache lines of B. Such a strategy can be used to minimize cache contention caused by "cache fitting", "cache thrashing" or "streaming" applications. Similarly, BIP can also be used to retain some portion of the working set of applications whose working set size exceeds the allocated cache space.

For the type of applications shown in Figure 2, adaptive insertion can be used to manage shared caches in the following manner. Since "cache friendly" applications benefit from the cache, using the traditional LRU policy works well for these applications. For "cache fitting" applications, LRU works well if the application can have the required cache capacity. If the required cache capacity is unavailable, thrashing can be avoided by using BIP. For "cache thrashing" applications, BIP helps reduce cache contention and provides an opportunity to receive cache hits by preserving some fraction of the working set in the cache. Finally, for "streaming" applications, BIP provides an easy means of filtering out the contention caused by such streaming applications. Thus, adaptive insertion can be used to obtain high performance from shared caches provided that the per-application insertion policy decisions are made judiciously. The next section describes adaptive insertion policies for managing shared caches.

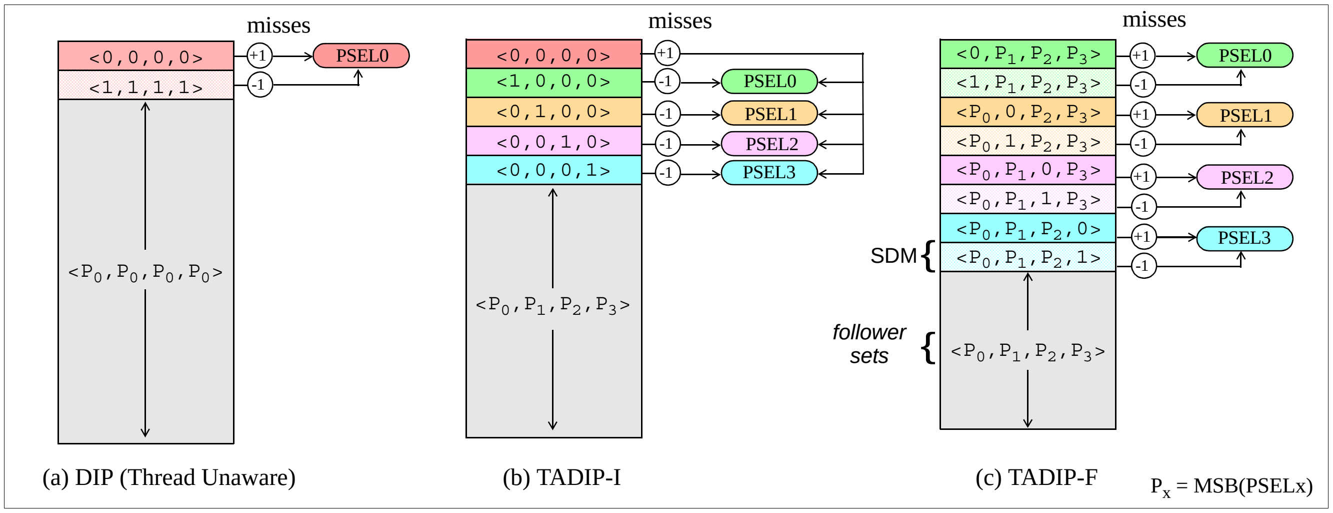

Figure 3: Adaptive Insertion Managed Shared Caches. Three schemes for managing a cache shared four by 4 applications. (a) DIP (b) TADIP-Isolation (c) TADIP-Feedback. Set Dueling Monitors (SDMs) estimate misses for a given policy and follower sets use the best performing policy. Given a binary string <P_0, P_1, P_2, P_3>, the insertion policy for Application 0 is P_0, Application 1 is P_1, and so on. Bimodal insertion policy (BIP) is used when P_x is 1, otherwise the LRU policy is used. P_x is the MSB of a policy selection (PSEL) counter. Both TADIP schemes require a per-core PSEL counter. Note that the figures are not drawn to scale.

3. ADAPTIVE INSERTION MANAGED SHARED CACHES

3.1. Thread-Unaware Adaptive Insertion

When the combined working set size of a workload mix exceeds the shared cache size, the traditional LRU replacement policy can cause cache thrashing. Cache performance can be improved by preserving some portion of the working set in the cache using BIP. We can simply extend the recently proposed Dynamic Insertion Policy (DIP) to determine whether or not a workload mix thrashes the cache. DIP dynamically chooses between BIP and LRU using two Set Dueling Monitors (SDM). An SDM estimates the misses for any given policy by permanently dedicating a few sets[2] of the cache to follow that policy. DIP uses the winning policy of the two SDMs (LRU and BIP) for the remaining sets (also known as *follower* sets) of the cache. Figure 3a illustrates DIP applied to a cache shared by four applications. A saturating counter, PSEL, keeps track of which of the two competing policies is doing well: misses in SDM-LRU increments PSEL while misses in SDM-BIP decrements PSEL. The PSEL value provides insight on whether or not the cache is being thrashed. PSEL values close to zero indicates the workload benefits from the LRU policy. On the other hand, PSEL values close to its maximum saturating value indicates the workload benefits from BIP. The follower sets use the most significant bit (MSB) of the PSEL counter to apply the winning policy to all applications sharing the cache. Since the decisions made by DIP are unaware of different applications running in the system, we call this *Thread-Unaware Dynamic Insertion Policy.* DIP requires two SDMs and one PSEL counter.

2. Prior work has shown that 32 sets are sufficient to estimate cache performance [13][12]. Throughout the paper an SDM consists of 32 sets.

DIP seeks to answer the question: "*Is the workload mix thrashing the cache?*" Since DIP does not differentiate between the behaviors of different applications in a workload mix, DIP is incapable of distinguishing between "cache friendly" and "cache thrashing" applications. As a result, managing shared caches using DIP applies a single policy for all applications that share the cache. Shared cache performance can be improved significantly by applying the LRU policy to "cache friendly" applications and BIP to applications that cause cache thrashing. This motivates the need for a thread-aware adaptive insertion policy.

3.2. Thread-Aware Adaptive Insertion

For a given reference stream, adaptive insertion tries to adapt between two policies: LRU and BIP. A *Thread-Aware Dynamic Insertion Policy (TADIP)* tries to make this binary decision (LRU=0, BIP=1) for each application executing on a core of the CMP. When there are N applications competing for the shared cache, the search space of TADIP is an N-bit binary string with 2^N possible decisions. The objective of TADIP is to converge on an N-bit binary string that performs the best for a given workload mix. The best performing binary string can be determined either statically or dynamically.

3.2.1 Profiling

A static way of finding the best performing binary string is to use profile information. This can be done by executing all 2^N possible combinations of the insertion policies on a target CMP and selecting the combination that yields the best performance. However, it may be impractical to run all 2^N possible combinations for each workload mix, especially when N is large. For example, when N=16 there are 65536 combinations. Therefore, it may be impractical to obtain representative profile information about all the applications that execute concurrently. The primary drawback of a profiling based approach is that the information gathered varies across systems with different cache sizes, is highly sensitive to the choice of input sets (which is only available at run time), and varies across different workloads. As a result, for the purpose of this paper, profiling is considered as an impractical solution.

3.2.2 Brute Force

When N is small we can use Set Dueling to do a runtime comparison of all possible binary strings and select the best. For example, when N=2 we can dedicate one SDM to each of the four possible insertion policies (00, 01, 10, 11) and use the best insertion policy out of the four for the remaining sets of the cache. However, the number of SDMs required increase exponentially with the number of applications making a brute-force approach impractical. For example, when N=16, 64K SDMs are required which is a total of 2 million (64K SDMs * 32 sets per SDM) cache sets—a cache of several gigabytes only for monitoring! Even if all the SDMs were to fit in the cache, dedicating a significant portion of the cache space to analyze different policies degrades performance because a large number of the SDMs continue to use the incorrect policy. In fact, attempting to find the best search string using brute force can degrade overall performance.[3] For the purpose of this paper, brute force is considered as an infeasible solution.

3.3. Scalable Thread-Aware Adaptive Insertion

When the number of concurrently executing applications is large, finding the best performing insertion string requires searching through an exponential number of decisions. The search space can be reduced by exploiting the fact that some of the bits in the best-performing insertion string can be determined independently. For example, applications that do not benefit from the cache (such as streaming applications) should always use BIP. The exponential search space can be reduced to a linear search space by individually learning the best insertion decision for each application in the presence of all other competing applications. We now describe practical mechanisms for implementing TADIP using a linear number of SDMs.

3.3.1 TADIP-Isolated

Our first approach attempts to independently discover whether or not an application benefits from the cache compared to the baseline LRU policy. We call this approach *TADIP-Isolated (TADIP-I)* because it attempts to learn the insertion decision of each application in isolation. For a cache shared by N applications, we use N+1 SDMs to learn the insertion policy of each application independently. The first SDM called the *baseline-SDM* uses the LRU policy for all the applications. The N remaining SDMs, called *bimodal-SDMs*, use BIP for one application and the LRU policy for the remaining applications. Figure 3b demonstrates this scheme for a cache shared by four applications. The baseline-SDM uses the binary string <0,0,0,0> implying that all applications use the LRU policy. For the four bimodal-SDMs, each bimodal-SDM uses the binary string <1,0,0,0>, <0,1,0,0>, <0,0,1,0>, and <0,0,0,1> respectively. The <1,0,0,0> bimodal-SDM uses BIP for the first application and the LRU policy for all other applications, the <0,1,0,0> bimodal-SDM uses BIP for the second application and the LRU for all other applications, and so on. Note that the hamming distance between the baseline-SDM and any bimodal-SDM is one. In other words, only one insertion policy is different between the baseline-SDM and any given bimodal-SDM. This helps TADIP-I decide the insertion policy for the given application.

TADIP-I decides the best insertion policy for each application by dedicating a PSEL counter to each application. Misses in the baseline-SDM increments all PSEL counters while misses in the bimodal-SDM only decrements the PSEL counter dedicated to the application. On a miss in the follower sets, the MSB of the PSEL counter of the miss causing application decides the insertion policy. If the MSB of PSEL is zero, LRU is used otherwise BIP is used.

The per-application PSEL counter values provide insight on whether or not the particular application is causing thrashing. Thrashing applications can be identified based on PSEL values that are close to maximum. Applying BIP to such applications provides two benefits: (a) it preserves the working set of applications that can co-exist in the cache (b) it preserves some portion of the working set of the thrashing application in the remaining portion of the cache.

TADIP-I can be thought of as a cache learning about each application: "*In the presence of other applications, is the given application degrading overall cache performance under the LRU policy? If so, use BIP for this application.*"

3.3.2 TADIP-Feedback

The problem with TADIP-I is that the insertion decision of one application may be dependent on the insertion decisions of another application. For example, consider three applications X, Y, and Z sharing a 2MB cache. Application X needs 0.5MB, application Y needs 1MB, and application Z needs 3MB. Although applications X and Y can co-exist in the cache, the presence of application Z can cause thrashing for application Y. Independently evaluating the insertion policies of applications X, Y, and Z using TADIP-I can result in BIP for Y. However, if application Z is doing BIP, then application Y can fit in the cache using the LRU policy. To incorporate such feedback, we propose *TADIP-with Feedback (TADIP-F)*.

TADIP-F requires 2N SDMs to learn the best insertion policy for each application given the insertion policies for all other applications. Of the 2N SDMs, a pair of SDMs are owned by every application. One of the SDMs in the pair is used for LRU and the other for BIP. For the owner applications, the LRU-SDM always uses the LRU policy and the BIP-SDM always uses BIP.

3. The performance overhead of SDMs can be reduced by using separate hardware structures external to the cache. However, this approach is expensive in terms of hardware, area, power, and verification overhead.

For the remaining applications, the SDMs use the insertion policy that is currently doing the best. Misses in the LRU-SDM increments the SDM owner's PSEL and misses in the BIP-SDM decrements the SDM owner's PSEL. On a miss in the follower sets, the MSB of the PSEL counter of the miss causing application decides the insertion policy.

Figure 3c illustrates TADIP-F for a cache shared by four applications. The cache has eight SDMs, one pair of SDMs for each application. For instance, the two SDMs for the first application use the binary strings $<0,P_1,P_2,P_3>$ and $<1,P_1,P_2,P_3>$ respectively, where P_x is the MSB of $PSEL_x$. The $<0,P_1,P_2,P_3>$ SDM always follows the LRU policy for the first application and the insertion policy predicted by the per-application SDMs for the remaining applications, whereas, the $<1,P_1,P_2,P_3>$ SDM always follows BIP for the first application and the best insertion policy predicted by the per-application SDMs for the remaining applications. Note that the hamming distance of the insertion policy strings between the per-application pair of SDMs is one. Also note that feedback is guaranteed since the decisions produced by one pair of SDMs is directly used by other SDMs.

TADIP-F can be thought of as a cache learning about each application: "*In the presence of other applications using their current insertion policy, is the given application degrading overall cache performance under the LRU policy? If so, use BIP for this application.*"

3.4. Summary of Insertion Policies

Table 1 presents a summary of the insertion policies for managing shared caches. The baseline LRU policy does not require any SDMs. DIP searches in the two extremes: LRU for all applications or BIP for all applications and requires two SDMs and one PSEL counter. To make thread aware decisions, a brute force approach searches through all the possible decisions and requires 2^N SDMs and 2^N counters. The brute force approach is impractical for large N. TADIP-I searches a hamming distance of 1 from the LRU policy and requires N+1 SDMs, but lacks feedback information. Finally, TADIP-F searches hamming distance 1 from the current best decision and requires 2N SDMs. Note that both TADIP-I and TADIP-F incur hardware overhead of N PSEL counters and logic to identify the SDMs.

4. EXPERIMENTAL METHODOLOGY

We use CMP$im [5], a Pin [9] based simulator for our performance studies. Our baseline system is a 4-core CMP with a three level cache hierarchy that does not enforce inclusion. The L1 and L2 caches are private to each core. The L1 instruction and data caches are 2-way 32KB each. The L2 cache is unified 8-way 256KB. Throughout this study, the size of the L1 and L2 caches are kept constant. The last-level cache (L3) is 16-way 4MB and shared by all four cores. All caches use a 64B cache line size. For replacement decisions, the L1 cache uses a true LRU replacement policy while the L2 and L3 cache use the pseudo-LRU policy with most recently used (MRU) bits [2]. Only demand references to the cache update the LRU state while non-demand references (e.g. write back references) leave the LRU state unchanged. The latency for accessing the L2 cache is 10 cycles, L3 cache is 24 cycles and memory is 350 cycles. To keep the simulations tractable we use a simple in-order core model that has an L1 hit IPC of 1. Without loss of generality, we assume that each application runs on one core.

4.1. Benchmarks

For our study, we use 17 SPEC CPU2006 benchmarks compiled using the *icc* compiler with full optimization flags. We created 15 workload mixes by combining four different SPEC CPU2006 benchmarks to form one workload. Simulation continues till all benchmarks in the workload mix have executed 1 billion instructions. When a benchmark reaches 1 billion instructions, it continues to execute so that all applications compete for the shared cache. However, statistics are collected only for the first one billion instructions of each benchmark. Table 2 lists the SPEC CPU2006 benchmarks used in the study and Table 3 lists the workload mixes. Figure 4 provides the cache sensitivity study of each application.

4.2. Metrics

For our studies we use the three metrics commonly used in literature for measuring the performance of multiple concurrently executing applications: throughput, weighted speedup, and fairness. The weighted speedup metric indicates reduction in execution time [16]. The "harmonic mean fairness" metric (which is harmonic mean of normalized IPCs) balances both fairness and performance [10]. The different metrics are defined as follows:

$$\text{Throughput} = \sum IPC_i \qquad \text{(Eq. 1)}$$

$$\text{Weighted Speedup} = \sum (IPC_i/SingleIPC_i) \qquad \text{(Eq. 2)}$$

$$\text{Harmonic Mean Fairness} = N/\sum (SingleIPC_i/IPC_i) \qquad \text{(Eq. 3)}$$

Table 1: Comparison of Adaptive Insertion Policies

Policy	Insertion Policy Search Space	# Set Dueling Monitors	# Counters
LRU Replacement	<0, 0, 0,... 0>	0	0
Dynamic Insertion	<0, 0, 0,... 0> and <1, 1, 1,... 1>	2	1
Brute Force Approach	<0, 0, 0,... 0>... <1, 1, 1,... 1>	2^N	2^N
TADIP-I Approach	<0, 0, 0,... 0> and Hamming Distance of 1	N+1	N
TADIP-F Approach	$<P_0, P_1, P_2 ... P_{N-1}>$ and Hamming Distance of 1	2N	N

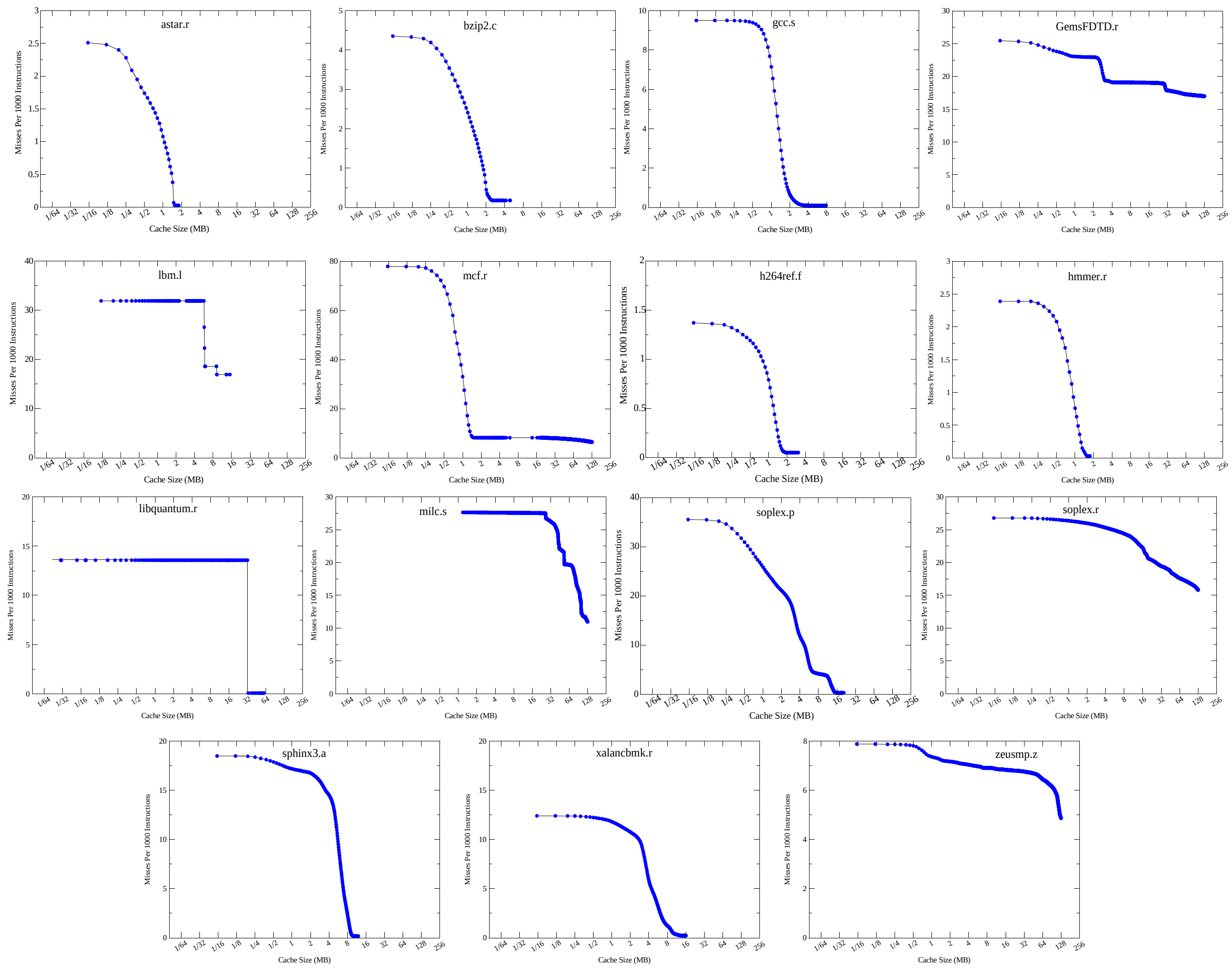

Table 2: Benchmarks

SPEC CPU2006 Benchmarks (Reference Input Sets)
gamess.c, perlbench.d, h264ref.f, hmmer.r, astar.r, gcc.s, bzip2.c, mcf.r, GemsFDTD.r, sphinx3.a, xalancbmk.r, soplex.p, lbm.l, zeusmp.z, libquantum.r, soplex.r, milc.s

Table 3: Workload Mixes

Mix Name	Benchmarks in Mix
MIX_00	xalancbmk.r | mcf.r | milc.s | gcc.s
MIX_01	sphinx3.a | mcf.r | hmmer.r | soplex.r
MIX_02	bzip2.c | zeusmp.z | lbm.l | hmmer.r
MIX_03	soplex.r | xalancbmk.r | h264ref.f | astar.r
MIX_04	libquantum.r | milc.s | soplex.p | bzip2.c
MIX_05	soplex.r | sphinx3.a | soplex.p | bzip2.c
MIX_06	zeusmp.z | h264ref.f | gcc.s | soplex.p
MIX_07	astar.r | GemsFDTD.r | hmmer.r | gcc.s
MIX_08	zeusmp.z | xalancbmk.r | sphinx3.a | soplex.r
MIX_09	mcf.r | xalancbmk.r | astar.r | perlbench.d
MIX_10	libquantum.r | soplex.p | perlbench.d | hmmer.r
MIX_11	soplex.p | milc.s | libquantum.r | zeusmp.z
MIX_12	lbm.l | xalancbmk.r | soplex.r | milc.s
MIX_13	zeusmp.z | libquantum.r | soplex.r | milc.s
MIX_14	gamess.c | perlbench.d | h264ref.f | hmmer.r

Figure 4: Cache Sensitivity of Individual SPEC CPU2006 applications. The applications *gamess.c* and *perlbench.d* are not shown because they have very few cache misses.

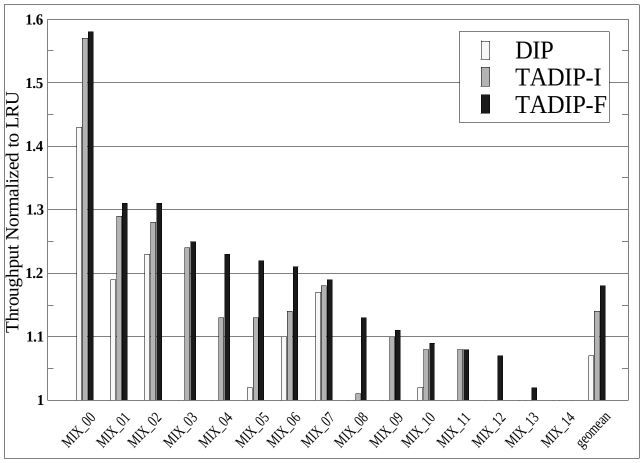

Figure 5: Comparison of TADIP for Throughput Metric.

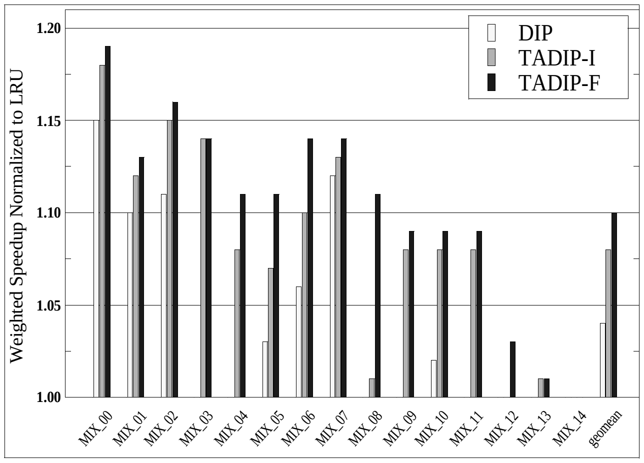

Figure 6: Comparison of TADIP for Weighted Speedup.

where IPC_i is the IPC of the *i*th application when it concurrently executes with other applications and $SingleIPC_i$ is IPC of the same application in isolation.

5. RESULTS AND ANALYSIS

5.1. Throughput

Figure 5 shows the throughput of the three adaptive insertion policies (DIP, TADIP-I, and TADIP-F) normalized to the baseline LRU policy. The x-axis represents the different workload mixes. The bar labeled geomean is the geometric mean of all 15 workloads. DIP improves throughput by more than 10% for four out of the fifteen workloads. For Mix-0, DIP improves throughput by 42%. TADIP-I provides further improvement compared to DIP. For example, for the Mix-3, Mix-4, Mix-5, and Mix-9 workloads DIP does not improve any performance, whereas, TADIP-I improves performance significantly. TADIP-I improves performance by more than 10% for eight out of the 15 workloads. The addition of feedback improves the performance of TADIP-I. For example, for Mix-4, Mix-5 and Mix-8 workloads, TADIP-F improves throughput by more than 10% when compared to TADIP-I. Note that the Mix-13 and Mix-14 workloads do not receive any benefit from adaptive insertion. This is because the Mix-13 workload contains only streaming applications and the Mix-14 workload contains applications whose combined working-set fits into the shared cache. For such mixes cache performance can not be improved compared to LRU and the proposed policy performs similar to LRU. On average, DIP improves throughput by 7%, TADIP-I improves throughput by 14% and TADIP-F improves throughput by 18%. This illustrates the importance of making insertion policies that are not only thread-aware but also incorporate feedback.

5.2. Weighted Speedup

Figure 6 shows the performance of DIP, TADIP-I, and TADIP-F insertion policies on the weighted speedup metric. The values are normalized to the weighted speedup of the baseline LRU policy. DIP improves performance by more than 10% for four out of the 15 workloads. TADIP-I improves performance by more than 10% for six out of the 15 workloads. By adding feedback, TADIP-F improves performance further and allows 9 out of the 15 workloads to have more than 10% performance improvement. On average, DIP improves weighted speedup by 4%, TADIP-I improves weighted speedup by 8% and TADIP-F improves weighted speedup by 10%. Again, we notice that a thread-aware insertion policy with feedback provides the best performance.

5.3. Fairness

Adaptive insertion schemes can improve the performance of some benchmarks at the expense of hurting other benchmarks in the workload mix. The fairness metric proposed by [10] considers both fairness and performance. Figure 7 shows the performance of the baseline LRU policy, DIP, TADIP-I, and TADIP-F for the fairness metric. The baseline LRU policy has an average fairness metric value of 0.80, DIP of 0.84, TADIP-I of 0.89, and TADIP-F of 0.91. Thus, on average, we notice that adaptive insertion with feedback improves the fairness metric by 12% compared to LRU.

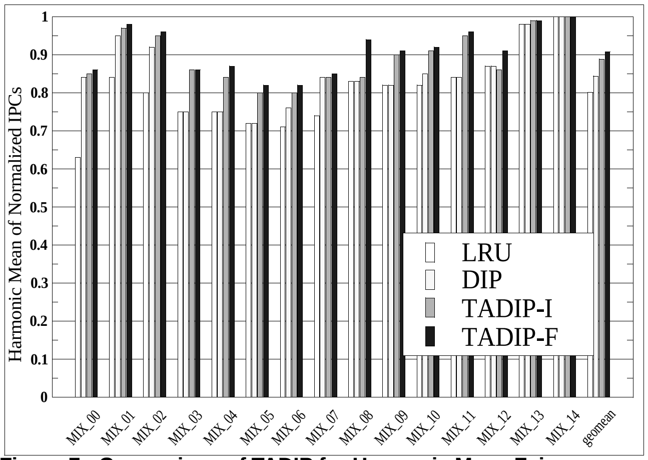

Figure 7: Comparison of TADIP for Harmonic Mean Fairness.

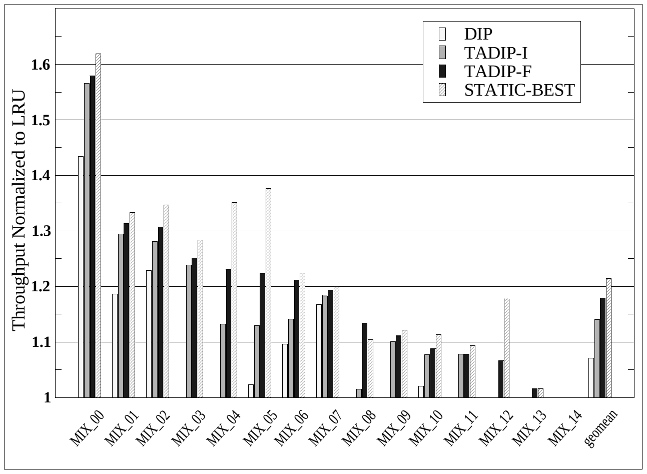

Figure 8: Comparison of TADIP to Profile Based “Static Best”.

5.4. Static Profiling

We compare our adaptive insertion policies to an insertion policy that uses statically generated profiling information. Profiling has the advantage of statically exploring all possible insertion decisions. However, such decisions are dependent on input sets, applications, and cache sizes. Nonetheless, a static policy can serve as a reference for comparing the performance of TADIP.

For our workload mixes, we statically run all 16 possible insertion policies. For each workload mix, the binary string with the best throughput is used as the “Static-Best” performance. Figure 8 compares the throughput performance of DIP, TADIP-I, and TADIP-F with Static-Best. For the MIX-08 workload, TADIP-F outperforms the Static-Best policy by 4% as it can dynamically adapt to changing phase behavior. For the remaining workloads, Static-Best performs better than all the adaptive insertion policies because it optimizes for the throughput metric while TADIP-F minimizes cache misses. Though reducing misses to memory is a good proxy for improving system performance, the cost of a miss to memory can vary within a single application [14] and even more so across multiple applications[4]. Variations of TADIP-F that optimize for throughput instead of cache misses can bridge the gap between Static-Best and TADIP-F.

Besides Static-Best optimizing for a different metric, other reasons for the difference between Static-Best and TADIP-F include: (a) the overhead of experimentation due to SDMs can degrade performance even when the global decision is identical to the Static-Best decision, (b) TADIP-F does not search exhaustively through the entire search space (c) per-application sampled SDMs do not completely reflect application behavior.

Despite the reasons mentioned above, the figure shows that on average TADIP-F achieves 82% of the performance improvement of the Static-Best policy (Static-Best improves performance by 22% and TADIP-F improves performance by 18%). Since TADIP-F does not require any prior knowledge of the workload mix or the choice of input sets and can adapt to different system configurations, it is a practical design alternative. For the rest of the paper, we only show results for TADIP-F.

4. The IPC loss for a given cache miss can vary across applications. For example, if Application A accesses the L2 cache more often than Application B. And if both A and B access the L3 cache at the same rate and have similar MPKI, then the IPC of A will be lower than the IPC of B. A miss for B will cause a greater IPC loss than a miss for A. TADIP-F can be made to optimize for throughput instead of misses by increasing or decreasing the PSEL counter by a value of the estimated IPC-loss instead of a constant value of 1.

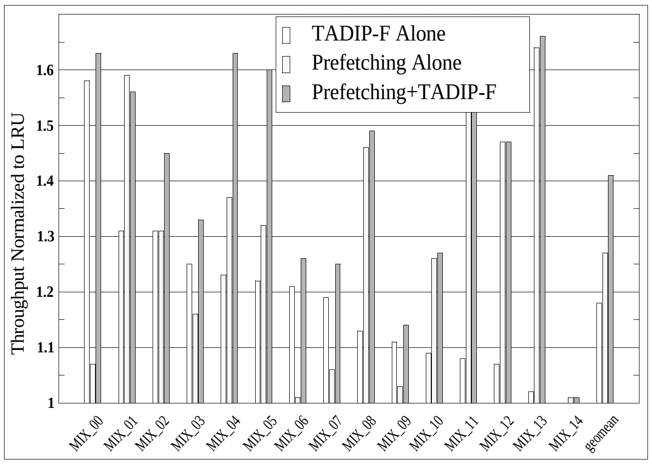

Figure 9: Performance Impact of TADIP-F Under Prefetching.

5.5. Sensitivity to Cache Size

Cache performance can not only be improved by managing cache resources intelligently but also by increasing the total cache capacity. Figure 10 shows the throughput normalized to a baseline 4MB shared cache for the following four cache configurations: 4MB shared cache with TADIP-F, 8MB shared cache with LRU policy, 8MB shared cache with TADIP-F, and finally 16MB shared cache with LRU policy. Increasing the size of an LRU managed cache from 4MB to 8MB and 4MB to 16MB improves the average throughput by 15% and 36% respectively. An adaptive insertion managed cache of 4MB increases throughput by 18% and an adaptive insertion managed cache of 8MB increases throughput by 33%. Thus, TADIP-F is particularly attractive as it provides performance benefits similar to an LRU managed cache twice its size while requiring negligible design and storage overhead (discussed in Section 5.7).

5.6. Interaction with Prefetching

Thus far we have focused on reducing cache misses by managing the shared cache intelligently. Hardware prefetching is yet another commonly used technique that improves cache performance by learning the access patterns in the reference stream. If the misses saved by TADIP belong only to the category that can easily be prefetched, then the misses saved by TADIP may not translate into performance improvement. Therefore, it is important to analyze the interaction between TADIP and prefetching. To study this interaction, for this section only, we add a per-core stream prefetcher similar to [19] to our baseline. Each core can have a maximum of two outstanding prefetches at anytime. Misses caused by the prefetcher are treated similar to

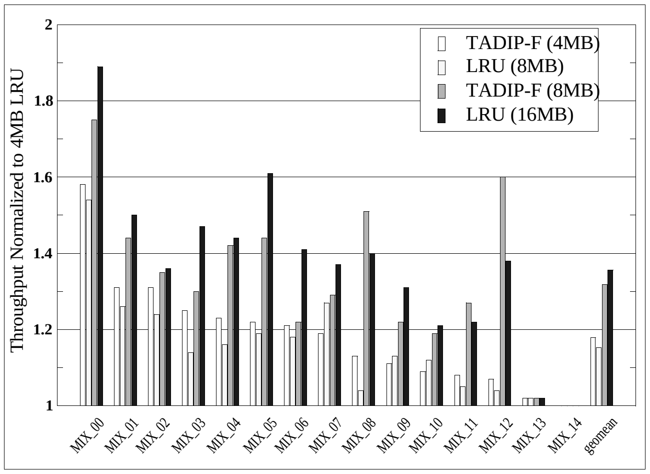

Figure 10: Sensitivity of TADIP to Cache Size.

demand misses[5]. Figure 9 shows the throughput normalized to a baseline 4MB LRU-managed shared cache without prefetching for the following configurations: TADIP-F without prefetching, LRU-managed cache with prefetching, and TADIP-F with prefetching. Prefetching significantly improves performance for 10 out of the 15 workloads. TADIP-F improves performance even in the presence of prefetching. On average, prefetching alone improves performance by 27% and when combined with TADIP-F improves performance by 41%. Thus, TADIP-F improves the performance of a baseline system with prefetching by 11% indicating that TADIP-F is applicable to systems with and without prefetching.

5.7. Hardware Design Overhead

The hardware requirement for TADIP-F includes set selection logic (to identify SDMs) and storage for per-application PSEL counters. Identifying the SDMs can be done either with a CAM or a hash function. CAMs require extra storage. To obviate this storage, we use a hash function that leverages readily available information when the miss occurs: cache set index and requesting core ID. Assuming a cache with 4096 sets, Figure 11 illustrates a hash function to identify LRU SDMs, BIP SDMs, and follower sets. The hash function works as follows. When the sum of the SetIndex[11:7] and ReqCoreID[1:0] equals SetIndex[6:0], the set

5. Further insertion policy optimizations are possible to improve system performance with prefetching. However, such optimizations are beyond the scope of this paper.

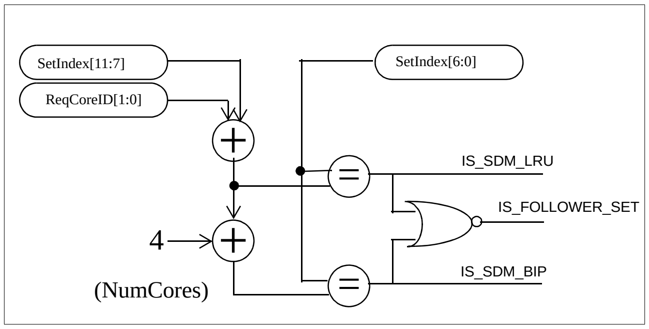

Figure 11: Set Selection Logic to Identify SDMs with TADIP-F .

is dedicated to SDM-LRU for core ReqCoreID of the CMP. Similarly, when the sum of SetIndex[11:7], ReqCoreID[1:0], and NumCores (four in our case) equals SetIndex[6:0], the set is dedicated to SDM-BIP for core ReqCoreID of the CMP. Note that this hash function does not require any storage and only uses two adders and two comparators to identify the SDMs. The only storage required for TADIP-F are the per-core PSEL counters (9-bits in our study). Since TADIP-F does not require any extra bits in the tag store entry, it avoids changes to the existing structure of the cache. Additionally, cache access latency remains unaffected since set selection and insertion logic are not on the critical path.

5.8. Scalability

This section analyzes the scalability of TADIP-F. We compare the performance of TADIP-F for a large number of workload mixes for 2-core, 4-core, 8-core, and 16-core systems with a shared cache of size 2MB, 4MB, 8MB, and 16MB respectively. For each CMP configuration we generate 50 workload mixes by randomly combining from the pool of 17 SPEC CPU2006 applications. Figure 12 shows "s-curves" for the four systems where the different workloads are on the x-axis and the percentage improvement obtained by TADIP-F on the y-axis. TADIP-F improves throughput by up to 95%, 66%, 33%, and 28% on 2-core, 4-core, 8-core, and 16-core systems. An important observation from these "s-curves" is that across the four systems and for the 200 workload mixes, TADIP-F does not degrade performance compared to the baseline LRU policy. Thus, TADIP-F is robust (worst case behavior of LRU) and scalable to a large number of cores (16 in our evaluation). On average, TADIP-F improves throughput for 2-core, 4-core, 8-core, and 16-core systems by 14%, 18%, 15%, and 17% respectively.

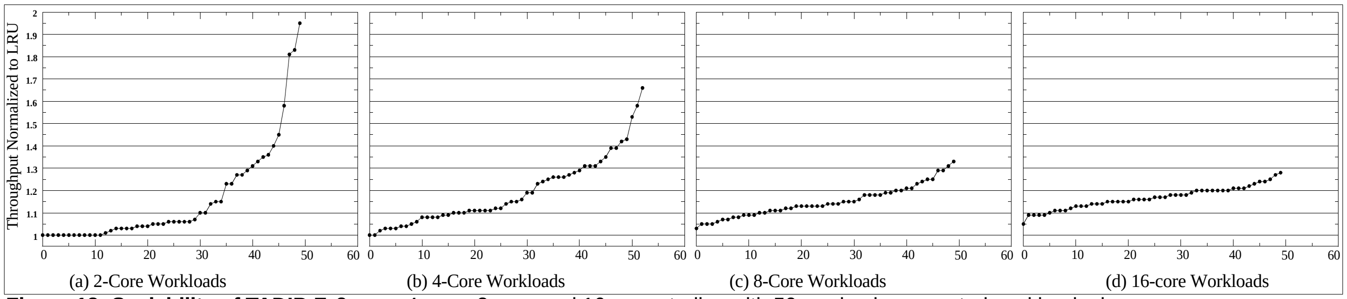

Figure 12: Scalability of TADIP-F. 2-core, 4-core, 8-core and 16-core studies with 50 randomly generated workload mixes.

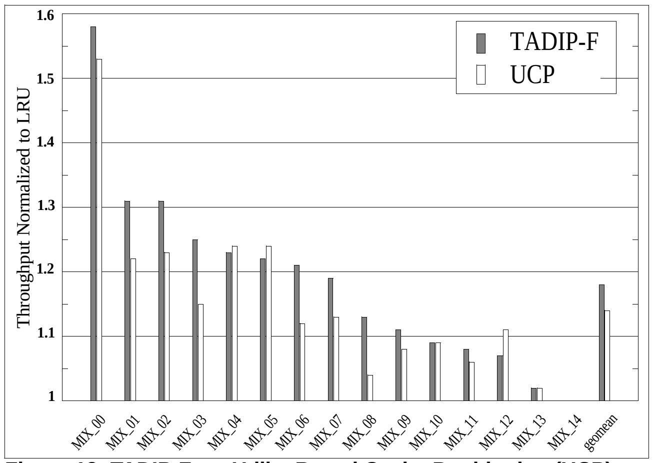

Figure 13: TADIP-F vs. Utility Based Cache Partitioning (UCP).

6. RELATED WORK

With CMPs and shared caches becoming common, there is an active body of research both in industry and academia for improving shared cache performance. One popular way of managing shared caches among competing applications is via cache partitioning. Cache partitioning allocates cache resources to competing applications either statically or dynamically.

Stone et al. [17] investigated optimal (static) partitioning of cache resources between multiple applications when the information about change in misses for varying cache size is available for each of the competing applications. However, such information is non-trivial to obtain dynamically for all applications as it is dependant on the input set of the application. The focus of our study is to dynamically manage cache resources without requiring any prior information about the application.

Dynamic partitioning of shared cache was first studied by Suh et al. [18]. They describe a mechanism to measure cache utility for each application by counting the hits to the recency position in the cache and used way partitioning to enforce partitioning decisions. The problem with way partitioning is that it requires core identifying bits with each cache entry, which requires changing the structure of the tag-store entry. Way partitioning also requires that the associativity of the cache be increased to partition the cache among a large number of applications.

Qureshi et al. [13] improved on [18] by separating the cache monitoring circuits outside the cache so that the information computed by one application is not polluted by other concurrently executing applications. They provide a set-sampling based utility monitoring circuit that requires a storage overhead of 2KB per core and used way partitioning to enforce partitioning decisions. TADIP-F is better able to respond to workloads that have working sets greater than the cache size while UCP does not. For example, when two applications (each with a working set 1.5x the cache size) execute concurrently, UCP causes thrashing under LRU replacement while TADIP-F retains some fraction of the working set for both applications. Figure 13 shows the throughput of the UCP policy and TADIP-F normalized to the baseline LRU policy. For the 15 workloads in our study, UCP provides an average performance improvement of 14% while TADIP-F outperforms UCP by providing an average improvement of 18%. Unlike UCP, TADIP-F does not require the storage overhead of monitoring (2KB per core), changes to the existing cache design for enabling way partitioning, and the complexity of doing look-ahead based partitioning between multiple applications.

Several recent studies have focused on metrics other than cache performance. For example, [7] describe fairness based cache partitioning which uses profile information to do partitioning such that all threads receive equal slowdowns. Mechanisms for partitioning the shared cache are described in detail in [4]. Recent work has also looked at different ways of managing a shared cache using capitalist, socialist, and utilitarian policies. Chang et al. [3] use timeslicing as a means of doing cache partitioning so that each application is guaranteed cache resources for a certain time quantum. Their scheme is still susceptible to thrashing when the working set of the application is greater than the cache size. Nesbit et al. [11] describe virtual private caches that focus on bandwidth and cache resource partitioning to enforce some level of quality of service.

Recent mechanisms [15] have also looked at reducing cache pollution by dynamically changing the insertion position depending on prefetcher accuracy. But this work inserts all incoming demand lines in the MRU position. Hence it is not well suited to managing shared caches. The recently proposed, Dynamic Insertion Policy (DIP), the basis of our proposed policies, provided a low overhead mechanism to provide high performance private caches. We show that a direct extension of DIP to shared caches does not perform as well because it cannot make thread aware decisions. DIP improves the average throughput over LRU by 7% while TADIP-F improves throughput over LRU by 18% while only consuming negligible hardware overhead of less than two bytes of storage per core.

7. SUMMARY AND FUTURE WORK

This paper investigates dynamic management of a shared cache among multiple applications. The commonly used LRU replacement policy can allocate cache resources to applications that have no benefit from the cache, resulting in inefficient use of cache space. We study adaptive insertion policies for managing shared caches and make the following contributions:

1. This is the first work that studies Dynamic Insertion Policy (DIP) for shared caches. We show that DIP improves shared cache performance over LRU. However, since DIP makes thread-unaware decisions, it leaves significant scope for improving performance.
2. We show that *Thread Aware Dynamic Insertion Policy (TADIP)*, which can adapt to requirements of different applications, can significantly outperform DIP. However, a brute force approach for finding the best thread-aware insertion decision requires hardware overhead that increases exponentially with the number of applications.
3. To avoid the exponential overhead, we propose a scalable mechanism, *TADIP-Isolated*, that performs thread-aware

insertion by independently learning the best insertion decision for each application in isolation. However, finding the best insertion decision without taking into account the insertion decisions of other competing applications leaves additional scope for improving performance.

4. We propose a scalable mechanism, *TADIP-Feedback* (TADIP-F), that incorporates feedback from other applications. TADIP-F learns the best insertion decision for each application by using current insertion decisions of all competing applications.

We evaluate TADIP for a variety of systems and for a large number of workload mixes. We show that TADIP-F improves throughput of a 2-core, 4-core, 8-core, and 16-core system by 14%, 18%, 15%, and 17% respectively. Furthermore, for a baseline 4-core CMP configuration with a 4MB shared cache, TADIP-F improves average throughput by 18%, DIP improves throughput by 7%, and the recently proposed cache partitioning policy (UCP) improves throughput by 14%. Unlike previous proposals for cache partitioning, TADIP-F does not require any additional hardware structures and design changes for cache partitioning, and is scalable and robust across a wide variety of workload and systems. Finally, TADIP-F provides performance similar to doubling the size of an LRU-managed cache while requiring a total storage overhead of less than two bytes per core.

This paper evaluates TADIP for a CMP system assuming one thread per core. However, TADIP is also applicable to SMT-enabled systems. In such a system, the total number of concurrently executing applications managed by TADIP will be equal to the total number of hardware threads per core. This paper uses adaptive insertion to minimize misses in a shared cache. Similarly, adaptive cache insertion can also be used to design cache management policies that optimize other metrics such as throughput, fairness, and Quality of Service (QoS). This study uses adaptive insertion to distinguish access streams from multiple applications. However, adaptive insertion can be extended to distinguish between multiple access streams of a single application. For example, different threads of the same application, demand and prefetch stream, instruction and data stream, read and write stream, and private and shared stream. Exploring these extensions is part of our on-going work.

8. ACKNOWLEDGEMENTS

The authors would like to thank Michael Adler, Eric Borch, Michelle Moravan, Angshuman Parashar, Paul Racunas, Ankush Varma and the anonymous reviewers for their valuable feedback in improving the quality of this paper.

9. REFERENCES

[1] Intel Corporation. Next leap in microprocessor architecture: Intel core duo. White paper. http://ces2006.akamai.com.edgesuite.net/yonahassets/CoreDuo_WhitePaper.pdf.

[2] H. Al-Zoubi, A. Milenkovic and M. Milenkovic. Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite. In ACMSE, 2004.

[3] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. ICS-21, 2007.

[4] R. Iyer. CQoS: a framework for enabling QoS in shared caches of CMP platforms. In ICS-18, 2004.

[5] A. Jaleel, R. S. Cohn, C. K. Luk, and B. Jacob. CMP$im: A Pin-Based On-The-Fly Multi-Core Cache Simulator. In MoBS, 2008.

[6] R. Kalla, B. Sinharoy, and J. M. Tendler. IBM Power5 chip: A Dual-Core Multi-Threaded Processor. IEEE Micro, 24(2):40{47, Mar. 2004.

[7] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In PACT-13, pages 111–122, 2004.

[8] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. IEEE Micro, 25(2):21{29, March/April 2005.

[9] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S.Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In PLDI, pages 190–200, 2005.

[10] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in smt processors. In ISPASS, pages 164–171, 2001.

[11] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual private caches. In ISCA-34, pages 57–68, 2007.

[12] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely Jr., and J. Emer. Adaptive insertion policies for high-performance caching. In ISCA-34, 2007.

[13] M. K. Qureshi and Y. Patt. Utility Based Cache Partitioning: A Low Overhead High-Performance Runtime Mechanism to Partition Shared Caches. In MICRO-39, 2006.

[14] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A Case for MLP-Aware Cache Replacement. In ISCA-33, 2006.

[15] S. Srinath, O.Mutlu, H. Kim, and Y. N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In HPCA-13, 2007.

[16] A. Snavely and D. Tullsen. “Symbiotic Jobscheduling for a Simultaneous Multithreading Processor”. In ASPLOS IX, 2000.

[17] H. S. Stone, J. Turek, and J. L. Wolf. Optimal partitioning of cache memory. IEEE Transactions on Computers., 41(9):1054–1068, 1992.

[18] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. Journal of Supercomputing, 28(1):7–26, 2004.

[19] J. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. IBM Technical White Paper, Oct. 2001.