LLM-shark – A Tool for Automatic Resource-boundness Analysis and Cache Partitioning Setup

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*—We present LLM-shark, a tool for automatic hardware resource-boundness detection and cache-partitioning. Our tool has three primary objectives: First, it determines the hardware resource-boundness of a given application. Secondly, it estimates the initial cache partition size to ensure that the application performance is conserved and not affected by other processes competing for cache utilization. Thirdly, it continuously monitors that the application performance is maintained over time and, if necessary, change the cache partition size. We demonstrate LLM-shark's functionality through a series of tests using six different applications, including a set of feature detection algorithms and two synthetic applications. Our tests reveal that it is possible to determine an application's resource-boundness using a Pearson-correlation scheme implemented in LLM-shark. We propose a scheme to size cache partitions based on the correlation coefficient applications depending on their resource boundness.**

I. Introduction

The internal memory subsystem of a processor is often limited, cache memories for instance, often host a memory area that ranges from two digit KB's to single digit MB. The small memory space means it is improbable that an application's entire memory footprint can fit within one of the caches. As such, the caches will, at some point during application execution, become full. When new memory is requested, and when the necessary memory block is not present within the cache, it is instead fetched from the main memory and brought into the cache, replacing old data. Fetching the DRAM data produces a significant delay and will produce processor pipeline stalls while waiting for the memory block to become available.

Modern computers often utilize multi-core processors to increase throughput. The multi-core processors typically implement a shared cache policy, where at least one cache is shared between all cores. When multiple applications execute on different cores while sharing the same cache, there is a high risk that one application's memory requests will repeatedly replace another application's data, causing a resource contention scenario called *cache contention*. Cache contention causes execution-time jitters and performance degradation [7].

We can also find contention in other resources such as the Translation Lookaside Buffer (TLB) [15], the memory bus [23] and even the DRAM [24], but in this paper we focus on the cache related issues. Several mitigation techniques for cache contention exist, including page coloring [22] (a memory allocation scheme implemented in the memory management unit), cache way-partitioning (provided by the hardware manufacturer), and cache locking schemes [21]. The mitigation techniques improve the execution-time jitters, at the cost of implementation complexity and/or execution-time overhead. Due to the disadvantages, it is desirable to only use mitigation techniques when necessary, i.e., when there is a risk for cache contention that would degrade the performance of the system.

Determining the necessity of a mitigation technique is not a straight-forward process, since engineers have to carefully investigate the run-time behavior of each individual application and may also have to inspect the application code. Due to the code complexity of many modern applications this can quickly turn into a time-consuming procedure.

This paper describes our tool, LLM-shark, that determines partitioning techniques and partitioning sizes. LLM-shark monitors the run-time behavior of applications using hardware performance counters and creates a characteristics profile of an application. We use the characteristics profile to determine how much the performance of an application depends on a certain resource. We make a resource-boundness estimations using the Pearson-correlation coefficient and then suggest the usage of specific mitigation techniques. We, furthermore, utilize the correlation coefficient to determine how much memory the selected mitigation technique must assign to the application. The main contributions in this paper are:

- A method to quantify resource-boundness automatically.
- An evaluation of how applications perform in a cache restricted environment and how the results are in line with our resource-boundness estimation.
- Automatic calculation of the amount of needed cache memory based on the above estimation.
- An implementation of the above aspects in the tool LLM-shark, and a proof of concept study showing the feasibility of the tool.

II. Background

A. Performance counters

Many architectures implement a performance monitoring unit (PMU) [1] to monitor hardware resource usage. The PMU follows a large set of events, including CPU pipeline resources, various internal memory events such as cache/TLB events, and also off-core events such as DRAM accesses and interrupts [25]. The PMU utilizes hardware-implemented performance monitor counters (PMC) that increments each time a specific event occurs. Modern processors typically contain several PMC's to simultaneously measure a set of different events. Our test environment processor is an Arm Cortex-a53 CPU, with capabilities for six simultaneous PMCs measuring six different hardware events simultaneously. In this paper, we utilize the Performance API (PAPI) [14] which is a front-end framework for the standard Linux API for performance counters - Perf [9]. PAPI has extensive default support for the branch unit and the last-level cache.

B. Resource-boundness

In the context of our paper, we define the performance of an application as the number of instructions completed per unit of time. In Eq. 1, P denotes the performance metric, I_t the number of instructions retired of an application and t as the time interval.

$$P = \frac{I_t}{t} \tag{1}$$

We use the definition from Eq. 1 to trace an application's performance through a PMU event called *instructions retired*, which increments each time an instruction leaves the final write-back stage of the processor pipeline. Thus, we define the application performance as the number of instructions retired at a specific sampling frequency, a higher number meaning higher performance. This definition is not suitable for all types of applications, such as network applications that heavily utilize tight and small busy-wait loops, causing a high instructions retired rate with no perceived system-level performance. The performance metric for such applications instead is packets per second or similar [11] system-level metrics. In this paper, we target non-I/O-bound applications, for which the above performance definition is applicable.

An application is typically built of millions of instructions where each instruction takes at least one clock cycle to complete, assuming an in-order, non-super-scalar processor. In ideal conditions, the processor executes an instruction without any delays, which means a one-cycle instruction will take one cycle to finish, a two-cycle instruction will take two cycles to finish, etc. An application running on a 1.2GHz processor, without any external disturbances will thus execute instructions equivalent to 1.2 billion cycles per second.

It is however unrealistic for an application to execute instructions close to the processor clock frequency, due to cycle-disturbances, such as branch mis-predictions, structural data hazards, memory wait operations and also operating system overheads. All cycle-disturbances causes result in stall penalties within the processor pipeline and means that an application's instruction is not allowed to execute for a certain number of clock ticks. One common source for disturbance is the register memory itself, which is typically very small and can therefore not host the complete application data set. When the register memory does not contain requested data, the data needs to be fetched from L_1D-cache and a one cycle penalty stall will be inserted into the processor pipeline, which halts the processor from executing the instruction. This cycle stall penalty, thus, halts the application from executing an instruction, which means that the application will suffer a performance degradation.

The cycle stall penalty varies depending on the hardware unit. Application data that is not present within L_1D-cache enforces an even greater pipeline stall penalty (e.g., 10 cycles) and needs to be fetched from the L_2-cache, etc,. Similar stall penalties are also present in other various computer components such as the branch predictor unit (BPU) and the Translation Lookaside Buffer (TLB). An application with a high stall cycle penalty count is able to execute less instructions per time interval than an application which contains few stall cycle penalties. Thus, an application's performance builds a dependency towards the stalls, where more stalls infer a decrease in performance.

The resource that causes the most cycle stalls to an application causes the greatest effect on the applications' performance and therefore presents the strongest resource-boundness. Identifying an applications' resource-boundness becomes of great importance in multi-core systems, since some resources are physically shared across different cores. Two applications that display noticeable resource-boundness towards the same shared resource such as the Last-level cache can lead to cache contention, causing both applications to suffer from (potentially severe) performance degradation.

C. Cache partitioning

Page-coloring is a software approach to partition a cache for mitigating cache contention. Page-coloring creates an allocation scheme for free pages and assigns the pages to a fixed position within the cache. Page coloring, thus, alters an application's data positioning within the cache. Fig. 1 demonstrates how page coloring maps the pages of three applications (B, G, R) to different positions in the cache memory.

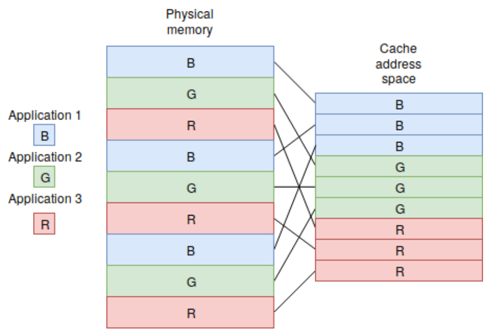

Fig. 1. Cache coloring

Applications executing within a page-colored environment will not suffer from shared cache contention since page coloring provides a clear border in the cache between where applications are allowed to position data. Page-coloring utilizes a cache's set-associative and presents an additional overhead to memory allocations due to algorithm complexity. The trade-off with cache partitioning, therefore, stands between performance and isolation.

D. Analyzing resource-boundness

We discuss here a statistical approach to quantify the resource-boundness using the Pearson correlation coefficient.

The Pearson-correlation coefficient has three types of outcomes, 1 - which means a complete positive correlation between two datasets; 0, which means no correlation between the datasets; and -1, which means complete negative correlation. We exemplify the three types of correlation in Figure 2.

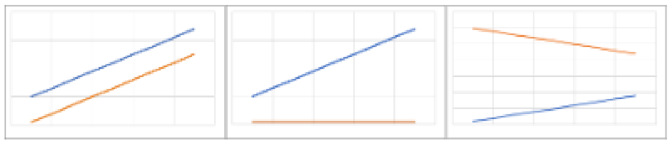

Fig. 2. Example of positive- (left) zero- (middle) and negative (right) correlation

We analyze resource-boundness by investigating negative relationships between the number of instructions completed and a resource event. Negative relationships mean either the number of instructions completed increase while the number of resource events decrease, or vice-versa. We utilize the PMU - that consists of several performance monitor counters (PMCs) - to monitor an applications' performance. We monitor the *Instructions retired* event, which counts the number of instructions that went through all processor pipeline stages, as a quantifiable performance metric. The second metric defines the kind of resource-boundness the user is interested in (e.g. if interested in cache related boundness we count the cache misses - via counters such as L_1D-cache-misses, etc).

In this paper, we utilize the analysis part of LLM-shark to determine the necessity of placing an application into a Last-Level Cache partition container [22]. Here, we mainly focus on the L_2-cache misses as performance counter-event.

The resource-boundness analysis further covers three actions:

- *Determine the event set.* To determine the resource-boundness, we first need to determine which counter events to sample. The complete counter set (perf+PAPI) lists a total of 116 events. For this paper, we chose to use only the 11 PAPI preset events.
- *Sample performance counters.* We sample selected performance counters during application execution at a fixed rate and save the data for resource-boundness analysis.
- *Assessment of the resource-boundness.* We utilize the performance counter samples from the previous step to calculate the Pearson-correlation coefficient. We quantify the magnitude of the resource-boundness according to the approach proposed by Mindrila and Balentyne [13]: the coefficients are compared using positive and negative values where 0-0.3 is considered none or weak, 0.3-0.5 weak, 0.5-0.7 moderate and greater than 0.7 is considered strong. Since the L_2-cache-misses provide a negative impact on the application, we are only interested in negative correlations. We therefore only consider applications for cache partitions if they present a correlation less than -0.25

III. Methodology

In previous work, we discussed the definition of resource-boundness [6] and also the consequences of running resource-bound loads simultaneously on different cores [3] [5]. The main take-away point is that resource-boundness is an important factor to consider when partitioning a system since the resource-boundness is an indicator of what resource an applications' performance depends on.

This section investigates the resource-boundness of six different applications, four of them implementing feature detection algorithms (SUSAN, Harris, SIFT, and FAST), and two presenting synthetic workloads (Matrix multiplication and Bubblesort). We illustrate the effects of resource contention on the execution of the six applications and discuss the relation to the respective correlation coefficient values.

A. System model

The relevant characteristics of the six mentioned applications are presented in Table I.

TABLE I
LLC-PC SPECIFICS

Application	Data input	type
Harris	2 MB bmp	Corner detection
SUSAN	2 MB bmp	Corner detection
FAST	12 MB bmp	Corner detection
SIFT	256 KB pgm	Object detection
Matmult	200x200 array	Synthetic
Bubblesort	20000 elements array	Synthetic

We use a variety of data input to showcase the usability of LLM-shark. The purpose here is not to create a comparison study on which application executes best in certain circumstances. Instead, our aim is to show that LLM-shark works for a variety of applications, independently on the applications memory footprint. Each application runs within the execution context of the tool, which starts an application, samples the desired performance events during the application execution, calculates a resource-boundness estimation and finally positions the application within a cache partition container. Figure 3 depicts the respective core functionality and execution flow. It shows the seven major execution steps of LLM-shark, which we describe in detail as follows:

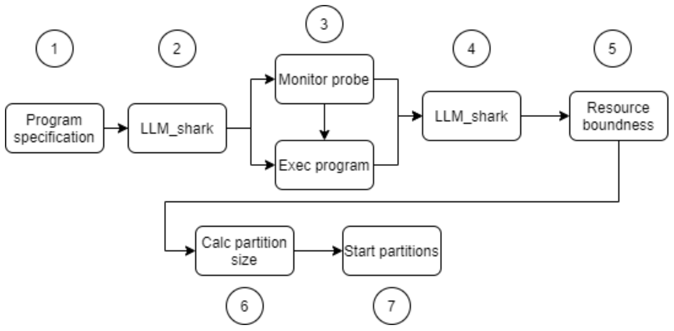

Fig. 3. LLM shark execution flow

1) *Application identification phase* - LLM-shark uses the filepath to an application executable to run the application within an LLM-shark context.
2) *Initialization phase* - initializes the instructions retired performance counter together with the desired counters.
3) *Fork phase* - LLM-shark executes a fork operation and runs the application within the context of child process. During the child process' execution, we monitor the performance counters of the child process continuously at a sampling frequency.
4) *Synchronization phase* - when the forked application has stopped, we store all the performance counter data from the child process and compute the correlation.
5) *Data store phase* - store the resource-boundness of the application so that appropriate actions such as cache partitions can be made.
6) *L_2-cache partition calculation phase* - calculate the L_2-cache partition size for each application based on the application correlation.
7) *L_2-cache partition actuation phase* - execute the applications within their respective L_2-cache partition containers.

LLM shark utilizes performance counters in a context of PAPI, a framework that includes preset performance counter events and the native counters that are specified by the Perf API. In this paper, we use a Xilinx zynq zcu102 evaluation kit which supports 13 preset PAPI counters and an additional 103 native performance counters. For the sake of readability, we only include results containing the preset PAPI counters, as data from 116 events are too much to present in one paper. We list the PAPI preset counter events with a short description in Table II.

TABLE II
PAPI PRESET COUNTER EVENTS

Event name	Brief explanation
PAPI_L1_DCM	L_1D-cache misses
PAPI_L1_ICM	L_1I-cache misses
PAPI_L2_DCM	L_2-cache misses
PAPI_TLB_DM	DTLB misses
PAPI_TLB_IM	ITLB misses
PAPI_TOT_INS	Instructions retired
PAPI_HW_INT	Hardware interrupts
PAPI_LD_INS	Memory load instructions
PAPI_SR_INS	Memory store instructions
PAPI_BR_INS	Branch instructions
PAPI_TOT_CYC	Processor cycles completed
PAPI_L1_DCA	L_1D-cache accesses
PAPI_L2_DCA	L_2-cache accesses

We limit this paper to only focus on non I/O-bound applications where the perceived performance of an application is equal to the number of instructions retired per time interval. As such, we omit the PAPI_HW_INT counter since that counter-event defines an application type that we are not interested in. PAPI_TOT_INS defines our performance metric and will be used in all correlation calculations versus another hardware resource. We, therefore, omit this counter from correlation calculation since correlating a value against itself always is one and will not be meaningful. We also omit PAPI_TOT_CYC since it presents the number of active clock cycles for an application and does not hold relevance to the internal memory hierarchy.

IV. Application Experiments

We have executed our experiment on two scenarios - baseline and contended. The baseline scenario is defined by the target application running without any deliberately disturbing load. The contended scenario presents the application running simultaneously with a leech application causing artificial L_2-cache contention. We list our test platform in Table III.

TABLE III
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. Baseline scenario

Table IV shows the median execution-time for each investigated application, taken over 100 measurements. We illustrate the instructions retired and the L_2-cache misses of the Harris algorithm in Figure 4 and the FAST algorithms in Figure 5 for demonstrative purposes.

TABLE IV
APPLICATION BASELINE EXECUTION-TIME

Application	Median execution-time (ms)
Harris	306
SIFT	750
SUSAN	189
Matmult	224
FAST	133.8
Sort	797

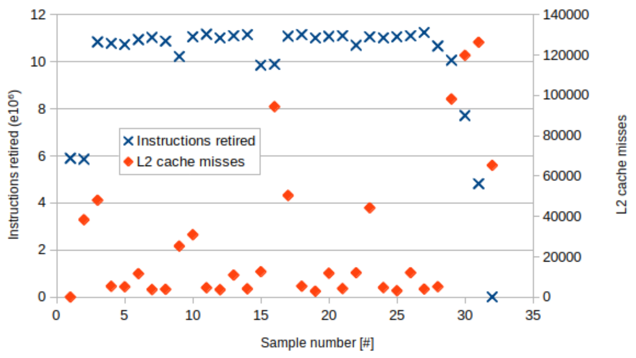

Fig. 4. Harris execution characteristics

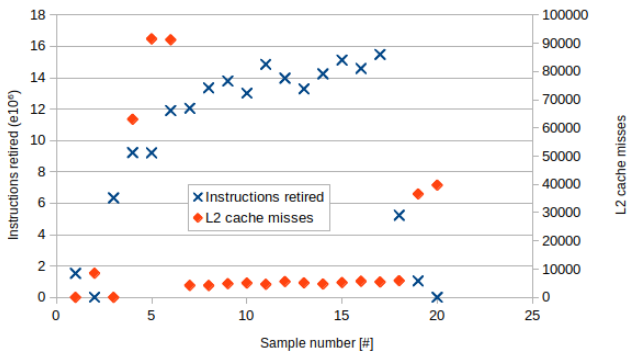

Fig. 5. FAST execution characteristics

Fig. 4 and Fig. 5 illustrates the Harris and FAST applications' execution profiles running within the context of LLMshark at a 200Hz sampling frequency, denoted by the x-axis. The left y-axis denotes the number of instructions retired with blue crosses, and the right y-axis denotes the number of L_2-cache misses with red diamonds. The execution profiles of these two applications are visibly different.The trend for the Harris application is that the number of instructions retired decreases while at the same time the L_2-cache misses increases. This trend is most notable at the last four measurement values; a similar, weaker trend can also be detected in the rest of the measurements. The FAST application do not have the same trend since the instructions retired continuously increases while the L_2-cache misses remaining the same at a count of 5000 for most of the application. There are small trends between sampling points 4 and 6. The majority of the application is, however, unaffected by L_2-cache misses. We list the correlation coefficients for all different counters for our six applications in Table V.

TABLE V
CORRELATION BETWEEN INSTRUCTIONS RETIRED AND PAPI PRESET COUNTERS

Counter	Harr	SUS	FAST	Matm	SIFT	Sort
BR_INS	0.69	0.48	0.82	0.19	0.63	0.98
BR_MSP	0.77	0.36	0.71	0.28	0.63	0.04
L1_DCA	0.76	0.65	0.83	-0.89	0.85	0.86
L1_DCM	-0.03	0.26	0.26	**-0.80**	0.75	0.23
L1_ICM	**-0.39**	**-0.29**	0.68	0.49	**-0.3**	-0.06
L2_DCA	**-0.25**	0.28	0.1	-0.83	0.49	0.15
L2_DCM	**-0.49**	0.2	0.12	**-0.84**	**-0.26**	-0.08
LD_INS	0.71	0.61	0.83	**-0.99**	0.87	0.91
SR_INS	0.79	0.43	0.16	-0.06	-0.36	0.79
TLB_DM	**-0.56**	-0.13	-0.05*	0.27	-0.01	-0.02
TLB_IM	**-0.53**	-0.13	0.23	0.31	-0.06	0.02

The table shows the correlation between the preset PAPI counters and the number of instructions retired for each application. Our theory is that applications that negatively correlate to a performance counter that implies pipeline stalls, such as the L_2-cache misses or DTLB misses will be sensitive in terms of execution-time when other applications utilize the same resources. The sensitivity depends on the correlation value magnitude; a higher correlation means the application will be more prone to performance implications if other applications are using that same resource. We summarize the negative correlation coefficients for each application in Table VI since these counters show an indication for resource-boundness.

TABLE VI
CORRELATION SUMMARY

Application	Resource-boundness
Harris	L_1I-cache misses (Weak) L_2I-cache accesses (Weak) L_2-cache misses (Moderate) DTLB misses (Moderate) ITLB misses (Moderate)
SUSAN	L_1I-cache misses (Weak)
FAST	No resource-boundness
Matmult	L_1D-cache misses (Strong) L_2-cache accesses (Strong) L_2-cache misses (Strong) L_2-cache misses (Strong)
Bubblesort	No resource-boundness
SIFT	L_1I-cache (Weak) L_2-cache misses (Weak)

Out of the six applications, three - SIFT, Harris and the Matrix multiplication - display weak, moderate and strong negative boundness to the L_2-cache misses counter, respectively. Harris furthermore displays a moderate relationship versus both TLBs.

B. *Resource contention*

In this section, we empirically show the relationship between the correlation coefficient and how our test applications' execution-time is affected by simultaneously executing artificial loads that utilize the shared cache. We define our hypothesis as follows.

Hypothesis The magnitude of the correlation coefficient indicates how closely tied an applications' performance is with a resource. Reducing this resource's size or capacity will affect the performance of applications with a high correlation towards this resource to a greater extent than applications with a low correlation towards this resource.

We chose to reduce the capacity of the L_2-cache through the execution of a memory-intensive program called leech [10]. The leech executes simultaneously as our our benchmark applications and is positioned on other non-occupied cores to enforce shared cache contention. The leech is built as a memory specific load and executes a read-then-write access pattern on an integer array of variable size. We run the leech using a specific stride pattern in the array to force as many cache line evictions from our benchmark applications as possible. To generate maximum L_2-cache contention, we run three separate instances of the leech on different cores (2,3,4) while the application runs on core 1. Table VII summarizes the leech specifics.

Fig. 6 depicts the execution characteristics of the Harris application running on core 1 in a leech contented environment. The red diamonds plot the L_2-cache misses on the right-hand side y-axis, and the blue crosses plot the instructions retired on the right-hand side y-axis.

TABLE VII
LEECH SPECIFICS

Property	Value
Iteration sleep	0
Array size	2 MB
Core affinity	Core 2,3,4
Stride Length	64 Byte
Access method	read-then-write

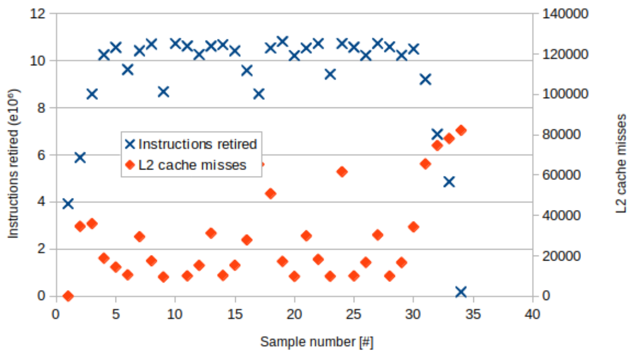

Fig. 6. The Harris application running together with leeches

There are two significant differences between the baseline Harris of Fig. 4 and the leech loaded Harris of Fig. 6. Firstly, the number of instructions retired in the leech loaded version is significantly less per 50 μs than in the baseline case. Since the number of instructions retired is considerably less, on average 23.7% less per measurement point, the performance becomes significantly worse. The graph also shows an apparent increase in the number of caches misses per time interval compared to the baseline Matrix multiplication with an average of 28% increased cache misses per measurement point.

For comparative purposes, we depict the execution characteristics for the non-L_2-cache-bound application FAST running in a leech setting in Fig 7.

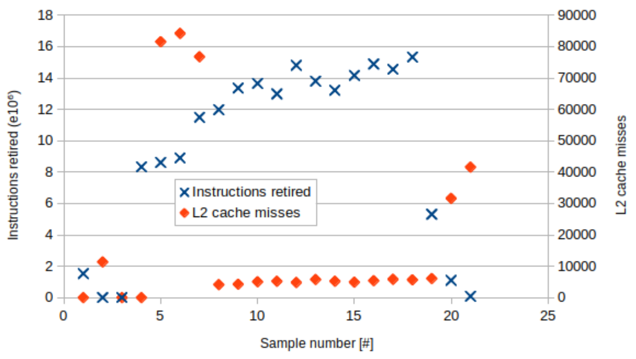

Fig. 7. FAST execution characteristics with leech

The baseline FAST version executes an average of 4.4% more instructions on average per 50 μs than the leech-loaded version. The difference in L_2-cache is also not overwhelming, (on average 1.1% more) in our leech version versus the baseline version. The small increase in execution-time, small decrease in instructions retired and small decrease in L_2-cache misses means FAST did not suffer notably from heavy L_2-cache contention. This goes in line with our assertion of the previous section that FAST, in fact, is not L_2-cache-bound. Harris, on the other hand, displays notably different behavior; the instructions retired are now jittery, especially in the middle sections of the execution. The L_2-cache misses are also more jittery and counts significantly more than the baseline version. Table VIII summarizes the results from our leech tests and shows the application execution-time from our leech loaded version (column 2) and the percentage difference in execution-time compared to the baseline execution (column 3). The table also shows the percentage difference in instructions retired per 50 μs (column 4) and L_2-cache misses (column 5) between the leech loaded version and the baseline execution of the applications.

TABLE VIII
SUMMARY OF PERFORMANCE LOSS DUE TO L_2-CACHE CONTENTION

Application	Ex. [ms]	Diff.[%]	Instr.[%]	L2.[%]
Harris	328	-7.41%	-5.76%	+64.14%
SUSAN	209	-7.3%	-0.75%	-16.62%
FAST	137.8	-3.02%	-4.4%	+1.1%
Matmult	381	-41.2%	-31.7%	+36.5%
SIFT	799.5	-6.6%	-10.6%	+6.3%
Sort	800.1	-0.38%	-0.93%	+7600%

The matrix multiplication suffered the worst execution-time losses due to resource contention, with a 42.1% increase in total execution-time, which is a drastic performance decrease. The FAST application suffered a 2.96% increase in execution-time, which is also in line with the low L_2-cache correlation listed in the previous subsection. Another standout measurement is the number of increased cache misses for the sort application of 7600%. This measurement is not an error but rather a natural consequence of the few misses in the baseline case (1 L_2-cache-miss on average) compared to the leech case (7600L_2-cache-misses on average). The increase is drastic, but the count is not enough to significantly reduce the performance.

The second worst is the Harris application (moderate correlation), which presents a 7.41% execution-time decrease. The third worst application is SUSAN (no correlation), which also presents an interesting case - the application displays no L_2-cache-boundness according to the correlation calculation. However, it still shows a notable performance decrease, while displaying a less L_2-cache misses average per 50 μs than the baseline case. Since SUSAN shows a performance degradation while simultaneously showing a decrease in L_2-cache misses, it cannot be L_2-cache-bound.

V. PARTITIONING EXPERIMENTS

In previous subsections, we show how we use the Pearson-correlation coefficient to determine the resource-boundness of an application and how L_2-cache-cache contention affects the application's performance. Here, we apply the knowledge of an applications' resource-boundness for assigning L_2-cache-partition sizes.

A. *Cache partitioning performance impacts*

In this section, we present experiments on the effects of executing our different applications within a cache partitioned environment. LLM-shark relies on the Palloc framework to implement page-coloring, which replaces the default *buddy* allocator algorithm in the Linux kernel. The page coloring algorithm utilizes the cache set-associative addressing bit for determining the memory location of new data. We list the L_2-cache specifications in detail in Table IX.

TABLE IX
L_2-CACHE SPECIFICATION OF ARM CORTEX-A53

Property	**Size**
Cache size	1 MB
Line length	64 Byte
Set-associativity	16
Set size	1024 Byte
Number of sets	1024
Replacement policy	Pseduo-random

The number of available cache partitions (colors) on a platform depends on the cache size, number of sets and the page size (4 KB), see Eq 2.

$$Nr.\ of\ Colors = \frac{Cache\_size}{Cache\_ways * page\_size} \quad (2)$$

Our platform provides 16 colors according to the formula and each color provides a 64 KB memory area. We showcase the effects of cache partitioning through a set of experiments where we measure our applications' execution-times under different color assignments, using 2-, 4-, 7-, 10-, 13-, and 16- assigned slices for the application. We assign one color for LLM-shark so that it can operate. We list the median execution-times of 100 measurements for each application using different cache partition sizes in Table X with the slices translated into the actual L_2-cache partition size.

TABLE X
APPLICATION EXECUTION-TIMES IN MILLISECONDS OF APPLICATIONS USING DIFFERENT L_2-CACHE PARTITION SIZES

	Partition size (KB)					
Application	128	256	448	640	832	1024
Harris	320	312	312	309	307	306
SIFT	780	778	777	775	774	771
Matmult	1214	1162	979	876	451	288
FAST	138	138	138	138	138	138
Sort	798	797	797	797	797	797
SUSAN	202	199	198	196	196	196

The table shows the difference in execution-time for each respective application, using different cache partitions where Matmult displays the most execution-time difference due to change in L_2-cache partition size. We further plot the number of instructions retired and the number of L_2-cache misses using different L_2-cache partition sizes in Fig. 8 and Fig. 9 respectively.

Fig. 8 plots the percentage difference in the number of instructions retired on the y-axis when scaling up the cache

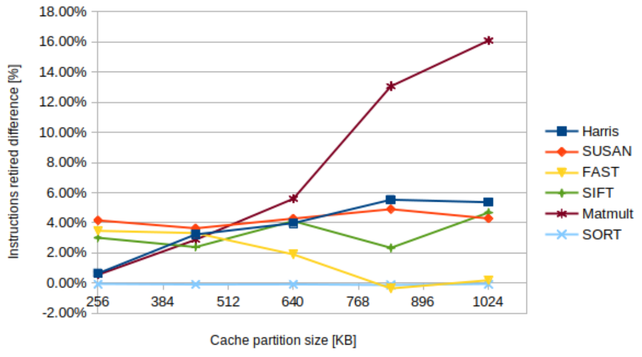

Fig. 8. Difference in L_2-cache misses using different L_2-cache partition sizes.

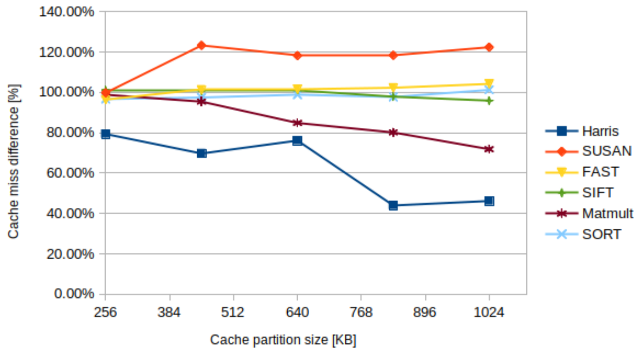

Fig. 9. Difference in instructions retired misses using different L_2-cache partition sizes.

partition size. A high percentage means more instructions retired per 50 milliseconds and is preferable to a low percentage difference. Fig. 9 plots the percentage change in the number of L_2-cache misses on the y-axis when increasing the cache partition size. The plots show an inverted scale, which means a positive percentage difference points to a decrease in L_2-cache misses compared to our reference measurement. Since Fig. 9 only plots the difference in L_2-cache misses, it is not possible to conclude that higher percentages are preferable to low percentages. Instead, the cache misses must be interpreted in an instructions retired context, where a decrease in the number of cache misses leads to an increased amount of instructions retired.

B. *Initial cache partitions*

Previously [4], we used a methodology called LLC-PC which tries to find the best Last-level cache partition size for an application. The methodology is an iterative process that continuously increases the Last-level cache partition for the application until a desired performance has been met. The method utilizes a run-time comparison scheme and measures an application's performance while increasing the cache partition size – if an increase in cache partition size positively affects the application's performance, LLC-PC continues increasing the partition size; if not, then then stop increasing the cache partition size. LLC-PC uses the smallest possible cache

TABLE XI

L_2-CACHE INITIAL CACHE PARTITION SUGGESTIONS COMPARISON

	LLM-shark			$C_{50\mu s}$			C_{tot}			Execution-times		
Application	**Corr.**	**Norm.%**	**Size**	**Median**	**%**	**Size**	**Number**	**%**	**Size**	**LLM**	$\mathbf{C_{50\mu s}}$	$\mathbf{C_{tot}}$
Harris	-0.49	30.97%	5	11274	15.36%	2	$8.7 * 10^5$	15.36%	2	309.2	313.9	313.9
Matmult	-0.84	52.69%	8	23871	33.95%	5	$2.2 * 10^6$	40.37%	6	897.5	1137.4	1075.3
SIFT	-0.26	16.34%	2	15401	21.9%	3	$1.3 * 10^6$	23.86%	3	780	779	779
SUSAN	**—Omitted—**		1*	2786	3.96%	1**	$4.5 * 10^5$	7.95%	1	212	210	205
FAST	**—Omitted—**		1*	16972	24.14%	4	$7.0 * 10^5$	12.37%	2	145.1	138.9	138.8
SORT	**—Omitted—**		1*	6	0.01%	1**	$4 * 10^3$ %	0.08%	1**	813.7	807.4	797.9

**Executing simulatenously on different cores using the same cache partition*

***Percentage not sufficient to justify a standalone partition, instead use a shared container*

partition size (which in our case is 64 KB) for starting point to avoid over-saturation. LLC-PC then increases the cache partition size by one (64 KB) for every iteration. Thus, it may take several iterations before reaching the desired performance, since the starting point is set at the smallest cache partition size possible.

Our optimization proposal is to have a “reasonable” starting point, i.e., an initial L_2-cache partition size from where to start scaling the partition sizes. Here, we illustrate the usage of the correlation coefficient to determine the size of an initial cache partition: applications with high correlations should receive more spacious partitions while low correlation applications should receive less space. In our methodology, we normalize all the correlation coefficient values with a magnitude $\geq$ to weak and partition assign partitions according to the percentage value of our 15 available colors. The initial correlation approach for determining resource-boundness was presented in [6], in here we apply that methodology to assign cache partitions. The normalized values gives a percentage and is used to calculate the cache partition sizes. Since the cache partition space only provides 15 colors, we also need to round-off decimal values to the closest integer values find an appropriate cache partition. E.g., an application that displays a normalization percentage of 30% ($15{*}0.3 = 4.5$) will result in a cache partition size of 4.5 – we round-off 4.5 to the closest integer value, 5, since it is not possible to use fraction numbers as a cache partition.

We discard applications with a lower than weak correlation since additional cache partitions since our assessment is that these application will receive little to no performance benefits from increased cache partition size. We instead execute these applications within a ”Junk” container - a partition with space of (64 KB) that holds all low correlation applications, and as such, we do not waste L_2-cache space on these applications. We argue our L_2-cache partition distribution methodology is preferable to other methodologies, such as assigning L_2-cache partitions based on L_2-cache misses or L_2-cache accesses [22] since our methodology includes the resource-boundness factor. We compare our correlation methodology versus two L_2-cache misses distribution policies ($C_{50\mu s}$ and C_{tot}). The table contains results using three methodologies, specified as follows:

1) LLM-shark (column 2-4) – distributes L_2-cache partitions based on correlation.
2) $C_{50\mu s}$ (column 5-7) – distributes L_2-cache partition size based on the median number of L_2-cache misses per $50\mu s$.
3) C_{tot} (column 8-10) – distributes L_2-cache partitions based on the total number of L_2-cache misses per given application.

The table shows the execution-time results and cache partition sizes of the three different methodologies, LLM-shark, $C_{50\mu s}$ and C_{tot}. The most significant difference is seen from the Matrix multiplication perspective, which receives the most spacious L_2-cache partition size, followed by Harris. SUSAN, SORT and FAST are all assigned to share “Junk” L_2-cache partition container. An application must utilize the locality of reference [20] principle to benefit from the L_2-cache and as such be L_2-cache-bound. Through code inspection of FAST [17], we conclude that FAST cannot be L_2-cache-bound due to a lack of locality utilization – even though the performance counters show a high count (relative to the other applications) of L_2-cache misses. Due to FAST’s high L_2-cache count, it receives L_2-cache using the cache miss-based policies, and can be seen as a waste of L_2-cache partition space since it’s performance is not affected notably. Due to FAST’s low correlation, it does not receive any individual L_2-cache partition space but is instead assigned to the Junk container. We summarize the comparison between the different methodologies in Table XII. The columns marks the best solution and specifies to performance degradation for each application compared to the best results. A higher value means an increase in execution time and is therefore a more significant performance degradation than a low value.

TABLE XII

COMPARISON SUMMARY

	Execution-time comparison (ms)		
Application	LLM-shark	$C_{50\mu s}$	C_{tot}
Harris	Best	+4.7	+4.7
Matmult	Best	+239.9	+177.8
SIFT	+1	Best	Best
SUSAN	+7	+5	Best
FAST	+6.2	Best	+0.1
SORT	+15.8	+9.8	Best

Our correlation-based methodology achieved the best execution-times for the matrix multiplication and the Harris applications. SORT displays the most significant downgrade using our approach, which is a fifteen milliseconds performance degradation, comparatively the cache-based distribution policies that display a 239.9ms ($C_{50\mu s}$) and (+177.8 C_{tot}) for our most cache heavy load, the matrix multiplication. The cache misses distribution policies instead focuses on

C. Discussion

The comparison shows that our correlation-based methodology assigns most cache partitions to the matrix multiplication than the cache misses distribution policies due to its high resource-boundness. The Harris application also receives most cache partitions using our correlation-based approach due to its resource-boundness, resulting in the best performance. SIFT suffers a 1ms performance degradation using LLM-shark. The junk cache partition displays a slight performance degradation for FAST, SUSAN, and SORT, indicating that cache contention occurs within this specific cache partition. However, the performance degradation of these three applications is slight compared to the performance gains of the matrix multiplication in LLM-shark compared to the cache-misses-based distribution policies. The matrix multiplication performance results display the main take-away point from this paper; it is not how frequently an application utilizes the cache that determines how the application responds to a change in cache partition size, but rather *how* an application use the cache. The matrix multiplication has a high data-reusage and tries to access the same cache memory several times during execution. If the cache space is reduced, the matrix multiplication cannot re-use data to the same extent since the cache is smaller and will suffer a significant performance penalty. Comparatively, FAST does not show cache-boundness due to low cache re-usage but still maintains a high cache-miss count. FAST fetches data from the main memory, leading to cache misses, but does not re-access the same data again; therefore, FAST is not cache-bound and does not benefit from increased cache partition space.

We argue that it is more beneficial to assign cache partitions based on their resource boundness rather than the cache-miss count since an increase in cache partition space provides more significant performance benefits to the highly cache-bound applications than non-cache bound applications.

VI. Related Work

Related work includes papers directed towards investigating the resource-boundness. Work such as Cache Pirating [7] and Bandwidth Bandit [8] are tools for generating shared resource contention and and can empirically pinpoint how much an application suffers from shared resource contention, which is similar to our leech methodology. Even though these works provide a structured methodology for pinpointing resource contention in one particular resource, the process can become time consuming since a bandit, a pirate or even a leech has to be designed specifically for each individual resource in order to generate and measure contention. Our opinion is that our correlation methodology can significantly decrease the complexity of such tests.

Other works such as Scarphase [18] divides program execution into phases and proposes a method for identifying how the resource usage changes over time in applications using the perf interface for measuring the performance counters. Sembrant et al. [19] expands on the same direction topic and explains the differences in phase behavior between serial and parallel applications.

The body of cache partitioning papers is relatively large [12]. Coloris [22] is an excellent example, with an approach that splits the cache partitions according to how many cache misses one process is responsible for. As we show in our paper, it is not necessary that the application with the most cache misses benefits the most from cache partitioning, instead we have to look at the significance of the cache misses and take it into relationship on how it affects the performance. Brock et al. [2] discusses how to optimally allocate cache partitions to different processes through exhaustive searches.

Perarnau et al. [16] argue that the cache partition size is best left to the user since it is the user that in the end knows what performance the application requires. The last level cache is, however, a very complex hardware resource due to the contention factor and also due to the limited size. This means the user needs expert knowledge on how the system applications utilizes the cache. Our solution eliminates the need for expert knowledge, through our partitioning scheme.

With LLM-shark, we cover all the aspects of performance improvements based on cache usage: i) we analyze the behavior of an application; ii) we assert potential problems such as resource contention; iii) we actuate mitigation strategies to avoid resource contention before it happens. While all the other similar approaches perform only one or two of these actions, we even provide an automated solution.

VII. Summary

We show that the LLM-shark tool can assign cache partitions automatically to applications based on their hardware resource-boundness. We have conducted a series of tests to assess the usability of the Pearson correlation coefficient as a tool for determining resource-boundness. We then verify that resource bound processes benefits to various extent depending on cache partitioning sizes and the level of hardware resource-boundness. Finally, we propose a normalization scheme to assign initial cache partitions to a system, based on the magnitude of the Pearson correlation coefficient. Our tool thus answers the following questions:

1) Which hardware components are limiting factors to application performance?
2) Which hardware resources are potential contention bottlenecks to the application?
3) How spacious should initial cache partitions be?

We focus only on the L_2-cache as a test subject. However, we argue that the correlation methodology is generalizable to all hardware events which imply a negative performance impact, such as TLB-, L_1D-cache-, L_1I-cache-misses and page-misses etc.

A. Future work

For future work, we plan to integrate our tool to different architectures that provide a more sizeable cache memory. Since our Last-Level Cache (the L_2-cache) is relatively small, the number of the available cache partitions is also small. More spacious cache memory will leave more room for flexibility in the cache partition assignments and, therefore, the execution-times of our L_2-cache-bound applications.

In the paper, we mention several mitigation techniques such as bandwidth reservation and TLB coloring which also can benefit from correlation based resource-boundness estimation. For future work, we intend to implement more mitigation techniques for LLM-shark, utilizing the same correlation-based scheme for determining parititon sizes. We furthermore plan to integrate LLM-shark with LLC-PC [4] to enable the dynamic adaptation of partition sizes once we have them assigned. Other future work includes investigating other metrics for formulating initial partition sizes. Our current strategy only looks at the magnitude of the correlation, which serves as a reasonable starting point. However, to provide an even more accurate partition estimation, it could be possible to model how much instructions retired can be gained from one single cache partition.

References

[1] ARM. Arm Cortex-A53 MPCore Processor Technical Reference Manual. URL https://developer.arm.com/documentation/ddi0500/j/.

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

[3] J. Danielsson, M. Jagemar, M. Behnam, M. Sjodin, and T. Seceleanu. Measurement-based evaluation of data-parallelism for opencv feature-detection algorithms. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, pages 701–710. IEEE, 2018.

[4] J. Danielsson, M. Jägemar, M. Behnam, T. Seceleanu, and M. Sjödin. Run-time cache-partition controller for multi-core systems. In *IECON 2019-45th Annual Conference of the IEEE Industrial Electronics Society*, volume 1, pages 4509–4515. IEEE, 2019.

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

[6] J. Danielsson, T. Seceleanu, M. Jägemar, M. Behnam, and M. Sjödin. Resource depedency analysis in multi-core systems. In *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 87–94. IEEE, 2020.

[7] 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.

[8] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten. Bandwidth bandit: Quantitative characterization of memory contention. In *Proceedings of the 2013 IEEE/ACM International*

[10] M. Jagemar, A. Ermedahl, S. Eldh, M. Behnam, and B. Lisper. Enforcing quality of service through hardware resource aware

Symposium on Code Generation and Optimization (CGO), pages 1–10. IEEE, 2013.

[9] T. Gleixner. Linux Performance Counter announcement, 2008. URL http://lkml.org/lkml/2008/12/4/401.

process scheduling. In *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, volume 1, pages 329–336. IEEE, 2018.

[11] R. Jain. *The Art Of Computer Systems Performance Analysis: Techniques For Experimental Measurement, Simulation, And Modeling*. john wiley & sons, 2008.

[12] 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 *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–14. IEEE, 2019.

[13] D. Mindrila and P. Balentyne. Scatterplots and Correlation. URL \url{https://www.westga.edu/academics/research/vrc/assets/docs/scatterplots_and_correlation_notes.pdf}.

[14] 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.

[15] S. A. Panchamukhi and F. Mueller. Providing task isolation via tlb coloring. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 3–13. IEEE, 2015.

[16] S. Perarnau, M. Tchiboukdjian, and G. Huard. Controlling cache utilization of hpc applications. In *Proceedings of the international conference on Supercomputing*, pages 295–304. ACM, 2011.

[17] E. Rosten and T. Drummond. Fusing points and lines for high performance tracking. In *Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1*, volume 2, pages 1508–1515. Ieee, 2005.

[18] A. Sembrant, D. Eklov, and E. Hagersten. Efficient software-based online phase classification. In *2011 IEEE International Symposium on Workload Characterization (IISWC)*, pages 104–115. IEEE, 2011.

[19] A. Sembrant, D. Black-Schaffer, and E. Hagersten. Phase behavior in serial and parallel applications. In *2012 IEEE International Symposium on Workload Characterization (IISWC)*, pages 47–58. IEEE, 2012.

[20] W. Stallings. *Computer organization and architecture: designing for performance*. Pearson Education India, 2003.

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

[22] 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), 2014 23rd International Conference on*, pages 381–392. IEEE, 2014.

[23] 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), 2013 IEEE 19th*, pages 55–64. IEEE, 2013.

[24] H. Yun, R. Mancuso, Z. 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), 2014 IEEE 20th*, pages 155–166. IEEE, 2014.

[25] G. Zellweger, D. Lin, and T. Roscoe. So many performance events , so little time. *APSys '16*, 2016. doi: 10.1145/2967360.2967375.