Contention-aware scheduling policies for fairness and throughput

Alexandros-Herodotos HARITATOS, Nikela PAPADOPOULOU,
Konstantinos NIKAS, Georgios GOUMAS and Nectarios KOZIRIS
National Technical University of Athens
Computing Systems Laboratory

Abstract. This paper presents a fast and simple contention-aware scheduling policy for CMP systems that relies on information collected at runtime with no additional hardware support. Our approach is based on a classification scheme that detects activity and possible interference across the entire memory hierarchy, including both shared caches and memory links. We schedule multithreaded applications taking into account their class, targeting both total system throughput and application fairness in terms of fair distribution of co-execution penalties. We have implemented a user level scheduler and evaluated our policy in several scenarios with different contention levels and a variety of multiprogrammed multithreaded workloads. Our results demonstrate that the proposed scheduling policy outperforms the established Linux and Gang schedulers, as well as a number of research contention-aware schedulers, both in system throughput and application fairness.

Keywords.

1. Introduction

Chip Multi-Processor (CMP) designs are the dominant paradigm in server, desktop, HPC and mobile computing environments. CMPs integrate two or more cores onto a single die with one or more private levels of cache memory, shared levels of cache memory, shared memory links and memory controllers. This high level of resource sharing can potentially create competition between executing threads that can in turn lead to severe performance degradation [29]. On the software side, the slowdown in the single core performance improvement has boosted multithreaded software implementation, leading CMP computing platforms to be heavily populated by multiprogrammed multithreaded (MPMT) workloads. This creates an interesting challenge for the operating system to optimize a number of critical metrics like throughput, fairness, energy/power consumption and others, within a non-trivial hardware resource configuration.

There exist two established scheduling approaches that can be applied to MPMT workloads: a) The *Completely Fair Scheduler* (CFS) [22] used in Linux, that targets fairness in terms of equal waiting time per thread, irrespective of the parent application of each thread. This is a sensible policy for desktop applications on single-core systems, where the CPU is a scarce resource and interaction with human users is a top priority. However, it does not seem an optimal fit for multicore/manycore systems and throughput-sensitive applications. Additionally, CFS does not consider resource sharing effects and is completely contention agnostic. b) The *Gang scheduler* [11], on the other hand, focuses on parallelism and schedules the threads of a parallel application simultaneously on the available cores, in an attempt to minimize communication and synchronization overheads, or *lock contention* [16]. Still, the Gang scheduler, similar to CFS, does not consider contention over shared hardware resources like caches and memory links and, thus, can suffer from performance degradation due to these factors.

A large number of research works has been devoted to alleviating the effects of resource contention using either the operating system [15,5,20,3,16], the data center resource manager [18,7,8], or the supercomputer scheduler [6]. Contention-aware scheduling focuses on the minimization of execution interference and typically relies on some kind of application *classification* derived from the utilization of resources or the explicit penalties under co-execution scenarios. Classifiers make use of a variety of application features, ranging from simple, like the LLC miss rate [5] or memory link bandwidth [3,20], to more elaborate, such as cache utilization patterns [27,17,13], contentiousness and sensitivity [26], and wider combinations of execution features, which are collected either online [16] or by application profiling [7] and analyzed with machine learning or data mining techniques. Once the classification has finished, the *scheduling* step may decide on a number of actions like node allocation in a cloud environment, space partitioning (thread mapping), time quantum allocation, DVFS levels, memory management policies, thread throttling and others, with the goal to maximize one or more objectives such as throughput, fairness or power consumption.

Despite this heavy research effort towards contention-aware scheduling, the schedulers of modern operating systems still remain contention agnostic. In our view, this is due to the fact that the contention-handling module of the OS needs to have a set of features not fully existing in prior work. In particular, we argue that a resource-aware policy should: a) Be simple to facilitate integration with existing OS projects, but accurate enough to identify the majority of contention situations. LLC miss rate and memory link bandwidth alone are unable to capture several interference scenarios as we discuss later on. b) Not rely on information that needs to be collected from compiler analysis (e.g. stack distance profiles [27]), additional hardware components [13], or execution information collected by heavy profiling and possible co-execution with behavior-revealing microbenchmarks [7,10,18]. c) Incur low overheads both in the classification and scheduling steps, relying on low complexity algorithms to support fast decisions in a dynamic OS environment, where processes enter and leave the system, perform I/O and exhibit execution phases with different behavior.

In this paper we propose a *contention-aware policy suitable for integration into an operating system*, that achieves low contention effects and distributes them among processes in a fair manner. Our policy relies on a simple classifier called SCN, that distinguishes between three application classes: *S*treaming applications, *C*ache intensive applications and applications restricting their work to the private parts of the hardware, and thus participate in *N*o contention situations. On top of SCN, we gradually build four collaborating scheduling techniques: a) the Thread Folding Technique (TFT) that halves the cores assigned to each application; b) the Link and Cache-Aware (LCA) scheduler [12] adapted to the SCN classification scheme, that avoids pairing highly competitive applications; c) the Fair On Progress (FOP) scheduler that compensates the applications suffering the most from contention by assigning them more time, thus efficiently distributing degradation among all applications, and d) the Folding LCA (F-LCA), a combination of all the above techniques aiming at improving both fairness and throughput without increasing the complexity of state-of-the-art scheduling schemes. Our policies are implemented in a user-space co-scheduling framework and applied to several scheduling scenarios with MPMT workloads on an 8-core CMP server. Our experimental results indicate that F-LCA is able to improve fairness over CFS while, at the same time, improving the average throughput, in some cases over 20%.

The rest of the paper is organized as follows: In Section 2 we define our scheduling problem. Section 3 presents the SCN classification method. The proposed scheduling strategies and the F-LCA scheduler are presented in Section 4. We compare F-LCA with Linux CFS and research scheduling policies in Section 5. Section 6 discusses further steps to incorporate our policy in a contention-aware OS scheduler and Section 7 presents related work. Finally, conclusions and future work can be found in Section 8.

2. Scheduling threads on CMPs

A generic co-scheduling scenario involving MPMT workloads comprises n applications $(r_1, ..., r_n)$ containing t_i threads each, that request to be executed on a CMP platform with p processors, l memory links and c shared caches. This is a typical scenario for time-shared desktop systems, cloud environments offered by oversubscribed data centers and compute nodes in HPC systems.

As threads running on different cores try to access shared resources, they interfere with each other leading to substantial performance degradation. Previous work has indicated shared memory links and LLC as spots of contention [15,4,20,23]. Memory bandwidth is a limited resource that can be easily exhausted when threads with high demands on data stored in main memory are running simultaneously using the same memory link. Moreover, an application's activity in LLC may lead a co-runner application to suffer from a large number of cache line evictions.

Contention-aware schedulers attempt to overcome contention by separating contending threads in *space*, i.e. by assigning them on cores that do not have shared resources, or in *time*, i.e. by scheduling them in different time slots. A special problem for multithreaded applications is lock contention. Gang scheduling assigns all threads of an application to be executed concurrently on different cores, minimizing the effects of lock contention.

Regarding their objectives, contention-aware schedulers aim to improve the overall throughput, provide QoS or minimize power consumption, while the default Linux scheduler (CFS) targets fairness based on the time each application is waiting to be executed. Although CFS is extremely fair on waiting time, it neglects fairness in terms of throughput and, due to contention on shared resources, applications meet different levels of slowdown. Our approach aims to improve the overall throughput and at the same time achieve fairness by distributing co-execution penalties fairly.

2.1. Scheduling threads on a flat CMP architecture

To better clarify the potential of the proposed scheduling policies, we work on a simplified scheduling problem, where each one of multiple multithreaded programs employ a number of threads equal to the available cores ($t_i = p$). In the target architecture all cores share a single LLC and memory link, i.e. $l = c = 1$. This simplification offers a clear picture of how applications interact in a system where resource sharing is unavoidable. In Section 6 we discuss how we these simplifications can be overcome.

We consider the following state-of-the art schedulers to address the aforementioned problem:

Gang scheduler [11]: It schedules all p threads of each application in a single time slot and circulates the applications in a round-robin fashion. This scheduler exposes each application's behavior when executed in isolation, although *self-interference*, i.e. interference between an application's threads, may still occur.

Link bandwidth balance (LBB) scheduler [20,3]: It sorts applications by bandwidth consumption and forms execution groups by selecting the first from the top and bottom of the list, then the second from the top and bottom and so on. All threads of an application are executed within the same time quantum.

LLC miss rate balance (LLC-MRB) scheduler [5,3]: Similar to the *LBB* scheduler but it sorts the applications by their LLC miss rate.

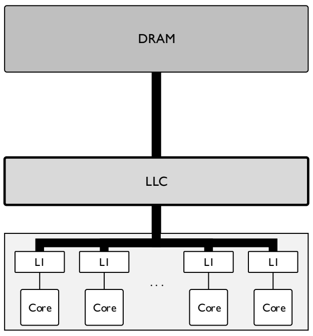
Completely Fair Scheduler (CFS) [22]: Incorporated in the Linux kernel since version 2.6.23, it maintains a run queue for each processing core implemented as a red black tree and treats every thread as an independent scheduling unit. CFS is contention unaware and serves as the state-of-the-art scheduling baseline for desktop, cloud and HPC systems.

Link and Cache contention aware scheduler (LCA) [12]: LCA is based on a four class classification method. A simple pairing algorithm prevents highly competitive applications from being co-scheduled.

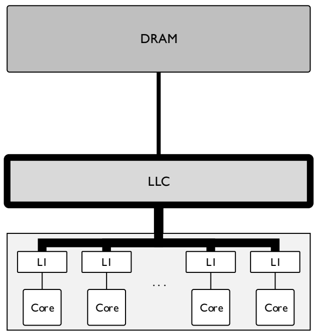
3. SCN Application Classification

In this section we present our scheme that classifies applications based on their activities on the two main shared resources, i.e. the memory link and the LLC. Our goal is to identify contention fast on both resources, relying solely on monitoring information that can be collected during execution time from the existing monitoring facilities of modern processors. Our scheme is named *SCN* after the three application classes that are identified as described next and illustrated in Figure 1:

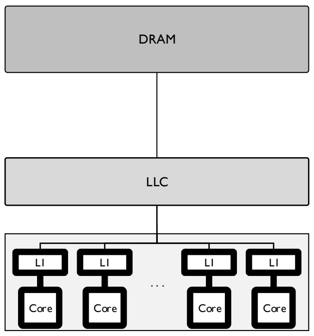
Class *S* (*Streaming*): Applications of this class have a stable data flow on all links of the memory hierarchy. This class typically includes applications that exhibit one or more of the following characteristics: they perform streaming memory accesses on data sets that largely exceed the size of the LLC, and have either no or large reuse distances. Although they fetch data on the entire space of the LLC due to their streaming nature, they do not actually reuse them either because their access pattern does not recur to the same data, or because they have been swept out of the cache. No level of cache memories help *S* applications accelerate their execution. Thus, they tend to pollute all levels of the cache hierarchy.



(a) class S



(b) class C



(c) class N

Figure 1. Activity in application classes

Class *C* (*Cache sensitive*): Applications with high activity on the shared LLC. This is a wide class including members with a combination of main memory accesses and LLC data reuse and members with varying characteristics, such as those that operate on small data sets with heavy reuse, optimized code for the LLC (e.g. via cache blocking with a block size fitting the LLC), or latency-bound applications that make irregular data accesses and benefit a lot from LLC hits.

Class *N* (*No contention*): Applications that restrict their activity either to the private part of the memory hierarchy or within the core. The members of this class create no contention on the shared system resources and include applications with heavy computations, very small working sets or optimized data reuse that can be serviced by the private caches.

3.1. Classification method

Having defined the application classes, we need a concrete method to perform the classification using runtime information. The core idea is to inspect the data path from the main memory down to the core to locate components with high utilization (Figure 2). We have focused only on the stream flowing towards the core, as we have empirically found that this direction concentrates the largest portion of contention.

Our classification method implements the decision tree shown in Figure 3. We follow a hierarchical approach in the classification. First, we look at the application's activity in the L1 cache, i.e. data flowing into the L1 cache. No or low activity in L1 means that only limited amount of data are fetched from the entire memory hierarchy, indicating that the application's activity is restrained within the core. Such applications are classified as N.

In the case of high activity, i.e. large amount of data are flowing towards the core, we need to check whether they are reused. We define the reuse factor of cache level i as the ratio $CR_i = \frac{B_{in_{i-1}}}{B_{in_i}}$, where B_{in_i} the inbound bandwidth to cache level i consumed by data flowing from higher levels of memory. The rationale is simple: if data flows out of a cache towards the direction of the core with a much higher rate than it flows in, then we can safely assume that reuse is present. We empirically set a threshold of $CR_i = 2$ to designate reuse.

If no reuse is detected, then the application has a streaming attitude and is classified as S. On the other hand, if reuse is detected, the application class depends on the reuse location. If reuse is higher in the private caches, then the application is classified as N; otherwise, reuse is higher in the shared LLC and the application is classified as C.

The classification cost is very small. It comprises the collection of information from the system's performance counters (typically a few dozens of CPU cycles in modern processors), the computation of the reuse factors (as many integer divisions as the levels of the memory hierarchy) and the traversal of the decision tree that requires at most three comparisons. Classification needs to take place whenever an application is entering the system or it changes its execution phase. Although phase change detection [9] is beyond the scope of this work, a change in the monitoring data could provide a hint of a possible phase change. We intend to further investigate this aspect in future work. Finally, for a safe classification, each application needs to be scheduled without any interference, i.e by applying gang scheduling for a few time quanta (in our experimentation we utilized two time quanta of 1 sec each). If there exist additional empty cores in the machine,

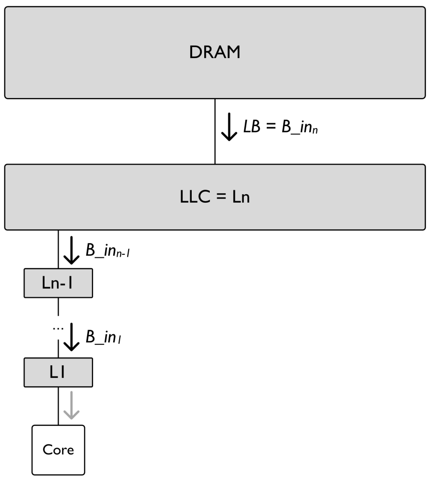


Figure 2. Inspected data flow in the memory hierarchy

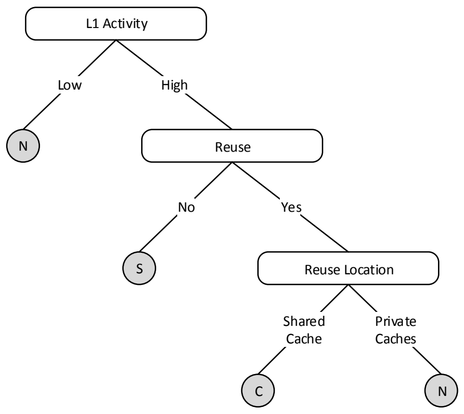


Figure 3. Decision tree for application classification

they could be employed by N class applications, since these will not interfere with the application under classification.

3.2. Co-execution effects

Despite the fact that inside a class one may find applications with quite different execution patterns, the classes themselves can be used to capture the big picture of how applications access common resources and how they interfere with each other. We define *xy* to be the co-execution of an application from class *x* with an application from class *y* and we use $*$ as a wildcard for "any class". Here is what we expect from all possible co-execution pairs:

N – *: As applications from class *N* do not actively employ any shared resource, this co-execution does not create interference to any of the applications.

S – *S*: Both applications compete for the memory link. The contention pattern in this case indicates that the shared resource, the memory link bandwidth, is divided, not necessarily equally, between the competing applications.

S – *C*: As *S* applications tend to pollute caches, a *C* application may suffer from the co-execution with *S* applications. When *S* is fetching data with at a low rate, it may cause little harm to *C* applications. If however, the data fetch rate is high, the co-execution can be catastrophic for the *C* application. The streaming nature of *S* applications causes data that might be heavily reused by *C* applications to be swept out of the shared cache rapidly, enforcing them to access main memory. This contention pattern can lead to a dramatic slowdown of *C* applications. On the other hand, *S* applications suffer no severe penalty from co-execution.

$C-C$: The effects of this co-execution are difficult to predict. In general we expect cache organization and replacement policies to be able to handle activity from different applications on the shared cache adequately. However, if both applications exhibit similar data access patterns, contention is expected to be high. To look at more details and better understand possible interactions and their effects, one should utilize information regarding the data allocated to each application on the cache and its access pattern. This would require either information from static analysis or additional hardware support, which are both out of the scope of this work.

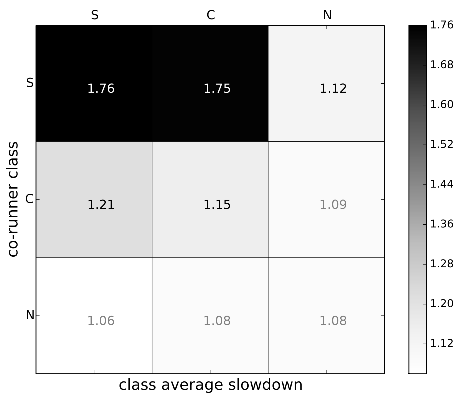


Figure 4. Average application slowdown due to co-execution. Along the x axis we show the slowdown imposed by each class, and along the y axis we show the slowdown suffered.

To evaluate our scheme we used multithreaded applications from the NAS [2], polybench [1], and stream benchmark suites [19], a pointer chasing benchmark [2] and two in-house implementations of a tiled Matrix Multiplication and a tiled LU decomposition. All applications were executed with Small, Medium and Large dataset sizes. More specifically, small datasets were a lot smaller than the LLC; medium datasets were slightly smaller than the LLC size, and Large were significantly larger than the LLC. Interestingly, depending on the dataset size, an application may populate different classes.

We co-executed all the possible pairs, measured the time to completion for each application and compared it to its standalone execution time. Figure 4 offers the general picture of the co-execution scenarios at the class level, that generally confirms our previous analysis: S are the most competitive applications, C are the most sensitive and N do not suffer from contention and do not cause any harm to their co-runners.

4. Scheduling policies for fairness and contention avoidance

Our approach towards a contention-aware scheduling module is composed of four steps. We first examine a thread folding technique, where pairs of applications are executed concurrently. Then, we discuss the LCA contention-aware scheduler, which is based on

[1]PolyBench: The Polyhedral Benchmark suite: `http://www.cs.ucla.edu/~pouchet/software/polybench`

[2]pChase benchmark: `https://github.com/maleadt/pChase`

the SCN classification and creates pairs of applications. The third step is a technique that can distribute performance degradation fairly across the entire workload. Finally, F-LCA is a scheduler that combines all the above techniques.

4.1. Thread Folding Technique – TFT

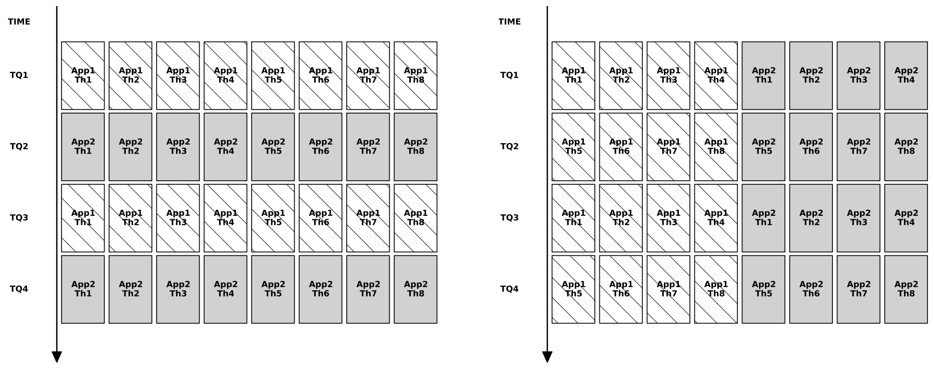
As mentioned before, a common practice for multithreaded applications is to request a number of threads equal to the number of cores provided by the underlying hardware. This practice however, is not always efficient and may lead to resource waste, as in several cases maximum throughput can be attained with less threads.

When multiple multithreaded applications are running concurrently on the system, there are two established scheduling approaches: the Gang scheduler and CFS. The main advantage of the Gang scheduler is that it minimizes lock contention. However, it turns out to be inefficient and waste resources for a large class of applications that fail to scale to the maximum available cores because of intra-application contention (self-interference). On the other hand, CFS does not necessarily execute all the threads of an application at the same time, as it treats each thread as a separate scheduling unit. Therefore, CFS may be more efficient by not wasting resources on applications exhibiting intra-application contention, but on the other hand it can increase lock contention degrading the performance of applications that would otherwise be able to scale.

Our approach is based on the observation that in a specific time window, the threads of two multithreaded applications can utilize the exact same amount of cores either by time or space sharing the system. In Figure 5a, the Gang scheduler enforces time sharing, i.e. at each time quantum only the threads of a single application are scheduled and the two applications are circulated in a round-robin fashion. In Figure 5b the cores assigned to each application are now halved and both applications are scheduled every time quantum. This thread folding technique (TFT) leads to pairs of threads from the same application time sharing a core, while at each time quantum the two applications are space sharing the system. In both cases, for a specific time window, an application is awarded the same amount of cpu time. However, in the presence of both scalable and non-scalable applications, we expect TFT to be more efficient than Gang scheduling, as it will be able to accelerate non-scalable applications by alleviating their self-interference, while imposing no harm to the scalable ones.

To validate this, we used an 8-core system (see Table 1) to execute each application of our suite (see Table 2) in isolation, using eight threads per application. We initially employ Gang scheduling; then, we rerun each application using TFT and restricting the 8 threads to only 4 cores. The first four threads run in a time quantum and the rest of the threads run in the next one, while the other 4 cores remain always empty. Figure 6 presents the execution time using TFT normalized to the execution time using the Gang scheduler. We use boxplots (Tukey boxplots) to present our experimental results for the different application classes. For an explanation of this form of plots please refer to Section 5.3.

It is evident that *S* applications do not suffer from folding, as they cannot scale to the maximum available cores. On the other hand, *N* and the majority of *C* applications nearly double their execution time when using half of the available cores. Based on these, when we employ TFT instead of Gang for pairs of multithreaded applications, we expect lower execution times for *S* and few *C* applications, as, due to space sharing, both applications



(a) Gang scheduler (b) TFT

Figure 5. Core allocations per time quantum for two 8-threaded applications on an 8-core system

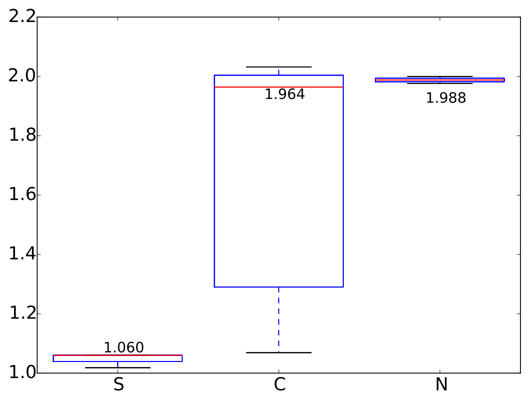


Figure 6. TFT vs full system execution time

are scheduled every time quantum instead of every other. Similarly, we expect *N* and the majority of *C* applications to maintain almost the same execution time. Of course, the execution times of the latter depend on the existence or not of contention created by a competitive co-runner.

To evaluate TFT we executed the applications in pairs, with every pair running for 100 seconds. During this time, whenever an application terminated, it was re-spawned. We define throughput as the number of times an application terminates within the time window, normalize it to the throughput achieved by Gang scheduling, and present it in Figure 7, side by side with the throughput obtained using CFS.

By folding the threads and space sharing the system, we manage to maintain and in some cases improve the performance for the majority of the applications. The only exception is, as expected, *C* when co-scheduled with *S*. In this case, the throughput of many *C* applications is reduced as a consequence of the cache thrashing caused by the *S* co-runner. On the other hand, CFS does not provide any similar improvements for *S*, while still degrading the performance of many *C* applications. Moreover, while in TFT the throughput of an *N* application is stable irrespective of its co-runner, when CFS is used, its performance may degrade, as depicted by the boxes for the S-N, C-N and N-N pairs.

In general, Figure 7 demonstrates that for MTMP workloads, TFT provides an effective scheduling mechanism for *S* and *N* applications. However, it can degrade the performance of *C* applications, indicating that further improvements are required.

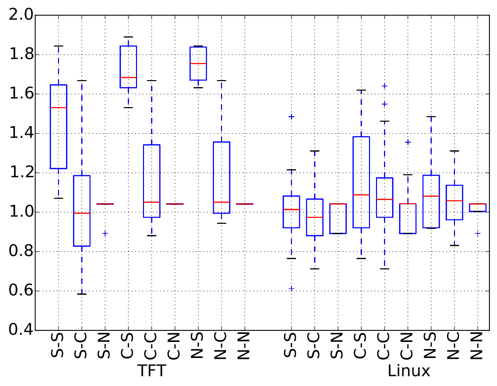


Figure 7. Applications' throughput normalized to gang scheduling. A-B denotes the throughput of B when co-running with A.

4.2. Link and Cache Awareness (LCA) for contention avoidance

As contention for shared resources can impact the performance, a number of contention aware schedulers have been proposed in the literature, most of which aim at separating competitors in time or space. Most of the proposed policies try to avoid co-executing memory bound applications and instead attempt to combine them with compute bound ones [20,1]. Prior work [12] has shown that some classes are very competitive and their coexistence should be avoided. We extend this work based on our SCN classification to present a contention avoidance scheme. The core idea behind this scheme is to prevent *C* and *S* applications from running at the same time under the same shared cache.

Algorithm 1. LCA co-scheduling algorithm

```
Input: S, C, N: lists of applications in classes S, C, N, respectively
foreach $x$ in $N$ do
    $y \leftarrow popFromFirstNonEmpty(S,C,N)$  co-schedule($x,y$)
end
foreach $x$ in $C$ do
    $y \leftarrow popFromFirstNonEmpty(C)$  co-schedule($x,y$)
end
foreach $x$ in $S$ do
    $y \leftarrow popFrom(S)$  co-schedule($x,y$)
end
```

The proposed algorithm, presented in Algorithm 1, is an adaptation of the Link and Cache-Aware (LCA) algorithm to the SCN classification scheme. The `popFromFirstNonEmpty` routine searches for the first available application in a collection of application lists, scanning them in the order they appear in the argument list.

The algorithm is greedy with $O(n)$ complexity and tries to form pairs in a predefined order. It starts with N applications and pairs them with S, which cause the greatest harm. When no more S can be found, it pairs them with C applications, which suffer the greatest harm. When all the N have been scheduled, if more applications exist, the algorithm proceeds pairing applications from the same class.

4.3. Fairness on Progress

On single core architectures or architectures where cores do not share any hardware, fairness can be achieved by assigning to the applications equal time shares of the core(s). On architectures, though, where cores are sharing resources, dividing time equally is not sufficiently fair. As we have shown, contention on shared resources impacts each application differently, depending both on its own characteristics and those of its co-runners.

To evaluate the fairness achieved by the state-of-the-art Linux scheduler, we create MPMT workloads by pairing the 8 threads of every application of our suite (see Table 2) first with the 8 threads of *Stream*, then with the 8 threads of *Jacobi* and, finally, with the 8 threads of *3mm*. These three applications were selected to create high contention scenarios, as our characterization revealed that the former two are the most resource consuming applications of class S, and the latter the most resource consuming application of class C. The 16 threads of each pair are executed on our 8-core system (see Table 1) for a time window of 5 secs using the CFS scheduler. Every time an application terminates inside this window, it is restarted. We define throughput as the number of times an application finishes and normalize it to the throughput achieved when running alone on the system. Figure 8 reports the average normalized throughput of each application together with its average running to waiting time ratio.

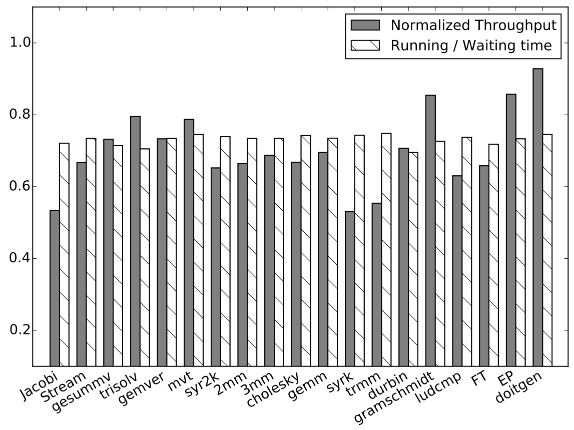


Figure 8. Average normalized throughput and $\frac{\text{running time}}{\text{waiting time}}$ ratio for Linux CFS.

It is obvious that CFS fulfills its goal; in our execution window, every application gets almost the same cpu time and thus, fairness according to CFS is achieved. However, being contention agnostic, CFS fails to treat all applications *fairly in terms of achieved progress*. Despite all receiving equal shares of the cpu, some applications, like EP and doitigen, progress at a similar pace to when running alone on the system, while others, such as Jacobi and syrk, are severely affected by their co-runners and their performance is halved.

Motivated by this observation, we present *Fair on Progress* (FOP), a simple and effective scheduling strategy that aims at enabling all the applications to progress at a similar pace. To accomplish this, the scheduler needs to disrupt CFS's main characteristic of fair cpu time and reward applications suffering from contention by granting them higher time shares. For each application we define its *Progress* as the ratio:

$$Progress = \frac{IPC_{last\_quantum}}{IPC_{standalone}}$$

the *Contention Penalty* (CP) as:

$$CP = 1 - Progress$$

and its *Age* as the sum:

$$Age = \underbrace{\frac{WT}{n \cdot TQ}}_{\text{ageing due to waiting}} + \underbrace{CP}_{\text{ageing due to contention}}$$

where WT is its waiting time, n is the number of applications and TQ is the system's time quantum. An application matures in two ways: first, waiting for cpu resources, expressed as their waiting time (WT) normalized to a time period equal to n time quanta ($n \cdot TQ$); second, suffering contention penalties which hinder its progress, expressed by the factor $1 - Progress$. Applications that suffer from resource contention will exhibit a CP close to one, leading to a higher age compared to others that progress at a similar pace to when running alone. Every time quantum, the scheduler selects the applications with the highest age, thus rewarding applications that have fallen behind.

To evaluate FOP we executed the applications of our suite on our 8-core system using first FOP and then CFS for 2400 secs, restarting every application that terminated inside this window. Figure 9 depicts the throughput normalized to standalone execution for all the applications. As demonstrated by the box size and the distance between the whiskers, when FOP is used, the variation between the progress of each application is significantly smaller compared to when CFS is employed. Therefore, FOP achieves fairness allowing all applications to progress at a similar pace.

An obvious shortcoming of FOP however, is that prioritizing applications whose progress is hindered over applications running at full speed, leads to a total system throughput degradation as demonstrated in Figure 9. This is primarily due to the fact that granting more time slots to applications suffering from interference, increases the possibility of interference itself, a fact that leads to an overall performance degradation. In the next paragraph we present our approach to address this issue as well.

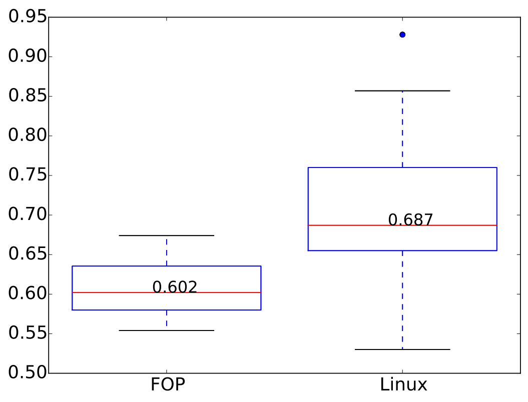


Figure 9. Progress achieved when using FOP and CFS.

4.4. F-LCA: Putting it all together

This section presents *Folding-Link and Cache Aware* (F-LCA), our complete contention-aware scheduling scheme. We base our approach on the potential of TFT to alleviate self-contention and employ LCA to create pairs that avoid contention as much as possible. To achieve fairness, we utilize FOP and reward applications that suffer from contention with more running time.

However, as folding is likely to cause additional slowdowns, we extend F-LCA to take the *Folding Penalty* (FP) into account as well. FP derives from the ratio of the IPC of the folded application to the IPC of the original one as follows:

$$FP = 1 - \frac{IPC_{folded}}{IPC_{standalone}}$$

and extend Age as follows:

$$Age = \underbrace{w_1 \cdot \frac{WT}{n \cdot TQ}}_{\text{ageing due to waiting}} + \underbrace{w_2\, CP}_{\text{ageing due to contention}} + \underbrace{w_3 \cdot FP}_{\text{ageing due to folding}}$$

Ageing in the F-LCA scheme is a function of the waiting time of an application, the slowdown incurred by contention and the penalty it suffers from folding. Our ageing formula includes weighting factors (w_1, w_2, w_3) to support the enforcement of different priorities to the ageing components. To enforce equal priorities between CP and FP, we set $(w_2, w_3) = (1, 2)$, since $CP \in [0, 1.0)$, but $FP \in (0, 0.5]$. Note that in our model we disregard applications that suffer from lock contention and consider that folding at most doubles an application's execution time (i.e. halves its IPC - hence $FP \in (0, 0.5]$). Finally we set $w_1 = 1$, to balance the three ageing factors fairly.

Whenever an application enters the system, our scheme executes it in isolation, classifies it and measures its $IPC_{standalone}$. It then folds the application, measures its IPC_{folded} and calculates its FP. $IPC_{standalone}$, IPC_{folded} and FP are calculated only during this characterization step and used subsequently for the scheduling decisions.

F-LCA sorts all applications by Age, and selects a constant subset of K applications with the highest Age to be scheduled in the next epoch, i.e. K time quanta. The K applications are folded, paired using LCA and scheduled to the system. In our experimentation we have set $K = 4$ for workloads with more than four applications, and $K = 2$ for workloads with less than four applications.

If the ready list of applications is maintained in a binary search tree (as CFS does using a Red-Black tree) keeping the applications sorted by age has a complexity of $O(logn)$. Applying LCA to a constant number of applications has a complexity of $O(1)$, leading to a total F-LCA complexity of $O(logn)$, equal to that of CFS. Note, finally, that as our ageing scheme incorporates waiting time, it guarantees starvation-free executions: eventually the waiting time of an application will dominate the CP and FP factors which are bounded, leading to the selection of the application by the scheduler.

5. Evaluation

5.1. Experimental Platform

We performed our experimental evaluation on an Intel® Xeon® CPU E5-4620 (Sandy Bridge) equipped with 8 cores, private L1 and L2, 16MB 16-way shared L3 and 64 GB DDR3 @1333MHz (see Table 1 for details). The platform runs Debian Linux with kernel 3.7.10.

Cores	8
L1	Data cache: private, 32 KB, 8-way, 64 bytes block size Instruction cache: private, 32 KB, 8-way, 64 bytes block size
L2	private, 256 KB, 8-way, 64 bytes block size
L3	shared, 16 MB, 16-way, 64 bytes block size
Memory	64 GB, DDR3, 1333 MHz

Table 1. Processor details

5.2. Experimental Setup

We implemented all our schedulers in user space using a pluggable infrastructure that allows for different scheduling algorithms. We use the control group infrastructure provided by the Linux kernel to implement the scheduling decisions. Because the control groups are inherited, our system can handle programs that create threads dynamically. In particular, we implement space scheduling via the cpuset library[3] that allows setting the CPU affinity for program threads, and time scheduling via the freezer control group[4] that allows pausing and resuming the execution of a program's threads. We compare F-LCA in terms of throughput and fairness with the established Linux (CFS) and Gang sched-

[3]CPUSETS: Processor and Memory Placement for Linux 2.6 kernel based systems `http://oss.sgi.com/projects/cpusets/`

[4]The freezer subsystem: `https://www.kernel.org/doc/Documentation/cgroups/freezer-subsystem.txt`

ulers, and LBB, LLC-MRB and LCA proposed in literature (see Section 2). The folding strategy is applied to LBB, LLC-MRB and LCA schedulers.

We implemented application profiling and execution monitoring using hardware performance counters. The data collected was used to classify applications and determine the applications' IPC during execution. More specifically, we collected information from the counters `UNHLT_CORE_CYCLES, INSTR_RETIRED, LLC_MISSES, L1D.REPLACEMENT`, `L2_LINES.IN`. Furthermore, we used `OFFCORE_REQUESTS (0xB7, 0x01; 0xBB, 0x01)` together with Intel's Performance Counter Monitor [5] utility to acquire information regarding bandwidth usage.

Name	Source	IPC	LLC miss rate ($\times 10^3$ misses/sec)	$B_{in_{3=LLC}}$ (MB/sec)	B_{in_2} (MB/sec)	B_{in_1} (MB/sec)	CR_{LLC}	CR_{L2}	Class
jacobi	polybench	0.487	2514339	6227	4138	4024	1.50	1.03	S
stream	[19]	0.664	85640	10627	10631	10411	1.00	1.02	
gesummv	polybench	0.486	34450	3374	2672	2542	1.26	1.05	
trisolv	polybench	0.438	25969	1462	1328	1112	1.10	1.19	
gemver	polybench	0.445	23039239	10333	8033	1777	1.29	4.52	C
mvt	polybench	0.393	26906052	10193	8527	1539	1.20	5.54	
syr2k	polybench	1.934	2622111	24215	22513	2739	1.08	8.22	
2mm	polybench	0.363	11048673	25128	24838	74	1.01	335.65	
3mm	polybench	0.351	9503846	26719	26128	570	1.02	45.84	
cholesky	polybench	1.611	7957523	20872	27631	3	0.76	9210.33	
gemm	polybench	0.377	11858539	24234	23918	19	1.01	1258.84	
syrk	polybench	2.287	3615742	20930	20929	32	1.00	654.03	
trmm	polybench	2.165	5825980	20389	24623	1450	0.83	16.98	
durbin	polybench	0.347	9399687	4940	3725	442	1.33	8.43	
gramschmidt	polybench	0.219	77439222	16730	16652	0	1.00	-	
ludcmp	polybench	0.431	8359724	13205	2290	6	5.77	381.67	
ft.A	NAS	0.957	425496	20667	4131	1916	5.00	2.16	
ep	NAS	0.747	143659	959	978	10	0.98	97.80	N
doitgen	polybench	1.429	72012	102081	334	224	305.63	1.49	

Table 2. Application suite

To evaluate the various policies we created 30 different workloads, 15 four-applications workloads and 15 eight-applications workloads. Workloads are divided into mixes leading to low, medium and high contention and consist of a combination of the 19 applications of our benchmark suite shown in Table 2. We selected benchmarks from various suites that exhibit a single phase throughout their execution. To avoid the effect of each workload having a different execution time, we define a time window, in which every application that terminates starts again. This way, we are able to evaluate the performance of the various schedulers while the system operates at full load. The window length has been appropriately selected to allow even long running applications to terminate at least around 10 times for every scheduler.

5.3. Results

We focus our evaluation on the critical metrics of throughput and fairness illustrated with the use of Tukey boxplots. Boxplots represent statistical populations without making any

[5] Intel® Performance Counter Monitor - A better way to measure CPU utilization `http://software.intel.com/en-us/articles/intel-performance-counter-monitor`

assumptions. The top and bottom of the box represent the first and the third quartiles, the band inside the box is the median value and the end of the whiskers represent the lowest datum still within 1.5 interquartile range (IQR) of the lower quartile and the corresponding higher. Cycles above and under the whiskers are values outside the defined ranges and square marks are the average values. To our data analysis shorter boxes represent higher level of fairness as the distribution of the values is smaller. Higher median values show better overall performance of the scheduling policy.

Figures 10 and 11 show the distribution of throughput for all workloads and schedulers. In terms of average/median throughput, as a metric for performance, we observe that F-LCA achieves the best performance in four out of six cases, and is very close to the best performance which is obtained by Linux in the case of four workloads with high contention.

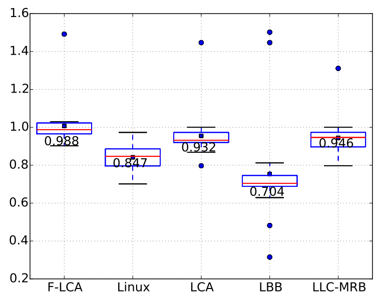
In general, in low contention cases the throughput of all schedulers is lower than that of the Gang scheduler. The Linux scheduler performs worse than Gang because of the way it scatters threads on cores that can lead, as we observed, to more than two threads of the same application time sharing the same core. On the other hand, F-LCA's performance is more stable and almost equal to that of the Gang scheduler. This is expected, as contention-aware schedulers cannot really offer any significant benefits in low contention scenarios. In cases of medium contention, F-LCA outperforms all other schedulers, mainly due to its folding strategy. CFS performance is closer to that of Gang than before, however F-LCA reaches a performance improvement of 20% over CFS.

Finally, in cases of high contention LCA, LBB, LLC-MRB achieve the same or better overall throughput than both F-LCA and CFS. However, F-LCA is more stable and for almost every workload performs better than the Gang scheduler, while for many workloads all the other schedulers perform worse, failing to alleviate performance penalties due to resource contention. In the most demanding case of workloads comprising 8 applications with high contention, CFS performs worse than the Gang scheduler for almost all the different workloads, while F-LCA obtains around 30% better performance than CFS and around 20% better than the Gang scheduler.

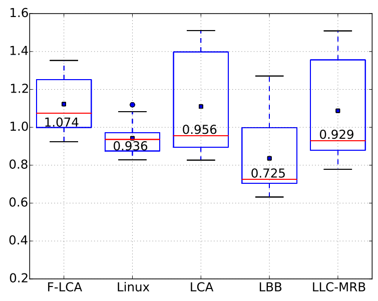
There are two more elaborate metrics of fairness we can utilize: a) the distribution of throughput compared to gang scheduling, i.e. the distance between the maximum and minimum points and the height of the boxes/whiskers; the lower these distances, the higher the scheduler's fairness; b) the distribution of co-execution penalty compared to gang scheduling, focusing only on applications that were harmed. In this case we look at the distance of the minimum points and low ends of the boxes/whiskers from 1.0 (gang scheduling).

Considering both these metrics, F-LCA clearly outperforms all other schedulers in terms of fairness. All schedulers have their best IQR under 10%. On the other hand, F-LCA's worst IQR is under 30% while all the others' worst IQR exceeds 50%. More specifically, for low contention cases both F-LCA and Linux schedulers are fair, while LBB seems to be the most unfair. CFS is fair in scenarios with medium and high contention for the 8 application workloads, because of the many co-execution variations that exist. If there are few variations however, CFS fails to be fair.

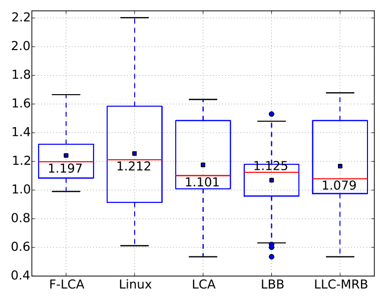
Finally, LCA's, LLB's and LLC-MRB's wide IQR are due to misclassifications that occur and to the fact that pairs are statically determined at the start of the scheduling. Therefore, a pair exhibiting high contention, once created by these schedulers, is used for the entire execution. This can be very unfair to a sensitive application. On the other hand,



(a) Low Contention

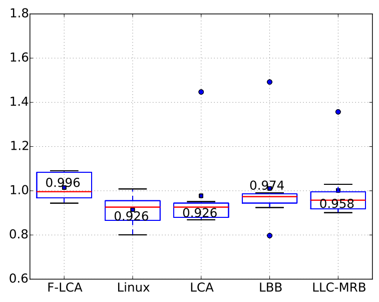


(b) Medium Contention

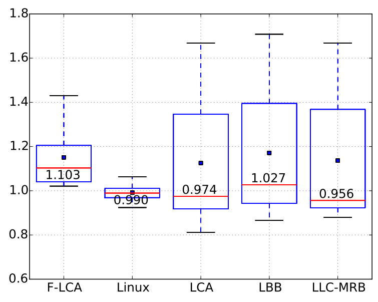


(c) High Contention

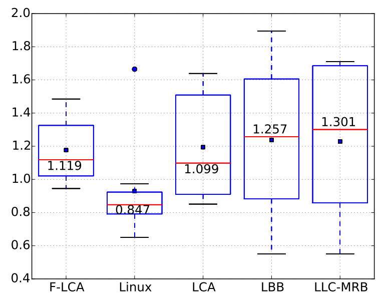
Figure 10. Throughput normalized to Gang scheduler for four application workloads.



(a) Low Contention



(b) Medium Contention



(c) High Contention

Figure 11. Throughput normalized to Gang scheduler for eight application workloads.

F-LCA, using the FOP mechanism, manages to deal with possible misclassifications as it dynamically creates pairs depending on each application age and hence progress.

6. Discussion – Towards a complete, OS-level, contention-aware scheduler

The F-LCA scheduling approach presented in this work focuses on the core mechanisms and policies that need to be incorporated in a contention-aware module that will operate in the context of a complete scheduling framework. To this direction, we discuss a number of issues that need to be further addressed by the collaboration of F-LCA with the OS scheduler:

Dynamic behavior of applications In a real execution environment applications may enter/leave the system at any time, perform I/O operations, or change their execution phase. One of the key design principles of F-LCA is to be fast both in its classification and scheduling steps in order to minimize the additional overheads when invoked under these conditions. Application spawn and exit is a trivial action for the operating system and F-LCA requires no additional modifications other than updating the relevant data structures. The same holds for the case of I/O operations that trigger the operating system, thus enabling straightforward management. However, as F-LCA does not target I/O intensive applications, once such an application is recognized (e.g. by exceeding a number of I/O operations within a time window) it could be excluded from the F-LCA policy and assigned to an accompanying scheduling module for I/O intensive applications. Such module needs to collaborate with F-LCA to share system resources.

Phase change is also a significant aspect in contention-aware scheduling. Although phase change detection [9] is orthogonal and beyond the scope of our work, F-LCA collects a lot of monitoring information like IPC and traffic within the memory hierarchy that could be utilized to identify potential phase changes. We leave this for future work. However, a simple and straightforward solution for our scheme, would be to re-classify an application, whenever a phase change is recognized.

Lock contention F-LCA targets contention in shared hardware resources like caches and memory links. However, lock contention has also been identified as a source of performance degradation for multithreaded applications that need to synchronize frequently. Gang scheduling has been proposed exactly for this reason. Our scheme needs to exclude applications that suffer from lock contention and resort to gang scheduling for them. During characterization, such applications can be easily identified as their folding penalty (FP) will be significantly higher than 2.

Richer memory hierarchies We have applied F-LCA on architectures where all cores share a single memory link and LLC. However, wide CMP systems, especially servers, are based on deeper and more complex memory hierarchies with several CPUs, memory links, NUMA organization and different cache sharings. In this case, F-LCA needs to operate in an hierarchical mode, with separate F-LCA modules closely collaborating. Our classification and scheduling approach can be easily applied to applications both space and time sharing NUMA/cache islands. Modification of F-LCA for richer memory hierarchies is left for future work.

Arbitrary number of threads per application In this paper we worked under the assumption that all applications request a number of threads equal to the number of CPU cores. Although this is the case for several applications, for a scheduling framework to be generic, we need to address the case of arbitrary numbers of application threads. The challenge in this case is to keep the complexity of the scheduling algorithm low, as packing arbitrary numbers of threads to fill the CPU cores may be based on algorithms that well exceed our current $O(logn)$ complexity. However, our simple and effective classification scheme can come to the rescue for this case as well. We can easily apply a number of simple techniques like building groups of residual threads with non-conflicting behaviors to form a scheduling entity for F-LCA, or reuse N applications to fill empty cores.

7. Related work

Resource-aware scheduling has been addressed in several contexts such as HPC systems [6] and data centers or cloud environments [18,25,7]. In this paper we focus on resource-aware scheduling for CMPs, where contrary to data center schedulers that are able to apply extensive profiling on the submitted applications to characterize them and take allocation decisions [18,7], actions need to be taken fast, on the fly, without the use of any external behavior-revealing microbenchmarks.

In this context, contention on shared caches was identified as a source of performance degradation early at the adoption of CMP systems, where elaborate hardware management mechanisms were proposed (e.g. replacement and partitioning policies [14,24,21]). However, as hardware approaches focus on the shared cache only, they are unable to alleviate resource contention on other resources, such as the shared memory link which can also be a source of severe performance degradation [29]. However, as modern processors are starting to include mechanisms for cache partitioning exposed to the software layers, we intend to incorporate such mechanisms as an additional action to our scheduling approach.

Software co-scheduling approaches have been proposed to handle contention in a more flexible way. Contention-aware mechanisms typically rely on a classification of the application execution behavior and a relevant prediction of the interference when two or more applications are co-executed under a resource sharing scenario. Initially, the shared cache utilization pattern was the one that received the most intense study. Xie and Loh [27] proposed an animal classification approach, where an application may belong to one of four classes named turtles, sheep, rabbits and tasmanian devils. Applications that do not make much use of the last level cache are called turtles. The ones that are not sensitive to the number of ways allocated to them but make use of the LLC belong to the sheep group. Rabbits are very sensitive to the ways allocated to them. Finally, devils make heavy use of the LLC while having very high miss rates. Targeting efficient cache partitioning, Lin et al. [17] worked on a scheme to allocate the LLC between two programs by cache coloring. Their program classification uses four classes (colors) and is based on program performance degradation when running on 1MB cache compared to running on 4MB cache. Programs belonging to the red class suffer a performance degradation greater than 20%. Yellow applications suffered a performance degradation from 5% to 20%. Programs with less than 5% degradation were further classified as green or black according to their number of misses per thousand cycles. Jaleel et al. [13] proposed a categorization scheme that employs the following application categories: Core Cache Fitting (CCF) have a dataset size fitting in the lower levels of the memory organization and do not benefit from the shared level of cache. LLC Thrashing (LLCT) have a data set larger than the available LLC; under LRU, these applications harm performance of any co-running application that benefits from the LLC usage. LLC Fitting (LLCF) applications require almost the entire LLC; their performance is reduced if there is cache thrashing. Finally, LLC Friendly (LLCFR) applications, though they gain in performance from the available cache resources, they do not suffer significantly when these resources are reduced. Although the aforementioned classification schemes may provide an elaborate view on the application's utilization of the shared cache, they suffer from two severe shortcomings: first, as mentioned above, they do not capture the entire contention picture as contention may arise in different resources than the LLC, and second they require information that needs to be collected from static analysis (e.g. memory reuse patterns of applications known as stack distance profiles, or comparison of execution with different cache sizes) or from additional hardware.

Prior works on contention-aware scheduling classify memory-intensive applications using the LLC miss rate [5,28,3,16]. Blagodurov et al. [5] show that, despite its simplicity, the LLC miss rate can be more accurate than more elaborate cache contention classification schemes. Merkel et al. [20] use the memory bandwidth as a metric to quantify memory intensity. Contentiousness and sensitivity [26] are two insightful metrics to quantify the penalty imposed and suffered by application in co-execution. Tang et al. [26] calculate contentiousness and sensitivity by considering LLC resource usage, link bandwidth and prefetching traffic which are fed to a linear regression model to calculate proper parameters. By considering wider activity than sole bandwidth or miss rates, this scheme is able to better capture application interference. Our classification scheme builds upon this idea and extends the inspection area from the main memory down to the compute core. A number of contention-aware co-schedulers have been proposed based on the aforementioned classification schemes. The Distributed Intensity Online (DIO) [5] and its variation for NUMA systems Distributed Intensity NUMA Online (DINO) [4] schedulers monitor the LLC miss rate and balance memory intensity by space scheduling applications to different LLC and memory links. This framework considers a total number of threads that is equal to or smaller than the available cores, and does not handle the general scheduling scenario discussed in Section 2. The LLC-MRB scheduler uses the notion of balancing the LLC misses in the scheduling problem addressed in this paper.

Bhadauria and McKee present two resource-aware scheduling schemes [3] focusing on the optimization of Energy-Delay products. Their schedulers aim to decide upon the number of threads for each application together with the coexistence of applications within the same time slot (quantum). Although they account for resource balance, contention avoidance is not central to their approach. In their first scheme (named FAIR), they try to balance resource requests of the co-scheduled applications in terms of either LLC cache miss rate (hits/misses - named FAIRMIS) or bus occupancy (named FAIRCOM) together with setting an efficient concurrency level. Their second scheme, a greedy scheduler, profiles the applications to get their most energy-efficient thread counts and schedules them using a bin-packing heuristic to maximize average system throughput in a resource-oblivious manner. The LLC-MRB and LBB schedulers used for our evaluation purposes use the notion of LLC miss rate balance and link bandwidth balance.

Merkel et al. [20] propose explicitly the LBB scheduler. The authors recognize the need to handle multiple resource contention points and employ the concept of task activity vectors for characterizing applications by resource utilization. They base their co-scheduling policies on the selection of applications that use complementary resources (sorted co-scheduling). In their implementation, however, they focus on one resource only (memory bandwidth) that is assumed to be mainly responsible for contention. The authors discuss that the scheme can be extended to adapt to the hardware and workload's characteristics by focusing on a different contention point each time. Our scheme, on the other hand, is able to handle different contention points at the granularity of a time quantum by avoiding potential contention in the LLC and memory link.

Pusukuri et al. [16] present ADAPT, a resource-aware co-scheduling framework that considers overheads due to memory contention, lock contention and thread latency. They characterize memory-intensive applications by their LLC miss rate. Their approach distributes space to applications based on a number of core preconfigurations. Multithreaded applications either enter a single group of space scheduling or are allocated the entire system. To this extent, although the approach can handle a total number of threads higher than the total number of cores, it cannot be extended in a straightforward way to address the general co-scheduling scenario. In our approach, we deal with hardware resource and lock contention separately and consider gang scheduling as a mechanism to handle lock contention.

8. Conclusion and future work

In this paper we presented F-LCA, a contention-aware scheduling policy for CMP systems that targets system throughput and application fairness. Our approach is based on a classification scheme that detects activity and possible interference across the entire memory hierarchy and relies on information that can be collected at runtime with no additional hardware support. We have implemented our scheduler at user level and evaluated our policy on several scenarios with different contention levels employing a variety of multiprogrammed workloads. Our experimental results demonstrate that the proposed scheduling method outperforms both the established Linux and Gang schedulers as well as a number of research contention-aware schedulers, both in terms of system throughput and application fairness.

As future work we intend to extend our scheduling scheme towards a complete OS level framework working in the following directions: a) expand our scheduling approaches to handle more complex scheduling problems with multiphased applications and I/O, b) extend our scheme to architectures with richer memory hierarchies, c) utilize the cache partitioning mechanisms of modern processors and d) augment our objectives with power and energy.

Acknowledgements

This research was partly funded by project I-PARTS: Integrating Parallel Run-Time Systems for Efficient Resource Allocation in Multicore Systems (code 2504) of Action ARISTEIA, co-financed by the European Union (European Social Fund) and Hellenic national funds through the Operational Program Education and Lifelong Learning (NSRF 2007-2013). Nikela Papadopoulou has received funding from IKY fellowships of excellence for postgraduate studies in Greece-SIEMENS program.

References

[1] Christos D. Antonopoulos, Dimitrios S. Nikolopoulos, and Theodore S. Papatheodorou. Scheduling algorithms with bus bandwidth considerations for SMPs. In *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, pages 547–554, Oct 2003.

[2] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks - Summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Supercomputing '91, pages 158–165, New York, NY, USA, 1991. ACM.

[3] Major Bhadauria and Sally A. McKee. An approach to resource-aware co-scheduling for CMPs. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 189–199, New York, NY, USA, 2010. ACM.

[4] Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. A case for numa-aware contention management on multicore systems. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 1–1, Berkeley, CA, USA, 2011. USENIX Association.

[5] Sergey Blagodurov, Sergey Zhuravlev, and Alexandra Fedorova. Contention-aware scheduling on multicore systems. *ACM Trans. Comput. Syst.*, 28(4):8:1–8:45, December 2010.

[6] Alex D. Breslow, Leo Porter, Ananta Tiwari, Michael Laurenzano, Laura Carrington, Dean M. Tullsen, and Allan E. Snavely. The case for colocation of high performance computing workloads. *Concurrency and Computation: Practice and Experience*, pages n/a–n/a, 2013.

[7] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 77–88, New York, NY, USA, 2013. ACM.

[8] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices*, 49(4):127–144, 2014.

[9] Ashutosh S Dhodapkar and James E Smith. Comparing program phase detection techniques. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 217. IEEE Computer Society, 2003.

[10] David Eklov, Nikos Nikoleris, David Black-Schaffer, and Erik Hagersten. Bandwidth bandit: Quantitative characterization of memory contention. In *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, pages 1–10. IEEE, 2013.

[11] Dror G. Feitelson and Larry Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16:306–318, 1992.

[12] Alexandros-Herodotos Haritatos, Georgios Goumas, Nikos Anastopoulos, Konstantinos Nikas, Kornilios Kourtis, and Nectarios Koziris. Lca: A memory link and cache-aware co-scheduling approach for cmps. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 469–470, New York, NY, USA, 2014. ACM.

[13] Aamer Jaleel, Hashem H. Najaf-abadi, Samantika Subramaniam, Simon C. Steely, and Joel Emer. Cruise: Cache replacement and utility-aware scheduling. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 249–260, New York, NY, USA, 2012. ACM.

[14] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 111–122, Washington, DC, USA, 2004. IEEE Computer Society.

[15] Evangelos Koukis and Nectarios Koziris. Memory bandwidth aware scheduling for SMP cluster nodes. *16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, 0:187–196, 2005.

[16] Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N. Bhuyan. ADAPT: A framework for coscheduling multithreaded programs. *ACM Trans. Archit. Code Optim.*, 9(4):45:1–45:24, January 2013.

[17] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *International Symposium on High Performance Computer Architecture*, pages 367–378, 2008.

[18] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: increasing utilization in modern warehouse scale computers via sensible co-locations. In *MICRO*, pages 248–259, 2011.

[19] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[20] Andreas Merkel, Jan Stoess, and Frank Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 153–166, New York, NY, USA, 2010. ACM.

[21] Konstantinos Nikas, Matthew Horsnell, and Jim D. Garside. An adaptive bloom filter cache partitioning scheme for multicore architectures. In *ICSAMOS*, pages 25–32, 2008.

[22] Chandandeep Singh Pabla. Completely fair scheduler. *Linux J.*, 2009(184), August 2009.

[23] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 423–432, Washington, DC, USA, 2006. IEEE Computer Society.

[24] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO*, pages 423–432, 2006.

[25] Alan Roytman, Aman Kansal, Sriram Govindan, Jie Liu, and Suman Nath. Pacman: Performance aware virtual machine consolidation. In *Presented as part of the 10th International Conference on Autonomic Computing*, pages 83–94, Berkeley, CA, 2013. USENIX.

[26] Lingjia Tang, Jason Mars, and Mary Lou Soffa. Contentiousness vs. sensitivity: improving contention aware runtime systems on multicore architectures. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, pages 12–21, New York, NY, USA, 2011. ACM.

[27] Yuejian Xie and Gabriel Loh. Dynamic classification of program memory behaviors in CMPs. In *Proceedings of the 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects*, 2008.

[28] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. *SIGPLAN Not.*, 45(3):129–142, March 2010.

[29] Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Comput. Surv.*, 45(1):4:1–4:28, December 2012.