Run-Time Cache-Partition Controller for Multi-Core Systems

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

***Abstract*—The current trend in automotive systems is to integrate more software applications into fewer ECU's to decrease the cost and increase efficiency. This means more applications share the same resources which in turn can cause congestion on resources such as such as caches. Shared resource congestion may cause problems for time critical applications due to unpredictable interference among applications. It is possible to reduce the effects of shared resource congestion using cache partitioning techniques, which assign dedicated cache lines to different applications. We propose a cache partition controller called LLC-PC that uses the Palloc page coloring framework to decrease the cache partition sizes for applications during run-time. LLC-PC creates cache partitioning directives for the Palloc tool by evaluating the performance gained from increasing the cache partition size. We have evaluated LLC-PC using 3 different applications, including the SIFT image processing algorithm which is commonly used for feature detection in vision systems. We show that LLC-PC is able to decrease the amount of cache size allocated to applications while maintaining their performance allowing more cache space to be allocated for other applications.**

I. Introduction

Recent trends in the automotive industry show an increasing interest in high-performance computational machines. A common way to address the increased demand for computational capacity is the use of multi-core CPUs, which is a significant benefit to the autonomous industry due to the reduced size, weight, and power (SWaP) area [3]. Increasing the number of cores adds additional computational capacity, however, it also increases the system complexity. Multi-core systems are infamous for performance variations, which can become problematic in time-sensitive systems [8]. These variations often occur due to inter-core resource sharing, such as shared caches, shared memory bus, Translation Lookaside Buffers (TLB), shared DRAM-banks and others. These resources can be shared between cores, which means an application (e.g. app_0), executing on one core, does not have exclusive ownership of a single resource, instead it shares the resource with another application, (e.g. app_1), executing on an adjacent core. Such scenario can lead to shared resource contention where app_0 unexpectedly stalls, since app_1 has access to the resource.

The shared last level cache (LLC) has been a performance bottleneck in multi-core systems for a long time because of simultaneous accesses from multiple cores. In recent years, several studies have proposed methods aiming to mitigate LLC contention through isolation. Some examples are cache partitioning which partition the LLC so that accesses from one application do not affect the performance of another [11]. An additional technique is cache locking [12], that forces applications to use only certain cache lines. Another example is cache scheduling [6] that schedules applications to minimize conflicts in the cache memory. Isolating the cache memory can however be a costly process in terms of lost memory space and increased overhead.

We have devised a new way to optimize LLC partition allocation, during run-time. We implement a controller that continuously reads the instructions retired event from the Performance Monitoring Unit (PMU) [5] to estimate the application's performance. This paper focuses on the LLC, but the PMU supports a broad set of events [15], and our method can be applied to other shared resources as well - to be investigated in the future. The controller correlates the performance metrics and the cache partition size, and decides if an application needs more cache memory to achieve the desired performance or Quality of service (QoS). Our main contribution is:

- Propose a method to *automatically* select the minimum cache-size to be allocated to an application for achieving a desired QoS.

The rest of the paper is structured as follows. We give background information in Section II and describe the LLC partition controller we have implemented in Section III. An empirical study of the correlation coefficient and also a comparison study of our LLC partition controller versus statically assigned LLC partitions is described in Section IV. Section V describe work related to ours and we conclude the paper in Section VI.

II. Background

In the following, we discuss cache partitioning and it's relations to application performance.

A. Partitioning to avoid LLC contention

LLC contention occurs when multiple applications compete for the same cache lines. This can drastically degrade the execution time. Page-coloring, a.k.a cache coloring [13] or cache partitioning, is a way of disqualifying applications from using certain cache lines. LLC partitioning in Linux can be done by replacing the standard Buddy allocator [14], forcing applications to take a subset of the total number of cache lines. Forming LLC partitions is often done by assigning colors to an application. The colors are then used to control where data requests from the physical memory should be put in the cache, see Fig 1.

978-1-7281-4878-6/19/$31.00 ©2019 IEEE

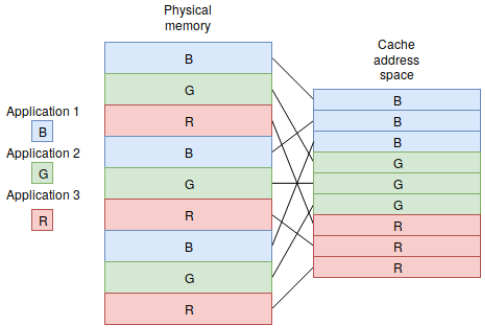


Fig. 1. Cache coloring

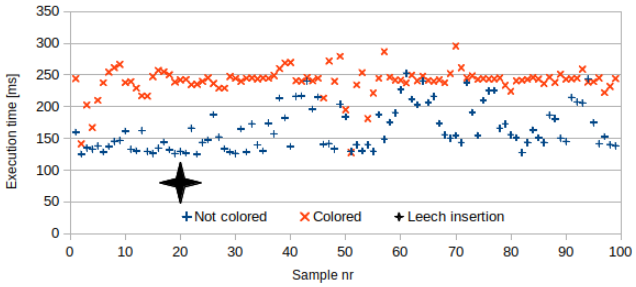


Fig. 2. Matrix multiplication - isolation example

The Figure shows three applications which split the cache memory equally. The applications are assigned three different colors in the physical memory which are then used to map memory rows to cache line locations. Cache colors are referenced using the set-associative bits of the LLC, calculated according to Equation 1 [13].

$$Nr.\ of\ Colors = \frac{Cache\_size}{Cache\_ways * page\_size} \quad (1)$$

We have used the combined DRAM-bank partitioning and LLC coloring tool called Palloc [14] to create LLC partitions. Palloc is a kernel module which runs partitions at the granularity of a page and replaces the regular Linux Buddy allocator with a colored page approach.

B. Cache partitioning effect

Page coloring can be very efficient for reducing the execution time oscillations of applications executing in a memory contentious environment [13]. We have illustrated such environment in Fig. 2 where one 512x512 matrix multiplication application runs iteratively 100 times on core 0. The blue pluses show 100 iterations of the matrix multiplication without page coloring. The red crosses show 100 iterations of the matrix multiplication using palloc page coloring with a cache partition size of 60. Another matrix multiplication starts at iteration 20, running on core 1. The purpose of the newly inserted matrix multiplication is to cause LLC contention, which happens as a consequence of sharing the same LLC.

Fig. 2 depicts a typical LLC contention scenario, where the execution time of the no-page-colored matrix multiplication starts to oscillate, after inserting the leech. The page-colored matrix multiplication is, on the other hand, undisturbed by the leech. It is, however, apparent that page coloring comes with

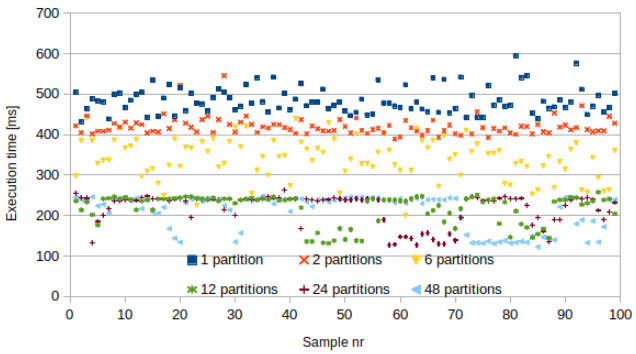


Fig. 3. Matrix multiplication using different cache partition sizes

an increased overhead due to extra latency in page allocations. Such trade-off can be worthwhile in time-critical systems when application time-predictability is essential. Overhead evaluations and Real-time performance impacts of the Palloc tool using bank partitions is extensively discussed in the Palloc paper.

Dimensioning the LLC partition sizes is one of the critical aspects when running multiple applications simultaneously. Assigning too small LLC partitions can significantly decrease the application performance. Fig. 3 shows the performance difference of the same matrix multiplication using various amount of LLC partition size.

Assigning only 1 LLC partition to the matrix multiplication significantly reduces the performance, compared to the execution in Fig. 2, which uses 60 LLC partitions. Increasing the LLC partition size to 2, significantly increases the performance compared to the 1 LLC partition assignment and so on. Fig. 3 also illustrates an "above LLC saturation point" scenario - when an application does not gain performance from being assigned more cache memory, which is a consequence of fully saturating the temporal locality of the matrix multiplication. For this dataset size, the number of cache misses cannot be reduced anymore and all data which can be re-used is being re-used. Thus, there is no increase in performance from increasing the LLC partition size further. In this case, the saturation point occurs at the 12 LLC partitions assignment. Further increasing the available LLC partitions, does not produce a significant performance impact on the application. Increasing the LLC size for this application will only allocate unnecessary resources. As a comparison, we could adopt a static partitioning strategy: for instance, assigning a 4^{th} of all cache partitions to each core in a 4 core system. In many cases, this may be a waste of valuable resources. Thus, we argue that it is beneficial to find the LLC saturation point at run-time, rather than statically assigning partitions.

III. Cache Partition Decision

There are many ways to create efficient LLC partitions. One possibility is to use exhaustive offline profiling for tasks, distributing the available cache partitions optimally to different tasks [2]. Offline profiling, however, needs complete knowledge of the applications running in the system. Changing the application set requires a complete re-profiling procedure before deploying new cache partitions. These limitations make

offline cache partitioning not feasible for most dynamic systems. In addition, some applications may also change their respective workload during execution, which can be very difficult to foresee at design-time.

This paper focuses on LLC-bound workloads, meaning that the respective performance is bound tightly with the amount of LLC misses, where more LLC misses equals less performance. It is possible to assume that an LLC-bound workload will benefit from receiving more LLC partitions and opens up ways for constructing re-partition methodologies.

For an app_0, the performance is denoted by the number of retired (reached the final step in the instruction pipeline) instructions. In the context of the used example, our theory is that:

- The performance of an LLC-bound process is strongly correlated to the number of LLC misses.
- Enlarging the corresponding partition size available for app_0 increases the performance and decrease the LLC misses.
- The correlation between performance and increased LLC partition size decreases as the number of LLC partitions increase, until a *LLC saturation point*, where other resources (may) become the bottleneck

Based on the theory above, we propose a correlation-based cache partition controller, *LLC-PC*, that tries to find the LLC saturation point - Fig. 4.

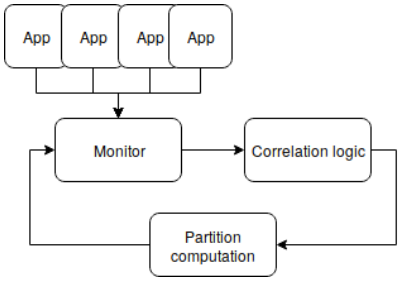


Fig. 4. LLC-PC

The cache controller is a correlation based control loop which regulates the cache partition size according to the correlation between a performance metric and the increase in cache size for a specific application. The controller will continuously increase the cache size for as long as the correlation between the increase in amount of cache partition size and the performance metric is high. Once the correlation starts to decline and reaches a certain threshold, an LLC saturation point has been found and the controller will stop assigning additional cache partitions to the specific application.

The correlation scheme to find the LLC partition saturation point is based on the *Pearson correlation coefficient* [1] - a statistics methodology to quantify the relationship between two datasets. The pearson correlation coefficient is calculated according to Equation 2, where r is the pearson correlation coefficient estimate, n is the number of samples, x is the first sample vector, $\overline{x}$ is the mean of the first sample vector, y is the second sample vector, $\overline{y}$ is the mean of the second sample vector and i is the iterator.

$$r = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \overline{x})^2(y_i - \overline{y})^2}} \quad (2)$$

The correlation coefficient ranges from values between -1 and 1. The absolute value of the correlation coefficient represents how strong the correlation is, where a higher value represents a stronger correlation. Correlation coefficients between 0.1 to 0.3 generally show a weak correlation, 0.4-0.5 show a medium correlation and greater than 0.5 show a strong correlation [4]. The correlation significance may, however, vary depending on the data set.

A. Controller implementation

We have implemented the LLC partition controller - *LLC-PC* - as a user-space application in the Linux operating system. LLC-PC employs the Palloc page-coloring interface, described in Fig. 5.

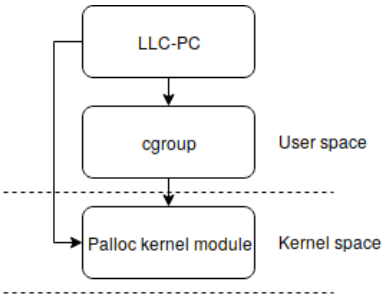


Fig. 5. System connections

LLC-PC handles application connections through message queues and assigns LLC partitions to the connected applications using the `cgroup` interface. The `cgroup` interface has an implemented file-system called `palloc`, which uses the LLC set associative bits for configuring LLC boundaries. The `palloc` kernel implementation creates the cache colors based on the information provided by the `cgroup` file-system. LLC-PC has also a connection to the `palloc` kernel space user interface to enable `palloc`.

The controller, see Fig. 4, consists of three parts. The monitor part, the correlation part and the partition computation part. The controller implementation is described in Algorithm 1.

The first forall block of the algorithm shows the connectivity part of LLC-PC, i.e., how the program deals with connected applications through message queues. Applications connect to LLC-PC by sending the application pid to a message queue. Applications furthermore notify LLC-PC of execution iteration ends by sending a ”done” message to the same message queue. If the system does not currently recognize the pid posted by an application, the create_application function is triggered. This function initializes an application variable and stores the newly created application to an array. If an ”end” message is received, an average value of instructions retired for the application is calculated, and the amount of average samples for the application is increased by 1.

The second forall block shows the actual LLC-PC controller part and starts with an application monitor part. The monitor

```
Initialize_palloc();
Initialize_PAPI();
while forever do
    /* Handle application connections */
    forall messages in message_queue do
        if message == new_application then
            initialize_application();
            tasks_in_system++;
        end
        if message == task_iteration_ended then
            calculate_avg_instructions_retired();
            avg_samples++;
            done = 1;
        end
    end
    /* Control loop segment */
    forall applications in tasks_in_system do
        /* Monitor application
           characteristics */
        instr_retired = read_pmu(pid);
        if avg_samples <= 3 then
            /* Calculate correlation */
            correlation =
             pearson(avg_instructions_retired[i..end],
             cache_partition_size[i..end]);
            /* Make partition decision */
            if correlation > 0.8 then
                partition_size++;
            end
        else
            /* Insufficient amount of data
               to calculate correlation */
            partition_size++;
            done = 0;
        end
        resize_cache_partition();
    end
    sleep();
end
```

Algorithm 1: LLC-PC pseduocode

continuously reads the instructions retired PMU event for all application pids which exists within the applications array. The instructions retired event is stored within another array, used for calculating the average instructions. If the amount of average samples for an application is less than 3, a correlation calculation will not be performed, since it is not possible to detect trends with so few values. Thus, if there are less than three available average samples, LLC-PC will increase the partition size by 1. If on the other hand, the amount of average samples is at least 3, LLC-PC will start to perform the correlation calculation. The correlation calculation uses the average instructions retired and partition history for one application as input data and provides a Pearson correlation coefficient as output data. The application input data to the Pearson calculation is provided as a sliding window filter ranging from i to the end of the vector. This window is implemented to ensure that only the most recent values are accounted for in the Pearson calculation, to provide a faster response of LLC-PC. Once the correlation calculation is complete, a partition decision can be made. If the correlation is over 0.8, the partition size of the application is increased by one. If not, the saturation point of the application has been found, and LLC-PC will not increase the partition size further.

The third step is to actuate the resize cache partition method, which goes through all currently active applications in LLC-PC and calls cgroup/palloc to create partitions accordingly. Finally, the sleep variable dictates the periodicity of the monitor loop and therefore controls the number of values given as input to the average performance calculations the average overhead of the LLC-PC monitor- and control loop averages at 73 μs. Decreasing the sleep timer will increase the amount of control-loop iterations per application samples and will thus increase the overhead while expanding the sleep timer will reduce overhead.

IV. Experiments

We here describe the experiment on the identification of a feasible correlation threshold, to be used to determine the LLC saturation point. We evaluate how well LLC-PC perform compared to a static LLC partitioning.

Our experiment platform is a desktop Intel® Core™ i5 computer, with specification details as in Table I.

TABLE I
HARDWARE SPECIFICATIONS INTEL® CORE™ I578850H

Feature	Hardware Component
Core	4xIntel® Core™ i578850H CPU (Skylake) 2.6GHz 32 KB 8-way set assoc. instruction caches/core +
L_1-cache	32 KB 8-way set assoc. data cache/core
L_2-cache	256 KB 4-way set assoc. cache/core
LLC	9 MB 12-way set assoc. shared cache
MMU	64 Byte line size, 64 Byte Prefetching, DTLB: 32 entries 2 MB/4 MB 4-way set assoc. + 64 entries 4 KB 4-way set assoc., ITLB: 128 entries 4 KB 4-way set assoc., L_2Unified-TLB: 1 MB 4-way set assoc., L_2Unified-TLB: 512 entries 4 KB/2 MB 4-way assoc.

A. Point of saturation - Correlation threshold

Finding the right correlation threshold value is essential to LLC-PC, since a too low threshold value can cause the LLC-PC to act too slowly and therefore assign too many LLC partitions to an application. A too high threshold value may, on the other hand, force LLC-PC to act too quickly, and to assign not-enough LLC partitions to an application. The following experiments describe how the correlation coefficient between performance and LLC partition size changes over time, using different workloads while increasing the LLC partition size.

The correlation-based approach is able to identify which resource has the dominant effect on the performance of the applications, and this might change after allocating a certain

Fig. 6. 512x512 matrix multiplication execution

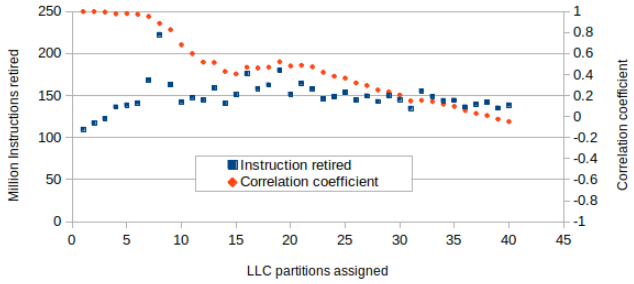
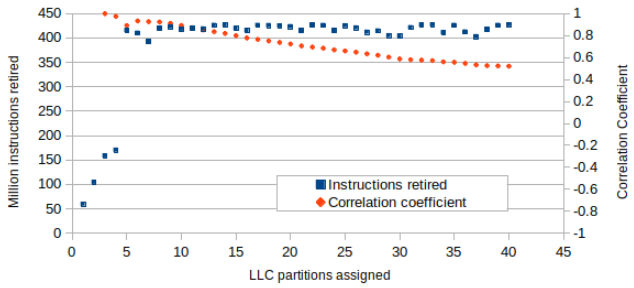


Fig. 7. 4 MB SIFT execution



amount of that particular resource, such as the LLC. Due to the space limitation, we will leave the management of multiple resources as future work and focus on a single resource which is the LLC.

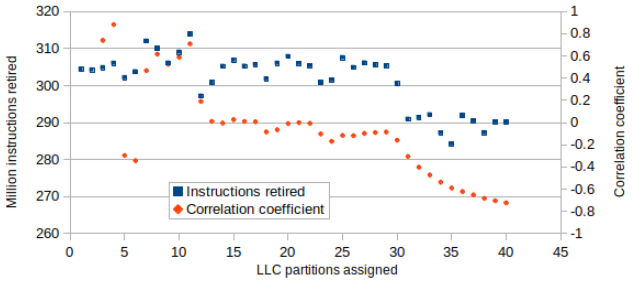
Matrix multiplication. This experiment exemplifies what happens when a cache intensive workload runs on different partition sizes. We chose a 512x512 matrix multiplication, which is a well-known cache optimization problem [7] to run, using an increasing amount of LLC. Fig. 6 depicts the matrix multiplication instructions retired on the left-hand side y-axis and the correlation relationship between the instructions retired and the cache partition size on the right-hand side y-axis.

The figure shows a gradually decreasing correlation curve and also a clear relationship between increased LLC partition size and instructions retired. The matrix multiplication reaches saturation at a partition size of 10.

SIFT. We test the SIFT algorithm, a commonly used feature detection algorithm to illustrate that our correlation theory works for not only synthetic workloads. Fig. 7 show as an execution of the SIFT algorithm run on a 4MB image with different cache partition sizes from 1 to 40. The figure shows an upwards going performance curve, with an absolute peak when assigned 37 cache colors. This peak is however very minor and can be explained as local deviation due to ”lucky” executions. The majority of the peak values are, however, within the 405 million - 425 million instructions retired interval, which is reached at a correlation coefficient of roughly 0.9 and continues to scale down.

Random Calculation. The purpose of this experiment is to exemplify what happens when a load is not LLC-bound. The random calculation program executes a set of random number requests and stores the random value into a variable. The variable is compared with another variable to find the highest value gained from the random number requests. We set the random number requests to 10^8 random number requests with

Fig. 8. Random calculation execution



a modulo of $5*10^5$ and increase the number of cache partitions assigned to this application by one each time the application is finished executing. Fig. 8 depicts the correlation coefficients from the random calculation test.

The figure shows an entirely different result from the matrix multiplication correlation graph. Instead of a continuously decreasing correlation, the correlation values are irregular at first but then saturates on iteration 13 to a correlation coefficient of 0.

B. Summary of experiments

There are two common nominators for the LLC-bound applications in these experiments. Firstly, the number of instructions retired increase when increasing the LLC partition size. The increase in instructions retired is reasonable since the application gets significantly more LLC. Secondly, there is a point where the instructions retired curve levels off to a stable state. The curve levels out when the application is assigned a certain number of LLC partitions. Thus, we have found the LLC saturation point for this given application. We can conclude that in our experiments, the LLC saturation point of the curve is a certainty at a correlation coefficient of 0.8. Using this conclusion, we set the correlation threshold to 0.8 in the subsequent LLC-PC experiments, which is the point from which LLC-PC will not assign more cache partitions to an application. Using a correlation over the entire dataset at all time, however, makes LLC-PC slow to saturate. The saturation of the system can, however, be hastened through introducing a sliding window, which only tracks the most recent cache partition and instructions retired measurements. Using a sliding window means the system will only react to current execution trends, not considering the earliest stages of the system execution.

C. LLC-PC evaluation

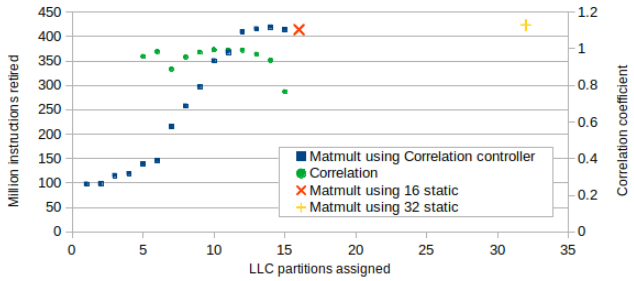
One static way of assigning LLC partitions is to split all available LLC partitions equally between the cores. Our test environment has 4 cores and 128 available cache partitions, thus each core gets $128/4 = 32$ static LLC partitions as a first reference value. We also use 16 partitions per core as a second reference value. Below, we show an evaluation of static partitioning vs. LLC-PC, using different sizes of the previously introduced LLC-bound workloads. We ran each test a total of 5 times. LLC-PC runs the experiment setup listed in Table II.

For the sake of test simplicity, re-partition regulations are made once each application iteration, however, in theory a re-partition decision could be made each time a memory manager

TABLE II
LLC-PC SPECIFICS

Property	Value
Available LLC partitions	128
Correlation window size	5
Correlation threshold	0.8
Control loop sleep	50ms

Fig. 9. Comparison of 756x756 matrix multiplication executions

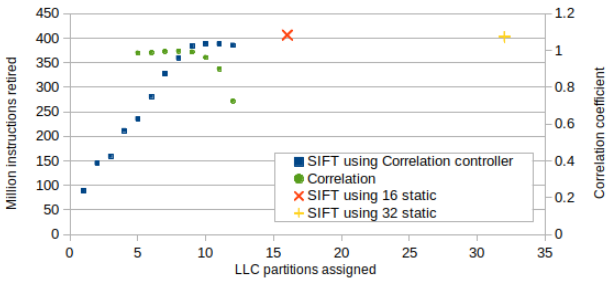


call is made. We execute each test sequentially for a more straightforward interpretation of the results. The control loop address each task individually, which means that it is possible for the controller to handle multiple tasks concurrently at the same time. It can also be argued that the control loop sleep time would be a coefficient of the execution time such that the sampling occurs only a certain amount of times every iteration, however since the execution time can be very hard to predict, we chose to go for a statically set sleep timer. Such a solution, however, requires accurate execution time prediction of an application, which becomes very troublesome since the execution time of each application can change dramatically due to cache re partitioning and would possibly mean more overhead to LLC-PC. We chose 50 ms as control loop sleep in order to get at least 100 measurement values for the average calculation for all application variations.

Matmult and SIFT running under LLC-PC. We evaluate LLC-PC versus a static partition based solution which uses a LLC partition size of 16 and 32. Fig. 9 and Fig. 10 depicts the execution flow of a 756x756 matrix multiplication and a 8MB sift execution respectively, using LLC-PC. The left-hand side y-axis of the graphs plots the median instructions retired (i.e., performance) per 50 milliseconds of the application using LLC-PC (blue squares), 16 statically assigned cache partitions (orange cross) and 32 statically assigned cache partitions (yellow plus). The right hand-side axis show the correlation over time using LLC-PC. A higher value on the left-hand side axis means more instructions executed per 50 milliseconds and is, therefore, better than a low value. The x-axis shows the number of partitions used, where a lower value is preferred since more cache partitions can be given to other applications.

Fig. 9 shows a full LLC-PC run of a 756x756 matrix multiplication, where the system saturates at 16 partitions, with comparable performance to that of the static partitions. For this particular matrix multiplication size, the static partition size was equal to the correlated size. Statically increasing the LLC sizes to 32 does not improve the matrix multiplication

Fig. 10. Comparison of 8MB SIFT executions



performance significantly. Furthermore, Fig. 10 show SIFT operating within the LLC-PC, with a final assignment of 13 LLC partitions at which point the correlation value has dropped from 0.89 to 0.72. The correlation-based methodology almost reaches the same performance achieved by the static LLC partition allocations.

Table III and Table IV further compares LLC-PC with a static partitioning strategy using different sizes of the workloads. W_{size} is the workload size and C_{size} is the LLC partition size assigned to the application, C_{instr}, S_{16} and S_{32} show the median million instructions retired per 50 milliseconds of the matrix multiplication using LLC-PC, 16 statically allocated LLC partitions and 32 statically allocated LLC partitions respectively.

TABLE III
MATRIX MULTIPLICATION TESTS

W_{size}	C_{size}	C_{instr}	S_{16}	S_{32}
256x256	7	432.73	428.17	419.49
512x512	13	420.11	432.78	428.54
756x756	16	414.36	424.63	428.39

TABLE IV
SIFT TESTS

W_{size}	C_{size}	C_{instr}	S_{16}	S_{32}
1MB	6	395.38	406.56	408.79
2MB	7	384.96	413.01	405.85
4MB	9	387.80	410.61	405.87
8MB	13	385.53	406.31	402.58

Table III shows the benefit of LLC-PC, especially using the smallest matrix multiplication size of 256x256, which saturates at a partition size of 7. Increasing LLC partition size to 16 and 32 does not increase the performance, and would thus be a wasteful LLC assignment since other applications could have used the LLC partitions. The larger 512x512 matrix multiplication size saturates at an LLC partition of 13, which is 3 LLC partitions less than the static 16 allocation, which does not notably change performance. Table IV further compares LLC-PC with the static partitioned strategy using different image sizes, where W_{size} is the image size used by the SIFT application and C_{size} is the LLC partitions assigned to SIFT by LLC-PC. C_{instr}, S_{16} and S_{32} show the median million instructions retired per 50 milliseconds for using LLC-PC, 16 statically allocated LLC partitions and 32 statically

allocated LLC partitions respectively. The table shows a close-to static performance for all different image sizes using less LLC partitions. The 8MB image receives 13 LLC partitions from LLC-PC and is which is relatively close to the S_{16} allocated partitions, which saves 3 LLC partitions from waste. Increasing the image size further could potentially trespass the S_{16} allocation using the correlation controller.

V. Related Work

Our work is based on the Palloc [14] page coloring framework, which can be used for partitioning both the cache and DRAM banks. While the authors show that Palloc efficiently can be used to counter resource contention where all cores gain the same amount of cache partitions, they do not consider to optimize the cache assignments for each application. We aim to further extend this approach by using correlation-based partitioning decisions and therefore gain more efficient cache partitions. Ye et al. [13] presented the Coloris cache coloring engine which uses a threshold scheme, based on performance counters. The Coloris approach forms cache partitions based on how many cache misses one process contributes to the total amount of cache misses of all processes. Our approach differs from Coloris, as we look at how the performance of a process correlates to the cache misses of the same process. Perarnau et al. [10] presents another cache coloring scheme and argues that creating feasible cache memory partitions is best left to the user, since they have most knowledge of the application. We argue that it is difficult to know beforehand how much cache an application needs, in order to achieve a certain performance level. It is therefore beneficial to use a method that makes the cache partition decision automatically at run-time.

VI. Conclusion

We have created a correlation based LLC partition controller, called LLC-PC, which can be used to find LLC partition sizes for workloads with unknown cache usage. We evaluate LLC-PC using two LLC heavy loads, a Matrix multiplication, and a SIFT feature detection algorithm. The results show that LLC-PC can be used for this set of workloads to reduce the amount of cache size given to an algorithm compared to a static 32 cache LLC partition assignment, and also in most cases a 16 LLC partition assignment - while still maintaining similar performance. We can probably find better cache partitions through thorough offline measurements and code analysis; however, our aim is not to find the absolute optimal cache partitions but rather find sufficient cache partition sizes during runtime of an algorithm.

Our prime focus has been to create a generalizable correlation model. We can apply the correlation model on any shared resource that has a performance counter event and a partitioning strategy which affect the shared resource, e.g., TLB partitioning [9]. Our future work includes introducing new partitioning strategies. We would also like to create a methodology for solving the multi-objective control problem when balancing multiple shared resources usage.

References

[1] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. Pearson correlation coefficient. In *Noise reduction in speech processing*, pages 1–4. Springer, 2009.

[2] Jacob Brock, Chencheng Ye, Chen Ding, Yechen Li, Xiaolin Wang, and Yingwei Luo. Optimal cache partition-sharing. In *2015 44th International Conference on Parallel Processing*, pages 749–758. IEEE, 2015.

[3] Alessio Bucaioni, Saad Mubeen, Federico Ciccozzi, Antonio Cicchetti, and Mikael Sjödin. Technology-preserving transition from single-core to multi-core in modelling vehicular systems. In *European Conference on Modelling Foundations and Applications*, pages 285–299. Springer, 2017.

[4] Jacob Cohen. *Statistical power analysis for the behavioral sciences*. Routledge, 2013.

[5] Thomas Gleixner. Linux Performance Counter announcement, 2008. URL `http://lkml.org/lkml/2008/12/4/401`.

[6] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. Cache-aware scheduling and analysis for multicores. In *Proceedings of the seventh ACM international conference on Embedded software*, pages 245–254. ACM, 2009.

[7] Monica D Lam, Edward E Rothberg, and Michael E Wolf. The cache performance and optimizations of blocked algorithms. In *ACM SIGARCH Computer Architecture News*, volume 19, pages 63–74. ACM, 1991.

[8] Abdelhafid Mazouz, Denis Barthou, et al. Study of variations of native program execution times on multi-core architectures. In *2010 International Conference on Complex, Intelligent and Software Intensive Systems*, pages 919–924. IEEE, 2010.

[9] Shrinivas Anand Panchamukhi and Frank Mueller. Providing task isolation via tlb coloring. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 3–13. IEEE, 2015.

[10] Swann Perarnau, Marc Tchiboukdjian, and Guillaume Huard. Controlling cache utilization of hpc applications. In *Proceedings of the international conference on Supercomputing*, pages 295–304. ACM, 2011.

[11] G Edward Suh, Larry Rudolph, and Srinivas Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28(1):7–26, 2004.

[12] Xavier Vera, Björn Lisper, and Jingling Xue. Data cache locking for higher program predictability. In *ACM SIGMETRICS Performance Evaluation Review*, volume 31, pages 272–282. ACM, 2003.

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

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

[15] Gerd Zellweger, Denny Lin, and Timothy Roscoe. So many performance events , so little time. *APSys '16*, 2016. doi: 10.1145/2967360.2967375.