Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems

Eiman Ebrahimi† Chang Joo Lee† Onur Mutlu§ Yale N. Patt†

†Department of Electrical and Computer Engineering
The University of Texas at Austin
{ebrahimi, cjlee, patt}@ece.utexas.edu

§Computer Architecture Laboratory (CALCM)
Carnegie Mellon University
onur@cmu.edu

Abstract

Cores in a chip-multiprocessor (CMP) system share multiple hardware resources in the memory subsystem. If resource sharing is unfair, some applications can be delayed significantly while others are unfairly prioritized. Previous research proposed separate fairness mechanisms in each individual resource. Such resource-based fairness mechanisms implemented independently in each resource can make contradictory decisions, leading to low fairness and loss of performance. Therefore, a coordinated mechanism that provides fairness in the entire shared memory system is desirable.

This paper proposes a new approach that provides fairness in the *entire shared memory system*, thereby eliminating the need for and complexity of developing fairness mechanisms for each individual resource. Our technique, Fairness via Source Throttling (FST), estimates the unfairness in the entire shared memory system. If the estimated unfairness is above a threshold set by system software, FST throttles down cores causing unfairness by limiting the number of requests they can inject into the system and the frequency at which they do. As such, our *source-based* fairness control ensures fairness decisions are made in tandem in the entire memory system. FST also enforces thread priorities/weights, and enables system software to enforce different fairness objectives and fairness-performance tradeoffs in the memory system.

Our evaluations show that FST provides the best system fairness and performance compared to four systems with no fairness control and with state-of-the-art fairness mechanisms implemented in both shared caches and memory controllers.

Categories and Subject Descriptors: C.1.0 [Processor Architectures]: General; C.5.3 [Microcomputers]: Microprocessors; C.1.2 [Multiple Data Stream Architectures (Multiprocessors)]

General Terms: Design, Performance.

1. Introduction

Chip-multiprocessor (CMP) systems commonly share a large portion of the memory subsystem between different cores. Main memory and shared caches are two examples of shared resources. Memory requests from different applications executing on different cores of a CMP can interfere with and delay each other in the shared memory subsystem. Compared to a scenario where each application runs alone on the CMP, this inter-core interference causes the execution of simultaneously running applications to slow down. However, sharing memory system resources affects the execution of different applications very differently because the resource management algorithms employed in the shared resources are unfair [22]. As a result some applications are unfairly slowed down a lot more than others .

Figure 1 shows examples of vastly differing effects of resource-sharing on simultaneously executing applications on a 2-core CMP system (Section 4 describes our experimental setup). When *bzip2* and *art* run simultaneously with equal priorities, the inter-core interference caused by the sharing of memory system resources slows down *bzip2* by *5.2X* compared to when it is run alone while *art* slows down by only *1.15X*. In order to achieve system level fairness or quality of service (QoS) objectives, the system software (operating system or virtual machine monitor) expects proportional progress of *equal-priority* applications when running simultaneously. Clearly, disparities in slowdown like those shown in Figure 1 due to sharing of the memory system resources between simultaneously running equal-priority applications is unacceptable since it would make priority-based thread scheduling policies ineffective [6].

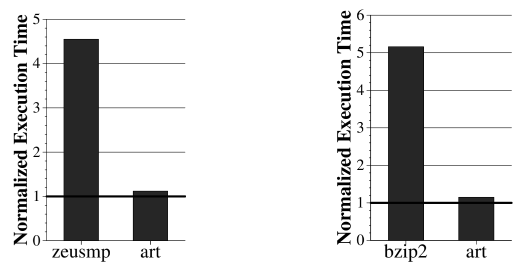


Figure 1: Disparity in slowdowns due to unfairness

To mitigate this problem, previous works [12, 13, 15, 22–25] on fair memory system design for multi-core systems mainly focused on partitioning a particular shared resource (cache space, cache bandwidth, or memory bandwidth) to provide fairness in the use of that shared resource. However, none of these prior works directly target a *fair* memory system design that provides fair sharing of *all resources together*. We define a memory system design as *fair* if the slowdowns of equal-priority applications running simultaneously on the cores sharing that memory system are the same (this definition has been used in several prior works [3, 7, 18, 22, 28]). As shown in previous research [2], employing separate uncoordinated fairness techniques together does not necessarily result in a fair memory system design. This is because fairness mechanisms in different resources can contradict each other. **Our goal** in this paper is to develop a low-cost architectural technique that allows system software fairness policies to be achieved in CMPs by enabling fair sharing of the *entire memory system*, without requiring multiple complicated, specialized, and possibly contradictory fairness techniques for different shared resources.

Basic Idea: To achieve this goal, we propose a fundamentally new mechanism that 1) gathers dynamic feedback information about

ASPLOS'10, March 13–17, 2010, Pittsburgh, Pennsylvania, USA.

the unfairness in the system and 2) uses this information to dynamically adapt the rate at which the different cores inject requests into the shared memory subsystem such that system-level fairness objectives are met. To calculate unfairness at run-time, a slowdown value is estimated for each application in hardware. Slowdown is defined as T_{shared}/T_{alone}, where T_{shared} is the number of cycles it takes to run simultaneously with other applications and T_{alone} is the number of cycles it would have taken the application to run alone. Unfairness is calculated as the ratio of the largest slowdown to the smallest slowdown of the simultaneously running applications. If the unfairness in the system becomes larger than the *unfairness threshold* set by the system software, the core that interferes most with the core experiencing the largest slowdown is throttled down. This means that the rate at which the most interfering core injects memory requests into the system is reduced, in order to reduce the inter-core interference it generates. If the system software's *fairness goal* is met, all cores are allowed to throttle up to improve system throughput while system unfairness is continuously monitored. This configurable hardware substrate enables the system software to achieve different QoS/fairness policies: it can determine the balance between fairness and system throughput, dictate different fairness objectives, and enforce thread priorities in the entire memory system.

Summary of Evaluation: We evaluate our technique on both 2-core and 4-core CMP systems in comparison to three previously-proposed state-of-the-art shared hardware resource management mechanisms. Experimental results across ten multi-programmed workloads on a 4-core CMP show that our proposed technique improves average system performance by 25.6%/14.5% while reducing system unfairness by 44.4%/36.2% compared respectively to a system with no fairness techniques employed and a system with state-of-the-art fairness mechanisms implemented for both shared cache capacity [25] and the shared memory controller [23].

Contributions: We make the following contributions:

1. We introduce a low-cost, hardware-based and system-software-configurable mechanism to achieve fairness goals specified by system software in the *entire* shared memory system.

2. We introduce a mechanism that collects dynamic feedback on the unfairness of the system and adjusts request rates of the different cores to achieve the desired fairness/performance balance. By performing *source-based* fairness control, this work eliminates the need for complicated *individual resource-based* fairness mechanisms that are implemented independently in each resource and that require coordination.

3. We qualitatively and quantitatively compare our proposed technique to multiple prior works in fair shared cache partitioning and fair memory scheduling. We find that our proposal, while simpler, provides significantly higher system performance and better system fairness compared to previous proposals.

2. Background and Motivation

We first present brief background on how we model the shared memory system of CMPs. We then motivate our approach to providing fairness in the entire shared memory system by showing how employing resource-based fairness techniques does not necessarily provide better overall fairness.

2.1 Shared CMP Memory Systems

In this paper, we assume that the last-level (L2) cache and off-chip DRAM bandwidth are shared by multiple cores on a chip as in many commercial CMPs [1, 11, 29, 31]. Each core has its own L1 cache. Miss Status Holding/information Registers (MSHRs) [16] keep track of all requests to the shared L2 cache until they are serviced. When an L1 cache miss occurs, an access request to the L2 cache is created by allocating an MSHR entry. Once the request is serviced by the L2 cache or DRAM system as a result of a cache hit or miss respectively, the corresponding MSHR entry is freed and used for a new request. The number of MSHR entries for a core indicates the total number of outstanding requests allowed to the L2 cache and DRAM system. Therefore increasing/decreasing the number of MSHR entries for a core can increase/decrease the rate at which memory requests from the core are injected into the shared memory system.

2.2 Motivation

Most prior works on providing fairness in shared resources focus on partitioning of a single shared resource. However, by partitioning *a single* shared resource, the demands on other shared resources may change such that neither system fairness nor system performance is improved. In the following example, we describe how constraining the rate at which an application's memory requests are injected to the shared resources can result in higher fairness and system performance than employing fair partitioning of a single resource.

Figure 2 shows the memory-related stall time[1] of equal-priority applications A and B running on different cores of a 2-core CMP. For simplicity of explanation, we assume an application stalls when there is an outstanding memory request, focus on requests going to the same cache set and memory bank, and assume all shown accesses to the shared cache occur before any replacement happens. Application A is very memory-intensive, while application B is much less memory-intensive. As prior work has observed [23], when a memory-intensive application with already high memory-related stall time interferes with a less memory-intensive application with much smaller memory-related stall time, delaying the former improves system fairness because the additional delay causes a smaller slowdown for the memory-intensive application than for the non-intensive one. Doing so can also improve throughput by allowing the less memory-intensive application to quickly return to its compute-intensive portion while the memory-intensive application continues waiting on memory.

Figures 2(a) and (b) show the initial L2 cache state, access order and memory-related stall time when no fairness mechanism is employed in any of the shared resources. Application A's large number of memory requests arrive at the L2 cache earlier, and as a result, the small number of memory requests from application B are significantly delayed. This causes large unfairness because the compute-intensive application B is slowed down significantly more than the already-slow memory-intensive application A. Figures 2(c) and (d) show that employing a fair cache increases the fairness *in utilization of the cache* by allocating *an equal number of ways* from the accessed set to the two equal-priority applications. This increases application A's cache misses compared to the baseline with no fairness control. Even though application B gets more hits as a result of fair sharing of the cache, its memory-related stall time does not reduce due to increased interference in the main memory system from

[1] Stall-time is the amount of execution time in which the application cannot retire instructions. Memory-related stall time caused by a memory request consists of: 1) time to access the L2 cache, and if the access is a miss 2) time to wait for the required DRAM bank to become available, and finally 3) time to access DRAM.

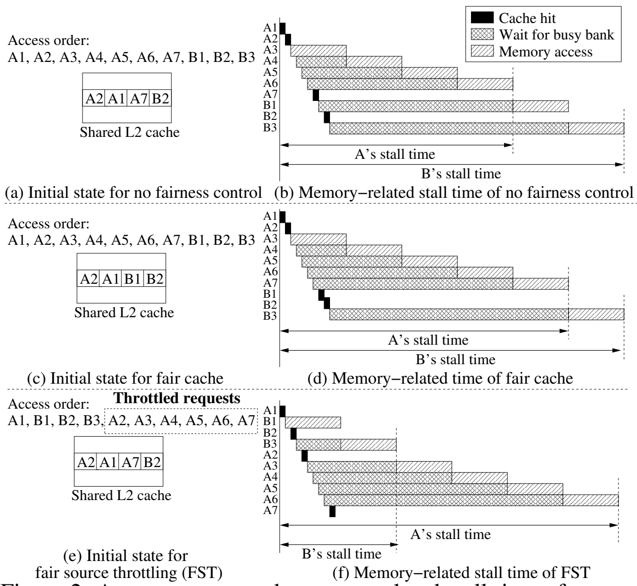


Figure 2: Access pattern and memory-related stall time of requests with (a, b) no fairness control, (c, d) fair cache, and (e, f) fair source throttling

application A’s increased misses. Application B’s memory requests are still delayed behind the large number of memory requests from application A. Application A’s memory-related stall time increases slightly due to its increased cache misses, however, since application A already had a large memory-related stall time, this slight increase does not incur a large slowdown for it. As a result, fairness improves slightly, but system throughput degrades because the system spends more time stalling rather than computing compared to no fair caching.

In Figure 2, if the unfair slowdown of application B due to application A is detected at run-time, system fairness can be improved by limiting A’s memory requests and reducing the frequency at which they are issued into the shared memory system. This is shown in the access order and memory-related stall times of Figures 2(e) and (f). If the frequency at which application A’s memory requests are injected into the shared memory system is reduced, the memory access pattern can change as shown in Figure 2(e). We use the term *throttled requests* to refer to those requests from application A that are delayed when accessing the shared L2 cache due to A’s reduced injection rate. As a result of the late arrival of these *throttled requests*, application B’s memory-related stall time significantly reduces (because A’s requests no longer interfere with B’s) while application A’s stall time increases slightly. Overall, this ultimately improves both system fairness and throughput compared to both no fairness control and just a fair cache. Fairness improves because the memory-intensive application is delayed such that the less intensive application’s memory related-stall time does not increase significantly compared to when running alone. Delaying the memory-intensive application does not slow it down too much compared to when running alone, because even when running alone it has high memory-related stall time. System throughput improves because the total amount of time spent computing rather than stalling in the entire system increases.

The **key insight** is that *both system fairness and throughput can improve by detecting high system unfairness at run-time and dynamically limiting the number of or delaying the issuing of memory requests from the aggressive applications*. In essence, we propose a new approach that performs *source-based* fairness in the entire memory system rather than *individual resource-based* fairness that implements complex and possibly contradictory fairness mechanisms in each resource. Sources (i.e., cores) can collectively achieve fairness by throttling themselves based on dynamic unfairness feedback, eliminating the need for implementing possibly contradictory/conflicting fairness mechanisms and complicated coordination techniques between them.

3. Fairness via Source Throttling

To enable fairness in the entire memory system, we propose *Fairness via Source Throttling* (FST). The proposed mechanism consists of two major components: 1) *runtime unfairness evaluation* and 2) *dynamic request throttling*.

3.1 Runtime Unfairness Evaluation Overview

The goal of this component is to dynamically obtain an estimate of the unfairness in the CMP memory system. We use the following definitions in determining unfairness:

1) We define a memory system design as *fair* if the slowdowns of equal-priority applications running simultaneously on the cores of a CMP are the same, similarly to previous works [3, 7, 18, 22, 28].

2) We define slowdown as T_{shared}/T_{alone} where T_{shared} is the number of cycles it takes to run simultaneously with other applications and T_{alone} is the number of cycles it would have taken the application to run alone on the same system.

The main challenge in the design of the runtime unfairness evaluation component is obtaining information about the number of cycles it would have taken an application to run alone, while it is running simultaneously with other applications. To do so, we estimate the number of *extra cycles* it takes an application to execute due to inter-core interference in the shared memory system, called T_{excess}. Using this estimate, T_{alone} is calculated as $T_{shared} - T_{excess}$. The following equations show how *Individual Slowdown (IS)* of each application and *Unfairness* of the system are calculated.

$$IS_i = \frac{T_i^{shared}}{T_i^{alone}}, \quad Unfairness = \frac{MAX\{IS_0, IS_1, ..., IS_{N-1}\}}{MIN\{IS_0, IS_1, ..., IS_{N-1}\}}$$

Section 3.3 explains in detail how the runtime unfairness evaluation component is implemented and in particular how T_{excess} is estimated. Assuming for now that this component is in place, we next explain how the information it provides is used to determine how each application is throttled to achieve fairness in the entire shared memory system.

3.2 Dynamic Request Throttling

This component is responsible for dynamically adjusting the rate at which each core/application[2] makes requests to the shared resources. This is done on an interval basis as shown in Figure 3.

An interval ends when each core has executed a certain number of instructions from the beginning of that interval. During each interval (for example *Interval 1* in Figure 3) the runtime unfairness evaluation component gathers feedback used to estimate the slowdown of each application. At the beginning of the next interval (*Interval 2*), the feedback information obtained during the prior interval is used to make a decision about the request rates of each application for that

[2] Since each core runs a separate application, we use the words core and application interchangeably in this paper. See also Section 3.4.

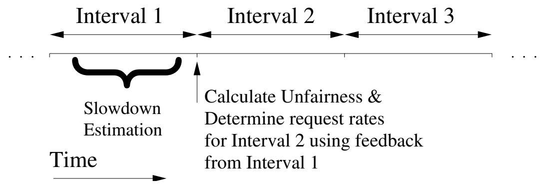


Figure 3: FST's interval-based estimation and throttling

interval. More precisely, slowdown values estimated during *Interval 1* are used to estimate unfairness for the system. That unfairness value is used to determine the request rates for the different applications for the duration of *Interval 2*. During the next interval (*Interval 2*), those request rates are applied, and unfairness evaluation is performed again. The algorithm used to adjust the request rate of each application using the unfairness estimate calculated in the prior interval is shown in Algorithm 1. To ease explanations, Algorithm 1 is simplified for dual-core configurations. Section 3.5 presents the more general algorithm for more than two cores.

We define multiple possible levels of aggressiveness for the request rate of each application. The dynamic request throttling component makes a decision to increase/decrease or keep constant the request rate of each application at interval boundaries. We refer to increasing/decreasing the request rate of an application as throttling the application up/down.

Algorithm 1 Dynamic Request Throttling

if $Estimated\ Unfairness > Unfairness\ Threshold$ **then**
  Throttle down application with the smallest $slowdown$
  Throttle up application with the largest $slowdown$
  Reset $Successive\ Fairness\ Achieved\ Intervals$

else
  if $Successive\ Fairness\ Achieved\ Intervals = threshold$ **then**
    Throttle all applications up
    Reset $Successive\ Fairness\ Achieved\ Intervals$
  else
    Increment $Successive\ Fairness\ Achieved\ Intervals$
  end if
end if

At the end of each interval, the algorithm compares the unfairness estimated in the previous interval to the unfairness threshold that is defined by system software. If the fairness goal has not been met in the previous interval, the algorithm reduces the request rate of the application with the smallest individual slowdown value and increases the request rate of the application with the largest individual slowdown value. This reduces the number and frequency of memory requests generated for and inserted into the memory resources by the application with the smallest estimated slowdown, thereby reducing its interference with other cores. The increase in the request rate of the application with the highest slowdown allows it to be more aggressive in exploiting Memory-Level Parallelism (MLP) [8] and as a result reduces its slowdown. If the fairness goal is met for a predetermined number of intervals (tracked by a $Successive\ Fairness\ Achieved\ Intervals$ counter in Algorithm 1), the dynamic request throttling component attempts to increase system throughput by increasing the request rates of all applications by one level. This is done because our proposed mechanism strives to increase throughput while maintaining the fairness goals set by the system software. Increasing the request rate of all applications might result in unfairness. However, the unfairness evaluation during the interval in which this happens detects this occurrence and dynamically adjusts the requests rates again.

Throttling Mechanisms: Our mechanism increases/decreases the request rate of each application in multiple ways: 1) Adjusting the number of outstanding misses an application can have at any given time. To do so, an *MSHR quota*, which determines the maximum number of MSHR entries an application can use at any given time, is enforced for each application. Reducing MSHR entries for an application reduces the pressure caused by that application's requests on all shared memory system resources by limiting the number of concurrent requests from that application contending for service from the shared resources. This reduces other simultaneously running applications' memory-related stall times and gives them the opportunity to speed up. 2) Adjusting the *frequency at which requests in the MSHRs are issued to access L2*. Reducing this frequency for an application reduces the number of memory requests per unit time from that application that contend for shared resources. This allows memory requests from other applications to be prioritized in accessing shared resources even if the application that is throttled down does not have high MLP to begin with and is not sensitive to reduction in the number of its MSHRs. We refer to this throttling technique as *frequency throttling*. We use both of these mechanisms to reduce the interference caused by the application that experiences the smallest slowdown on the application that experiences the largest slowdown.

3.3 Unfairness Evaluation Component Design

T_{shared} is simply the number of cycles it takes to execute an application in an interval. Estimating T_{alone} is more difficult, and FST achieves this by estimating T_{excess} for each core, which is the number of cycles the core's execution time is lengthened due to interference from other cores in the shared memory system. To estimate T_{excess}, the unfairness evaluation component keeps track of inter-core interference each core incurs.

Tracking Inter-Core Interference: We consider three sources of inter-core interference: 1) cache, 2) DRAM bus and bank conflict, and 3) DRAM row-buffer.[3] Our mechanism uses an $InterferencePerCore$ bit-vector whose purpose is to indicate whether or not a core is delayed due to inter-core interference. In order to track interference from each source separately, a copy of $InterferencePerCore$ is maintained for each interference source. A main copy which is updated by taking the union of the different $InterferencePerCore$ vectors is eventually used to update T_{excess} as described below. When FST detects inter-core interference for core i at any shared resource, it sets bit i of the $InterferencePerCore$ bit-vector, indicating that the core was delayed due to interference. At the same time, it also sets an $InterferingCoreId$ field in the corresponding *interfered-with* memory request's MSHR entry. This field indicates which core interfered with this request and is later used to reset the corresponding bit in the $InterferencePerCore$ vector when the *interfered-with* request is scheduled/serviced. We explain this process in more detail for each resource below in Sections 3.3.1-3.3.3. If a memory request

[3] On-chip interconnect can also experience inter-core interference [4]. Feedback information similar to that obtained for the three sources of inter-core interference we account for can be collected for the on-chip interconnect. That information can be incorporated into our technique seamlessly, which we leave as part of future work.

has not been interfered with, it's $InterferingCoreId$ will be the same as the core id of the core it was generated by.

Updating T_{excess}**:** FST stores the number of *extra cycles* it takes to execute a given interval's instructions due to inter-core interference (T_{excess}) in an $ExcessCycles$ counter per core. Every cycle, if the $InterferencePerCore$ bit of a core is set, FST increments the corresponding core's $ExcessCycles$ counter.

Algorithm 2 shows how FST calculates $ExcessCycles$ for a given core *i*. The following subsections explain in detail how each source of inter-core interference is taken into account to set $InterferencePerCore$. Table 5 summarizes the required storage needed to implement the mechanisms explained here.

Algorithm 2 Estimation of T_{excess} for core *i*

Every cycle
  if $inter\text{-}core\ cache\ or\ DRAM\ bus\ or\ DRAM\ bank\ or\ DRAM\ row\text{-}buffer\ interference$ **then**
    set $InterferencePerCore$ bit *i*
    set $InterferingCoreId$ in delayed memory request
  end if
  if $InterferencePerCore$ bit *i* is set **then**
    Increment $ExcessCycles$ for core *i*
  end if

Every L2 cache fill for a miss due to interference OR
Every time a memory request which is a row-buffer miss due to interference is serviced
  reset $InterferencePerCore$ bit of core *i*
  $InterferingCoreId$ of core *i* = *i* (no interference)

Every time a memory request is scheduled to DRAM
  if Core *i* has no requests waiting on any bank which is busy servicing another core *j* (*j* != *i*) **then**
    reset $InterferencePerCore$ bit of core *i*
  end if

3.3.1 Cache Interference

In order to estimate inter-core cache interference, for each core *i* we need to track the last-level cache misses that are caused by any other core *j*. To do so, FST uses a pollution filter for each core to approximate such misses. The pollution filter is a bit-vector that is indexed with the lower order bits of the accessed cache line's address.[4] In the bit-vector, a set entry indicates that a cache line belonging to the corresponding core was evicted by another core's request. When a request from core *j* replaces one of core *i*'s cache lines, core *i*'s filter is accessed using the evicted line's address, and the corresponding bit is set. When a memory request from core *i* misses the cache, its filter is accessed with the missing address. If the corresponding bit is set, the filter predicts that this line was previously evicted due to inter-core interference and the bit in the filter is reset. When such a prediction is made, the $InterferencePerCore$ bit corresponding to core *i* is set to indicate that core *i* is stalling due to cache interference. Once the interfered-with memory request is finally serviced from the memory system and the corresponding cache line is filled, core *i*'s filter is accessed and the bit is reset.

3.3.2 DRAM Bus and Bank Conflict Interference

Inter-core DRAM bank conflict interference occurs when core *i*'s memory request cannot access the bank it maps to, because a request from some other core *j* is being serviced by that memory bank. DRAM bus conflict interference occurs when a core cannot use the DRAM because another core is using the DRAM bus. These situations are easily detectable at the memory controller, as described in [22]. When such interference is detected, the $InterferencePerCore$ bit corresponding to core *i* is set to indicate that core *i* is stalling due to a DRAM bus or bank conflict. This bit is reset when no request from core *i* is being prevented access to DRAM by the other cores' requests.

3.3.3 DRAM Row-Buffer Interference

This type of interference occurs when a potential row-buffer hit of core *i* when running alone is converted to a row-buffer miss/conflict due to a memory request of some core *j* when running together with others. This can happen if a request from core *j* closes a DRAM row opened by a prior request from core *i* that is also accessed by a subsequent request from core *i*. To track such interference, a *Shadow Row-buffer Address Register (SRAR)* is maintained for each core for each bank. Whenever core *i*'s memory request accesses some row *R*, the SRAR of core *i* is updated to row *R*. Accesses to the same bank from some other core *j* do not affect the SRAR of core *i*. As such, at any point in time, core *i*'s SRAR will contain the last row accessed by the last memory request serviced from that core in that bank. When core *i*'s memory request suffers a row-buffer miss because another core *j*'s row is open in the row-buffer of the accessed bank, the SRAR of core *i* is consulted. If the SRAR indicates a row-buffer hit would have happened, then inter-core row-buffer interference is detected. As a result, the $InterferencePerCore$ bit corresponding to core *i* is set. Once the memory request is serviced, the corresponding $InterferencePerCore$ bit is reset.[5]

3.3.4 Slowdown Due to Throttling

When an application is throttled, it experiences some slowdown due to the throttling. This slowdown is different from the inter-core interference induced slowdown estimated by the mechanisms of Sections 3.3.1 to 3.3.3. Throttling-induced slowdown is a function of an application's sensitivity to 1) the number of MSHRs that are available to it, 2) the frequency of injecting requests into the shared resources. Using profiling, we determine for each throttling level l, the corresponding slowdown (due to throttling) f of an application *A*. At runtime, any estimated slowdown for application *A* when running at throttling level l is multiplied by f. We find that accounting for this slowdown improves the performance of FST by 1.9% and 0.9% on the 2-core and 4-core systems respectively. As such, even though we use such information in our evaluations, it is not fundamental to FST's benefits.

3.3.5 Implementation Details

Section 3.3 describes how separate copies of $InterferencePerCore$ are maintained per interference source. The main copy which is used by FST for updating T_{excess} is physically located close by the L2 cache. Note that shared resources may be located far away from each other on the chip. Any possible timing constraints on the sending of updates to the $InterferencePerCore$ bit-vector from the shared resources can be eliminated by making these updates periodically.

[4] We empirically determined the pollution filter for each core to have 2K-entries in our evaluations.

[5] To be more precise, the bit is reset "row buffer hit latency" cycles before the memory request is serviced. The memory request would have taken at least "row buffer hit latency" cycles had there been no interference.

3.4 System Software Support

Different Fairness Objectives: System-level fairness objectives and policies are generally decided by the system software (the operating system or virtual machine monitor). FST is intended as architectural support for enforcing such policies in shared memory system resources. The *fairness goal* to be achieved by FST can be configured by system software. To achieve this, we enable the system software to determine the nature of the condition that triggers Algorithm 1. In the explanations of Section 3.2, the *triggering condition* is

"$Estimated\, Unfairness_i > Unfairness\, Threshold$"

System software might want to enforce different triggering conditions depending on the system's fairness/QoS requirements. To enable this capability, FST implements different triggering conditions from which the system software can choose. For example, the fairness goal the system software wants to achieve could be to keep the maximum slowdown of any application below a threshold value. To enforce such a goal, the system software can configure FST such that the triggering condition in Algorithm 1 is changed to

"$Estimated\, Slowdown_i > Max.\, Slowdown\, Threshold$"

Thread Weights: So far, we have assumed all threads are of equal importance. FST can be seamlessly adjusted to distinguish between and provide differentiated services to threads with different priorities. We add the notion of *thread weights* to FST, which are communicated to it by the system software using special instructions. Higher slowdown values are more tolerable for less important or *lower weight* threads. To incorporate thread weights, FST uses *weighted slowdown* values calculated as:

$WeightedSlowdown_i = Measured\, Slowdown_i \times Weight_i$

By scaling the real slowdown of a thread with its weight, a thread with a higher weight appears as if it slowed down more than it really did, causing it to be favored by FST. Section 5.4 quantitatively evaluates FST with the above fairness goal and threads with different weights.

Thread Migration and Context Switches: FST can be seamlessly extended to work in the presence of thread migration and context switches. When a context switch happens or a thread is migrated, the interference state related to that thread is cleared. When a thread restarts executing after a context switch or migration, it starts at maximum throttle. The interference caused by the thread and the interference it suffers are dynamically re-estimated and FST adapts to the new set of co-executing applications.

3.5 General Dynamic Request Throttling

Scalability to More Cores: When the number of cores is greater than two, a more general form of Algorithm 1 is used. The design of the *unfairness evaluation* component for the more general form of Algorithm 1 is also slightly different. For each core *i*, FST maintains a set of *N-1* counters, where *N* is the number of simultaneously running applications. We refer to these *N-1* counters that FST uses to keep track of the amount of the inter-core interference caused by any other core *j* in the system for *i* as $ExcessCycles_{ij}$. This information is used to identify which of the other applications in the system generates the most interference for core *i*. FST maintains the total inter-core interference an application on core *i* experiences in a $TotalExcessCycles_i$ counter per core.

Algorithm 3 shows the generalized form of Algorithm 1 that uses this extra information to make more accurate throttling decisions in a system with more than two cores. The two most important changes are as follows. First, when the algorithm is triggered due to unfair slowdown of core *i*, FST compares the $ExcessCycles_{ij}$ counter values for all cores $j \neq i$ to determine which other core is interfering most with core *i*. The core found to be the most interfering is throttled down. We do this in order to reduce the slowdown of the core with the largest slowdown value, and improve system fairness. Second, cores that are neither the core with the largest slowdown (App_{slow}) nor the core generating the most interference ($App_{interfering}$) for the core with the largest slowdown are throttled up every $threshold1$ intervals. This is a performance optimization that allows cores to be aggressive if they are not the main contributors to the unfairness in the system.

Preventing Bank Service Denial due to FR-FCFS Memory Scheduling: First ready-first come first serve (FR-FCFS) [27] is a commonly used memory scheduling policy which we use in our baseline system. This memory scheduling policy has the potential to starve an application with no row-buffer locality in the presence of an application with high row-buffer locality (as discussed in prior work [21–24]). Even when the interfering application is throttled down, the potential for continued DRAM bank interference exists when FR-FCFS memory scheduling is used, due to the greedy row-hit-first nature of the scheduling algorithm: a throttled-down application with high row-buffer locality can deny service to another application continuously. To overcome this, we supplement FST with a heuristic that prevents this denial of service. Once an application has already been throttled down lower than $Switch_{thr}\%$, if FST detects that this throttled application is generating greater than

Algorithm 3 Dynamic Request Throttling - General Form

```
if Estimated Unfairness > Unfairness Threshold then
  Throttle down application that causes most interference
  (App_interfering) for application with largest slowdown
  Throttle up application with the largest slowdown (App_slow)
  Reset Successive Fairness Achieved Intervals
  Reset Intervals To Wait To Throttle Up for App_interfering.

  // Preventing bank service denial
  if App_interfering throttled lower than Switch_thr AND causes
  greater than Interference_thr amount of App_slow's total interference
  then
    Temporarily stop prioritizing App_interfering due to row hits in
    memory controller
  end if
  if App_RowHitNotPrioritized has not been App_interfering for
  SwitchBack_thr intervals then
    Allow it to be prioritized in memory controller based on row-buffer
    hit status of its requests
  end if

  for all applications except App_interfering and App_slow do
    if Intervals To Wait To Throttle Up = threshold1 then
      throttle up
      Reset Intervals To Wait To Throttle Up for this app.
    else
      Increment Intervals To Wait To Throttle Up for this app.
    end if
  end for

else
  if Successive Fairness Achieved Intervals = threshold2 then
    Throttle up application with the smallest slowdown
    Reset Successive Fairness Achieved Intervals
  else
    Increment Successive Fairness Achieved Intervals
  end if
end if
```

$Interference_{thr}\%$ of App_{slow}'s total interference, it will temporarily stop prioritizing the interfering application based on row-buffer hit status in the memory controller. We refer to this application as $App_{RowHitNotPrioritized}$. If $App_{RowHitNotPrioritized}$ has not been the most interfering application for $SwitchBack_{thr}$ number of intervals, its prioritization over other applications based on row-buffer hit status will be re-allowed in the memory controller. This is done because if an application with high row-buffer locality is not allowed to take advantage of row buffer hits for a long time, its performance will suffer.

4. Methodology

Processor Model: We use an in-house cycle-accurate x86 CMP simulator for our evaluation. We faithfully model all port contention, queuing effects, bank conflicts, and other major DDR3 DRAM system constraints in the memory subsystem. Table 1 shows the baseline configuration of each core and the shared resource configuration for the 2 and 4-core CMP systems we use.

Execution Core	6.6 GHz out of order processor, 15 stages, Decode/retire up to 4 instructions Issue/execute up to 8 micro instructions 256-entry reorder buffer
Front End	Fetch up to 2 branches; 4K-entry BTB 64K-entry Hybrid branch predictor
On-chip Caches	L1 I-cache: 32KB, 4-way, 2-cycle, 64B line L1 D-cache: 32KB, 4-way, 2-cycle, 64B line Shared unified L2: 1MB (2MB for 4-core), 8-way (16-way for 4-core), 16-bank, 15-cycle (20-cycle for 4-core), 1 port, 64B line size
DRAM Controller	On-chip, FR-FCFS scheduling policy [27] 128-entry MSHR and memory request buffer
DRAM and Bus	667MHz bus cycle, DDR3 1333MHz [20] 8B-wide data bus Latency: 15-15-15ns (^tRP-^tRCD-CL) 8 DRAM banks, 16KB row buffer per bank Round-trip L2 miss latency: Row-buffer hit: 36ns, conflict: 66ns

Table 1: Baseline system configuration

Workloads: We use the SPEC CPU 2000/2006 benchmarks for our evaluation. Each benchmark was compiled using ICC (Intel C Compiler) or IFORT (Intel Fortran Compiler) with the -O3 option. We ran each benchmark with the reference input set for 200 million x86 instructions selected by Pinpoints [26] as a representative portion for the 2-core experiments. Due to long simulation times, 4-core experiments were done with 50 million instructions per benchmark.

We classify benchmarks as *highly memory-intensive/with medium memory intensity/non-intensive* for our analyses and workload selection. We refer to a benchmark as highly memory-intensive if its L2 Cache Misses per 1K Instructions (MPKI) is greater than ten. If the MPKI value is greater than one but less than ten, we say the benchmark has medium memory-intensity. If the MPKI value is less than one, we refer to it as non-intensive. This classification is based on measurements made when each benchmark was run alone on the 2-core system. Table 2 shows the characteristics of some (due to space limitations) of the benchmarks that appear in the evaluated workloads when run on the 2-core system.

Workload Selection We used 18 two-application and 10 four-application multi-programmed workloads for our 2-core and 4-core evaluations respectively. The 2-core workloads were chosen such that at least one of the benchmarks is highly memory intensive. For this purpose we used either *art* from SPEC2000 or *lbm* from SPEC2006. For the second benchmark of each 2-core workload, applications of different memory intensity were used in order to cover a wide range of different combinations. Of the 18 benchmarks combined with either *art* or *lbm*, seven benchmarks have high memory intensity, six have medium intensity, and five have low memory intensity. The ten 4-core workloads were randomly selected with the condition that the evaluated workloads each include at least one benchmark with high memory intensity and at least one benchmark with medium or high memory intensity.

Benchmark	Type	IPC	MPKI	Benchmark	Type	IPC	MPKI
art	FP00	0.10	90.89	milc	FP06	0.30	29.33
soplex	FP06	0.28	21.24	leslie3d	FP06	0.41	20.88
lbm	FP06	0.45	20.16	bwaves	FP06	0.46	18.71
GemsFDTD	FP06	0.46	15.63	astar	INT06	0.37	10.19
omnetpp	INT06	0.36	10.11	gcc	INT06	0.45	6.26
zeusmp	FP06	0.82	4.69	cactusADM	FP06	0.60	4.51
bzip2	INT06	1.14	2.61	h264ref	INT06	1.46	1.28
vortex	INT00	1.01	1.24	gromacs	FP06	1.06	0.29
namd	FP06	2.25	0.18	calculix	FP06	2.28	0.05
gamess	FP06	2.32	0.04	povray	FP06	1.88	0.02

Table 2: Characteristics of 20 SPEC 2000/2006 benchmarks: IPC and MPKI (L2 cache Misses Per 1K Instructions)

FST parameters used in evaluation: Table 3 shows the values we use in our evaluation unless stated otherwise. There are eight aggressiveness levels used for the request rate of each application: 2%, 3%, 4%, 5%, 10%, 25%, 50% and 100%. These levels denote the scaling of the MSHR quota and the request rate in terms of percentage. For example, when FST throttles an application to 5% of its total request rate on a system with 128 MSHRs, two parameters are adjusted. First, the application is given a 5% quota of the total number of available MSHRs (in this case, 6 MSHRs). Second, the application's memory requests in the MSHRs are issued to access the L2 cache at 5% of the maximum possible frequency (i.e., once every 20 cycles).

$Fairness\ Threshold$	$Successive\ Fairness\ Achieved\ Intervals\ Threshold$	$Intervals\ Wait\ To\ Throttle\ Up$	$Interval\ Length$
1.4	4	2	25Kinsts

$Switch_{thr}$	$Interference_{thr}$	$SwitchBack_{thr}$
5%	70%	3 intervals

Table 3: FST parameters

Metrics: To measure CMP system performance, we use *Harmonic mean of Speedups (Hspeedup)* [18], and *Weighted Speedup (Wspeedup)* [28]. These metrics are commonly used in measuring multi-program performance in computer architecture. In order to demonstrate fairness improvements, we report *Unfairness* (see Section 3.1), as defined in [7, 22]. Since *Hspeedup* provides a balanced measure between fairness and system throughput as shown in previous work [18], we use it as our primary evaluation metric. In the metric definitions below: N is the number of cores in the CMP system, IPC^{alone} is the IPC measured when an application runs alone on one core in the CMP system (other cores are idle), and IPC^{shared} is the IPC measured when an application runs on one core while other applications are running on the other cores.

$$Hspeedup = \frac{N}{\displaystyle\sum_{i=0}^{N-1} \frac{IPC_i^{alone}}{IPC_i^{shared}}}, \quad Wspeedup = \sum_{i=0}^{N-1} \frac{IPC_i^{shared}}{IPC_i^{alone}}$$

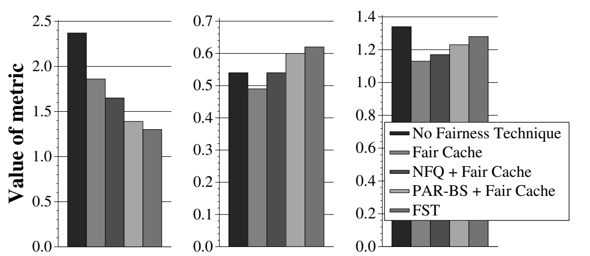
5. Experimental Evaluation

We evaluate our proposed techniques on both 2-core and 4-core systems. We compare FST to four other systems in our evaluations:

1) a baseline system with no fairness techniques employed in the shared memory system, using LRU cache replacement and FR-FCFS memory scheduling [27], both of which have been shown to be unfair [15, 21, 24]. We refer to this baseline as *NoFairness*, 2) a system with only fair cache capacity management using the virtual private caches technique [25], called *FairCache*, 3) a system with a network fair queuing (NFQ) fair memory scheduler [24] combined with fair cache capacity management [25], called *NFQ+FairCache*, 4) a system with a parallelism-aware batch scheduling (PAR-BS) fair memory scheduler [23] combined with fair cache capacity management [25], called *PAR-BS+FairCache*.

5.1 2-core System Results

Figure 4 shows system performance and unfairness averaged (using geometric mean) across 18 workloads evaluated on the 2-core system. Figure 5 shows the Hspeedup performance of FST and other fairness techniques normalized to that of a system without any fairness technique for each of the 18 evaluated 2-core workloads. FST provides the highest system performance (in terms of Hspeedup) and the best unfairness among all evaluated techniques. We make several key observations:



(a) Unfairness (b) Hspeedup (c) Wspeedup

Figure 4: Average performance of FST on the 2-core system

1. Fair caching's unfairness reduction comes at the cost of a large degradation in system performance. This is because fair caching changes the memory access patterns of applications. Since the memory access scheduler is unfair, the fairness benefits of the fair cache itself are reverted by the memory scheduler.

2. NFQ+FairCache together improves system fairness by 30.2% compared to *NoFairness*. This degrades Wspeedup (by 12.3%). The combination of PAR-BS and fair caching improves both system performance and fairness compared to the combination of NFQ and a fair cache. The main reason is that PAR-BS preserves both DRAM bank parallelism and row-buffer locality of each thread better than NFQ, as shown in previous work [23]. Compared to the baseline with no fairness control, employing PAR-BS and a fair cache reduces unfairness by 41.3% and improves Hspeedup by 11.5%. However, this improvement comes at the expense of a large (7.8%) Wspeedup degradation.

NFQ+FairCache and PAR-BS+FairCache both significantly degrade system throughput (Wspeedup) compared to employing no fairness mechanisms. This is due to two reasons both of which lead to the delaying of memory non-intensive applications (Recall that prioritizing memory non-intensive applications is better for system throughput [23, 24]). First, the fairness mechanisms that are employed separately in each resource interact negatively with each other, leading to one mechanism (e.g. fair caching) increasing the pressure on the other (fair memory scheduling). As a result, even though fair caching might benefit system throughput by giving more resources to a memory non-intensive application, increased misses of the memory-intensive application due to fair caching causes more congestion in the memory system, leading to both the memory-intensive and non-intensive applications to be delayed. Second, even though the combination of a fair cache and a fair memory controller can prioritize a memory non-intensive application's requests, this prioritization can be temporary. The deprioritized memory-intensive application can still fill the shared MSHRs with its requests, thereby denying the non-intensive application entry into the memory system. Hence, the non-intensive application stalls because it cannot inject enough requests into the memory system. As a result, the memory non-intensive application's performance does not improve while the memory-intensive application's performance degrades (due to fair caching), resulting in system throughput degradation.

3. FST reduces system unfairness by 45% while also improving Hspeedup by 16.3% and degrades Wspeedup by 4.5% compared to *NoFairness*. Unlike other fairness mechanisms, FST improves both system performance and fairness, without large degradation to Wspeedup. This is due to two major reasons. First, FST provides a coordinated approach in which both the cache and the memory controller receive less frequent requests from the applications causing unfairness. This reduces the starvation of the applications that are unfairly slowed down as well as interference of requests in the memory system, leading to better system performance for almost all applications. Second, because FST uses *MSHR quotas* to limit requests injected by memory-intensive applications that cause unfairness, these memory-intensive applications do not deny other applications' entry into the memory system. As such, unlike other fairness techniques that do not consider fairness in memory system buffers (e.g., MSHRs), FST ensures that unfairly slowed-down applications are prioritized in the entire memory system, including all the buffers, caches, and schedulers.

Table 4 summarizes our results for the 2-core evaluations. Compared to the previous technique that provides the highest system throughput (i.e. NoFairness), FST provides a significantly better balance between system fairness and performance. Compared to the previous technique that provides the best fairness (PAR-BS+FairCache), FST improves both system performance and fairness. We conclude that FST provides the best system fairness as well as the best balance between system fairness and performance.

	Unfairness	Hspeedup	Wspeedup
FSTΔ over No Fairness Mechanism	-45%	16.3%	-4.5%
FSTΔ over Fair Cache	-30%	26.2%	12.8%
FSTΔ over NFQ + Fair Cache	-21.2%	16%	8.8%
FSTΔ over PAR-BS + Fair Cache	-6.3%	4.3%	3.4%

Table 4: Summary of results on the 2-core system

5.2 4-core System Results

5.2.1 Overall Performance

Figure 6 shows unfairness and system performance averaged across the ten evaluated 4-core workloads. FST provides the best fairness and Hspeedup among all evaluated techniques, while providing Wspeedup that is equivalent to that of the best previous technique. Overall, FST reduces unfairness by 44.4%[6] and increases system performance by 25.6% (Hspeedup) and 2.6% (Wspeedup) compared to *NoFairness*. Compared to PAR-BS+FairCache, the best performing previous technique, FST reduces unfairness by 36.2% and in-

[6] Similarly, FST also reduces the coefficient of variation, an alternative unfairness metric, by 45%.

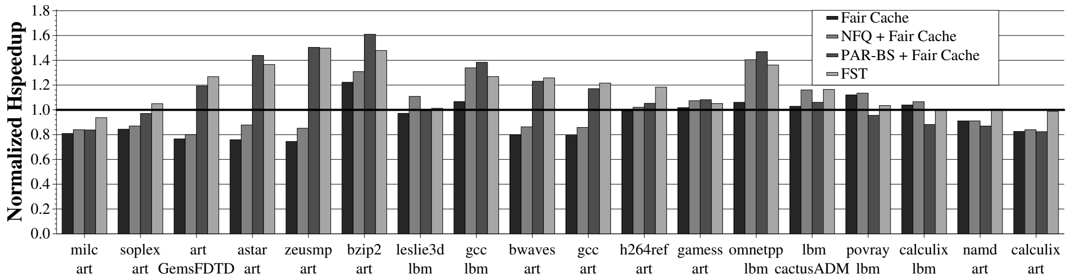
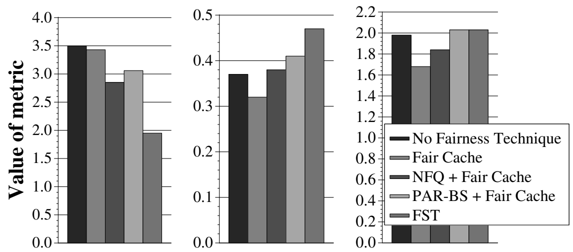


Figure 5: Hspeedup of 18 2-core workloads normalized to no fairness control

creases Hspeedup by 14.5%. FST's large performance improvement is mainly due to the large reduction in unfairness.[7]



(a) Unfairness (b) Hspeedup (c) Wspeedup

Figure 6: Average performance of FST on the 4-core system

Note that the overall trends in the 4-core system are similar to those in the 2-core system except that previous fairness mechanisms do not significantly improve fairness in the 4-core system. As explained in detail in Section 5.2.2, this is due to prioritization of non-intensive applications in individual resources by previous fairness mechanisms regardless of whether or not such applications are actually slowed down.

Figure 7 shows the harmonic speedup performance of FST and other fairness techniques normalized to that of a system without any fairness technique for each of the ten workloads. We make two major conclusions. First, FST improves system performance (both Hspeedup and Wspeedup) and fairness (not shown in this figure) compared to no fairness control for almost all workloads. Second, FST provides the highest Hspeedup compared to the best previous technique (PAR-BS+FairCache) on eight of the ten workloads. In the two workloads that PAR-BS+FairCache has higher performance, it is due to PAR-BS's ability to gain higher system throughput by preserving bank parallelism at the memory controller. Using bank parallelism preserving techniques in conjunction with FST can improve its performance further in such cases. We conclude that FST's performance and fairness benefits are consistent across workloads.

5.2.2 Case Study

To provide more insight into the performance and fairness improvements of FST, we analyze one 4-core workload in detail. This workload is a mix of applications of different levels of memory intensity. *Art* and *astar* are both highly memory-intensive, while *h264ref* is less so and *gromacs* is non-intensive (as shown in Table 2). When these applications are run simultaneously on a 4-core system with no

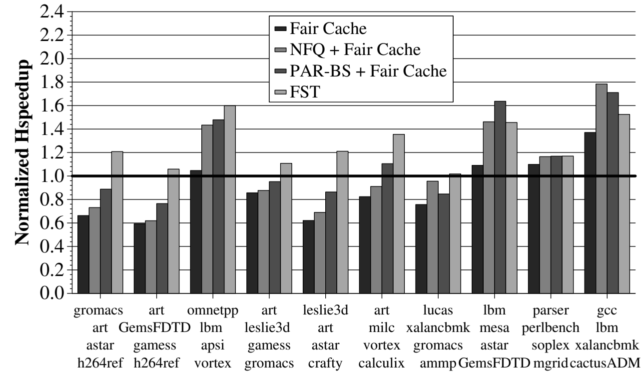


Figure 7: Normalized speedup of 10 4-core workloads

fairness control, the two memory-intensive applications (especially *art*) generate a large amount of memory traffic. *Art*'s large number of memory requests to the shared resources unfairly slows down all other three applications, while *art* does not slow down significantly. Figures 8 and 9 show individual benchmark performance and system performance/fairness, respectively (note that Figure 8 shows speedup over alone run which is the inverse of individual slowdown, defined in Section 3.1). Several observations are in order:

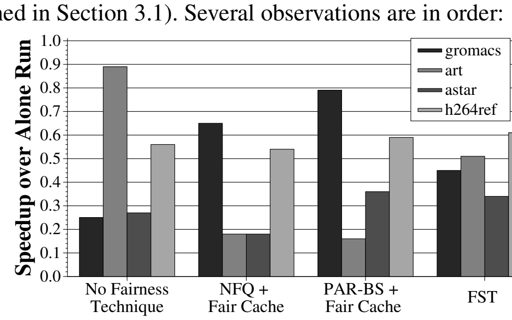
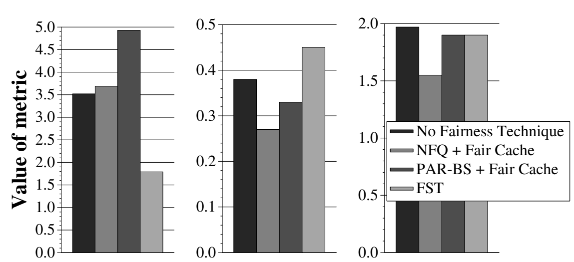


Figure 8: Case Study: individual application behavior



(a) Unfairness (b) Hspeedup (c) Wspeedup

Figure 9: Case study: system behavior

1. NFQ+FairCache significantly degrades system performance by 26.9% (Hspeedup) and 21.2% (Wspeedup) compared to no fairness control. All applications but the non-intensive *gromacs* are

[7] Since relative slowdowns of different applications are most important to improving unfairness and performance using FST, highly accurate T_{excess} estimations are not necessary for such improvements. However, we find that with the mechanisms proposed in this paper the application which causes the most interference for the most-slowed-down application is on average identified correctly in 70% of the intervals.

slowed down, causing unfairness to also increase by 4.9% compared to employing no fairness technique. The largest slowdowns are experienced by the memory-intensive *art* and *astar*, which are deprioritized in the memory system by NFQ because they are memory-intensive. Second, PAR-BS+FairCache degrades system performance (Hspeedup) by 11% and makes the system 40% more unfair. As *astar* is less memory-intensive than *art*, PAR-BS prioritizes *astar* over *art* within each batch of memory requests due to its shortest-job-first (i.e., least-memory-intensive-thread-first) prioritization policy among threads. This causes *art* to slow down significantly while all other applications speed up, thereby leading to large system unfairness compared to no fairness control.

2. We found that both PAR-BS+FairCache and NFQ+FairCache overly deprioritize memory-intensive applications in this workload, because they do not explicitly detect when such applications cause slowdowns for others. These techniques simply prioritize non-intensive applications all the time regardless of whether or not they are actually slowed down in the memory system. In contrast, our approach explicitly detects when memory-intensive applications are causing unfairness in the system. If they are not causing unfairness, FST does not deprioritize them. As a result, their performance is not unnecessarily reduced. This effect is observed by examining the most memory-intensive application's (*art*'s) performance with FST. With FST, *art* has higher performance than with any of the other fairness techniques.

3. FST increases system performance by 20.8% (Hspeedup) while reducing unfairness by 49.2% compared to no fairness control. This comes at a small loss in system throughput. In this workload, the memory-intensive *art* and *astar* cause significant interference to each other in all shared resources and to *gromacs* in the shared cache. Unlike other fairness techniques, FST dynamically tracks the interference and the unfairness in the system in a fine-grained manner. When the memory-intensive applications are causing interference and increasing unfairness, FST throttles the offending *hog* application(s). In contrast, when the applications are not interfering significantly with each other, FST allows them to freely share resources in order to maximize each application's performance. The fine-grained dynamic detection of unfairness and enforcement of fairness mechanisms only when they are needed allow FST to achieve higher system performance (Hspeedup) and a better balance between fairness and performance than other techniques.

To provide insight into the dynamic behavior of FST, Figure 10 shows the percentage of time each core spends at each throttling level. FST significantly throttles down *art* much of the time (but not always) to reduce the inter-core interference it generates for all other applications. As a result, *art* spends almost 50% of its execution time at 10% or less of its full aggressiveness. However, even at low throttling levels, *art* can prevent bank service to the continuous accesses of *astar* to the same bank. FST detects this and disallows *art*'s requests to be prioritized based on row-buffer hits for 40% of all intervals, preventing *art* from causing bank service denial as described in Section 3.5. Note that *art* spends approximately 15% of its time at throttling level 100, which shows that FST detects times when art is not causing large interference and does not penalize it. Figure 10 also shows that FST detects interference caused by not only *art* but also other applications. *Astar*, *h264ref*, and even *gromacs* are detected to generate high inter-core interference for other applications in certain execution intervals. As such, FST dynamically adapts its fairness control decisions to the interference patterns of applications rather than simply prioritizing memory non-intensive applications. Therefore, unlike other fairness techniques, FST does not overly deprioritize *art* in the memory system.

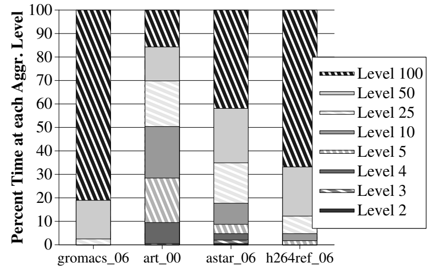


Figure 10: Case Study: application throttling levels

We conclude that FST provides a higher-performance approach to attaining fairness than coarsely tracking the memory-intensity of applications and deprioritizing memory-intensive applications without dynamic knowledge of interference and unfairness. FST achieves this by tracking unfairness in the system and making fairness/throttling decisions based on that tracking in a finer-grained manner.

5.3 Effect of Throttling Mechanisms

As described in Section 3.2, FST uses the combination of two mechanisms to throttle an application up/down and increase/decrease its request rate from the shared resources: 1) Applying an *MSHR quota* to each application, 2) Adjusting the frequency at which requests in the MSHRs are issued to access L2. Section 3.5 explains how to prevent bank service denial from FR-FCFS memory scheduling within FST. Figure 11 shows the effect of each of the different throttling mechanisms and FST on the 4-core system. Using *MSHR quotas* is the more effective of the two mechanisms. By itself, using *MSHR quotas* achieves 75% of the performance and the fairness improvement provided by FST. We conclude that using all mechanisms of FST is better than each throttling mechanism alone in terms of both fairness and performance.

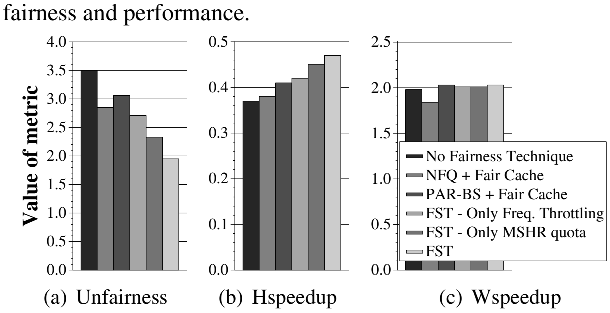


Figure 11: Effects of different throttling mechanisms for FST

5.4 Evaluation of System Software Support

Enforcing Thread Priorities: As explained in Section 3.4, FST can be configured by system software to assign different weights to different threads. As an example of how FST enforces thread weights, we ran four identical copies of the *GemsFDTD* benchmark on a 4-core system and assigned them *thread weights* of 1, 1, 4 and 8 (recall that a higher-weight thread is one the system software wants to prioritize). Figure 12 shows that with no fairness technique each copy of *GemsFDTD* has an almost identical slowdown as the baseline does not support thread weights and treats the applications identically in the shared memory system. However, FST prioritizes the applications proportionally to their weights, favoring applications with higher weight in the shared memory system. FST also slows down the two copies with the same weight by the same amount. We conclude that FST approximately enforces thread weights, thereby easing the development of system software which naturally expects a CMP to respect thread weights in the shared memory system.

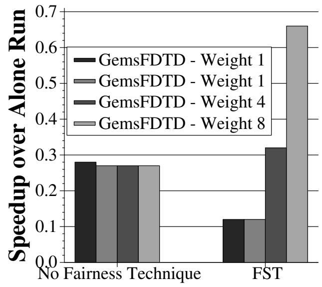


Figure 12: Enforcing thread weights with FST

Enforcing an Alternative Fairness Objective (Maximum Tolerable Slowdown): Section 3.4 explained how FST can be configured to achieve a *maximum slowdown threshold* as determined by system software, that dictates the maximum tolerable slowdown of any individual application executing concurrently on the CMP. Figure 13 shows an example of how FST enforces this fairness objective when four applications are run together on a 4-core system. The figure shows each application's individual slowdown in four different experiments where each experiment uses a different maximum slowdown threshold (ranging from 2 to 3) as set by the system software. As tighter goals are set by the system software, FST throttles the applications accordingly to achieve (close to) the desired maximum slowdown. The fairness objective is met until the maximum slowdown threshold becomes too tight and is violated (for *mgrid and parser*), which happens at threshold value 2. We conclude that FST can enforce different system-software-determined fairness objectives.

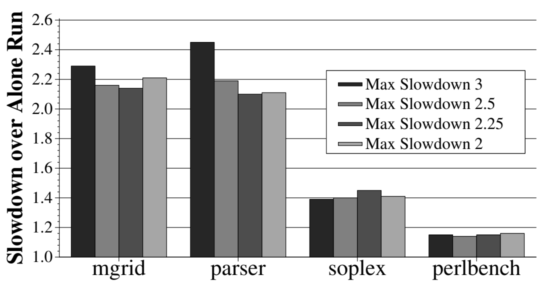


Figure 13: Enforcing maximum slowdown with FST

5.5 Hardware Cost

Table 5 shows FST's required storage. FST does not require any structure or logic that is on the critical path.

6. Related Work

To our knowledge, this paper provides the first comprehensive and flexible hardware-based solution that enables system-software-specified fairness goals to be achieved in the entire shared memory system of a multi-core processor, without requiring fairness mechanisms to be implemented individually in each shared resource.

Prior work in providing fairness in different shared resources of CMP systems focused on fair caching [12, 13, 15, 25], fair memory scheduling [22–24], and fair on-chip interconnects [4, 9, 17]. We have already provided extensive qualitative and quantitative comparisons showing that our mechanism significantly improves system fairness and performance compared to systems employing the combination of state-of-the-art fair cache capacity management [25] and fair memory scheduling [23, 24].

Bitirgen et al. [2] propose implementing an artificial neural network that learns each application's performance response to different resource allocations. Their technique searches the space of different resource allocations among co-executing applications to find a partitioning in the shared cache and memory controller that improves performance. In contrast to FST, this mechanism requires that resource-based fairness/partitioning techniques are already implemented in each individual resource. In addition, it requires relatively more complex, black-box implementation of artificial neural networks in hardware.

Herdrich et al. [10] observe that the interference caused by a lower-priority application on a higher-priority application can be reduced using existing clock modulation techniques in CMP systems. However, their proposal does not consider or provide fairness to equal-priority applications. Zhang et al. [32] propose a software-based technique that uses clock modulation and prefetcher on/off control provided by existing hardware platforms to improve fairness in current multi-core systems compared to other software techniques. Neither of these prior works propose an online algorithm that dynamically controls clock modulation to achieve fairness. In contrast, FST provides: 1) hardware-based architectural mechanisms that continuously monitor shared memory system unfairness at run-time and 2) an online algorithm that, upon detection of unfairness, throttles interfering applications using two new hardware-based throttling mechanisms (instead of coarse-grained clock modulation) to reduce the interfering applications' request rates.

Jahre and Natvig [14] observe that adjusting the number of available MSHRs can control the total miss bandwidth available to each thread running on a CMP. However, this prior work does not show how this observation can be used by an online algorithm to dynamically achieve a well-defined fairness or performance goal. In contrast to this prior work, our work 1) provides architectural support for achieving different well-defined system-software fairness objectives while also improving system performance, 2) shows that using complementary throttling mechanisms and preventing bank service denial due to FR-FCFS, as done by FST, provides better fairness/performance than simply adjusting the number of available MSHRs (see Section 5.3), 3) shows that FST's approach of throttling sources based on unfairness feedback provides better system fairness/performance than implementing different fairness mechanisms in each individual shared resource.

Prior work on SMT processors (e.g., [3, 18, 19, 30]) propose fetch policies to improve performance and/or fairness in such processors. These techniques are not applicable to the problem we address, since they mainly address sharing of execution pipeline resources and not the shared memory system. Eyerman and Eeckhout [5] propose a technique to estimate the execution times of simultaneously running threads had they been run alone. This work estimates interference in the execution resources and does not deal with memory system interference in a detailed manner. As such, our proposed memory interference/slowdown estimation and source throttling techniques are orthogonal to this prior work.

	Cost for N cores	Cost for N = 4
$ExcessCycles$ counters	N × N × 16 bits/counter	256 bits
Interference pollution filter per core	2048 entries × N × (1 pollution bit + (log_2 N) bit processor id)/entry	24,576 bits
$InterferingCoreId$ per MSHR entry	32 entries/core× N × 2 interference sources × (log_2 N) bits/entry	512 bits
$InterferencePerCore$ bit-vector	3 interference sources × N × N × 1 bit	48 bits
Shadow row-buffer address register	N × # of DRAM banks (B) × 32 bits/address	1024 bits
$Successive\ Fairness\ Achieved\ Intervals$ counter $Intervals\ To\ Wait\ To\ Throttle\ Up$ counter per core $Inst\ Count\ Each\ Interval$ per core	(2 × N + 1) × 16 bits/counter	144 bits
Core id per tag store entry in K MB L2 cache	16384 blocks/Megabyte × K × (log_2 N) bit/block	65,536 bits
Total hardware cost for N-core system	Sum of the above	92092 = **11.24 KB**
Percentage area overhead (as fraction of the baseline K MB L2 cache)	Sum (KB) × 100 / (K × 1024)	11.24KB/2048KB = **0.55%**

Table 5: Hardware cost of FST on a 4-core CMP system

7. Conclusion

We proposed a low-cost architectural technique, Fairness via Source Throttling (FST), that allows system-software fairness policies to be achieved in CMPs by enabling fair sharing of the entire memory system. FST eliminates the need for and complexity of multiple complicated, specialized, and possibly contradictory fairness techniques for different shared resources. The key idea of our solution is to gather dynamic feedback information about the slowdowns experienced by different applications in hardware at run-time and, based on this feedback, collectively adjust the memory request rates of sources (i.e., cores) to balance applications' slowdowns. Our solution ensures that fairness decisions in the entire memory system are made in tandem, thereby significantly improving both system performance and fairness compared to the state-of-the-art *resource-based* fairness techniques implemented independently for different shared resources. We have also shown that FST is configurable by system software, allowing it to enforce thread priorities and achieve different fairness objectives. We conclude that FST provides a promising low-cost substrate that can not only improve the performance and fairness of future multi-core systems but also ease the task of future multi-core system software in managing shared on-chip hardware resources.

Acknowledgments

Many thanks to Khubaib, Boris Grot, Carl Waldspurger, the HPS research group, and the anonymous reviewers for their comments and suggestions. We also thank Benjamin Lee for his feedback in earlier stages of this work. We gratefully acknowledge the support of the Cockrell Foundation, Intel, Microsoft, GSRC, and the Carnegie Mellon University Software Engineering Institute and CyLab. We also acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources.

References

[1] Advanced Micro Devices. AMD's six-core Opteron processors. http://techreport.com/articles.x/17005, 2009.

[2] R. Bitirgen et al. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *MICRO-41*, 2008.

[3] F. J. Cazorla et al. QoS for high-performance SMT processors in embedded systems. *IEEE Micro*, 2004.

[4] R. Das et al. Application-aware prioritization mechanisms for on-chip networks. In *MICRO*, 2009.

[5] S. Eyerman and L. Eeckhout. Per-thread cycle accounting in SMT processors. In *ASPLOS*, 2009.

[6] A. Fedorova et al. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *PACT*, 2007.

[7] R. Gabor et al. Fairness and throughput in switch on event multithreading. In *MICRO-39*, 2006.

[8] A. Glew. MLP yes! ILP no! In *ASPLOS Wild and Crazy Idea Session '98*, Oct. 1998.

[9] B. Grot et al. Preemptive virtual clock: A flexible, efficient, and cost-effective QoS scheme for networks-on-a-chip. In *MICRO*, 2009.

[10] A. Herdrich et al. Rate-based QoS techniques for cache/memory in CMP platforms. In *ICS*, 2009.

[11] Intel. First the tick, now the tock: Next generation Intel microarchitecure (Nehalem). *Intel Technical White Paper*, 2008.

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

[13] R. Iyer et al. QoS policies and architecture for cache/memory in CMP platforms. In *SIGMETRICS*, 2007.

[14] M. Jahre and L. Natvig. A light-weight fairness mechanism for chip multiprocessor memory systems. In *Computing Frontiers*, 2009.

[15] S. Kim et al. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *PACT*, 2004.

[16] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *ISCA-8*, 1981.

[17] J. W. Lee et al. Globally-synchronized frames for guaranteed quality-of-service in on-chip networks. In *ISCA-35*, 2008.

[18] K. Luo et al. Balancing throughput and fairness in SMT processors. In *ISPASS*, 2001.

[19] K. Luo et al. Boosting SMT performance by speculation control. In *IPDPS*, 2001.

[20] Micron. *Datasheet: 2Gb DDR3 SDRAM, MT41J512M4 - 64 Meg x 4 x 8 banks, http://download.micron.com/pdf/datasheets/dram/ddr3*.

[21] T. Moscibroda and O. Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *USENIX Security*, 2007.

[22] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *MICRO-40*, 2007.

[23] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *ISCA-35*, 2008.

[24] K. J. Nesbit et al. Fair queuing memory systems. In *MICRO-39*, 2006.

[25] K. J. Nesbit et al. Virtual private caches. In *ISCA-34*, 2007.

[26] H. Patil et al. Pinpointing representative portions of large intel itanium programs with dynamic instrumentation. In *MICRO-37*, 2004.

[27] S. Rixner et al. Memory access scheduling. In *ISCA-27*, 2000.

[28] A. Snavely and D. M. Tullsen. Symbiotic job scheduling for a simultaneous multithreading processor. In *ASPLOS-IX*, 2000.

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

[30] D. M. Tullsen et al. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *ISCA-23*, 1996.

[31] O. Wechsler. Inside Intel core microarchitecure. *Intel Technical White Paper*, 2006.

[32] X. Zhang et al. Hardware execution throttling for multi-core resource management. In *USENIX*, 2009.