Automatic Quality of Service Control in Multi-core Systems using Cache Partitioning

Jakob Danielsson[1], Tiberiu Seceleanu[1], Marcus Jägemar[1,2], Moris Behnam[1], Mikael Sjödin[1]
[1] Mälardalen University, Västerås, Sweden
[2] Ericsson AB, Stockholm, Sweden
jakob.danielsson@mdh.se

***Abstract*—In this paper, we present a last-level cache partitioning controller for multi-core systems. Our objective is to control the Quality of Service (QoS) of applications in multi-core systems by monitoring run-time performance and continuously re-sizing cache partition sizes according to the applications' needs. We discuss two different use-cases; one that promotes application fairness and another one that prioritizes applications according to the system engineers' desired execution behavior. We display the performance drawbacks of maintaining a fair schedule for all system tasks and its performance implications for system applications. We, therefore, implement a second control algorithm that enforces cache partition assignments according to user-defined priorities rather than system fairness. Our experiments reveal that it is possible, with non-instrusive (0.3-0.7% CPU utilization) cache controlling measures, to increase performance according to setpoints and maintain the QoS for specific applications in an over-saturated system.**

I. Introduction

Hardware resources are often limited for automotive control-systems. Especially when executing multiple applications on electronic control-unit, resource allocation must be carefully considered before deployment to achieve the desired Quality of Service (QoS). Multi-core computers are gaining increased popularity for the automotive industry due to increased available resources (processor cores) on the same chip. Multi-core processors offer higher computational capacity than their single-core predecessors while utilizing less size, weight, and power (SWaP) than older single-core ones. Multi-core computers often utilize a shared resource principle, where the ownership of multiple resources such as cache memories and the memory bus are shared simultaneously across different cores. The resource-sharing principle makes multi-core's prone to a state called shared resource contention, which causes severe execution-time fluctuations for applications and is seen as one of the major bottlenecks for bringing multi-core's into time-critical computing.

It is possible to counter shared resource contention using partitioning techniques such as cache coloring for the cache [12] and thus make multi-core systems more time-predictable. However, the partition boundaries are hard to assign appropriately at system boot since an application can change run-time behavior during the lifespan of the system. As such, too small partition sizes can cause an application to display lower QoS than desired, and a too large partition sizes wastes hardware resources without any QoS gain.

In this work, we try to automate the allocation of cache memory to meet QoS needs. We present experiments containing two different distribution policies; fair and prioritized. Fair distribution prioritizes assigning cache partitions based on the current performance of all system applications and tries to optimize the allocated cache size such that all applications reach as close as possible to their maximum performance. Priority distribution instead distributes cache memory based on an application's setpoint QoS and prioritizes this application to receive cache partition space while the setpoint QoS is not met. Our experiments are done for over-saturated systems where it is impossible to reach maximum QoS for all applications. We demonstrate that our controller can instead reach and maintain a setpoint QoS for our system using our fair strategy. We further demonstrate how to prioritize an application and meet QoS needs for one specific, prioritized application. We list our contributions as follows:

- An automated process of monitoring application performance continuously, without the need of a complex communication scheme.
- A cache partitioning control scheme that automatically adjusts the cache partition size of monitored applications to meet their respective QoS.
- A working implementation in Linux using the above mentioned contributions.

978-1-7281-2989-1/21$31.00 ©2021 IEEE

II. Background

A. Application Quality of Service

We define *QoS* as a function of the number of instructions retired in a time interval. One instruction retired means that the instruction has passed through all stages within the processor pipeline. This means that for higher QoS, an application will execute/retire a higher number of instructions. We can configure the performance monitor counters (PMC) of a CPU to monitor the instructions retired rate for a process ID (pid), and thus monitor the performance of an application online. However, the measurement approach means that we put requirements on the application's functionality – network and I/O applications that utilize busy-wait loops typically display a high number of instructions retired in the loops but not doing practical work. The prerequisites for our performance measurement approach to work is that the applications are not utilizing busy-wait loops (such as waiting for sensors to become ready) but instead continuously doing "actual" processing (such as identifying obstacles in an image frame from a video stream). We assume our the QoS of our applications is correlated to the number of instructions retired, where an increase in number of instructions retired leads to a decrease in response time.

B. Cache contention

Cache memories are relatively small, temporary memory storage units that affects the system's overall performance. Cache memories in multi-core systems are prone to reach a state called cache contention, which causes dramatic execution-time fluctuation of system applications [3] and can cause problems to a system that expects execution-time predictability. The main reason for cache contention is the small

memory size of the cache combined with simultaneous utilization from multiple tasks on different cores. The cache's are so small that it is *very* improbable that an entire application's memory foot-print fits within the cache and it is almost a certainty that the cache will become full at some point during an application's execution.

Cache memories implement a data eviction policy to mitigate out-of-cache memory scenarios and replaces old data with new data when the cache is full. The cache selects one data block (cache line) according to a policy (e.g., LRU, random), evicts the selected cache line from the cache and then finally inserts the new data onto the address of the previously evicted data. Data replacement is necessary to counter the small memory space of a cache, but is also the main reason for severe execution-time fluctuations and QoS decrease.

Execution-time fluctuation often appears when two or more cache dominant applications utilize the same cache memory [2]. Consider the following scenario; two applications App_1 and App_2 executes simultaneously in a dual-core system with a 4 MB cache. App_1 runs on core 1 and App_2 runs on core 2. Both applications require a memory footprint of 4 MB– i.e. the same size as the available cache and both application have a cache usage that is linear with the execution. The cache will be full and start to evict data that belongs to either App_1 or App_2 once the applications have executed roughly half of their execution. The data evictions means the data is no longer present within the cache and needs to re-fetched from the main memory into the cache if referenced again, which has a significant latency.

Cache contention causes dramatic execution-time fluctuations for applications in multi-core systems [2] and is one of the major bottlenecks for introducing multi-core chips in to time-critical computing. In this paper, we focus of the shared last-level cache (LLC) as the location of the contention which can be partitioned according to the page-coloring algorithm that mitigates cache contention [12].

C. Cache partitioning

The main idea behind cache partitioning is to reserve a portion of the cache memory to only certain processes such that shared cache contention never occurs. There exist a variety of solutions to implements cache partitioning, including the static, hardware-supported cache way-partitioning, MMU-based page coloring [12] [5] and also the programmatic cache locking solution [10]. In this paper, we utilize page coloring; an MMU-implemented a policy that redirects how page addresses are translated into the cache memory. There exists a large body of variations on page coloring including Palloc [14], Coloris [12], Jailhouse hypervisor adaptation [5] etc. Page coloring creates borders in the cache memory disqualifying processes from accessing certain data blocks in the cache memory (cache-lines).

We exemplify page coloring using three applications (A,B, and C) in Fig. 1 with a cache that contains nine cache-lines. Fig 1 shows how the MMU maps addresses to the cache in a page-colored environment. Memory requests that belong to application 1 are only allowed to access cache lines 1-3 while application 2 is only allowed to access the cache lines 4-6. Page-coloring thus means an application can only evict its' data from the cache memory and not by other applications on different cores.

We showcase the effects of cache contention in Fig. 2 and illustrate how these effects are countered using page coloring.

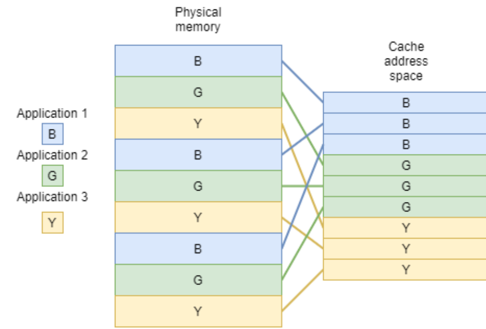

Fig. 1: Cache coloring

Fig. 2 shows two experiments that runs 150 executions of one 256x256 matrix multiplication on core 0. The blue squares marks the execution time of a matrix multiplication running in a non-isolated environment, while the orange crosses marks the the execution time of a matrix multiplication running in a cache partitioned environment. We generate cache contention by starting another 256x256 matrix multiplication at iteration 75 on core 1.

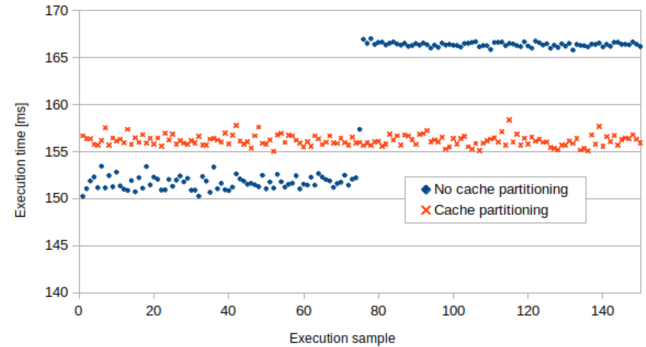

Fig. 2: Cache contention

Cache contention has a dramatic effect on the non-isolated matrix multiplications' (blue squares) execution time. The figure shows how the execution-time for the matrix multiplication increases by 17 milliseconds, just from running another matrix multiplication on another core. The cache partitioned version is roughly 5-6 milliseconds slower than our un-partitioned case, but remains unchanged when the new matrix multiplication. Cache partitioning however comes with drawbacks in terms of complexity which causes execution-time overhead as (5-6 ms in the example). The cache memory is also very small, which means the partition sizes for different applications must be handled with utmost care to avoid wasting a valuable resource. The total number of available colors depends on the way-set associativity and also the total amount of cache space. The number of available cache partitions for a processor is calculated according to Equation 1 [12].

$$Nr.\ of\ Colors = \frac{Cache_size}{Cache_ways * page_size} \quad (1)$$

D. Related work

There exists a large body of dynamic cache partitioning [12, 14, 6, 8, 9] that investigates how to optimize cache partitions to achieve maximum performance of the SPEC CPU benchmark

suite. However, it is hard to reach maximum performance in an over-saturated system where all cache-partitions are assigned; wherefore, we focus on creating a controller that lets the system engineer decide the performance thresholds. Other related works focuses on enforcing quality of service [4] and isolation [13] through bandwidth restrictions. Kloda et al. introduces page-coloring into the Jailhouse hypervisor, which also introduces an entirely new dimension to solving cache contention as the cache partitioning spans overall application that belongs to a specific guest OS. However, the solution is limited to re-partitioning at guest OS boot, which makes it dynamic but less flexible. Our work differs from the previous work since we introduce a priority fashion-based assignment policy into the cache allocation policy. Our goal is not to optimize overall system throughput but to provide isolation for applications while prioritizing performance for specific applications. Xu et al. [11] presents CaM, a resource partition allocation scheme using the intel's built-in cache partition utility called CAT and combines it with the memory bus reservation scheme memguard [13]. CaM proposes an algorithm that contains multiple procedures that optimizes task schedulability in a system. The main similarities between our work and CaM lie in resource allocation and load balancing. CaM takes the approach of allocating the minimum partition size to tasks executing on different cores and then re-allocates partitions until all tasks are schedulable. CaM also executes load balancing by migrating tasks from unschedulable cores to schedulable cores. CaM presents WCET guarantees and focuses on the schedulability of tasks. Our work instead focuses on tweaking the performance of specific applications in an oversaturated system in an online fashion. Our controller does not evaluate all possible task permutations in a system but instead focuses on tweaking the cache partition size of already running applications to satisfy performance needs.

III. Cache partition distribution

Cache partitioning offers isolation and counters execution-time fluctuations that happen as a consequence of cache contention. Cache partitioning, however, comes at the expense of performance degradation due to a complex memory management mechanism. Allocating cache partitions statically to applications is the most simplistic distribution policy. However, it can be very non-optimized as it is hard to assign suitable partition sizes beforehand unless performing time-consuming exhaustive searches [1]. Various online approaches instead tune the cache partition sizes to optimize application execution time [12] and maximizes system throughput.

There are industrial use-cases where the maximum throughput of the entire system is not the primary goal but is to instead maintain the QoS for one or perhaps two particular applications. Additional performance benefits to the system are just bonuses. Consider a simplistic multi-core system for an autonomous vehicle that contains three applications executing on different cores on the same chip; App_1 – feature detection algorithm that detects visual obstacles; App_2 – stores log-metrics in a database. App_3 – DPDK that sends and receives log-data packets over the network.

App_1 has the most critical task in detecting obstacles for the autonomous vehicle, while App_2 and App_3 posts log data. These applications, however, run in the same multi-core system and thus share the same cache. Assigning cache partitions based only on increasing system overall performance (that is, increasing the number of instructions retired in a time interval) can lead to a scenario where App_2 and App_3 reserves all available cache partitions, while App_1 only gets one cache partition. Distributing the system fairly based on cache usage can also lead to the same scenario if App_2 and App_3 utilizes the cache more than App_1. Both scenarios will lead to a decreased QoS for the feature detection algorithm while the system's overall performance increases. We target our use-case towards systems that prioritizes QoS for specific applications above an overall system throughput.

IV. Implementation

We implement our QoS cache partitioning controller in C using a petalinux 4.14 kernel. The controller utilizes the performance API (PAPI) [7] for monitoring the PMC; the cgroup interface for controlling an application's core affinity, and also the palloc [14] interface for adjusting the cache partition sizes. Our primary focus lies not in optimizing performance but to meet a specific application's QoS. We present our controller architecture in Fig 3.

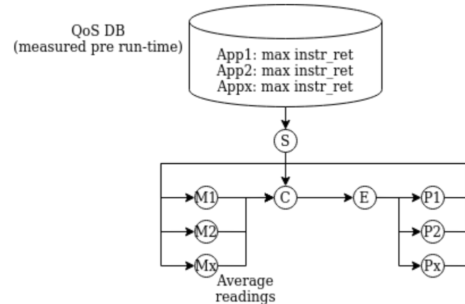

Fig. 3: Cache partition controller architecture

The controller samples the number of instructions retired (QoS) of all system applications (M1..Mx) every 20 ms and calculates the average QoS every 100 values. We assume each application to run in parallel on different cores. Our controller utilizes a QoS database to provide an estimate setpoint for the controller. The QoS database contains instructions retired measurements of all system applications and is measured pre-execution. However, the purpose of the QoS database is only to provide estimates for QoS setpoints, as they might not be theoretically achievable due to the limited cache space on-chip. The system engineer has to make the final verdict for each application on valid setpoints and use the database values as a reference for deciding a setpoint QoS. The controller compares the setpoints (S) of all running applications to the current QoS and selects two applications, one application that receives cache partition space and one application that loses cache partition space in step C. We implement two different controller modes that select applications for cache partition re-distribution; fair-oriented and priority-oriented.

Fair-oriented In this mode, we compare the differences between an application's current performance and its setpoint. The controller will increase the cache partition size for the application that displays the greatest difference between the setpoint and the current performance and decrease the cache partition size for the application that displays the least difference.

Priority-oriented In this mode, the system assigns priorities to each application. The application that has the highest priority will always receive cache partition size first-hand. The controller will assign cache partition space to lower priority

applications only when the higher priority applications display a QoS equal to the setpoint.

The change in cache partition size for the two selected applications is always 1 and the minimum cache partition is one for each application. If an application already has the minimum cache partition size, another application will instead be selected. The controller actuates the cache partition space of the two selected applications in step E utilizing the palloc API. The outcome from the cache partition actuation is a cache partition space for each application (P1..Px).

V. Experiment setup

We utilize the Xilinx-zynq zcu102 evaluation kit as testbed platform, with the processor specifics as of Table. I. Our chip provides a 16-way set associative cache, which means we may consider 16 ($1\text{ MB}/16 * 4\text{ KB} = 16$) available colors in the system according to equation 1.

TABLE I: Hardware specifications Xilinx Zynq UltraScale+ MPSoC

Feature	Hardware Component
Core	4xArm Cortex A-53 @ 1.2GHz
L_1I-cache	32 KB 2-way set assoc cache/core
L_1D-cache	32 KB 4-way set assoc cache/core
L_2-cache	1 MB 16-way set assoc. shared Last-level Cache
MMU	L_1ITLB: 10 entries
	L_1DTLB: 10 entries
	L_2TLB 512 entries, 4-way set assoc.

A. Test applications

Our execution scenario is inspired from industrial use-cases that execute applications on different cores. The system contains resource draining applications, that will drain the entire cache and as such cause severe execution-time jitters for other applications in the system. We exemplify the industrial use-case with three continuously running applications, two cache draining matrix multiplications that are common in computer graphics as synthetic loads and one feature detection algorithm, SUSAN to serve as a realistic load, listed as follows:

1) Matmult$_{ijk}$ (200x200) - Naïve implementation of the matrix multiplication that utilizes the traditional IJK traversion strategy.
2) Matmult$_{ikj}$ (100x100) - Cache prefetcher friendly traversing strategy of a matrix multiplication, designed to generate a higher cache hit rate than the naïve version. We chose a different size of this matrix multiplication to show more diverse results.
3) SUSAN - This application represents our realistic application and is used to detect corners in a frame. It is commonly used combination with other algorithms to identify visual obstacles for autonomous vehicles.

B. Controller setup

The controller is run as a standalone process that is running on its own core. It continuously monitors the applications' performance counters (instructions retired and L_2-cache misses) every 20 milliseconds – the sampling rate is a trade-off value. More frequent sampling rate reduces the controller's sleep time and thus results in a significant CPU utilization increase. Less frequent values will instead decrease the controller's responsiveness since we are dependent on average samples to estimate the current performance. In this paper we wanted to maintain a CPU utilization below 1% while still being able to re-partition on a second basis, wherefore we chose 20ms. The controller stores the performance counters in a history database, which is used for calculating the average readings. We calculate the average performance counter readings based on 100 samples from the history database and use these average readings as basis for the re-partitioning decision. Table II summarizes the controller variables for our tests.

TABLE II: Controller configuration

Property	Value
Sampling frequency	50HZ
Average window size	100

VI. Partitioning experiments

In this section, we perform several experiments to show the benefits of an online partitioning controller. We perform four different experiments, including a baseline experiment, a proof-of-concept experiment focusing on application fairness, a QoS-focused cache distribution policy, test and finally a priority-based cache distribution policy. We affine each application to different cores; Cache partition controller (core 0), Matmult$_{ijk}$ (core 1), Matmult$_{ikj}$ (core 2) and SUSAN (core 3). The controller has a CPU utilization of 0.3-0.7% and always runs using one cache partition. Due to the controller's CPU low utilization, it is possible to run other other applications on the same core as the controller if 0.3-0.7% loss of CPU utilization is acceptable. In this paper we focus only on partitioning the cache, wherefore we opt out of optimizing scheduling applications together with the controller.

A. Initial experiment

Here, we present the setpoint QoS of the applications utilizing the maximum available cache partitions for each application. This value will be our reference QoS and used to compare the quality of a cache partition. We measure the maximum QoS by monitoring the number of instructions retired while all available partitions are assigned to an application running in isolation. We sample the instructions retired every 20ms for 10 seconds and then calculate the average instructions retired. We use 10 seconds as interval to capture PMC events of at least 10 full iterations of each application. We present the reference values in Table III.

TABLE III: Cache partition maximum configuration

Application	Partition size	Reference QoS
Matmult$_{ijk}$	15	$5.4 * 10^6$
Matmult$_{ikj}$	15	$8.22 * 10^6$
SUSAN	15	$10.4 * 10^6$

The table shows the average number of instructions retired per 20 milliseconds of our applications, we denote this metric as **reference QoS**. However, the conditions of this experiment are not possible in a real system with concurrently running tasks, as we only have 15 available cache partitions and cannot distribute 15 colors to all concurrently running tasks without risking cache contention through cache-partition sharing.

B. Naïve cache partitioning

We can statically assign cache partitions in a naïve fashion by distributing the available cache partition space equally to all concurrently running applications, see Table IV.

TABLE IV: Initial setup

Application	Partition
Controller	1
Matmult$_{ijk}$	5
Matmult$_{ikj}$	5
SUSAN	5

In our naïve scenario, we split all available cache partitions among our different applications, which means our test applications receives 5 cache partitions while the controller receives 1. Table V shows the number of instructions retired per 20 ms (denoted as QoS), the L_2-cache misses per 20 ms, and the difference in QoS compared to the reference QoS for each application.

TABLE V: Performance comparison: reference versus equal partitions

Application	QoS	L_2-cache misses	Diff
Matmult$_{ijk}$	$3.23 * 10^6$	27901	41 %
Matmult$_{ikj}$	$7.19 * 10^6$	45380	13%
SUSAN	$9.83 * 10^6$	81073	6.2%

The table shows how an initial cache partition setup changes the QoS of our applications compared to the reference QoS in our previous experiment. Matmult$_{ijk}$ performs worst (due to nature of the naïve traversion strategy) comparing to the reference QoS, and SUSAN performs best.

C. Fair distribution

The equally shared cache distribution experiment shows a significant QoS degradation as compared to the reference QoS. We introduce a control mechanism to regulate the cache partition sizes according to the distance to the reference QoS. The controller balances the QoS of the applications to minimize the difference between the application's current QoS, and their reference QoS. We list the controller steps as follows:

1) Monitor current QoS of all applications in the system
2) Select application with highest difference compared to the reference QoS (App$_{high}$)
3) Select application with lowest difference compared to the reference QoS and cache partition size > 1 (App$_{low}$)
4) Increase partition size of (App$_{high}$) by one and decrease partition size of (App$_{low}$) by one
5) Go to step 1

The above algorithm embraces fairness, prioritizing poorly performing applications over better-performing applications. Figures 4, 5 and 6 depicts the cache partitioning assignments done by the controller over a time-period of 90 seconds. The red line marks the current average QoS on the left-hand side y-axis, the green line marks the reference QoS as measured in the initial experiment, and the blue line marks the cache partitioning sizes on the right-hand side y-axis.

The three figures show an example of an over-saturated system as the controller cannot assign partitions that meets any reference QoS. The difference of Matmult$_{ijk}$ remains the highest until a cache partition size of 11. Once this mark is met, the controller starts continuously change partition size between Matmult$_{ijk}$ and Matmult$_{ikj}$. The algorithm partitions the system fairly, but fails to meet the QoS requirements of any application. Figure 6 displays high performance fluctuations due to the sensitivity of the performance to the cache size, we discuss this further in Section VII.

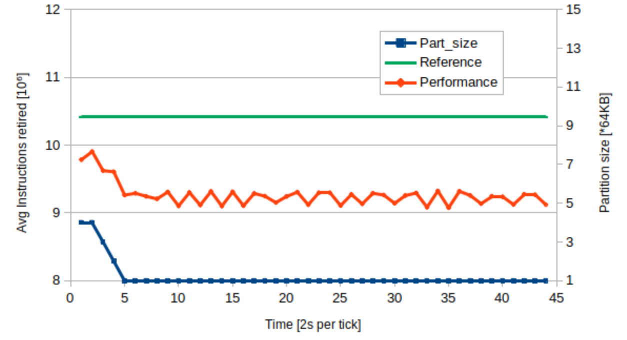

Fig. 4: Susan fair cache partitioning

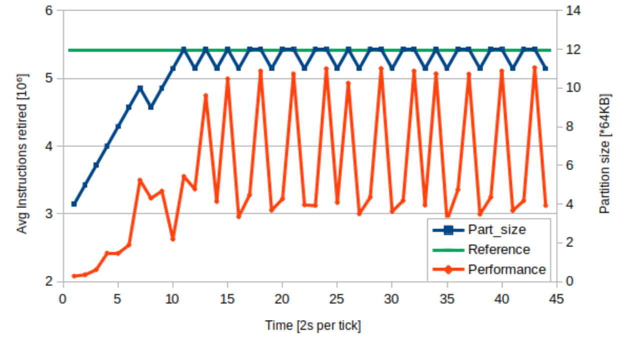

Fig. 5: Matmult$_{ijk}$ fair cache partitioning

D. Reference distribution

An application's desired QoS does not necessarily have to be the maximum achieve-able QoS. A system engineer can, for example, decide that it is acceptable that a task is operating at a percentage value of its maximum capacity when cache isolation is more prioritized. In this subsection, we tune down the expectations of our two matrix multiplications and instead use a "desired QoS" as metric for the controller to chase. We leave SUSAN's desired QoS unchanged at 10.4 million instructions per 20 ms. We show these new desired QoS values in Table VI and compare them with our reference values.

TABLE VI: Initial setup

Application	Reference	Desired	% Difference
Matmult$_{ijk}$	$5.4 * 10^6$	$3 * 10^6$	55%
Matmult$_{ikj}$	$8.22 * 10^6$	$7.5 * 10^6$	91%
SUSAN	$10.4 * 10^6$	$10.4 * 10^6$	100%

The table shows that we have tuned down the QoS requirement of Matmult$_{ijk}$ by 45% to an average of 3 million instructions per 20ms. We have also tuned down the requirement of Matmult$_{ikj}$ by 9%, to 7.5 million instructions per 20ms. In Figures 7, 8 and 9 we show how our controller operates with these new QoS requirements.

The figures show how the controller adapts the partitions according to the new desired QoS values. SUSAN still gets the minimum number of partitions, but Matmult$_{ijk}$ and Matmult$_{ikj}$ present a different scenario. Matmult$_{ijk}$ gets priority on receiving partitions first-hand since the distance to the desired QoS is highest. Matmult$_{ikj}$ starts to receive partitions from Matmult$_{ijk}$ at controller iteration six.

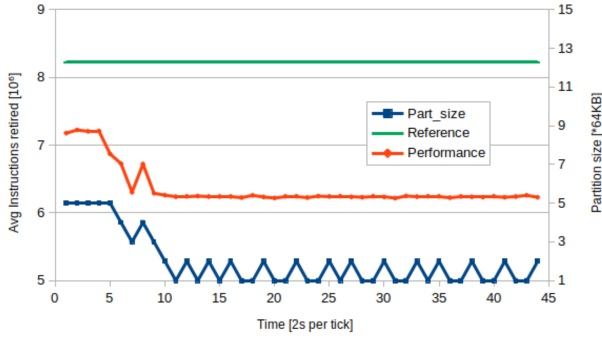

Fig. 6: Matmult$_{ikj}$ fair cache partitioning

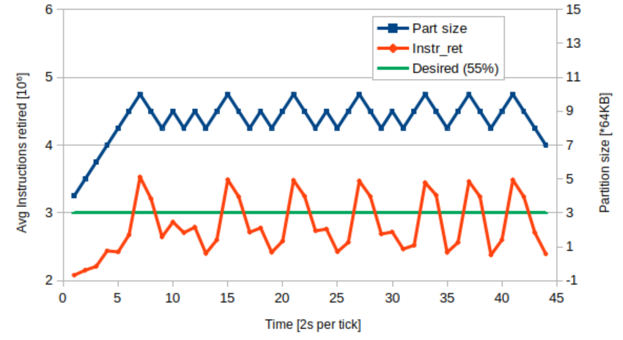

Fig. 8: Matmult$_{ijk}$ 55% target performance

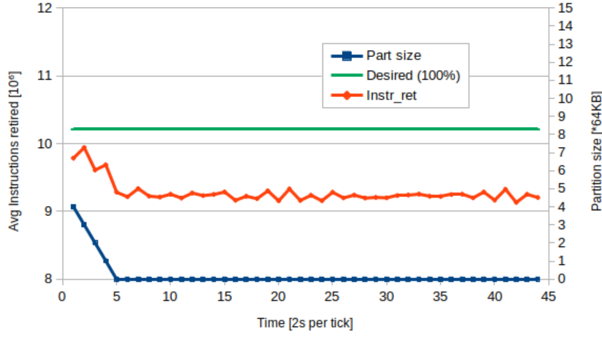

Fig. 7: SUSAN 100% target performance

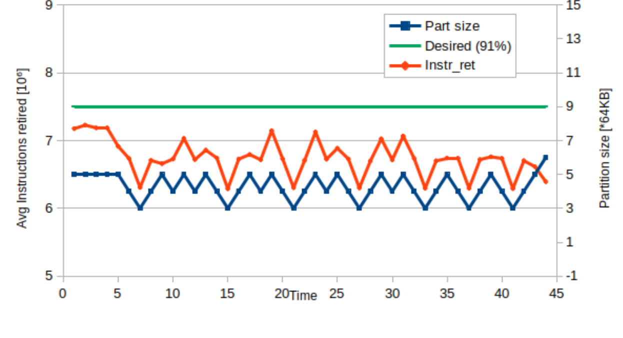

Fig. 9: Matmult$_{ikj}$ 91% target performance

E. Priority distribution

Different applications in a system can be of different importance. Our system utilizes two matrix multiplications as synthetic loads and one "real" scenario application, SUSAN. In this experiment, we assign priorities to our applications to force partitions into a specific application. We chose SUSAN to receive the highest priority, Matmult$_{ikj}$ to receive medium priority, and Matmult$_{ijk}$ to receive low priority. Introducing priorities means we also shift our distribution rules, presented as follows:

1) Monitor the QoS of active tasks in the system
2) Select the highest priority application (App$_{high}$) that does not have a current QoS higher than a desired QoS
3) Select the lowest priority application that has cache partition size > 1 (App$_{low}$)
4) Distribute one cache partition from App$_{low}$ to App$_{high}$
5) Go to step 1

Once the high-priority application meets its target QoS, the controller will actively shift focus to the second-highest priority task and so on. Our priority policy means a medium priority task will only get partitions once the high priority task has its QoS requirements fulfilled etc. We exemplify the priority distribution policy using a QoS threshold in our applications. The controller will shift cache distribution focus once an application runs at a higher QoS than its threshold. Table VII presents the experiment setup and contains application priorities and QoS threshold values. Matmult$_{ijk}$ has a non-applicable threshold since it is the lowest priority.

SUSAN has the highest priority, Matmult$_{ikj}$ has medium priority and Matmult$_{ijk}$ has low priority. Once SUSAN counts a presents a higher count of instructions retired than $7.5 * 10^6$ (95% of measured max), Matmult$_{ikj}$ will start to receive partitions. Figures 10, 11 and 12 shows the cache partition distributions for SUSAN, Matmult$_{ijk}$ and Matmult$_{ikj}$ using our prioritization policy.

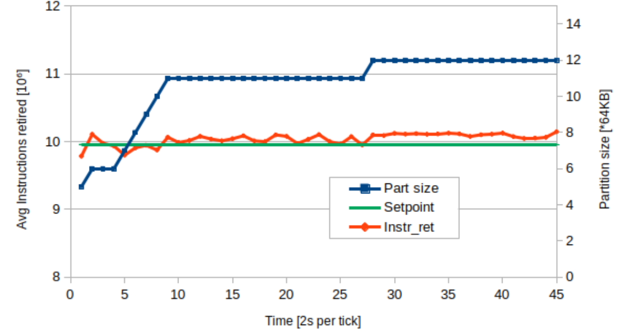

Fig. 10: SUSAN high priority

These experiments show a different cache partitioning distribution compared to the previous two experiments. SUSAN, now on high priority,receives a size increase at the first controller iteration, which increases the QoS to above the threshold. Since SUSAN now is above the threshold, Matmult$_{ikj}$ receives cache partitions from Matmult$_{ijk}$ for two iterations while also increases the QoS above the desired threshold. SUSAN, however, displays a QoS degradation during this time and is prioritized once again for cache partitions. This time, SUSAN takes cache partition size from Matmult$_{ijk}$ for six iterations, and the QoS finally hits the QoS threshold again. At iteration 30, SUSAN's QoS once again is below the threshold

TABLE VII: Initial setup

Application	Priority	Threshold	Value
Matmult_{ijk}	Low	N/A	N/A
Matmult_{ikj}	Medium	95%	$7.5 * 10^6\%$
SUSAN	High	95%	$9.95 * 10^6$

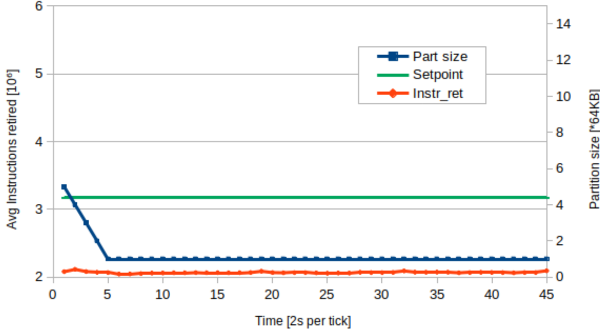

Fig. 11: Matmult_{ijk} low priority

and thus receives another cache partition from Matmult_{ijk}. From this point, the controller does not change the cache partition distribution.

F. Equal priority distribution

Our last experiment presents our prioritization policy when applications run the same priority. This policy presents the most complex problem since we here combine both fairness and priority. When two applications have the same priority, we trigger the fairness calculation and calculate the application's distance to its desired QoS. In this experiment, we assign SUSAN the same priority as in the previous experiment (high), but we lower the threshold by 0.2% to create a more interesting execution scenario. We furthermore lower the priority of Matmult_{ikj} to low, see Table VIII for experiment specification. We maintain the desired QoS from our previous experiment.

TABLE VIII: Initial setup

Application	Priority	Threshold	Value
Matmult_{ijk}	Low	N/A	N/A
Matmult_{ikj}	Low	N/A	N/A
SUSAN	High	93%	$9.95 * 10^6$

The graphs show the re-distribution policy when Matmult_{ijk} and Matmult_{ikj} run with the same priority (low). The controller immediately assigns one cache partition to SUSAN, which increases SUSAN's QoS to above the 93% threshold. The controller then triggers the fairness calculation for both matrix multiplications. Matmult_{ijk} has the most significant distance to the desired QoS and gets cache partitions from Matmult_{ikj} for five controller iterations. The increased cache space results in an increased QoS for Matmult_{ijk} but also a decreased QoS for Matmult_{ikj}. The controller starts to fluctuate at iteration 8, since Matmult_{ikj} has now the furthest distance to its desired QoS. The controller thus assigns one cache partition from Matmult_{ijk} to Matmult_{ikj}, a behavior maintained throughout the rest of the experiment execution.

G. Discussion

Our results show that it possible with relatively non-intrusive algorithms (0.3-0.7& CPU utilization) to control

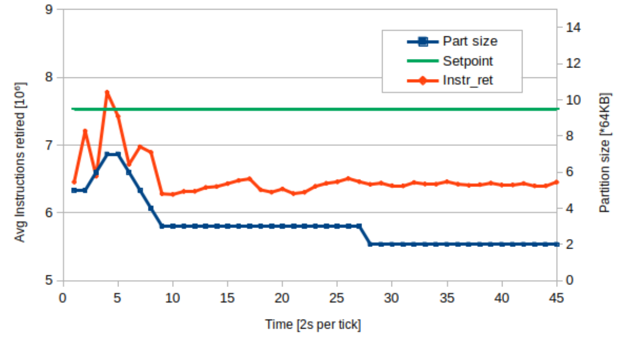

Fig. 12: Matmult_{ikj} medium priority

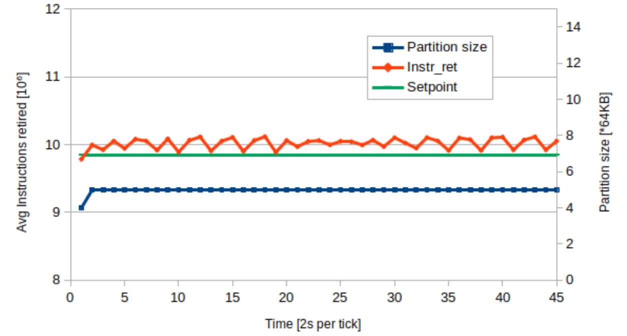

Fig. 13: SUSAN high priority

an application's QoS using only the means of cache re-partitioning. The first and most engaging discussion point is when to stop assigning cache partitions. All of our experiments, except for the all-different priority case, display a self-fluctuating state of the controller, which repeatedly decreases/increases the partition size of the same two applications. The fluctuating behavior is a consequence of a traditional constant controller that is working, but that case might not be practical. It is possible to add sanity checks within the controller that detects such behaviors since we have access to historical data and stop the re-partitioning procedure once detecting a fluctuating behavior. Stopping the re-partitioning procedure will increase the complexity of the controller significantly, since we then also need to add decision making for starting a stopped controller again.

The SUSAN's fluctuating performance - Fig. 10 - can be explained as follows. SUSAN trespasses the performance threshold setpoint already at controller iteration 1. SUSAN reaching the performance threshold mark this early is however an outlier and could be a result of SUSAN executing a couple of "lucky" executions. SUSAN goes below the threshold QoS setpoint again at controller iteration 4, wherefore the controller re-starts to assign partitions to SUSAN. Implementing a freezing functionality for the controller could, in this particular case, have led to a scenario where the controller freezes the partitions for SUSAN at partition size 1, while the current detected performance was a result due to a measurement anomaly.

VII. Summary

We presented here the idea of building an online cache partitioning controller that focuses on maintaining QoS for

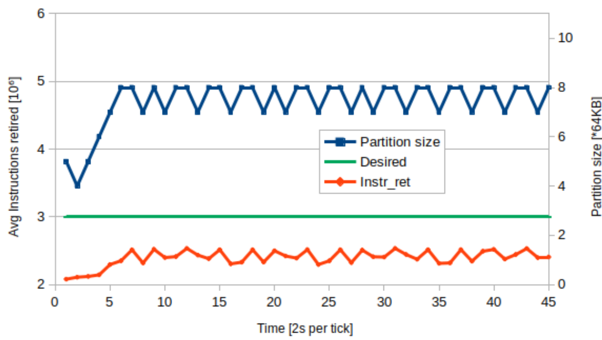

Fig. 14: Matmult$_{ijk}$ low priority

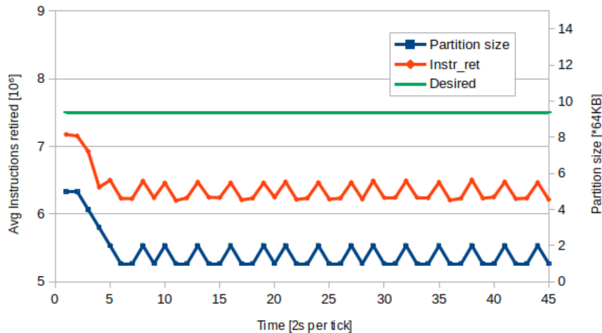

Fig. 15: Matmult$_{ikj}$ low priority

prioritized applications. We presented two primary cases; maintaining QoS based on a user-defined reference value and maintaining QoS through prioritization. Our results show that it is possible to control the execution time of several cache-bound tasks in a multi-core system by adjusting cache partition sizes. We introduced two controller modes: *fair* and *prioritized* and execute experiments using three applications. Our fair partitioning algorithms display favoritism towards the matrix multiplications because the difference between their current QoS and their setpoint performance is always greater than SUSAN's. The two matrix multiplications also show a more significant sensitivity towards an increased cache space which results in SUSAN never receiving cache partition space according to the fair policy. We therefore implement a priority policy that will assign partitions for applications with

A. Future work

Our controller uses a minimum cache partition size of one, but there is also the possibility of investigating cache partition sharing such that applications which are not important share the same cache partition. Sharing the same cache partitions will cause cache partition contention and reduce the QoS dramatically for the affected applications but will on the other hand free more cache partitions for the applications that do not share cache partitions. We also envision using more sophisticated controller techniques with other hardware that provides more available cache partitions. More available cache partitions means it can be possible to include a proportional element to the controller and change the cache redistribution

higher priority on first-hand. We show that our priority scheme prioritizes the QoS of SUSAN and increases its performance by 5% compared to a fairly partitioned system.

to more than just one per iteration. Other interesting works include investigating effective ways to freeze the system and thus counter the self-fluctuating effect resulting from our controller operating.

References

[1] J. Brock, C. Ye, C. Ding, Y. Li, X. Wang, and Y. Luo. Optimal cache partition-sharing. In *44th International Conference on Parallel Processing*, pages 749–758. IEEE, 2015.

[2] J. Danielsson, T. Seceleanu, M. Jägemar, M. Behnam, and M. Sjödin. Testing performance-isolation in multi-core systems. In *43rd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 604–609. IEEE, 2019.

[3] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten. Cache pirating: Measuring the curse of the shared cache. In *2011 International Conference on Parallel Processing*, pages 165–175. IEEE, 2011.

[4] M. Jagemar, A. Ermedahl, S. Eldh, M. Behnam, and B. Lisper. Enforcing quality of service through hardware resource aware process scheduling. In *23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, volume 1, pages 329–336. IEEE, 2018.

[5] T. Kloda, M. Solieri, R. Mancuso, N. Capodieci, P. Valente, and M. Bertogna. Deterministic memory hierarchy and virtualization for modern multi-core embedded systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–14. IEEE, 2019.

[6] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *14th International Symposium on High Performance Computer Architecture*, pages 367–378. IEEE, 2008.

[7] P. J. Mucci, S. Browne, C. Deane, and G. Ho. Papi: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, volume 710, 1999.

[8] S.P. Muralidhara, M. Kandemir, and P. Raghavan. Intra-application cache partitioning. In *International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–12. IEEE, 2010.

[9] M. K Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *39th Annual International Symposium on Microarchitecture (MICRO'06)*, pages 423–432. IEEE, 2006.

[10] X. Vera, B. Lisper, and J. Xue. Data cache locking for higher program predictability. *SIGMETRICS Performance Evaluation Review*, 31(1):272–282, 2003.

[11] Meng Xu, Linh Thi Xuan Phan, Hyon-Young Choi, Yuhan Lin, Haoran Li, Chenyang Lu, and Insup Lee. Holistic resource allocation for multicore real-time systems. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 345–356. IEEE, 2019.

[12] Y. Ye, R. West, Z. Cheng, and Y. Li. Coloris: a dynamic cache partitioning system using page coloring. In *Parallel Architecture and Compilation Techniques (PACT), 23rd International Conference on*, pages 381–392. IEEE, 2014.

[13] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 19th*, pages 55–64. IEEE, 2013.

[14] H. Yun, R. Mancuso, Z.P. Wu, and R. Pellizzoni. Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 20th*, pages 155–166. IEEE, 2014.