Testing Performance-Isolation in Multi-Core Systems

Jakob Danielsson[1], Tiberiu Seceleanu[1,3], Marcus Jägemar[1,2], Moris Behnam[1], Mikael Sjödin[1]
[1] Mälardalen University, Västerås, Sweden, [2] Ericsson AB, Stockholm, Sweden, [3] ABB AB, Västerås, Sweden
{jakob.danielsson, moris.behnam, marcus.jagemar, mikael.sjodin}@mdh.se, tiberiu.seceleanu@se.abb.com

***Abstract*—In this paper we present a methodology to be used for quantifying the level of performance isolation for a multi-core system. We have devised a test that can be applied to breaches of isolation in different computing resources that may be shared between different cores. We use this test to determine the level of isolation gained by using the Jailhouse hypervisor compared to a regular Linux system in terms of CPU isolation, cache isolation and memory bus isolation. Our measurements show that the Jailhouse hypervisor provides performance isolation of local computing resources such as CPU. We have also evaluated if any isolation could be gained for shared computing resources such as the system wide cache and the memory bus controller. Our tests show no measurable difference in partitioning between a regular Linux system and a Jailhouse partitioned system for shared resources. Using the Jailhouse hypervisor provides only a small noticeable overhead when executing multiple shared-resource intensive tasks on multiple cores, which implies that running Jailhouse in a memory saturated system will not be harmful. However, contention still exist in the memory bus and in the system-wide cache.**

I. Introduction

While great advancements in virtualization and partitioning techniques nowadays allow logical and functional partitioning of a system into a set of independently executing subsystems (referred to as partitions) [5], there exists no practical and efficient methods to guarantee that different partitions have no negative impact on each others performance. That is, contemporary techniques give logical isolation but not performance isolation. In this paper we propose a method for testing the performance isolation between different subsystems running on different cores in a multi-core architecture. Furthermore, the method tests isolation of different computing resources such as CPUs, caches and memory-system. Thus, it allows to pinpoint any sources of breached isolation and it enables mitigation of such breaches by introduction of specific isolation techniques for specific resources. With the introduction of multi-core architectures as the standard platforms for performance-critical application-domains like embedded systems and real-time systems, the issues of performance guarantees on these architectures becomes paramount. In multi-cores, isolation is hampered since a wealth of computing resources are shared between cores, such as caches, TLBs (Translation Lookaside Buffers), memory controllers and memory banks.

Our work is a step towards allowing empirical evaluation of performance isolation in complex multi-core architectures. We demonstrate the use of our model by evaluating performance isolation obtained by the Jailhouse hypervisor [1] and comparing it with running a non-partitioned Linux system.

Isolation is a complex topic and a clear terminology needs to be defined, for example: what is shared resource isolation?

The *performance isolation* is defined here by the slowdown in execution of an application while running in a context where access to resources is contended by other applications, too. An application that runs with a specific performance without any disturbing processes (in *isolation*) runs at a *baseline performance*. An application running with deliberately disturbing processes is running at a *loaded performance*. If the loaded version runs with the same performance as the baseline version, the application is performance isolated. Performance isolation of applications targeting specific hardware can be accomplished by using methods such as page coloring [11], hypervisors [8], bus-scheduling [19]. Many different techniques are available for isolating hardware from disturbances generated by other processes, but most techniques cover only one or two parts of the hardware resources. The resource partitioning hypervisor Jailhouse developed by Siemens can become one significant step towards achieving full isolation in multi-core systems. Due to its small code size, it is now much easier to understand the hypervisor and therefore implement new partitioning strategies into it.

The main contributions of this paper are:

- We present a methodology for measuring performance isolation of a system.
- A study on the performance isolation gained using the Jailhouse hypervisor.

Related work. We here identify previous studies that analyze shared resource contention caused by multiple cores, or address performance measurements on the Jailhouse hypervisor on ARM processors. Bansal et al. [4] investigated resource contention of the memory subsystem of the Xilinx ZCU 102 and proposes a Jailhouse based architecture to solve the contention. The authors effectively show a latency performance degradation of their benchmark when using multiple cores and propose mitigation techniques. In our work, we employ a different methodology, using the performance counting unit as a tool for identifying the sources of the performance degradation. Toumassian et al. [16] investigate the overhead of the Xen and Jailhouse hypervisors, where overhead is defined as Hypervisor performance/Linux performance. We complement this work, by deliberately adding the disturbing loads for estimating resource contention effects, while looking for application performance isolation. As listed by Deshane et al. [6], there exist a large body of reporting the impact of hypervisors on performance. However, since the Jailhouse hypervisor is relatively new, there is not so much research done on this subject. Up to our knowledge, there is no reporting of work investigating cache contention and memory bus contention in a Jailhouse environment, such effects being described as ”yet to be measured” in a Linux Journal article [1]. Furthermore, there is no reported work trying to verify what Jailhouse can accomplish in the area of task isolation, wherefore we research here the performance degradation on a Linux system caused by CPU sharing.

II. Background

Shared resource contention has become an increasingly important topic due to the phasing out of single-core systems and the adoption of multi-core systems. Important shared resources can be divided into three categories: CPU, memory, and I/O [17] which may all be subject to contention. The CPU sharing takes place in the scheduling level, where two or more processes share the execution capacity of the same CPU. If one process executes and a higher priority task interrupts, the swapped out process will not get to execute anymore, and may, therefore, expose an increased latency. The second level of resource contention occurs in the memory layer of a computer and can come in the form of *thrashing* - a state where much of the processing time is spent on handling cache misses or page faults due to several processes/threads continuously replacing each other. The third level of resource contention occurs in the I/O layer and can be illustrated very well by the ARM v8 case where a generic interrupt controller (GIC) handles all general purpose interrupts (such as general purpose I/O interrupts).

Partition-based virtualization is one of the solutions that addresses the sharing of resources across multiple processes [17], [7], [13]. Hypervisors such as Xen [5] and KVM [8] can effectively partition the cores of a system such that the resource is protected from usage of processes which do not belong to the specific partition. These hypervisors come with an overhead [9] and a significant code size. New virtualization techniques such as the Jailhouse hypervisor give promise of better task isolation through statically disallowing inter partition sharing of resources and also come with a relatively small code size.

A. Jailhouse hypervisor

The Jailhouse hypervisor (version 0.1 released in august 2014) partitions hardware resources through virtualization, and enables asymmetric multiprocessing on top of the Linux system [15]. It also enables the insertion of *cells* through a kernel module. A cell is a virtual machine that is created in a partitioned environment. Once created, the host operating system loses knowledge of the core where the cell is created. In a similar fashion, programs running within the Jailhouse cell do not know that they run within a virtual machine, nor have they any knowledge of cores outside of the cell.

Fig. 1 shows a regular Linux system - a) and a Jailhouse partitioned system which runs one Linux partition (core 0, 1, 2) and one real-time (RT) partition (core 3) - b).

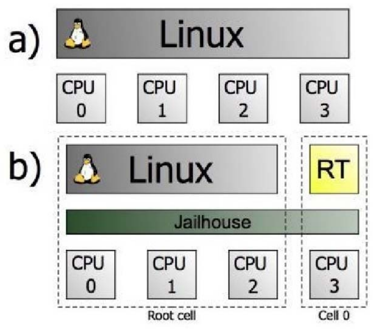

Fig. 1. a) Usual Linux deployment. b) Linux with Jailhouse configuration [1]

III. Shared resource contention

We describe the performance degradation of a process in Equation 1, where performance is equal to the execution time of an application.

$$I = \frac{P}{C} - 1 \quad (1)$$

We denote I as the *isolation coefficient*, representing the resulting slow-down of the execution of a task in the presence of other tasks. P denotes the loaded performance of an application, and C is the baseline performance. Both C and P values are measured in time units; moreover, it is expected that the P will always be higher than C, that is, the execution time of an application will always be longer in the presence of additional load as compared to the “ideal” case when the application executes alone on the computing platform. It is also important to note that the measured values of both C and P are platform dependent. Measurements are relying on processor specifics such as cache memory mechanisms, clock frequency and bus bandwidth, but also on the operating system. Therefore, C should not be seen as an absolute value of the best achievable performance (that is, cross-platform), but instead, the highest performance achievable using the respective setup. We refer to C as **baseline** in subsequent sections of the paper.

As an example, consider an application running on one core of a multi-core processor, exposing a baseline of 100ms. To perform tests on cache memory isolation, we apply a heavy cache intensive load, which runs on a different core than the application, and re-execute the application in these conditions. Both cores have a shared LLC. In case the loaded performance is observed to be 100ms, the isolation coefficient $I = 100\text{ms}/100\text{ms} - 1 = 0$. Hence, and the application is isolated from LLC disturbances. Alternatively, if the loaded performance is 110ms (for exemplification purposes), the isolation coefficient becomes $I = 110\text{ms}/100\text{ms} - 1 = 0.1 = 10\%$ which means that the application has suffered a 10ms performance penalty due to cache contention.

In the following subsections, we will discuss resource contention on shared resources, including CPU, cache, memory bus. We will discuss each shared resource in the context of a Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit using 4 Cortex A-53 cores, specified in Table I.

TABLE I
HARDWARE SPECIFICATIONS XILINX ZYNQ ULTRASCALE+ MPSOC

Feature	Hardware Component
Core	4xArm Cortex A-53 @ 1.5GHz
	2xArm Cortex-R5 @ 1.4GHz
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 platform cache
MMU	L_1ITLB: 10 entries
	L_1DTLB: 10 entries
	L_2TLB 512 entries, 4-way set assoc.

A. CPU utilization

Two applications sharing the same CPU can have dramatic effects on either applications response time. When sharing the CPU, one task may get to execute up to 50% of compared to the non-shared situation. Thus, the response time of the application could increase to at least the double of the baseline. We can avoid the CPU sharing effect by not scheduling other applications to the same core. However, if all cores are currently loaded, it is not possible to enforce such a policy, since the newly created application needs an execution environment. Consider our ARM system with 4 cores, running App_1..App_4 on core 0..3 respectively. In case a 5th application, App_5, enters the scheduling queue, there is no un-occupied core, which means App_5 has to share one of the cores with one of the other applications. This will increase the response time of both applications. This situation may not become a problem in real-time systems since tasks with high importance often are given a higher priority and will therefore not share execution time with other tasks during their respective time quanta. Thus, scheduling applications properly is usually a solution to this problem. Another solution can be static partitioning of the system, where the cores of one partitioned sub-system are hidden from another partitioned sub-system [12], disallowing partitions from using each other's designated cores.

B. Internal Memory Contention

The internal memory is often a source of execution time unpredictability - the so-called *jitter* - in multi-threaded systems [3]. Whenever the data requested by applications is not in the L_1D-cache or the L_2-cache, we need to fetch the data from the main memory. If the L_2-cache is already full, a cache-line is evicted from the cache to make space for the incoming data. Since the L_2-cache is shared between multiple cores, processes scheduled on different cores can evict the cache-lines of each other whenever the shared cache becomes full.

Within our ARM system, with a 1 MB L_2-cache, cache contention is exemplified as follows: App_1 and App_2 with a memory footprint of 1 MB each are executing on core 0 and 1 respectively. The applications are each using 1 MB of data, which, combined, is above the limit of L_2-cache - 1 MB. If the tasks are continuously running on different cores, App_1 will continuously try to write 1 MB of data into the shared cache. Since the cache is not large enough to contain the total amount of 2 MB data requested by both tasks, 1 MB of data will continuously have to be replaced according to the cache replacement policy. Cache coloring can be applied here, to restrict cache access of different applications to assigned cache lines only. Thus, one may mitigate problems such as performance losses [11], jitter [18], and even energy efficiency [10]. In our example though, this limits the amount of L_2-cache available to either of the applications.

C. Memory bus contention

The memory bus that interconnects the cache memory with the main memory is also a subject for contention. It is used for serving read and write requests from each core, which can become problematic when multiple memory intensive tasks are running on several cores. The bus can become a significant bottleneck concerning throughput, and a source of jitter.

Once again, consider the ARM system which has a measured bandwidth capacity of roughly 4.7 GB/s. The system hosts four applications (App_1, ..., App_4 running on core 1..4 respectively) which executes write operations at 2 GB/s individually. If the data is not present in the cache, it has to be fetched from the main memory via the memory bus. The bus, however, can only handle a certain amount of writes per second, as specified. Since we use multiple cores executing writes at 2GB/s, the bus bandwidth will be fully saturated. If any of the applications were the only one executing memory transactions, it could operate at the intended 2MB/s capacity. However, since multiple applications are executing, the bus has to distribute the capacity over the set of cores, which can dramatically decrease the individual memory throughput and increase the jitter of each application. It is possible to limit the effects of bus contention by restricting processes to execute under a certain memory bandwidth budget [19] [20] - with potential important overhead for each budgeted application.

IV. PERFORMANCE ISOLATION

We have used a matrix multiplication of various sizes as the application to benchmark the isolation that can be achieved using the Jailhouse hypervisor. The execution time of the application is measured by inserting wall-clock time-stamps at the start and at the end of the multiplication. Further, the matrix multiplication is co-executed with additional load programs denoted *leeches* to enforce shared resource contention. We use the previously defined Xilinx Zynq ZCU 102 platform (Table I) running a Petalinux 4.9 kernel and reserving 2 GB of RAM for the Jailhouse hypervisor using the *mem* kernel argument.

In the following subsections we show isolation measurements for the CPU, L_2-cache and memory bus resources with the matrix multiplication running in unfavourable (leech-disturbed) execution environments and compare them to the baseline executions.

A. CPU isolation test

We devised a test including a kernel module to serve as a CPU stealing leech and a matrix multiplication to show the contention problems in a CPU. We exemplify the problems using the following scenario, assuming equal application priority.

1) Applications P_0, P_1, P_2 and P_3 are ready to execute.
2) The applications are pinned as following $P_0 \rightarrow C_0$, $P_1 \rightarrow C_1$, $P_2 \rightarrow C_2$, $P_3 \rightarrow C_3$.
3) Kernel application KP_5 becomes ready to execute, all cores are currently occupied.
4) The kernel has to chose one available core for KP_5, in this case, C_3 is chosen.
5) P_4 and P_5 now share the same core and execute

To instantiate the above contention scenario, we co-run a 256x256 matrix multiplication as workload together, with a calculation-heavy program called a CPU leech, implemented as a kernel module. Kernel modules often are executed at seemingly random times and also at a higher priority than user-space modules. The CPU-stealing leech performs 100000 random number calculations, searches for the highest value read and then goes to sleep for a specified amount of time. This process takes between 79-80 milliseconds to execute. Since the time measurement of the matrix multiplication is dependent on context switches from another workload, we will call the time measurement *response time* in this test case. We statically set the core affinity of the matrix multiplication and the CPU leech to the same core C_3.

We also execute the same tests using the Jailhouse hypervisor, where the matrix multiplication is run within a Jailhouse Linux cell executing on C_3. The results of the CPU isolation tests are depicted in Fig. 2 where the y-axis shows the response time of the matrix multiplication run under Linux (blue dash) compared to a matrix multiplication run within a Jailhouse Linux cell (orange dash). Each data point is the median response time of 50 executions. The y-axis is a logarithmic scale of the response time measured in milliseconds, and the x-axis shows the sleep timer of the kernel module - the period between executions. A low value on the Y-axis - meaning a low response time - would be better than a high value. The calculated isolation coefficient of the matrix multiplication is listed in Table II.

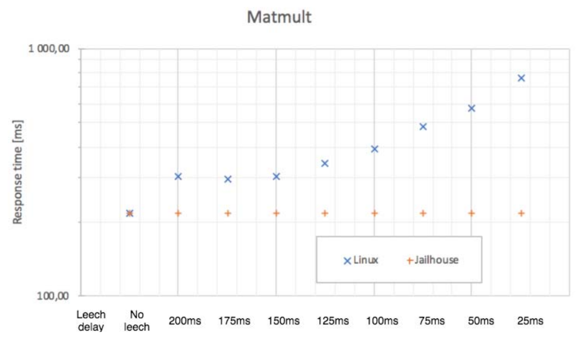

Fig. 2. CPU isolation test

Fig. 2 shows a Linux matrix multiplication which suffers heavily from the CPU stealing caused by the leech, even at the relatively large sleep periods of 200 ms. In these conditions, according to Table II and using Equation 1, Linux alone offers an isolation coefficient of 0.40, which is an indicator of significant resource contention. The CPU leech will always

TABLE II
I COEFFICIENT IN CPU CONTENTION TEST (PERCENTAGE)

Sleep	I_{Linux}	I_{Jhouse}	**Sleep**	I_{Linux}	I_{Jhouse}
200	41,22%	0,62%	100	81,99%	0,79%
175	38,25%	0,15%	75	124,00%	0,50%
150	40,76%	0,57%	50	166,47%	0,86%
125	59,88%	0,57%	25	250,79%	-0.15%

get a high priority when ready to execute, running with kernel priority. Hence, when the associated sleep period goes under a certain value, the isolation coefficient even surpasses 0.50. When running the matrix multiplication within a jailhouse partition, however, the response time is almost constant, with an isolation coefficient of 0.0086, which is in the range of an error margin.

Concluding, the Jailhouse hypervisor performs as promised regarding the CPU isolation, while the Linux system shows a significant downgrade in the performance of the matrix multiplication, as expected, too.

B. L_2-cache isolation test

Here, we intend to provide a measurement of the isolation coefficient for the matrix multiplication, verifying to what extent it suffers of L_2-cache cache contention.

We use a 512x512 matrix multiplication for benchmarking workload, and a tweaked version of a maximum bandwidth benchmark called Tinymembench [14] as a leech, for loading the L_2-cache. The Tinymembench load continuously reads 32-bit integers from a N-sized buffer and writes them into another N-sized buffer. The isolation test was conducted as follows.

1) Run baseline execution of the matrix multiplication
2) Initialize cache load process with size N (initially 64 KB)
3) Assign cache load process to C_0
4) Start matrix multiplication on C_3
5) Re-iterate from step 1 and multiply size N by 2

The results of the matrix multiplication running within a regular Linux environment are depicted in Fig. 3, and the results of running it within a Jailhouse Linux cell are shown in Fig. 4. The graphs point the execution time (blue dash) on the left-hand side y-axis and the L_2-cache misses (orange dash) on the right-hand side y-axis. The x-axis marks the leech buffer size. The graphs also include error bars where the upper dash shows the maximum value, and the lower dash shows the minimum value of 50 measurements. As previously, low values are better than high values of the execution times. Also, a large error bar is worse than a small one, since small variability in both L_2-cache misses and execution time is preferable. Table III lists the calculated isolation for the matrix multiplication when co-run with the Tinymembench load.

We observe a typical "knee" effect, i.e., the performance degradation of the matrix multiplication halts at a certain point. This halt occurs when the matrix multiplication co-run with a L_2-cache leech cannot produce more cache misses, as every cache line request will be a miss. This comes to a full effect when N is 1 MB, which is aligned with the 1 MB-sized

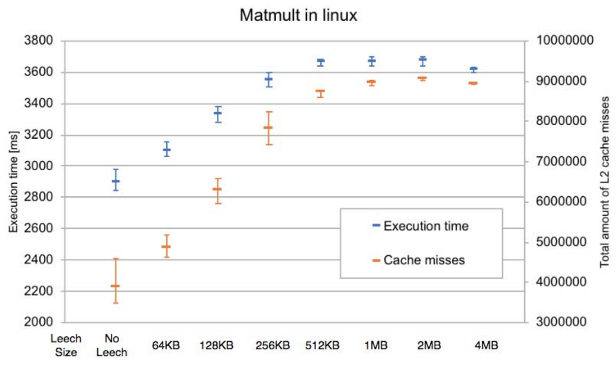

Fig. 3. Linux L_2-cache isolation test

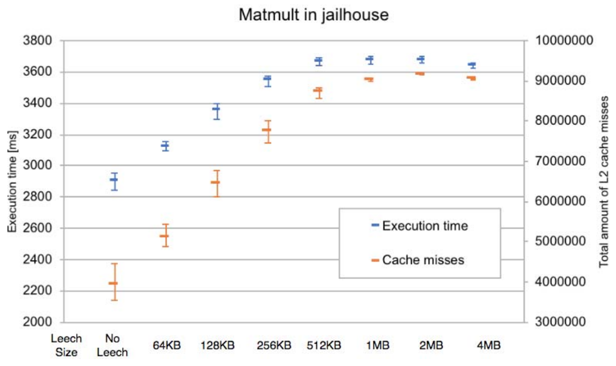

Fig. 4. Jailhouse L_2-cache isolation test

L_2-cache. From the isolation coefficient values- Table III, we see almost no difference between the Jailhouse measurement and the Linux measurement. This is motivated by the fact that the Jailhouse hypervisor (in the reported version) does not mitigate this problem. Also, there is almost no difference in execution time, nor cache misses. This suggests that it is potentially is possible to migrate tasks from regular Linux system to a Jailhouse partition without having to re-calculate the execution characteristics of the algorithm.

C. Memory bus isolation test

In this section, we describe memory bus contention which occurs due to multiple processes on different cores requesting non-cached memory. In the previous test, we discovered the knee effect occurring at a buffer size of 1 MB, which means all data requested by a process will be a cache miss and it has to be fetched from the main memory through the bus. If multiple processes from different cores request data from the main memory, the bus has to arbitrarily chose which process gets the access. This may lead to further performance degradation. To investigate memory bus contention, we run a test as follows, where we employ the same kind of leech as previously, with a buffer size of 8 MB (or any size larger than the 1 MB limit described above).

1) Start a 512x512 matrix multiplication on C_3
2) Insert one memory bus leech on a non-occupied core

TABLE III
I COEFFICIENT IN L_2-CACHE CONTENTION TEST (PERCENTAGE)

Size	I_{Linux}	I_{Jhouse}
128 KB	7,17%	7,74%
256 KB	15,27%	15,84%
512 KB	22,78%	22,33%
1 MB	26,62%	26,69%
2 MB	26,92%	26,87%
4 MB	25,14%	25,51%

3) Repeat step 3 until all cores are occupied

To ensure that full cache contention occurs during the entire execution of the test, we measure the L_2-cache misses of the system. Their number should remain constant - any change reflecting the fact that there were also some cache-hits, which is to be avoided.

Fig. 5 depicts the results of the regular Linux matrix multiplication execution, and Fig. 6 depicts the results of the execution under Jailhouse protection. The left-hand side y-axis plots the calculated median execution time of 50 measurements, the x-axis shows the number of leeches inserted into the system and the right-hand side y-axis shows the L_2-cache misses of the system. The graphs also include error bars where the upper dash shows the maximum value and the lower dash shows the minimum value of the 50 measurements. We list the calculated isolation coefficient for the matrix multiplication using regular Linux and Jailhouse in Table IV.

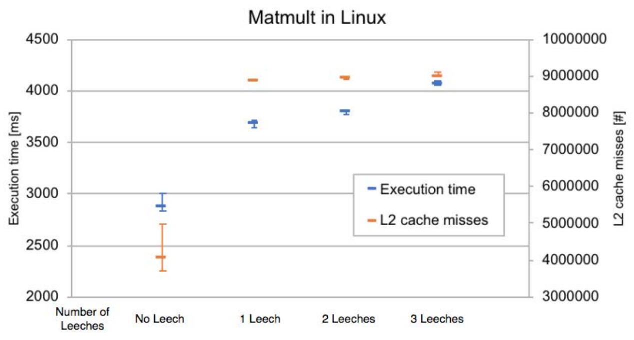

Fig. 5. Linux memory bus isolation test

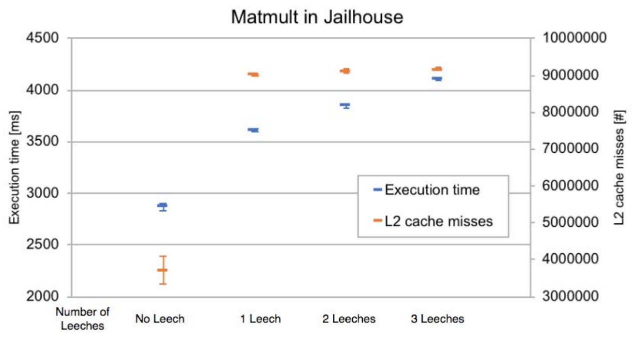

Fig. 6. Jailhouse memory bus isolation test

The graphs above show a significant performance degradation of the matrix multiplication due to memory bus contention running in Linux as well as in Jailhouse. The baseline execution time remains the same as in the matrix L_2-cache isolation case, since we used the same matrix size. Furthermore, the observed effects when using one leech are also similar to the L_2-cache isolation test, as the cache is fully loaded. However, the interesting effects on execution times occur when inserting two or more leeches. Firstly, we can read an isolation coefficient of 0,3168 and 0,326 for the Linux and Jailhouse matrix multiplications, respectively. The values mean that the Jailhouse hypervisor does not provide any sorts of bus isolation, as expected. In addition, the execution time of the matrix multiplication will be increased with any added leech. Once again, the performance impact of using the Jailhouse hypervisor is within a measurement error margin, suggesting that using the Jailhouse hypervisor does not come with any overhead penalties.

TABLE IV
I COEFFICIENT IN MEMORY BUS CONTENTION TEST, (PERCENTAGE)

Size	I_{Linux}	I_{Jhouse}
1 Leech	28,91%	25,96%
2 Leeches	31,75%	34,12%
3 Leeches	41,30%	43,50%

V. CONCLUSION

We have measured the effects of contention on computing resources such as CPUs, L_2-cache and memory bus. As an example of an application with high need for both CPU and memory, we used a matrix multiplication. We executed the application in a standard Linux context and compared it with the execution in a Jailhouse hypervisor cell context. In order to test the isolation, we disturbed the application by executing leeches designed to consume particular computing resources.

Our measurements focusing on the CPU resource show that the Jailhouse hypervisor provides isolation between different partitions, enabling the application to exhibit a performance very close to the baseline even in the presence of leeches. Jailhouse does not, however, provide any memory bus or L_2-cache isolation. These said, there is a very small difference in performance degradation for the application execution between the Jailhouse hypervisor and a standard Linux system during heavy shared resource congestion. This further suggests that using Jailhouse in a heavily loaded shared resource environment provides an at least as performant execution context as Linux.

We leave investigating TLB, DRAM bank and I/O contentions for future work. There also exists a newly published patch [2] for Jailhouse which provides a cache coloring configuration for Jailhouse cells. Investigating the page coloring mechanisms using our methodology is also relevant future work in the Jailhouse case.

REFERENCES

[1] *https://www.linuxjournal.com/content/jailhouse*, 2015.

[2] *https://groups.google.com/forum/#!topic/jailhouse-dev/rSSE8Yyjmbo*, accessed 2019-05-16.

[3] Federal Aviation Administration. Addressing cache in airborne systems and equipment. 2003.

[4] A. Bansal, R. Tabish, G. Gracioli, R. Mancuso, R. Pellizzoni, and M. Caccamo. Evaluating the memory subsystem of a configurable heterogeneous mpsoc. *OSPERT 2018*, page 55.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *ACM SIGOPS operating systems review*, volume 37, pages 164–177. ACM, 2003.

[6] T. Deshane, Z. Shepherd, J. Matthews, M. Ben-Yehuda, A. Shah, and B. Rao. *Xen Summit, Boston, MA, USA*, pages 1–2, 2008.

[7] S Han and H Jin. Full virtualization based arinc 653 partitioning. In *Digital Avionics Systems Conference (DASC), 2011 IEEE/AIAA 30th*, pages 7E1–1. IEEE, 2011.

[8] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230. Ottawa, Ontario, Canada, 2007.

[9] J. Li, Q. Wang, D. Jayasinghe, J. Park, T. Zhu, and C. Pu. Performance overhead among three hypervisors: An experimental study using hadoop benchmarks. In *Big Data (BigData Congress), 2013 IEEE International Congress on*, pages 9–16. IEEE, 2013.

[10] S. Mittal, Z. Zhang, and Y. Cao. Cashier: A cache energy saving technique for qos systems. In *26th International Conference on VLSI Design and 12th International Conference on Embedded Systems*, pages 43–48. IEEE, 2013.

[11] 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.

[12] R. Ramsauer, J. Kiszka, D. Lohmann, and W. Mauerer. Look mum, no vm exits!(almost). *arXiv:1705.06932*, 2017.

[13] S. Santos, J. Rufino, T. Schoofs, C. Tatibana, and J. Windsor. A portable arinc 653 standard interface. In *27th Digital Avionics Systems Conference.*, pages 1–E. IEEE/AIAA, 2008.

[14] S. Siamashka. https://github.com/ssvb/tinymembench. *Retrieved January*, 2019.

[15] V. Sinitsyn. Understanding the jailhouse hypervisor, part 1. *https://lwn.net/Articles/578295/*, 2014.

[16] S. Toumassian, R. Werner, and A. Sikora. Performance measurements for hypervisors on embedded arm processors. In *Advances in Computing, Communications and Informatics (ICACCI), International Conference on*, pages 851–855. IEEE, 2016.

[17] S. H. VanderLeest. Arinc 653 hypervisor. In *Digital Avionics Systems Conference (DASC), 29th*, pages 5–E. IEEE/AIAA, 2010.

[18] 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.

[19] 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 *19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64. IEEE, 2013.

[20] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory bandwidth management for efficient performance isolation in multi-core platforms. *IEEE Transactions on Computers*, 65 (2):562–576, 2016.