CRUISE: Cache Replacement and Utility-aware Scheduling

Aamer Jaleel[†], Hashem H. Najaf-abadi[*], Samantika Subramaniam[†], Simon C. Steely Jr.[†], and Joel Emer[†‡]

[†]Intel Corporation, VSSAD
Hudson, MA

[*]Intel Corporation, VPG
Folsom, CA

[‡]Massachusetts Institute of Technology (MIT)
Cambridge, MA

{aamer.jaleel, hashem.hashemi, samantika.subramaniam, simon.c.steely.jr, joel.emer}@intel.com

Abstract

When several applications are co-scheduled to run on a system with multiple shared LLCs, there is opportunity to improve system performance. This opportunity can be exploited by the hardware, software, or a combination of both hardware and software. The software, i.e., an operating system or hypervisor, can improve system performance by co-scheduling jobs on LLCs to minimize shared cache contention. The hardware can improve system throughput through better replacement policies by allocating more cache resources to applications that benefit from the cache and less to those applications that do not.

This study presents a detailed analysis on the interactions between intelligent scheduling and smart cache replacement policies. We find that smart cache replacement reduces the burden on software to provide intelligent scheduling decisions. However, under smart cache replacement, there is still room to improve performance from better application co-scheduling. We find that co-scheduling decisions are a function of the underlying LLC replacement policy. We propose Cache Replacement and Utility-aware Scheduling (CRUISE)—a hardware/software co-designed approach for shared cache management. For 4-core and 8-core CMPs, we find that CRUISE approaches the performance of an ideal job co-scheduling policy under different LLC replacement policies.

Categories and Subject Descriptors D.4.1 [Process Management]: Scheduling, B.3.2 [Design Styles]: Cache memories, C.1.4 [Parallel architectures]: Distributed architectures

General Terms Algorithms, Measurement, Performance, Design.

Keywords Scheduling, Cache Replacement, Shared Cache

1. Introduction

Emerging technologies such as virtualization, multi-core, and multi-socket systems have enabled the consolidation of multiple applications onto a single system. In doing so, a wide variety of applications with differing memory demands can concurrently execute and compete for shared resources in the system. Since modern multi-core and multi-socket CMP systems typically contain one or more shared last-level caches (LLC), system performance is typically determined by how well the shared LLC is managed.

ASPLOS'12, March 3–7, 2012, London, England, UK.

Traditionally, there have been two approaches to manage shared LLCs: intelligent software scheduling [37, 38, 31, 32, 7, 15, 17, 18, 21] and smart hardware cache replacement [4, 12, 10, 11, 14, 25, 34, 36]. Software scheduling puts the onus of co-scheduling applications on a layer between the hardware and the applications themselves, like the operating system or a hypervisor. The advantage of software scheduling is that the underlying hardware is agnostic to the scheduling of applications and can be designed independently. However, the drawback is that the scheduler is unaware of the runtime memory requirements of the applications scheduled. To address this problem, researchers have proposed using existing hardware performance counters (or adding new counters) to guide scheduling [31, 21, 38]. For example, a recent proposal measures per-application miss-rate using performance counters and then co-schedules applications based on the degree of memory intensity [38]. However, this approach can cause sub-optimal co-scheduling decisions since per-application miss-rate while sharing a cache can be significantly different from the miss-rate while running in isolation. Nonetheless, hardware performance counter based scheduling algorithms, to some degree, have addressed shared cache contention.

Researchers from the hardware community, on the other hand, have addressed heterogeneity in memory demand by proposing to build resource sharing decisions directly into the cache replacement policy. Such smart replacement policies dynamically allocate more cache resources to applications that benefit from the cache and less cache resources to applications that do not benefit from the cache. While cache replacement does not specify which applications can be co-scheduled to share a cache resource, it can change the impact that these applications have on shared cache usage. Consequently, we believe that it is important to study the interaction between the software and hardware approaches to shared cache management.

This paper presents the first comprehensive study on the interactions between intelligent software scheduling and smart cache replacement. In general, we find that smart cache replacement policies minimize the burden on software to provide intelligent scheduling decisions. However, they do not eliminate the need for intelligent scheduling decisions. Based on our observations, we propose a hardware/software co-designed approach for shared cache management called *Cache Replacement and Utility-aware Scheduling (CRUISE)*. We also propose a novel hardware mechanism, *Runtime Isolated Cache Estimator (RICE)*, which provides a low overhead mechanism to dynamically estimate the isolated cache performance of an application while still sharing the cache with other applications. RICE requires no changes to the existing shared cache structure and merely requires two counters per application. The software component of our proposal uses cache utility information provided by RICE *and* the knowledge of the underlying LLC replacement policy to intelligently co-schedule

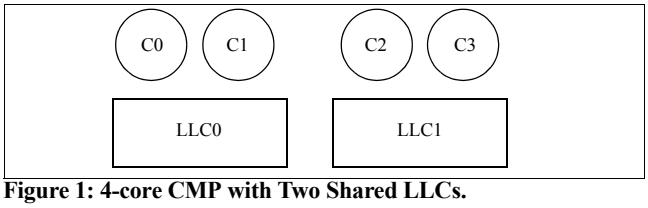

Figure 1: 4-core CMP with Two Shared LLCs.

applications. Our evaluations on 4-core and 8-core CMPs with two and four threads sharing an LLC show that CRUISE consistently provides *near-optimal* co-scheduling decisions.

2. Motivation

Recent studies have shown that the commonly used LRU replacement policy (and its approximations) perform poorly on shared caches [12, 34, 10, 30, 29, 25]. This is because LRU replacement allocates cache resources based on demand instead of cache utility [30, 25, 12]. To improve shared cache performance, researchers from the computer architecture community propose smart replacement policies to improve upon LRU [12, 34, 10, 30, 29, 25]. On the other hand, other researchers have proposed intelligent co-scheduling policies [37, 38, 32, 31, 21] that an operating system or hypervisor can use to minimize shared cache contention.

Figure 2 illustrates a 4-core CMP with two shared LLCs. When four applications (A, B, C, and D) concurrently execute on this CMP, there are three possible *application schedules*: AB|CD, AC|BD, and AD|BC. Depending on the amount of shared cache contention, the three application schedules may observe different system and per-application performance. For a given performance metric (e.g., throughput, weighted speedup), we refer to the application schedule that yields the best system performance as *Optimal Application Schedule (OAS)* and the application schedule that yields the lowest system performance as *Worst Application Schedule (WAS)*.

In our 4-core CMP system, application performance and system performance is largely dependent on the degree of cache contention between the two applications *co-scheduled* (i.e. share) on the same LLC. To avoid the worst application schedule, conventional wisdom prevents co-scheduling a CPU bound application with a memory bound application. However, such practice is only necessary for an LRU managed LLC. In an LLC managed by smart cache replacement, co-scheduling a CPU bound application with a memory bound application no longer hurts the performance of a CPU bound application. This is because smart cache replacement policies allocate cache resources based on utility instead of demand. Since the majority of existing intelligent scheduling studies have been evaluated in the presence of an inefficient LLC replacement policy (i.e., LRU), the question arises as to whether intelligent application scheduling is necessary in the presence of smart cache replacement.

In efforts to understand the interaction between cache replacement and application scheduling, Figure 2 illustrates the effects of optimal scheduling in the presence and absence of smart cache replacement. The study is based on 1365 4-core heterogeneous SPEC CPU2006 application mixes simulated on our 4-core CMP illustrated in Figure 1. We use DRRIP [10], a recently proposed low overhead, high performing shared cache replacement policy.

In the figure, the x-axis represents the system performance ratio OAS_{LRU}/WAS_{LRU} while the y-axis represents the system performance ratio OAS_{DRRIP}/WAS_{DRRIP}. This ratio (illustrated as a percent) represents the performance variability that intelligent scheduling policies attempt to minimize. We use system throughput as our performance metric. Each data point in the figure represents a 4-core workload mix. Some mixes are emphasized using large shapes (circle, square, diamond, and triangle) to highlight representative behavior. Circles are in "Region I", squares are in "Region II", diamond in "Region III", and finally triangle in "Region IV".

An interesting observation from the figure is the behavior of workloads in Region II (emphasized by squares). Under LRU replacement, these workloads have significant performance variation (up to 28% on the x-axis). Hence, under LRU, these workloads significantly benefit from intelligent scheduling decisions. However, under a smarter replacement policy, like DRRIP, these workloads have lesser performance variation (<4% on the y-axis). Consequently, under DRRIP, there is little benefit from improving scheduling decisions. Upon inspection, these workload mixes consist of memory bound applications paired with CPU bound applications. Smart cache replacement policies, such as DRRIP, are particularly designed to address such workload mixes.

In general, the figure shows that the majority of workload mixes lie below the linear bisector. Such behavior suggests that smarter cache replacement policies minimize the burden on software (i.e. operating system or hypervisor) to provide intelligent co-scheduling decisions. However, note that smarter replacement policies do not eliminate the need for intelligent scheduling decisions. For example, workloads in Region IV (emphasized by triangle) can have as much as 20% performance variation under both LRU and DRRIP. Furthermore, for workload mixes in region III, smart replacement policies introduce up to 20% performance variation. Note that these workload mixes had less than 4% performance variation under LRU replacement. Upon inspection, these workload mixes consist of several non-memory bound applications that benefit from more intelligent co-scheduling decisions.

For the workloads studied, we find that DRRIP alone improves performance over LRU by roughly 4.8%. Intelligent scheduling by itself in an LRU-managed cache improves performance by 4.4%. We find there is up to 8.4% performance potential when intelligent scheduling is employed in a DRRIP-managed cache. This shows that

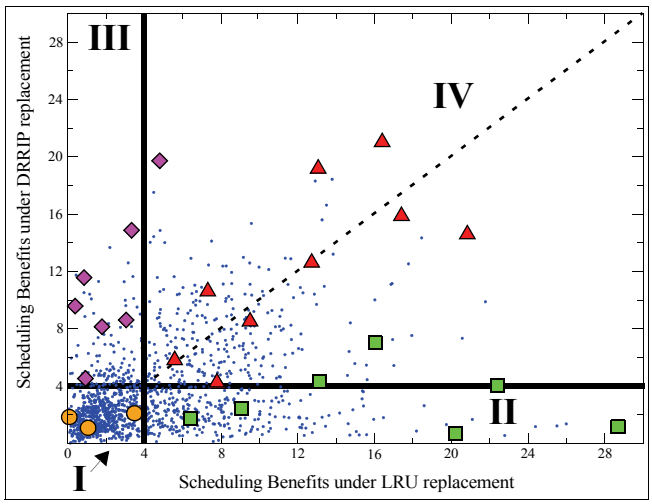

Figure 2: Behavior of Optimal Co-Scheduling in Presence and Absence of Intelligent Replacement. Some workload mixes are emphasized with different sized shapes and are analyzed more closely in the results section of this paper.

the underlying replacement policy *and* intelligent scheduling combined can together improve performance significantly. The next section proposes such an intelligent scheduling policy.

3. Cache Replacement and Utility-aware Scheduling (CRUISE)

Determining a suitable application co-schedule requires runtime knowledge on the cache requirements of an application in isolation [31, 38]. The next section describes our run-time mechanism to determine isolated cache *utility* of an application. Given the cache utility information, we then present our *Cache Replacement and Utility-aware Scheduling (CRUISE)* proposal that co-schedules applications based on knowledge of the underlying LLC replacement policy. Finally, we discuss how an operating system (or hypervisor) can incorporate CRUISE to improve system performance.

3.1. Classifying Application Cache Utility

Recent work used *colors* [18] and *animals* [35] as analogies to characterize the memory intensity of applications. Instead of using analogies, we propose to classify applications based on their cache performance on the available CMP cache hierarchy. We categorize application cache utility into four different categories:

- "Core Cache Fitting" (CCF) Applications: These applications have a working set size that fits in the smaller levels of the cache hierarchy. CCF applications have no benefit from the shared LLC.
- "LLC Thrashing" (LLCT) Applications: These applications have a working set size that is greater than the available LLC. We consider *streaming* applications as LLCT applications. Under LRU, LLCT applications degrade performance of any application that benefits from the shared LLC.
- "LLC Fitting" (LLCF) Applications: These applications require the majority of the available shared LLC capacity to perform well. If these applications do not receive the bulk of the shared LLC, like LLCT applications, their performance degrades significantly due to cache thrashing. As such, LLCF applications perform best when running in isolation or when sharing the LLC with CCF applications. LLCF application performance degrades when co-executed with any other application.
- "LLC Friendly" (LLCFR) Applications: These applications benefit from the available shared LLC and continue to do so as they are given more cache resources. Unlike LLCF applications, LLCFR application performance does not degrade significantly when they do not receive the bulk of the shared LLC. LLCFR application performance degrades only when co-executed with LLCT or LLCF applications.

Based on the above application classifications, intelligent scheduling decisions can be developed to manage a system with multiple shared LLCs. For example, since CCF applications do not require the shared LLC, co-scheduling them with LLCF applications can significantly improve LLCF application performance. Similarly, it is best to co-schedule LLCFR applications with LLCFR applications.

Before we provide a detailed description of our proposed application co-scheduling policies, we first discuss mechanisms to classify application cache *utility*.

3.1.1 Profiling based Classification of Applications

Cache utility of an application can be determined statically using profile information. Applications can be executed beforehand and classified into one of the four categories described above. However, the primary drawback of a profiling based approach is that the information gathered is highly sensitive to the choice of input sets (which is only available at run time), application phase, and also varies across different types of applications.

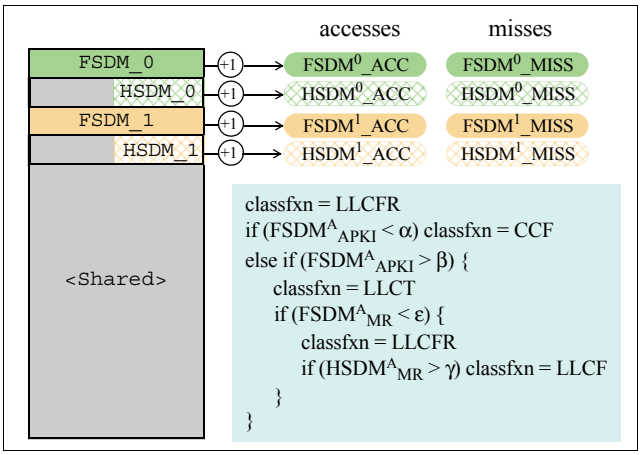

Figure 3: Runtime Isolated Cache Estimator (RICE) and Classification Algorithm. In the classification algorithm, to compute APKI, we multiply FSDMA_ACC by (total sets / SDM size). For this paper, we use $\alpha = 1$, $\beta = 4$, $\varepsilon = 25$, $\gamma = 50$.

3.1.2 Runtime Classification of Applications

A straightforward way of classifying application cache utility in isolation is to periodically pause all cores on a CMP and measure the application cache performance. However, this approach degrades performance of all other concurrently executing applications. Alternatively, external shadow tags can be used to monitor cache utility [25, 35]. However, such approaches require extra hardware and power overhead for the shadow tags. To avoid additional hardware overhead and changes to the existing shared cache structure, we propose a *Runtime Isolated Cache Estimator (RICE)*. RICE *estimate*s cache utility of an application by using a *Set Dueling Monitor (SDM)* [24]. An SDM estimates the misses for any given policy by permanently dedicating a few sets[1] of the cache to follow that policy. RICE dedicates two SDMs per application in the cache. For each SDM, two 32-bit counters track the number of accesses and misses to that SDM (illustrated in Figure 3). The first SDM, referred to as Full-SDM (FSDM) estimates the caching behavior of an application if it were to have sole access to the cache. The FSDM for an application A ($FSDM^A$) only allows A to allocate lines in these cache sets. All other applications are required to *bypass*[2] these cache sets. The second SDM, referred to as Half-SDM (HSDM) estimates the caching behavior of an application if it were to have sole access to only half the cache. All other applications can share the remaining ways in the HSDM. An HSDM can be implemented using way partitioning [25]. For example, in $HSDM^A$ of a 4-way set associative cache, application A only allocates lines in way 0 and way 1 while all other applications allocate lines in way 2 and way 3.

1. Prior work has shown that 32 sets are sufficient to estimate cache performance [25, 24]. Throughout the paper an SDM consists of 32 sets.
2. For an inclusive LLC, a single way in each set can be dedicated for all other applications. This effectively causes the SDM to behave like a direct mapped cache for all other applications.

RICE can be enabled permanently or can be software controlled through privileged instructions. Providing the dynamic on/off ability reduces RICE overhead especially when the FSDM or HSDM tend to be "hot" sets for other applications. The drawback however is that privileged instructions must be introduced into the instruction set architecture (ISA) to explicitly enable or disable RICE.

When RICE is enabled, periodically sampling the RICE counters and calculating the accesses per kilo instructions (APKI) and miss rate (MR) metrics in each SDM can be used to classify the cache utility of application A at runtime. For example, if the $FSDM^{A}_{APKI}$ of the application is small (e.g. < 1), then the application is a CCF application. On the other hand, if the $FSDM^{A}_{APKI}$ of the application is high, the $FSDM^{A}_{MR}$ is small but the $HSDM^{A}_{MR}$ miss rate is high, then the application is a LLCF application. However, if the $FSDM^{A}_{APKI}$ of the application is high and the $FSDM^{A}_{MR}$ is also very high, then the application is an LLCT application. If none of the above conditions apply, then the application can be classified as a LLCFR application. Note that the primary use of the HSDM is to classify an LLCF application. Without the HSDM, an LLCF application would incorrectly be classified as an LLCFR application. Since LLCF applications behave like LLCT applications in the presence of any other application, it is imperative that such applications be classified correctly. A summary of our runtime application classification algorithm is provided in Figure 3.

3.2. Cache Replacement and Utility-aware Scheduling

Given statically profiled or dynamically gathered cache utility information of all concurrently executing applications, we now discuss the design of an intelligent scheduling algorithm that is *co-designed* to the underlying shared LLC replacement policy.

3.2.1 CRUISE for Demand-based Cache Replacement

LRU-managed shared caches allocate cache resources based on demand instead of cache utility. As a result, under LRU replacement, LLCT applications receive more cache resources despite the fact that they do not benefit from the cache. Since LLCT applications can degrade the performance of LLCF and LLCFR applications, it is best to co-schedule them with LLCT or CCF applications.

When scheduling CCF applications, it is best to separate these applications across all available shared LLCs. Since CCF applications require little LLC resources, such a strategy allows co-scheduled applications to utilize more of the LLC.

When scheduling LLCF applications, it is best to co-schedule them with CCF applications. This is because an LLCF application requires majority of the shared cache while a CCF application requires little LLC resources. In the absence of CCF applications, LLCF applications behave like LLCT applications and thus should be scheduled in a similar fashion as LLCT applications.

Finally, LLCFR applications perform well when co-scheduled with other applications that benefit from the cache. Once LLCT, CCF, and LLCF applications are scheduled onto the appropriate LLCs, LLCFR applications can be co-scheduled anywhere.

Since an LRU-managed shared LLC allocates cache resources on demand, we propose *CRUISE-LRU (CRUISE-L)*. To ensure a proper schedule, steps must be followed in the order listed below:

1. Group LLCT applications on the same LLC
2. Spread CCF applications across all LLCs
3. Co-schedule LLCF with CCF applications
4. Fill in the LLCFR applications

During the scheduling process, if all computing resources on one shared LLC are occupied, then the scheduling algorithm overflows onto the next available LLC (selected at random). Additionally, if there exists more than one application from the same category, then the scheduling algorithm selects an application at random.

3.2.2 CRUISE for Utility-based Cache Replacement

Unlike LRU, DRRIP-managed shared caches allocate cache resources based on utility instead of demand. As a result, LLCT applications receive very few cache resources when co-scheduled with applications that benefit from the shared cache. Consequently, in a DRRIP-managed cache, LLCT applications can be treated like CCF applications that do not benefit from the available cache.

When co-designing the scheduling policy for an LRU-managed cache, an entire shared cache was used to "contain" LLCT applications and prevent them from hurting LLCFR and LLCF applications. In doing so, CRUISE-L is inefficient since it can "waste" one or more shared caches. However, in a system with more than one DRRIP-managed shared cache, we propose *CRUISE-DRRIP* (*CRUISE-D)* to efficiently utilize all available shared caches. Again, to ensure a proper schedule, steps must be followed in the order listed below:

1. Spread LLCT applications across all LLCs
2. Spread CCF applications across all LLCs
3. Co-schedule LLCF with CCF/LLCT applications
4. Fill in the LLCFR applications

Note that CRUISE-D is similar to the recently proposed *Distributed Intensity (DI)* proposal [38]. Like DI, CRUISE-D separates LLCT applications onto separate LLCs. However, CRUISE-D differs from DI in that it explicitly detects LLCF applications and co-schedules them intelligently to maximize system performance.

3.2.3 CRUISE-L vs. CRUISE-D

CRUISE-L and CRUISE-D primary differ in how they co-schedule LLCT applications. CRUISE-D treats LLCT applications like CCF applications. This is because a DRRIP managed cache naturally allocates fewer cache resources to LLCT applications since they do not benefit from cache.

Note that while CRUISE-L is co-designed for an LRU-managed cache, CRUISE-L is applicable for any un-managed cache replacement policy. Similarly, though CRUISE-D is co-designed for a DRRIP-managed cache, CRUISE-D is applicable for any utility-based cache replacement policy. Therefore, applying CRUISE-L to a utility-managed cache or applying CRUISE-D to an un-managed cache will yield suboptimal co-schedules.

3.3. Integrating CRUISE into Existing Software

Existing software (i.e., operating systems or hypervisor) can utilize CRUISE by using statically profiled utility information or by dynamically learning the application utility if RICE hardware is available. If an operating system implements CRUISE using statically profiled cache utility information, then the natural opportunity to determine the best application co-schedule is when the operating system (or hypervisor) schedules a new application onto a CPU (e.g., after a context switch). With statically profiled cache utility information, co-scheduling decisions are limited to context switches and/or when an application finishes execution. As a result, the operating system (or hypervisor) cannot adapt to dynamic application phases.

Table II. MPKI of Representative SPEC CPU2006 Applications In the Absence of Prefetching

	ast	bzi	cal	dea	gob	h26	hmm	lib	mcf	per	pov	sje	sph	wrf	xal
L1 MPKI (64KB)	29.29	19.48	21.19	0.95	10.56	11.26	4.67	38.83	21.51	0.42	15.08	0.99	19.03	16.50	27.80
L2 MPKI (256KB)	17.02	17.44	14.06	0.22	7.91	1.57	2.76	38.83	20.43	0.20	0.18	0.37	16.20	15.18	3.38
LLC MPKI (4MB)	2.02	1.05	0.51	0.05	7.70	0.03	0.80	34.28	18.72	0.60	0.01	0.30	1.21	8.23	1.72
Classification (4MB)	LLCFR	LLCFR	LLCFR	CCF	LLCT	LLCFR	LLCFR	LLCT	LLCT	CCF	CCF	CCF	LLCF	LLCT	LLCFR

Alternatively, if the underlying processor supports RICE, then an operating system (or hypervisor) can dynamically adapt to application phase behavior. To do so, the operating system can periodically read and classify all applications using the RICE counters. Upon classifying application cache utility, the operating system (or hypervisor) can periodically apply CRUISE.

4. Experimental Methodology

We use CMP$im [9], a Pin [19] based trace-driven x86 simulator for our performance studies. Our baseline system is a 4-core CMP with two shared LLCs (see Figure 1). Each core in the CMP is a 4-way out-of-order processor with a 128-entry reorder buffer and a three level cache hierarchy. We assume single-threaded cores with the L1 and L2 caches private to each core. The L1 instruction and data caches are 4-way 32KB each while the L2 cache is unified 8-way 256KB. The L1 and L2 cache sizes are kept constant in our study. We support two L1 read ports and one L1 write port on the data cache. Each last-level cache (LLC) is a unified 16-way 4MB cache that is shared by two cores in the CMP. We assume a banked LLC with as many banks as there are cores in the system. All caches in the hierarchy are non-inclusive and use a 64B line size. For replacement decisions, the L1 and L2 caches always use the Not Recently Used[3] (NRU) replacement policy. To evaluate interactions with intelligent scheduling, we only vary the LLC replacement policy between NRU and DRRIP [10]. We model a stream prefetcher that trains on L2 cache misses and prefetches lines into the L2 cache (i.e. the prefetcher is private to each core). The prefetcher has 16 stream detectors. The load-to-use latencies for the L1, L2, and LLC are 1, 10, and 24 cycles respectively. We model an interconnect with a fixed average latency. Bandwidth onto the interconnect is modeled using a fixed number of MSHRs. Contention for the MSHRs models the increase in latency due to additional traffic introduced into the system. We use a queuing model to model off chip contention. We model a 150 cycle unloaded latency penalty to main memory and support 16 outstanding misses to memory. The cache hierarchy organization and latencies are based on the Intel Core i7 processor [2]. Note that our proposed scheduling policies do not rely on the specific latencies used.

We do not use an operating system, but instead augment our CMP$im model with an application scheduler. Applications are initially assigned to the first available free CPU at the beginning of simulation. Our dynamic scheduler periodically (every 1ms) determines whether applications need to be re-scheduled. When classifying an application at runtime, RICE uses hysteresis to ensure steady state. This is to prevent CRUISE from doing frequent reschedules due to slight phase changes in the application. When an application is re-scheduled to a new CPU, we model all compulsory misses to warm up all levels of the cache hierarchy on the new CPU (and possibly new LLC). Additionally, when CRUISE attempts to reschedule applications, CRUISE tries to minimize the number of applications that need to be rescheduled. This is to minimize the compulsory miss overhead for all applications.

3. NRU is the hardware approximation for LRU replacement. NRU performs similar to LRU for a wide variety of workloads [10] and is the commonly used LLC replacement policy in majority of microprocessors today [10].

4.1. Benchmarks

For our study, we use benchmarks from the SPEC CPU2006 suite. We first grouped the SPEC CPU2006 benchmarks into four different categories based on their L1, L2, and LLC cache hit behavior. Of all the SPEC CPU2006 benchmarks, we selected few applications from each category to cover the spectrum of hit/miss behavior in the different levels of the cache hierarchy. A total of 15 representative SPEC CPU2006 benchmarks were selected. Each benchmark was compiled using the *icc* compiler with full optimization flags. Representative regions for the SPEC benchmarks were all collected using PinPoints [23]. Table II lists the 15 SPEC CPU2006 benchmarks and their misses per 1000 instructions (MPKI) in the L1, L2, and LLC when run in isolation. To illustrate application cache utility, the MPKI numbers are reported in the absence of a prefetcher.

To evaluate our proposed scheduling algorithms, we ran all possible four-threaded combinations of the 15 SPEC CPU2006 benchmarks, i.e. 15 choose 4—1365 workloads. To provide insights on when scheduling policies are beneficial, we selected 26 workload mixes (listed in Table I) to showcase results. These 26 workload mixes correspond to the same workload mixes emphasized with different shapes in Regions I, II, III, and IV of Figure 2. Recall that Region I workload mixes have no performance variation under different application schedules. Region II workload mixes have significant performance variation under LRU-managed shared caches. These workload mixes mostly consist of LLCT applications. Region III workload mixes have significant performance variation under DRRIP-managed shared caches. These workload mixes mostly consist of LLCFR applications. Finally Region IV workload mixes

Table I. Workload Mixes and Their Types

Name	Apps	Type	Name	Apps	Type
MIX_00	mcf,pov,sje,wrf	I	**MIX_13**	bzi,cal,h26,mcf	III
MIX_01	bzi,dea,per,xal	I	**MIX_14**	bzi,hmm,mcf,sph	III
MIX_02	ast,bzi,gob,hmm	I	**MIX_15**	ast,cal,sph,wrf	III
MIX_03	bzi,dea,gob,mcf	II	**MIX_16**	ast,cal,mcf,sph	III
MIX_04	ast,h26,hmm,lib	II	**MIX_17**	cal,per,sph,xal	IV
MIX_05	lib,sje,sph,wrf	II	**MIX_18**	ast,cal,dea,sph	IV
MIX_06	gob,lib,mcf,sph	II	**MIX_19**	hmm,lib,mcf,wrf	IV
MIX_07	lib,sph,wrf,xal	II	**MIX_20**	hmm,mcf,per,sph	IV
MIX_08	lib,mcf,sph,xal	II	**MIX_21**	ast,lib,sph,gob	IV
MIX_09	lib,mcf,pov,sph	II	**MIX_22**	ast,lib,sph,wrf	IV
MIX_10	ast,cal,sph,xal	III	**MIX_23**	ast,lib,sph,mcf	IV
MIX_11	cal,h26,mcf,sph	III	**MIX_24**	ast,lib,sph,per	IV
MIX_12	bzi,h26,wrf,xal	III	**MIX_25**	ast,lib,sph,pov	IV

have performance variation under both replacement policies. These workload mixes consist of both LLCT and LLCFR applications. To provide a thorough analysis, we also provide results for all 1365 workloads wherever applicable.

We simulated half billion instructions for each benchmark. Simulations continue to execute until all benchmarks in the workload mix execute at least half billion instructions. If a faster thread finishes its required instructions, it continues to execute to compete for cache resources. We only collect statistics for the half billion instructions committed by each application. This methodology is similar to existing work on shared cache management [25, 12, 34].

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 [28]. The "harmonic mean fairness" metric (which is harmonic mean of normalized IPCs) balances both fairness and performance [20]. The different metrics are defined as follows:

$$\text{Throughput} = \Sigma \text{IPC}_i \quad \text{(Eq. 1)}$$

$$\text{Weighted Speedup} = \Sigma(\text{IPC}_i/\text{SingleIPC}_i) \quad \text{(Eq. 2)}$$

$$\text{Harmonic Mean Fairness} = N/\Sigma(\text{SingleIPC}_i/\text{IPC}_i) \quad \text{(Eq. 3)}$$

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

To decouple the performance of CRUISE and the accuracy of our application classifier RICE, we first illustrate CRUISE using cache utility information from static profiling. In doing so, we can measure the overall accuracy of the CRUISE algorithm that is specifically co-designed to the replacement policy. RICE is just our low overhead hardware classification mechanism.

Figure 4 presents the relative throughput of five different application scheduling policies (*random*, *CRUISE-L, CRUISE-D*, *Distributed Intensity (DI)*[4] [38], and *Optimal Application Schedule (OAS)*) normalized to the *Worst Application Schedule (WAS)*. Random schedule refers to selecting a co-schedule at random from the different possible co-schedules. The x-axis represents the different workload mixes. The bar labeled geomean is the geometric mean of all 26 workloads. In an LRU-managed LLC (Figure 4a), as expected, workload mixes from category 'I' do not benefit from scheduling since all applications in these workload mixes do not benefit from the shared LLC. Workload mixes from category III also do not benefit from scheduling because they mostly consist of LLCFR applications. For such applications, LRU works best. Workload mixes from categories II and IV benefit from intelligent scheduling decisions because they consist of LLCT applications. In these workloads, we find that CRUISE-L performs very similar to OAS. In some scenarios CRUISE-L does not perform as well as OAS (e.g. MIX_04, MIX_06, MIX_19). CRUISE-L reaches suboptimal decision because these workload mixes consist of multiple applications that belong to the same cache utility category (e.g.,

4. For DI, we use isolated cache miss rate based on profile information.

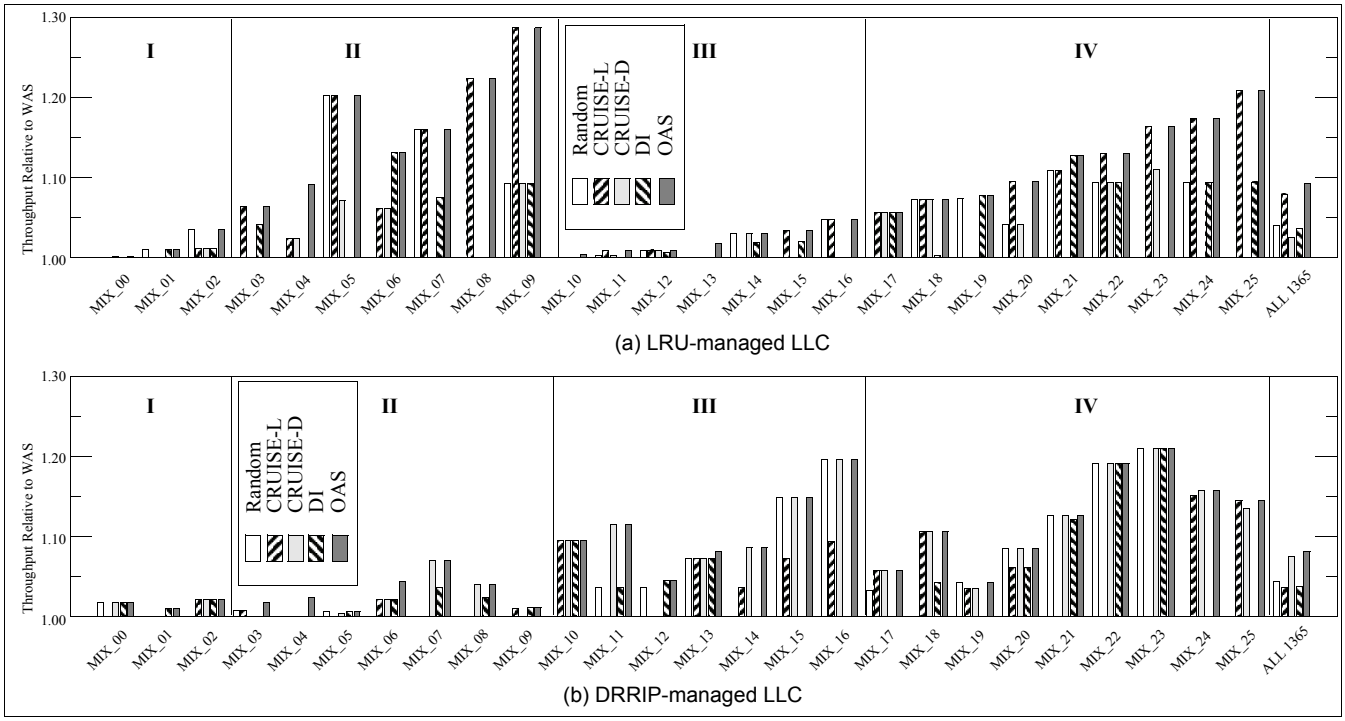

Figure 4: Throughput Comparison of Scheduling Policies to Worst Application Schedule (WAS) on a 4-core CMP. (a) LRU-managed LLC (b) DRRIP-managed LLC

multiple LLCFR applications). When co-scheduling applications, CRUISE randomly selects an application and does not distinguish between applications that belong to the same cache utility category. Though further enhancements can be made to CRUISE to make this distinction, the first order benefits are from distinguishing applications that belong to different cache utility categories.

Additionally, the figure also shows that in an LRU-managed cache, CRUISE-D is not as high performing as CRUISE-L (e.g. MIX_05, MIX_07, MIX_20). Unlike CRUISE-L, CRUISE-D is specifically designed for a utility managed cache and thus does not perform well on a demand managed cache.

The figure also shows that DI scheduling does not perform as well as CRUISE-L (e.g. MIX_05, MIX_07, MIX_09). This is because these workload mixes consist of an LLCF application, and DI incorrectly co-schedules LLCF applications. For example, consider MIX_05, the workload mix consists of applications *lib,sje,sph,wrf*. Based on cache miss rates, DI co-schedules these applications as *lib,sje|sph,wrf*. However, the OAS is *sph,sje|lib,wrf*. This is because *sph* is an LLCF application that benefits from being co-scheduled with *sje*, a CCF application. Furthermore, DI has suboptimal performance under LRU because it co-schedules memory bound applications with CPU bound applications. In doing so, DI degrades CPU bound application performance. As a result, DI is unable to arrive at the OAS.

On average in an LRU-managed cache, DI performs similar to random scheduling, both performing roughly 4% better than WAS. CRUISE-D and CRUISE-L perform roughly 3% and 8% better than WAS respectively. CRUISE-L bridges 90% of the gap between WAS and OAS—OAS performs roughly 9% better than WAS.

Figure 4b shows the performance of the different scheduling policies on a DRRIP-managed cache. Recall that DRRIP allocates cache resources based on utility instead of demand. We again observe that workloads from category 'I' do not benefit from scheduling since these mixes do not benefit from the shared LLC. Unlike an LRU-managed cache, workload mixes in category II do not benefit from scheduling because DRRIP inherently *knows* how to handle LLCT applications in a workload mix. However, CRUISE-D finds opportunity to improve workload mixes from categories III and IV because they consist of multiple LLCFR and LLCT applications. In a DRRIP-managed cache, we again observe CRUISE-D almost always outperforms CRUISE-L. In a utility managed shared LLC, random, DI, CRUISE-L, and CRUISE-D improve performance over WAS by 4.4%, 3.7%, 3.8%, and 7.5% respectively. Again, note that CRUISE-D bridges 90% of the performance gap between WAS and OAS—OAS performs roughly 8% better than WAS.

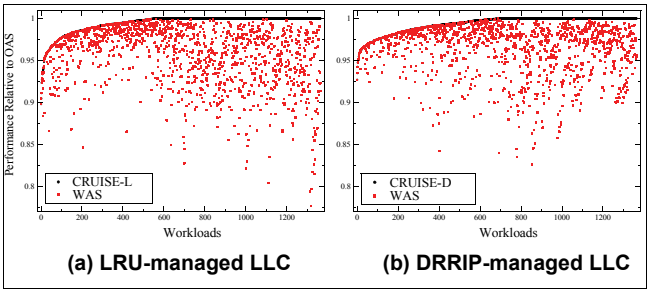

Figure 5: Per workload Comparison of CRUISE to WAS.

To illustrate the behavior of CRUISE-L and CRUISE-D across a much wider set of workload mixes, Figure 5 shows per-workload performance comparison of each scheduling policy on an LRU-managed and DRRIP-managed cache. The x-axis in the figures show the 1365 workload mixes and the y-axis shows the performance relative to OAS. Values at 1 imply that CRUISE application schedule is identical to OAS, and values below 1 illustrate the relative throughput difference between CRUISE and OAS. Each graph is sorted in ascending order based on the CRUISE algorithm. Both figures show that CRUISE significantly reduce the performance variation between WAS and OAS. For workload mixes where the performance variation is significantly high, CRUISE frequently arrives at an optimal schedule. In both LRU-managed and DRRIP-managed shared caches, the majority of performance variation is less than 5%. For cases where the performance variation is greater than 5%, further refinement can be made by intelligently selecting between applications that belong to the same cache utility category.

5.2. Weighted Speedup and Fairness

Again, assuming statically profiled utility information, Figure 6 and Figure 7 presents the throughput, weighted speedup, and fairness comparison of CRUISE on both LRU-managed and DRRIP-managed caches. Each figure shows the workloads on the x-axis and the corresponding performance metric on the y-axis. Like Figure 5, the throughput and weighted speedup metrics in these figures are normalized to OAS. The throughput and weighted speedup figures compare five different scheduling policies: *WAS, random*, *CRUISE-L, CRUISE-D, and DI*. We present 's-curves' independently sorted for each scheduling algorithm.

In an LRU-managed cache, across all 1365 workloads, for the throughput and weighted speedup metrics, there is roughly 0.5%

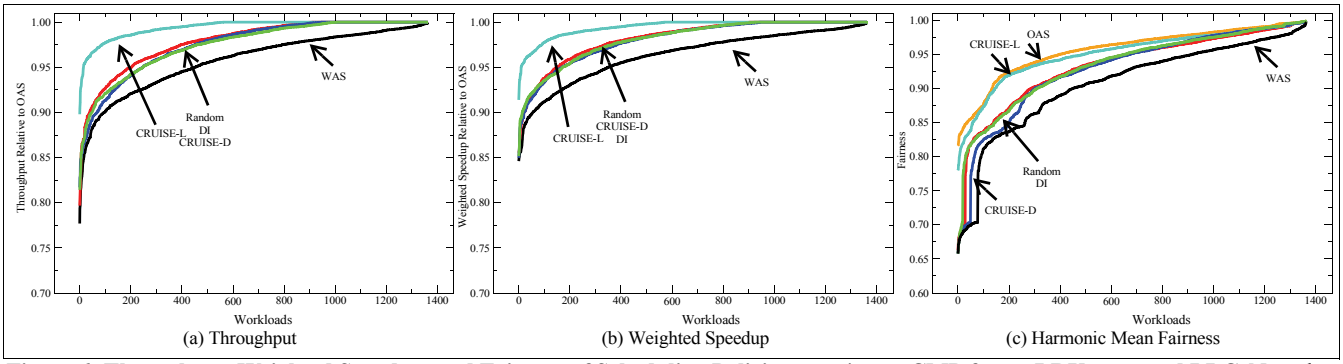

Figure 6: Throughput, Weighted Speedup, and Fairness of Scheduling Policies on a 4-core CMP for an LRU-managed LLC. Note that these curves are independently sorted for each scheduling policy.

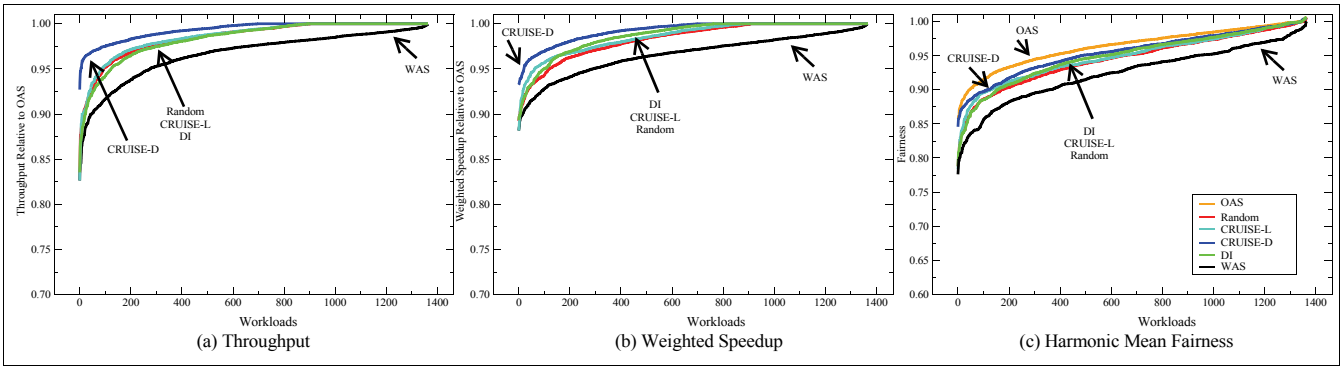

Figure 7: Throughput, Weighted Speedup, and Fairness of Scheduling Policies on a 4-core CMP for a DRRIP-managed LLC. . Note that these curves are independently sorted for each scheduling policy.

performance variation for CRUISE-L, roughly 2.5% for CRUISE-D, random, and DI scheduling, and finally roughly 4% for WAS. Similarly, for a DRRIP-managed cache, we observe roughly 0.8% performance variation for CRUISE-L, roughly 1.8% for CRUISE-D, random, and DI scheduling, and finally roughly 3.5% for WAS. Also, note that a randomly selected co-schedule tends to always be better than the WAS. However, CRUISE-D and CRUISE-L always provide better performance than a random schedule. Additionally, both CRUISE-L and CRUISE-D are within 1% of OAS for the fairness metric. In summary, Figure 6 and Figure 7 reveal that CRUISE performs similar to OAS for the throughput, weighted speedup, and fairness metrics—essentially CRUISE is robust across all metrics.

5.3. Scalability

We now discuss scalability of CRUISE by varying the number of shared LLCs and also varying the number of threads sharing an LLC. We construct 8-core workload mixes and consider two 8-core systems (a) four shared LLCs—two cores each sharing an LLC (b) two shared LLCs—four cores each sharing an LLC. For both these systems, we evaluate the performance of different scheduling policies for LRU and DRRIP-managed caches. We evaluate all possible 8-core combinations of the 15 applications—6435 workload mixes. For each system, we also evaluate all possible co-schedules. For a four LLC system with two cores per LLC, there are 105 possible co-schedules while for a two LLC system with eight cores per LLC, there are 35 possible co-schedules. Since we have already illustrated that CRUISE performs well across all performance metrics, we now limit ourselves to the throughput metric. We observe similar behavior for both the weighted speedup and fairness metrics.

Figure 8 illustrates CRUISE performance for the different 8-core configurations. As the number of shared LLCs increase, CRUISE consistently bridges the performance variation between OAS and WAS (Figure 8 (a) and (b)). Similarly, when increasing the number of threads per LLC, CRUISE still manages to bridge the performance variation (Figure 8 (c) and (d)). In these larger systems, CRUISE consistently performs better than all other scheduling policies.

5.4. Dynamic Classification

Thus far we have investigated the accuracy of CRUISE using profiled utility information. We now evaluate CRUISE with dynamically observed utility information gathered by RICE. Note that any performance deviation from OAS can primarily be attributed to the accuracy and sampling overhead of RICE. Figure 9 illustrates CRUISE-L and DI performance compared to OAS for our baseline 4-core CMP with two shared LLCs. We do not account for the RICE overhead due to sampling for OAS. We observe that CRUISE-L consistently tracks the performance of OAS, thereby showing that RICE performs well in dynamically classifying the application. We also observe that CRUISE-L consistently outperforms DI. On average, we find that CRUISE-L, DI, and OAS improve performance over WAS by roughly 2%, -0.2%, and 4% respectively. The difference between CRUISE-L and OAS is roughly 2% due to the sampling overheads of RICE (we observed similar comparison of CRUISE-D to OAS in DRRIP-managed caches). Note that these single digit performance gains are not insignificant since our studies span more than 1000 workload mixes that include a significant number of workloads that do not benefit from intelligent scheduling.

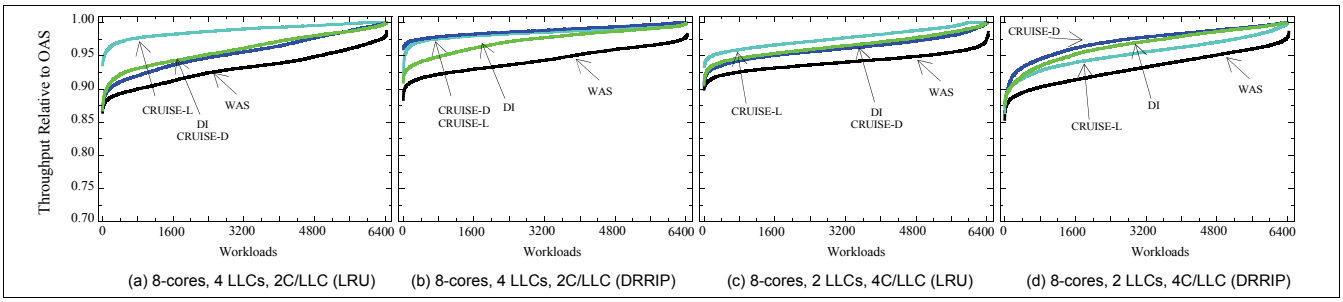

Figure 8: Performance of CRUISE when Increasing Number of Shared LLCs or when Increasing Number of Cores Per LLC (C/LLC). Note that these curves are independently sorted for each scheduling policy.

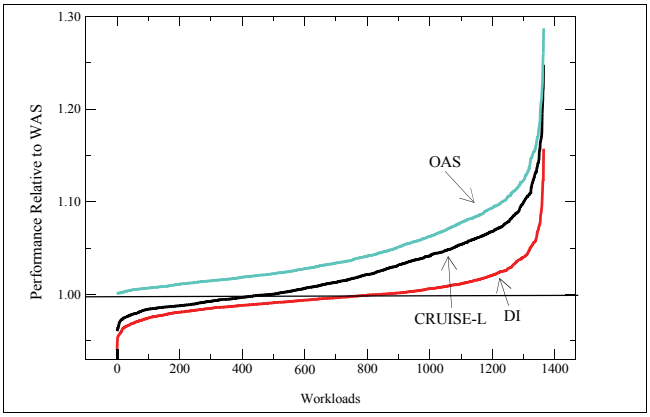

Figure 9: CRUISE Using Dynamic Classification From RICE for LRU-managed caches. Note that these curves are independently sorted for each scheduling policy.

Both CRUISE-L and DI degrade performance slightly (less than 5%) compared to WAS. The negative outliers can be attributed to the sampling overhead of RICE (for CRUISE-L) or the overhead of compulsory misses from to context switches. Our investigations revealed that for the duration of the runs, the application classification reaches steady state and the number of context switches are low. This indicates that there is opportunity to reduce the negative overheads and improve CRUISE performance by periodically sampling RICE counters. Investigating this is part of on-going work.

5.5. Sensitivity to Cache Size

CRUISE performance is dependent on the accuracy of the per-application cache utility. We evaluated CRUISE using profile based cache utility information on our baseline CMP system with smaller shared caches (2MB) and larger shared caches (8MB). With perfect cache utility knowledge, we found that CRUISE behaves similar to results illustrated in Figure 7 for all performance metrics. Similarly, when using RICE to dynamically classify applications, CRUISE behaves similar to results in Figure 9.

With regards to RICE-based dynamic *utility* classification, we found that classification of applications is a function of the available LLC size. For example, RICE correctly classifies *sphinx* as a LLCF application on a 4MB and an LLCT application on a 2MB cache. This is because *sphinx* has a roughly 4MB working-set size. Similarly, RICE correctly classifies *h264ref* as a LLCFR application on a 4MB cache but a LLCF application on a 2MB cache. This is because *h264ref* has a roughly 2MB working-set size. In general, as expected, we observe transitions in application classification from cache fitting/friendly to cache thrashing/fitting when the LLC size is reduced. Similarly, we see transitions in application classification from cache thrashing/fitting to cache fitting/friendly as the LLC size is increased. This shows that RICE can dynamically classifying the isolated application cache utility regardless of the available LLC size.

6. Related Work

Several researchers from the software and hardware community have independently studied and proposed ways to mitigate contention in shared caches. In this section we briefly describe recent work that is relevant to our proposal.

Perhaps the effort that closely resembles our work is the recent *Distributed Intensity (DI)* proposal [38]. Unlike our simulation based study, DI was evaluated on real hardware composed of Intel and AMD systems with shared LLCs. To the best of our knowledge, these LLCs use a pseudo-LRU replacement policy [10]. DI proposes to capture runtime cache miss rates of applications (using hardware performance counters), sort the miss rates, and then separate the memory intensive applications onto different LLCs, in effect co-scheduling memory intensive applications with non-memory bound applications. For an LRU managed cache, this co-scheduling strategy is at odds with all recent work on shared cache management [12, 25, 34, 8]. These studies have illustrated that in an LRU managed cache, memory intensive applications significantly degrade the performance of non-memory intensive applications. Perhaps the primary difference is because the authors correlate performance degradation to not only sharing the LLC but also from sharing prefetchers and the DRAM memory controller. More recent Intel and AMD processors, however, use a three-level hierarchy, where the prefetchers are private and located at the L2 cache [2]. Furthermore, the DRAM controller is on-die and more sophisticated than earlier designs [2]. Thus, the sharing effects of the prefetcher and DRAM controller is less pronounced in modern day processors with LRU replacement. As such, we show that DI does not perform well on LRU-managed caches. Furthermore, while CRUISE-D is similar to DI, it outperforms DI on DRRIP-managed caches as well.

There has been considerable work in shared cache management via software cache partitioning [6, 17, 36]. The general idea in these designs is to allocate a portion of the cache to each of the applications and modify physical memory allocation such that every application's cache lines map into its reserved cache space. These schemes have non-trivial and intricate interactions with the operating system kernel and require complicated changes to virtual memory management.

Some software-centric scheduling algorithms attempt an ad-hoc approach to choosing which threads should be co-scheduled by attempting several thread allocation strategies and picking the best-performing ones [28]. This period during which the algorithm is sampling and learning can have sub-optimal system performance to dynamic phase changes. Furthermore, such a scheme is impractical for large number of threads since the number of permutations of co-schedules exponentially increases.

Recent work in understanding the software co-scheduling of threads has also shown that data sharing is an important parameter by which to classify threads [32]. Tang et al. focus on datacenter application workloads and use a heuristic based algorithm to predict which threads should be mapped to which cores. Since our runtime classification is based on miss rates it works in the same manner as a heuristic based algorithm.

More recent efforts attempt to design an intelligent scheduler that is aware of underlying caches and that can differentiate between applications with differing memory demands [7, 37, 38]. These proposals, however do not consider the impact that the underlying hardware replacement and allocation policies may have on the benefit of the scheduling decisions. Additionally, to differentiate between and classify the applications, these mechanisms do not take into account that hardware cache contention between the threads can lead to misguided classifications.

In the hardware management of shared caches, some of the earlier work focused on hardware cache partitioning [30, 25]. Hardware cache partitioning allocates cache resources to competing applications either statically or dynamically. These proposals use way partitioning in the cache and extra identifying bits per cache line to reserve portions of the cache for each application. Such designs work

well independently to manage caches at a hardware level but cannot reach optimal performance since they do not guide the software scheduler to make intelligent decisions in the first place.

More recent work in hardware management of caches has focused on smart insertion and replacement policies. These policies do not influence application co-schedules, but instead can mitigate the impact that thrashing and scans have on the performance of individual applications [12, 24].

The recent state-of-the-art work in smart hardware cache management focuses on more fine-grained classification of each application by differentiating between references that have near and distant re-reference intervals [10]. The RRIP and DRRIP proposals improve the performance of shared caches considerably and are good starting points to consider a hardware-software co-design where a smart cache replacement removes some of the inefficiencies of shared caches such that intelligent software scheduling is able to reach near optimal performance as we showed in this paper.

There has also been some independent work done in classifying the memory intensity of applications [5, 15, 35, 31]. These proposals draw analogies to differentiate between the memory requirements of several competing applications. However, the proposed mechanisms are more complex than our RICE proposal. As we show in this paper, RICE has negligible hardware overhead, and is practical design that can obtain online and dynamic classification information of concurrently executing applications.

7. Summary and Future Work

The use of virtualization, multi-core, and multi-socket systems have enabled multiple concurrently executing applications on the same system. In doing so, applications with varying memory demands contend for shared resources. Since the on-chip shared LLC serves as the last-line of defense before the long-latency penalty to memory, it is crucial that shared LLCs be efficiently managed.

Shared LLCs commonly found in microprocessors today use the LRU replacement policy. Several studies have shown that LRU performs poorly for shared LLCs because LRU allocates cache resources based on demand instead of benefit. To address the LRU problem, shared LLCs have been managed independently by the software or the hardware. Software tries to intelligently co-schedule applications to avoid shared cache contention while hardware tries to reduce shared cache contention by improving the replacement policy. However, to-date there exists no comprehensive study that evaluates the interaction between improved cache replacement and intelligent scheduling decisions. With this in mind, this paper makes the following contributions:

- We conduct a detailed study on the interactions between intelligent scheduling and smart cache replacement. We find that smart cache replacement reduces the burden on software for intelligent scheduling but does not completely eliminate the need for finding optimal application co-schedules.
- We propose *Cache Replacement and Utility-aware Scheduling (CRUISE)*, a hardware/software co-designed application scheduling policy that uses knowledge of the underlying LLC replacement policy *and* application cache utility information to determine how best to co-schedule applications.
- Finally, we propose a *Runtime Isolated Cache Estimator (RICE)*, a hardware mechanism that dynamically determines isolated LLC performance while concurrently sharing the LLC with other applications. RICE requires no changes to the existing cache structure and requires storage overhead of only eight bytes per hardware-thread in the system.

For a large number of heterogeneous workload mixes, we evaluate CRUISE for a variety of systems with multiple shared LLCs. Our evaluations included LLCs shared by two and four applications. We show that CRUISE significantly reduces the performance variation between different static application co-schedules. In the majority of cases, CRUISE provides *near-optimal* performance for the throughput, weighted speedup, and fairness metrics.

In this paper, we evaluated CRUISE assuming as many cores as there are running applications. However, CRUISE is also applicable when the number of running applications is significantly larger than the number of cores. In such a system, the operating system or hypervisor can utilize CRUISE and RICE to dynamically select applications from the waiting queue while guaranteeing some degree of Quality of Service (QoS). Furthermore, we find that there is opportunity to reduce RICE overhead by dynamically controlling when to gather cache utility information. Exploring these extensions is part of our on-going work.

Acknowledgements

The authors would like to thank Bushra Ahsan, Malini Bhandaru, Jaideep Moses, Moinuddin Qureshi, Paul Racunas, Puneet Zaroo, and the anonymous reviewers for their feedback in improving the quality of this paper.

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] Intel. Intel Core i7 Processor. http://www.intel.com/products/processor/corei7/specifications.htm

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

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

[5] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting Inter-thread Cache Contention on a CMP. In HPCA, 2005.

[6] S. Cho and L. Jin. Managing distributed shared L2 caches through OS-level Page Allocation. In MICRO-39, 2006.

[7] A. Fedorova, M. I. Seltzer, and M. D. Smith. Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler. In PACT, 2007.

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

[9] 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.

[10] A. Jaleel, K. Theobald, S. Steely, and J. Emer. High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP). In ISCA-2010.

[11] A. Jaleel, E. Borch, M. Bhandaru, S. Steely, and J. Emer. Achieving Non-Inclusive Cache Performance With Inclusive Caches -- Temporal Locality Aware (TLA) Cache Management Policies, In MICRO, 2010.

[12] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, and J. Emer. Adaptive Insertion Policies for Managing Shared Caches. In PACT, 2008.

[13] R. Kalla, B. Sinharoy, and J. M. Tendler. IBM Power5 chip: A Dual-Core Multi-Threaded Processor. IEEE Micro, 2004.

[14] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a CMP architecture. In PACT-13, 2004.

[15] R, Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using OS Observations to Improve Performance in Multicore Systems. In IEEE-MICRO 2008.
[16] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. IEEE Micro, March/April 2005.
[17] J. Liedtke, H. Haertig, and M. Hohmuth. OS-Controlled Cache Predicatbility for Real-Time Systems. In RTAS-97.
[18] J. Lin, Q. Lu, X. Ding, Z. Zhang, and P. Sadayappan. Gaining Insights into Multicore Cache Partitioning: Bridging the Gap between Simulation and Real Systems. In HPCA, 2008.
[19] 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, 2005.
[20] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in smt processors. In ISPASS, pages 164–171, 2001.
[21] J. Moses, K. Aisopos, A. Jaleel, R. Iyer, R. Illikkal, D. Newell, and S. Makineni. CMPSched$im: Evaluating OS/CMP Interaction on Shared Cache Management, In ISPASS, 2009.
[22] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual private caches. In ISCA-34, pages 57–68, 2007.
[23] H. Patil, R. Cohn, M. Charney, R. Kapoor, and A. Sun. Pinpointing Representative Portions of Large Intel Itanium Programs with Dynamic Instrumentation. In MICRO, 2004.
[24] 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.
[25] 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.
[26] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A Case for MLP-Aware Cache Replacement. In ISCA-33, 2006.
[27] 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.
[28] A. Snavely and D. Tullsen. Symbiotic Jobscheduling for a Simultaneous Multithreading Processor. In ASPLOS IX, 2000.
[29] H. S. Stone, J. Turek, and J. L. Wolf. Optimal partitioning of cache memory. IEEE Transactions on Computers, 1992.
[30] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. In Journal of Supercomputing, 2004.
[31] G. E. Suh, S. Devadas, and L. Rudolph. A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning. In HPCA'2002.
[32] L. Tang, J. Mars, N. Vachharajani, R. Hundt, M. Soffa. The Impact of Memory Subsystem Resource Sharing on Datacenter Applications. In ISCA, 2011.
[33] J. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. IBM Paper, Oct. 2001.
[34] Y. Xie and G. H. Loh. PIPP: Promotion/Insertion Pseudo-Partitioning of Multi-Core Shared Caches. In ISCA, 2009.
[35] Y. Xie and G. H. Loh. Dynamic Classification of Program Memory Behaviors in CMPs. In CMP-MSI, 2008.
[36] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring - based multicore cache management. In EuroSys, 2009.
[37] S. Zhuravlev, S. Blagodurov and A. Fedorova. AKULA: A Toolset for Developing Scheduling Algorithms on Multicore Systems. In PACT, 2010.
[38] S. Zhuravlev, S. Blagodurov and A. Fedorova. Addressing Shared Resource Contention in Multicore Processors via Scheduling. In ASPLOS, 2010.