Heterogeneous multicore SDRAM interference analysis

Alfonso Mascareñas González
alfonso.mascarenas-gonzalez@isae-supaero.fr
ISAE-SUPAERO, Université de Toulouse
France

Frédéric Boniol
frederic.boniol@onera.fr
ONERA, Université de Toulouse
France

Youcef Bouchebaba
youcef.bouchebaba@onera.fr
ONERA, Université de Toulouse
France

Jean-Loup Bussenot
jean-loup.bussenot@onera.fr
ONERA, Université de Toulouse
France

Jean-Baptiste Chaudron
jean-baptiste.chaudron@isae-supaero.fr
ISAE-SUPAERO, Université de Toulouse
France

ABSTRACT

The purpose of this paper is to describe a set of DDR3 SDRAM interference estimation cost functions. The arbitration system of the SDRAM controller heavily impact the interference analysis. In this work, three arbitration are considered, corresponding to the situations where the accessed memory address belongs to the same block address, different memory banks and different rows. The aim of these functions is to estimate the instructions interference overhead may suffer when concurrently accessing these three logical addresses in a SDRAM saturation context. To develop these interference expressions, specific measurement systems, micro-benchmarks and theory on SDRAM controllers have been used.

CCS CONCEPTS

• **Computer systems organization → Real-time system architecture**; **Real-time system specification**; **Real-time system architecture**.

KEYWORDS

Multicore, DDR3 SDRAM Controller, WCET, Cost functions

ACM Reference Format:
Alfonso Mascareñas González, Frédéric Boniol, Youcef Bouchebaba, Jean-Loup Bussenot, and Jean-Baptiste Chaudron. 2021. Heterogeneous multicore SDRAM interference analysis. In *29th International Conference on Real-Time Networks and Systems (RTNS'2021), April 7–9, 2021, NANTES, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3453417.3453426

1 INTRODUCTION

The pursuit of better computing performance has been a priority since the first processors. At the beginning, it was mainly focused on the increase of the clock frequency of these. Afterwards, when the processor clock speed was sufficiently high and limited by the barrier of temperature, a new way to increase performance was studied. Instead of using single core processors, i.e. monocore platforms, new architectures were designed to create processors with multiple cores, i.e. multicore platform. As a consequence of parallel computing, overall processing improved. This approach has been used since the start of the century in general purpose systems. Nevertheless, this has not been the same for safety critical real time systems. Stacked cores in the same system share resources leading to the rise of new challenges [21]. Inevitably interference emerge among the different cores that make up the processor. These have to be reduced as much as possible so that critical systems can perform correctly in the required period of time. Many different approaches have appeared in order to tackle this issue [16], where we can point out task scheduling [13], cache partitioning [27], timing analysis [15] or task and memory mapping [6]. These different approaches are generally synergistic and can be implemented together. This paper focuses in the latter approach, task-memory mapping on multicore platforms, which has proven to be not trivial [4]. The way tasks are mapped can decide the overall behavior of the application to run, e.g. low energy consumption over performance, increment parallelism at expense of inter-core communication cost, maximize the performance, etc... The platform heterogeneity level add extra complexity when mapping tasks. A homogeneous platform, i.e. a platform with processors of the same type and symmetric architecture, are less complex when it comes to behavior modeling. In addition, it may be considered other challenges such as the level of criticality of tasks. In a mix-criticality context, interference coming from non-critical tasks must be considered in order to avoid exceeding the imposed execution time limits of the critical ones [9]. Failing to execute a task within the imposed time may lead to a system failure. Therefore, the Worst Case Execution Time (WCET) must be reduced as much as possible to ensure these timing conditions. Strategically placing tasks on cores to minimize the interference among each other should be a priority. As it is done for other optimizations in the task mapping context [4, 8], e.g. load, memory or communication variance, a cost function dedicated to the DDR3 SDRAM interference should also be defined. To do so, in this work, three groups of expressions are designed to describe the behavior of

RTNS'2021, April 7–9, 2021, NANTES, France
https://doi.org/10.1145/3453417.3453426

the SDRAM controller when accessing addresses of the same memory block, belonging to different banks and to different rows. These expressions intend to explain the resulting arbitration for each situation in a WCET context, where priority, commands bursting, blocking and starvation take place. For elaborating these expressions (Section 7), self-made micro-benchmarks are used for testing the behavior of the cores, the DDR controller and the instructions involved (Section 6). Besides, three measurement frameworks are used for capturing the data which are needed for the expressions themselves but also for checking their validity (Sections 5 and 7 respectively). Finally, the available information about the SDRAM controller arbitration system is also used. An extensive SDRAM overview can be found in [22], e.g. address mapping, reordering mechanisms. For more specific details related to the controller used in experimentation, the datasheets and manuals from the manufacturer are used [25]. The heterogeneous multicore platform used in this work is the Texas Instruments (TI) SOC Keystone II [26].

2 RELATED WORK

Multicore interference analysis. Multicore interference analysis is an extensive topic which has been the center of the attention since real-time applications intended to be implemented on these platforms. Interference may occur in many different places of the platform as a result of resource sharing, e.g. shared cache, main memory. Therefore, interference identification [3] and quantification are key aspects. To achieve the latter, simple and complex tests are used. Micro-benchmarks [18, 20] tend to focus on a specific feature, e.g. how loads or stores affect a certain memory. Benchmarks have a wider view, stressing multiple features at the same time. Report [5] evaluates how interference impact tasks execution time on Linux and RTEMS on ESA's Next Generation MicroProcessor: it describes (1) a set of micro-benchmarks, e.g. pointer chasing for load execution time measurement in different memory hierarchy levels, and (2) benchmarks, e.g. EEMBC AutoBench. Another important aspect is how measurements are retrieved from the platform. This can be done via hardware performance counters or relying on high level simulation platforms. Work [7] introduces a new set of micro-benchmarks to explore the different memory hierarchy levels and proposes new metrics for quantifying the qualitative aspects of memory behavior. To overcome the limitation of their hardware performance counters it implements a profiling tool using the VALGRIND framework (Simulation Framework). A measurement may often include other undesired factors, specially when using an operating system. Paper [12] describes how to create a controlled environment by (1) isolating hardware and operating system elements that affect the reliability (frequency throttling, thread migration, etc), and (2) applying statistics to remove outliers.

SDRAM devices analysis. Paper [1] proposes a DDR2 memory controller design that is able to upper-bound the latency. For that purpose, memory access groups are defined and the arbiter called Credit-Controlled Static-Priority is used. The former consists of read, write and refreshing groups whose efficiency is computed. The latter is an arbiter made up of a rate regulator and a static-priority scheduler. Similarly, paper [19] describes a DDR2 SDRAM memory controller for obtaining predictable access times. It offers a more general solution with respect to [1] as it can be applied to any kind of JEDEC [14] compliant DDRx SDRAM devices. As well, it does not require to set the tasks priorities or knowing the bandwidth requirements. Besides, their equations are designed to support mix-criticality. [10] offers great descriptions of the SDRAM features and memory controllers functioning and analyzes the SDRAM controller architectures thought for real-time applications. The proposed memory controllers are analyzed, e.g. [19], through a benchmark suite (EEMBC) and a simulation engine. WCET and latency studies are made as function of the benchmarks, number of requestors or data bus width. Paper [11] introduces a new perspective: It states that DDR DRAM are not suitable for hard real-time systems because of (1) highly variable access latencies due to various factors, e.g. access patterns, and (2) overly pessimistic latency bounds. Therefore, their paper focuses on the analysis of the Reduced Latency DRAM (RLDRAM) type of DRAM which seems (according do them) much more suitable for real-time predictability. The analysis is based on a set of mathematical descriptions and formulas as well as some measurements made in the simulation engine MacSim [1].

Contribution. This work offers a hybrid approach combining theoretical and experimental aspects to tackle DDR3 SDRAM interference based on the design of cost functions on a Commercial On The Shelf (COTS) platform. These can estimate the worst case impact that requests from a specific core suffer from others when accessing the DDR3. The functions focus on the SDRAM device functioning, the memory controller scheduling and the heterogeneity of the multicore platform. Instead of proposing new controllers to improve SDRAM predictability like in [1], [19] or those in [10], this paper analyzes the behavior of the platform memory controller and describes it as a set of mathematical cost functions which are validated through measurements. To do so, performance counters and time stamps are used like in [17] in contrast to [7] and [12] where simulators are also employed. Moreover, the cost modeling includes the heterogeneity effects of the platform, considering the processors execution time difference and the core packaging asymmetry.

3 PLATFORM

3.1 Multicore platform architecture

The heterogeneous multicore platform used in this paper is the Keystone II model TCI6636K2H [26] integrated in the EValuation Module (EVM) TCIEVMK2H. The main features of this Keystone II version (see Figure 1) are the following: (I) Four ARM Cortex A15 (architecture version 7), with an out-of-order execution pipeline, which together make up the ARM pack [2]. We can find an individual Level 1 instruction cache (L1P) and Level 1 data cache (L1D) of 32KB size each. The Level 2 cache (L2) of 4MB size is shared among the four cores. (II) Eight Texas Instruments C66x DSPs [23], made up of two data paths each. The L1D and L1P have also a 32KB capacity each. However, the 1MB L2 cache are dedicated for each core, i.e. not shared. (III) A Multicore Shared Memory Controller (MSMC) to where the 12 cores are connected [24]. The ARM pack, i.e. the four ARM cores, are connected to a single slave port while the DSPs have their own. A 6MB Multicore Shared Memory (MSM) SRAM can be found inside the controller. (IV) A 2GB DDR3 memory.

[1]https://github.com/gthparch/macsim

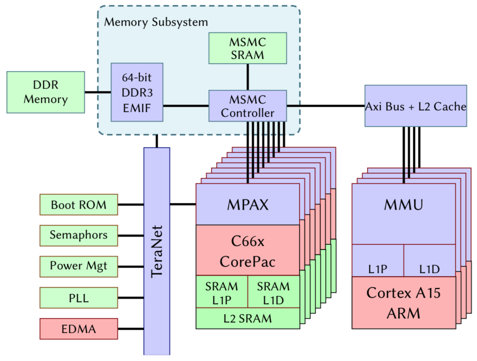


Figure 1: Keystone II TCI6636K2H platform overview

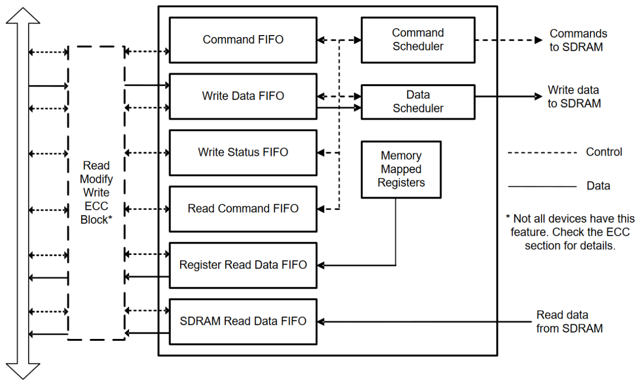


Figure 2: DDR3 Memory Controller Block Diagram [25]

3.2 External Memory Controller Interface

The TI DDR3 memory controller [25] is used on the Keystone II to interface with SDRAM devices. These devices are based on the JEDEC standard JESD79-3E [14], which describes the main specifications that define a DDR3 memory. A vast number of features are explained, being key aspects the functional description and commands operation. The DDR3 memory can be seen as a set of states where the commands are in charge of moving from one to another. In this work, the most important commands are the *Active* (ACT), which opens a row for a given bank, *Precharge* (PRE), which deactivates an opened row for a particular bank, *Read* (RD) for loading data and *Write* (WR) for storing data. The state transition time vary depending on the current state and the applied command. Two important transition timings are those from the RD/WR turnaround and the PRE/ACT sequence. It is crucial to note that the RD and WR operations, which are burst oriented, need to wait for an amount of time given by CAS and CWL respectively (see Table 1). This is the elapsed time since the command execution and the data availability. The data arrives as a burst of length 8 starting from the targeted address. The JEDEC specification also defines the addressing. The SDRAM is a 3-dimensional array made up of a set of banks which are in turn composed of columns and rows. Each bank has a buffer attached to it, where the last accessed row for that bank is stored and manipulated. The targeted physical address bits define to which column, row and bank it is being accessed, e.g. bits 3-11 of address 0x80014048 define the column.

The TI controller supports different features configuration, e.g. banks, page size, timings. The most important parameters for the interference cost functions detailed in Section 7 are summarized in Table 1. Figure 2 depicts the controller as a set of FIFOs with specific sizes each of them, e.g. Command FIFO with a depth of 16, Write Data FIFO with 20, Read Command FIFO with 28 or the SDRAM Read Data FIFO with 28. The Write Data FIFO stores the data to store into the SDRAM memory. The Read Command FIFO stores the read transactions to be issued to the respective requestors. The SDRAM Read Data FIFO contains the values loaded from the memory to be delivered. The Command FIFO stores the commands from the requestors to be issued to the SDRAM. The FIFO elements are reordered by a command scheduler to optimize the system performance according to some rules. Among these it can be pointed out (1) the reads prioritization over the writes as the former tends to stall the processors, (2) the execution of separated size-defined blocks of write and read commands to reduce the turnaround penalty frequency and (3) the execution of already opened rows commands before opening new ones to reduce the PRE/ACT commands execution frequency. Although not explicitly said by TI, this last rule is similar to an open-page policy. Nevertheless, the put into practice of the three previous optimizations are conditioned by the physical addresses being aimed. The location where transactions are pointing to can leave us with three arbitration systems that may permit the application or not of the rules mentioned before. The resultant arbitrations, inferred via theory and measurements, are:

- **Fixed block address**: Incoming transactions operate within the same memory block address. The arbitration works like a classical First In First Out queue (first come, first served), neglecting any command priority associated to read/write requests. This forces the command scheduler to block any read command (load instruction) in favor of any older write command (store instruction). Therefore, any read command, which normally has a higher priority (see priorities in [25]), waits until the older write commands of the Command FIFO have been executed. Thus, this arbitration doesn't use any of the optimizations recently explained.
- **Bank switching**: Each arriving transaction aims to a different SDRAM bank, allowing to interleave among them without penalties. The arbitration works similarly to a preemptive priority queue, where lower priority commands can be interrupted by higher ones. Therefore, a read command has priority over the writes (rule 1) unless a starvation mechanism is triggered, e.g. priority elevation. Furthermore, command bursts are applied (rule 2). During bursts, a non-preemptive priority queue policy is applied instead.
- **Row switching**: Each transaction destination points to a different SDRAM row within the same bank. The arbitration system works accordingly to optimization rule 3 unless a priority elevation from a command in another row is produced. In this case, the row switch is forced. Therefore, the arbitration schedules the Command FIFO in line with a non-preemptive priority policy.

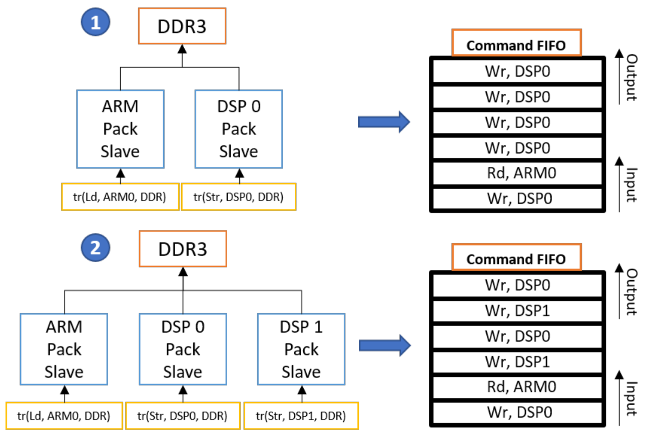


Figure 3: Command FIFO enqueuing behavior

Table 1: Typical Controller Features - 800 MHz Controller

Feature	Value
Number of ranks	1 rank
Number of banks	8 banks
Page size	10 bits
SDRAM width	64 bits
WR to RD	5 cycles
RD to WR	7 cycles
Column Address Strobe (CAS) latency	11 cycles
CAS Write Latency (CWL)	8 cycles
PRE to ACT	11 cycles
ACT to RD/WR command	11 cycles
RD to PRE delay	5 cycles

The way the Command FIFO entries are enqueued is another key factor of the controller behavior. It was seen through experimentation that MSMC core pack slave requests were equally added (fair queuing) to the FIFO. However, it was also observed that the read/write commands ratio remained untouched when having a fixed number of load requestors and a variable number of store requestors. This suggests that either the read enqueuing has priority or read/write entries addition to the Command FIFO alternates, i.e. after a write, a read follows and vice versa. Figure 3 depicts the experimented behavior. Situation 1 shows two MSMC slaves (ARM pack plus one DSP) interfacing with the DDR3 while Situation 2 considers three slaves (ARM pack plus two DSPs). Internally, the FIFO changes but the total reads/writes proportion doesn't.

3.3 Considerations

The measurements, expressions and results presented in this paper have been obtained under some specific criteria:

- **No caches**: All the caches except for the instruction ones, i.e. L2 and L1D, have been disabled. The aim is to ease the study of the DDR3 behavior by avoiding data caching or cache maintenance.
- **Bare-metal**: The work has been carried out without a real time operating system in order to have more control of the data traffic among cores as well as the registers configuration.
- **Compiler optimization**: The compilers optimizations have been disabled or reduced to CPU register level to assure that the micro-benchmarks execute as expected (Section 6).
- **Single thread**: Each core executes a single thread.
- **Arbitration**: Priority and starvation counters are left as default, where all cores are equally treated.
- **Burst and Saturation**: A continuous flow of transactions able to saturate DDR3 FIFOs is assumed.
- **Instructions**: Only load and store instructions of 32 bits size are involved. Instructions are not mixed for the same core, i.e. either stores or loads are executed but not both.

4 NOTATIONS

Along the paper there are some used terms according to the working context that should be explained. The processors instructions considered are the Load (Ld) and the Store (Str). Let us note:

$$Instruction = \{store, load\}$$

Our interference analysis focuses on transactions addressing the memory. The sources, i.e. the physical entities that start the transactions, are restricted to the platform cores, i.e. ARMs and DSPs:

$$Source = \{ARM_0, \ldots, ARM_3, DSP_0, \ldots, DSP_7\}$$

The destination of the transactions is the memory region aimed by the source. Although there are several memories in the Keystone II, only the DDR3 SDRAM is used. Inside this memory, we distinguish among banks and rows.

$$Destinations = \{DDR3\_bank_i\_row_j \mid (i = 0..7, j = 0..7)\}$$

A transaction is defined as the execution of an instruction, from a source to a destination. Let us note tr a transaction defined by:

$$tr = < inst, src, dest >$$

where $inst \in Instructions$, $src \in Sources$ and $dest \in Destinations$. In the following $tr.inst$ will denote the instruction of tr, $tr.src$ the source of tr, and $tr.dest$ the destination of tr.

In this article, we consider that the set of all transactions is given. Let us note:

$$Transactions = \{tr_0, \ldots, tr_N\}$$

In the next sections, the transaction from who the data is analyzed is called transaction under consideration or under analysis which will often be identified with an apostrophe tr'.

The execution time of a burst of N transactions is what the measurement frameworks records (see Figure 4). The time it takes for a single transaction to be completed is derived from the previous recording. We also consider that the execution time in isolation (i.e. without interference) of each transaction is given by the function:

$$\Delta(tr) = \text{Execution time of } tr \text{ in isolation}$$

5 MEASUREMENT FRAMEWORK

5.1 ARM - Performance Monitor Unit

The measurement framework consists in using the dedicated performance monitor hardware of the ARM, more specifically the performance counters. In this ARM version, there are a cycle execution specific counter and 6 general counters where we can set the event to study from a list (see [2]). In order to obtain the data, a Start-Stop pattern is used. The main advantage of this pattern with respect to the Start-Read (see its set up in [17]) is that the performance counters can be run in parallel for the same execution (7 events for this ARM version) without introducing extra overhead. Therefore, data correlations can be established for the same execution and behavior analysis is eased. This pattern consists in: (1) Selection of the counter, (2) Selection of the event to keep track of, (3) Reset counters value, (4) Enable counters, (5) Execute tasks, (6) Disable counters, (7) Read counters and (8) Check overflows. Steps 1 and 2 are done for the 6 general counters once. Steps 3 to 8 can be looped as many times as performance data it is wanted to take from the task.

5.2 DSP - Time Stamp Counter

The DSPs have a time stamp register [23] made up of two 32-bit registers (TSCL and TSCH), which can be used for studying the execution time of a task. To start the counter a first write to TSCL is done. To obtain the correct instant of time, first the TSCL value must be retrieved, triggering the copy of the counter upper-half to TSCH. In this way, the 32 MSB can be safely read without the risk of being incremented. Afterwards, a 32-bit shift has to be done to the TSCH. The implementation is shown in Algorithm 1.

Algorithm 1 DSP measurement framework

```
TSCL = 0; // Initiate counter
(t1_L, t1_H) = atomic_read(TSCL, TSCH)
task_execution(); // Execute task
(t2_L, t2_H) = atomic_read(TSCL, TSCH)
result = ((t1_H<<32) + t1_L) - ((t2_H<<32) + t2_L);
```

5.3 DDR3 Memory Controller - Performance Counters

The TI Keystone II DDR3 controller [25] has three performance counters that are indispensable for understanding the behavior of data management inside the controller. One of these counters (PERF_CNT_TIM) is fixed as the controller time counter. The other two (PERF_CNT_1 and PERF_CNT_2) accept an event chosen from a list, e.g. number of RDs, WRs or commands priority elevations. Some of these events can be studied from a system point of view or filtered by master (MSTID) or memory regions (REGION_SEL). In this work, these performance counters are used for analyzing the Command FIFO enqueing, saturation, arbitrations (see Subsection 3.2) and priority elevations due to starvation.

6 MICRO-BENCHMARKS

Several micro-benchmarks have been designed in order to analyze and measure different data about the interference impact on the DDR3 and its behavior. The load and the store micro-benchmarks are the most important for the development of the interference cost functions. Both micro-benchmarks are based on the N sequential execution of the aimed instruction, i.e. store or load. The N value is varied so that we can see the execution cycles evolve. By doing so, we achieve two things: (1) to prorate the measurement framework and the CPUs registers initialization overhead until they are considered negligible and (2) to obtain the execution times of the targeted instructions for a core type when accessing the DDR3. To ensure that the micro-benchmark is working as expected, i.e. only stores or loads are being executed and nothing more, this must be designed in a special way like described in [5]. Instead of using loops to easily manage the executed number of instructions, these must be written N times one after another. Otherwise, the instructions belonging to the loop would be executed once per instruction.

In the case of the load micro-benchmark, we have to previously define the address from where we are loading. This one may vary depending on the location of the DDR3. For our Keystone II, this one ranges from the 0x80000000 to 0xFFFFFFFF. As this value is stored in the CPU register, we must ensure that it is not overwritten with other values. The store instruction requires, apart from the destination address, a CPU register containing the value to store. The ARM micro-benchmark for the store instruction implementation is shown in Algorithm 2.

Algorithm 2 Store micro-benchmark for ARM

```
// Save value to store in register
__asm__ (" mov r2, %0" : "=r" (value));
// Store target address in register
__asm__ (" mov r1, %0" : "=r" (address));
// Store value to target N times
__asm__ __volatile(" str r2, [r1]");
...
__asm__ __volatile(" str r2, [r1]");
// Data Synchronization Barrier
__asm__ __volatile("dsb");
```

Optimally speaking, the load and store instructions code should be written in assembly to avoid any inadequate translation by the compiler. If not, the compiler optimization level has to be properly set so that instructions reordering doesn't take place. Anyway, the volatile modifier must be added to assure that the code executes as is. Before running the micro-benchmark, it should be considered to open or close the DDR3 row it is aimed at.

Figure 4 depicts the average execution cycles of stores and loads as function of N for the DSP and ARM. The measurements have been taken using the corresponding frameworks described in Section 5. At the beginning both execution cycles drop (not visible for the loads) due to the overhead prorate. The maximum value of both instructions tends to decrease as N gets bigger. To check the status of the SDRAM controller, e.g. command FIFO saturation, the DDR3 controller measurement framework was used. Some results are shown in Figure 5 as cycles per instructions. The time the FIFO is full when executing stores decreases as function of N tending to zero, meaning that in isolation achieving the FIFO saturation is very rare. Note that the Command FIFO is never full when loads are taking place because at processor pipeline level a limited number of loads can be concurrently executed. The pending commands per

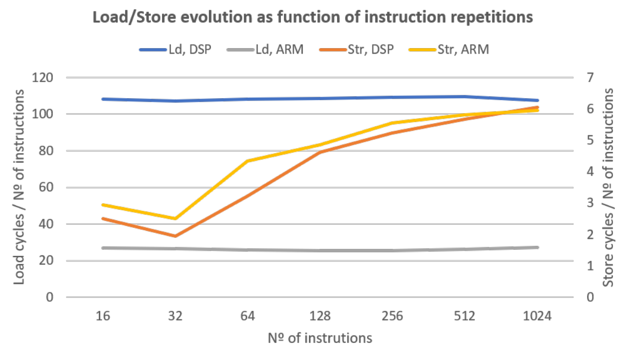


Figure 4: DSP/ARM average execution cycles per instruction as function of N. DDR3 address accessed: 0x88012000

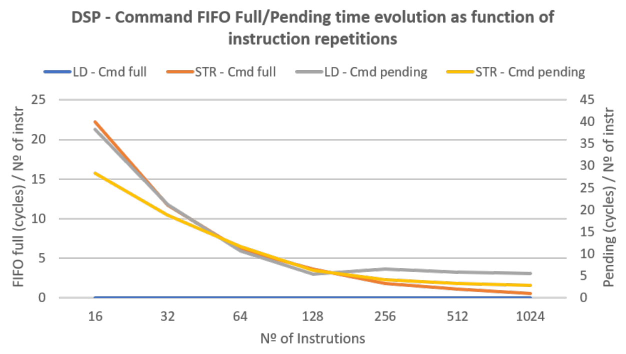


Figure 5: DSP SDRAM command analysis as function of N. DDR3 address accessed: 0x88012000

instruction, i.e. the time a command has waited before being served, also decreases but stabilizes at the end.

The results provided by Figure 4 define the value of $\Delta(tr)$ in isolation, which is used by the interference cost functions. The metric for the value selection depends on the user. In a concurrency context, if it is assumed that all the instructions involved do always their worst time, then the value for $\Delta(tr')$ and $\Delta(tr)$ should be the maximum, else the average. The most pessimistic case would be to use the maximum for $\Delta(tr')$ and the average for $\Delta(tr)$ (the minimum is considered as unusual). Furthermore, for choosing the transaction value it is required to know the total number of instructions N that are running. The interference cost equations validation is done with a *tr'* running a micro-benchmark with 128 instructions, i.e. N = 128, and the interfering *tr* running an endless micro-benchmark, i.e. N ≥ 1024. Table 2 shows the considered transaction timing values. The execution time values chosen are for N = 128 and estimations for $N \longrightarrow \infty$. The latter is done by taking more measurements with a higher N (up to N = 4096) and then looking at the figures tendency. The first row data is only used in isolation or low interference conditions. The latter situation is found when dealing only with loads as their larger execution time makes difficult to have several RD commands in the Command FIFO and hence produce interference, e.g. tr(LD, DSP, DDR) || tr(LD, DSP, DDR). The second row is used when concurrent transactions always interfere with each other, e.g. tr(STR, DSP, DDR) || tr(STR, DSP, DDR) or tr(STR, DSP, DDR) and tr(LD, DSP, DDR). In this last

Table 2: WCET of transaction (Δ(tr)) for 2000 measurements

Max	Ld, DSP	Str, DSP	Ld, ARM	Str, ARM
N = 128	122.1	13.1	35.7	17.27
$N \longrightarrow \infty$	110.57	8.21	27.70	8.36

situation, the $\Delta(tr')$ behaves like $N \longrightarrow \infty$, where the max-average difference is minimal. All this is checked using the DDR3 controller performance counters.

7 INTERFERENCE IMPACT

7.1 Impact calculation

The context where the interference impact calculation takes place is where the DDR3 controller is saturated by transactions from different sources. The same type of interfering transactions is assumed to try to access the controller continuously one after another. In order to determine the most affected transaction for a given scenario we need to analyze how each of them is affected by the others. This implies analyzing the *tr'* according to its composition i.e. its instruction (Ld/Str), source (ARM/DSP) and destination (DDR3), and the data arbitration rules for the given scenario. To carry out this task, we rely on the design of mathematical expressions that can describe the overall functioning of the DDR3 controller and more specifically its Command FIFO. This FIFO behaves as a deterministic queue (D/D/1) and its behavior is ruled by the scheduling policy applied by the arbitration (see Subsection 3.2). The arrival rate depends on the time of transaction (Table 2) and the queue saturation. The service rate is conditioned by the previously executed command, the actual command and the effective data and command burst size. Independently of the controller arbitration, there are other aspects that affect the cost function design. The next are considered:

- **Turnaround Time**: Total time that it takes from switching from a read command to a write command (RTWC) and vice versa (WTRC). The delay for a single switch is given by the expressions *T_RTW + 1* and *T_WTR + 1 + CAS* respectively (provided by TI [25])[2]. The designed expression for denoting the Turnaround Time Cost (TTC) is Equation 1.

$$TTC = \begin{cases} RTWC + WTRC & \text{if N \& M} > 0. \\ 0 & \text{otherwise}. \end{cases} \tag{1}$$

  Where *N* and *M* are the number of transactions with loads and stores working concurrently (*tr'* included).

- **Platform Heterogeneity**: In heterogeneous platforms, it is dealt with different type of processors. Their internal architecture differ and so their performance. Therefore, it is necessary to handle the possible DDR3 access time difference of these (ARMs and DSPs), especially for load instructions (store execution times are similar). The Average Heterogeneity Rate (AHR) is implemented to sort out this execution difference. AHR is shown in Equation 2.

$$AHR(tr, I) = min\left(\frac{\Delta(tr)}{\underset{\substack{tr_i.instr = I \\ tr_i \neq tr}}{Avg}(\Delta(tr_i))}, 1\right) \tag{2}$$

[2]TI nomenclature for it is T_WTR + 1 + CL, being CL the CAS latency.

Furthermore, the platform asymmetry must be considered. The ARMs behave as a single pack due to its unified L2 controller and pack slave in the MSMC unlike the DSPs. This causes transactions coming from ARMs to suffer from extra interference when there are other ARMs transactions. This also affects the Command FIFO enqueuing, where the DSPs can request concurrently while ARMs jsut can one at a time.

- **Number of Commands in Line**: The Generalized Number of Commands in Line (GNCL) is a function that returns the number of interfering transactions (aggressors) that the Command FIFO takes for a given transaction used as reference. This gives information about the number of interfering commands executing per *tr* command. This quantity is computed with Equation 3.

$$GNCL(tr, I) = min\left(\sum_{\substack{tr_i.instr=I \\ tr_i \neq tr}} \frac{\Delta(tr)}{\Delta(tr_i)},\ CFS\right) \tag{3}$$

The parameter *tr* denotes a transaction (tr_i or tr') and I the instruction type used by tr_i. Therefore, if I is a store, the interfering transactions involved are those with stores and vice versa. If I is a store and a load then both are considered. The maximum interfering transactions that a given transaction waits for before it is served is limited by the Command FIFO Size (CFS), i.e. 16 transactions. Hence, if the GNCL value is higher it should be cast to the FIFO size. The effect that the common ARM slave port has on Equation 3 vary. In this work, the impact is computed through experimentation, e.g. for a $tr(Ld, ARM, DDR3)$ interfered by $tr_i(Str, ARM, DDR3)$ the value of $\Delta(tr)$ is updated to $\Delta(tr) + \Delta(tr_i)$. Its effects should be extensively analyzed in a future work.
The GNCL function alone is not sufficient to explain the enqueuing behavior of the Command FIFO. The Scheduled GNCL (SGNCL), shown in Equation 4, describes this conduct according to the explanations in Subsection 3.2. The first line returns the number of identical commands than the selected *tr* executing before it. The second line does the same but with commands of different type.

$$\begin{aligned} SGNCL(tr, I1, I2) = min[GNCL(tr, I1) + \\ (GNCL(tr, I1) + 1) * AHR(tr, I2), CFS] \end{aligned} \tag{4}$$

Where tr is the reference transaction. *I1* and *I2* are two parameters that indicate the instruction types to use, i.e. store or load. *I1* contains the same instruction type as *tr* while $I2 \neq I1$, e.g. SGNCL(tr', Ld, Str) being tr'.instr equal to Ld.

- **Column Address Strobe latency**: It is the delay time after the execution of the RD/WR command and the moment when data is actually available. The RD and WR latencies are set in the CAS and CWL fields respectively (see Table 1). Keystone II register: SDRAM Configuration Register (SDCFG).
- **SDRAM Timing Registers**: These values configure the DDR3 controller to match the parameters of the DDR3 device and, therefore, setting the transitions cost, e.g. WR to RD delay (T_WTR), RD to WR delay (T_RTW) or PRE to ACT. Some of these values are found in Table 1 or fully in [25]. Keystone II registers: SDTIM1, SDTIM2, SDTIM3, SDTIM4.

Subsections 7.1.1, 7.1.2 and 7.1.3 describe the interfering cost functions, whose quantity goes as function of the number of arbitrations (fixed block address, bank switching and row switching) and commands (RD and WR), making a total of 6 functions.

7.1.1 *Fixed block address.*

The key point of this situation is the arbitration, which implies no read/write reordering inside the Command FIFO, and hence prioritizing the executing of the older instructions first, i.e. it behaves as a classical First In First Out. Therefore, waiting for the execution of other commands turns to be a compulsory source of interference independently of the command type.

Along with Equation 4, two other equations are defined. R1 and R2 (Equations 5 and 6) describe the percentage of transactions with stores and loads that may be inside the Command FIFO when the transaction *tr'* arrives. Although transactions with loads takes much longer than the stores and hence its presence in the FIFO may be outnumbered by stores, its interference impact weight is fairly represented by applying these two equations.

$$R1(tr, I1, I2) = \frac{SGNCL(tr, I1, I2) - GNCL(tr, I1)}{SGNCL(tr, I1, I2)} \tag{5}$$

$$R2(tr, I1, I2) = 1 - R1(tr, I1, I2) = \frac{GNCL(tr, I1)}{SGNCL(tr, I1, I2))} \tag{6}$$

The expressions that explain the behavior of those transactions that point to the same destination address are Equations 7 and 8. The two first lines of both equations describe the time cost due to the arbitration and the third represents the turnaround cost. The latter is always present when the *tr'* operation is a load owing to the faster memory access speed of the store (check Figure 4). The turnaround cost produced among the interfering transactions themselves and which may indirectly affect *tr'* is considered in $min(n, m)$ or n, followed by the turnaround impact. n and m represent the number of interfering transactions (aggressors) involved in the interference scenario with loads and stores instructions respectively. AHR returns the average instruction periodicity of the interfering transactions. Note that $C(tr')$ is the cost function.

If the *tr'* has a load as instruction:

$$\begin{aligned} C(tr') = & R1(tr', Ld, Str) * SGNCL(tr', Ld, Str) * CWL + \\ & R2(tr', Ld, Str) * SGNCL(tr', Ld, Str) * CAS + \\ & TTC * (min(1, m) + min(n, m) * AHR(tr', Ld)) \end{aligned} \tag{7}$$

If the *tr'* has a store as instruction:

$$\begin{aligned} C(tr') = & R2(tr', Str, Ld) * SGNCL(tr', Str, Ld) * CWL + \\ & R1(tr', Str, Ld) * SGNCL(tr', Str, Ld) * CAS + \\ & n * AHR(tr', Ld) * TTC \end{aligned} \tag{8}$$

7.1.2 *Bank switching.*

When transactions point to different banks, the arbitration is based on the default priority levels of the commands, where the read commands go first unless a burst is ongoing. The command burst size of the store and load play an important role in the arbitration. It may happen that the read commands gets blocked or forced to constantly switch to the write commands. This occurs if the burst size value of the former is very low compared to the size of the latter while having a considerable read command traffic. The burst size is set in the Read/Write Execution Threshold (RWET) register. The default value used by this controller is 5 for the reads and 3 for the writes, which avoids the described problem.

Since bursts take place, the *tr'* may stack and execute one after another (limited by RWET). The stack of load commands are quite volatile, depending on previous executions due to is relatively lower execution frequency, i.e. the execution of other commands make time for *tr'* to stack. This is contemplated by the Load Command Burst Size (LCBS) function. LCBS returns the number of load commands that might have been stacked inside the Command FIFO.

$$LCBS(tr) = max\left(1, \frac{TTC + WRTH * CWL + GNCL(tr, Ld) * CAS}{\Delta(tr)}\right)$$

Where WRTH is the threshold for write command bursts. The bursts also happen for the interfering transactions, provoking the commands to stack in groups of a size given by RWET. Equation Write Command Burst Cost (WCBC) and Read Command Burst Cost (RCBC) return the cost produced from write and read command stacking. The worst case for *tr'* would be that in its arrival a burst has just started and thus suffering a temporal block. From the point of view of a read command, the arbitration would temporally behave as in the Fixed Block Address situation, forcing to wait for the WR CAS latency (CWL). The number of commands to wait for is determined by the threshold value WRTH, i.e. the command write burst size. If the interfering burst involves read commands, owing to their larger execution periodicity (see Figure 4), jointly with their higher priority and the default burst size, it would be rare to stack the same transaction more than once. In consequence, it is assumed that the cost is the RD CAS latency itself by the times it enters the Command FIFO with respect to *tr'*.

$$WCBC = \begin{cases} WRTH * CWL & \text{if m > 0} \\ 0 & \text{otherwise} \end{cases}$$

$$RCBC\,(tr) = \begin{cases} GNCL(tr, Ld) * CAS & \text{if n > 0} \\ 0 & \text{otherwise} \end{cases}$$

Where *n* and *m* are the number of interfering transactions with load and store instructions respectively. Expressions 9 and 10 describe the behavior of the interference impact in a bank switching context. It is assumed that each *tr*, belongs to a different bank. For Equation 9, the first line represents the reordering cost due to a higher priority. The second line computes the cost due to command bursts per *tr'*. The LCBS is used for computing from the point of view of a single *tr'* rather from the point of view of *tr'* bursts. The load burst is already assumed to be made of a single command. The last line belongs to the turnaround cost. In Equation 10, the first line represents the cost coming from the execution of write commands. The second line considers the cost from the execution of load commands, command reordering and turnaround.

If the *tr'* instruction is a load:

$$\begin{aligned} C(tr') &= Reordering + \\ &LCBS(tr')^{-1} * WCBC + RCBC(tr') + \\ &TTC * (LCBS(tr')^{-1} + h) \end{aligned} \tag{9}$$

If the *tr'* instruction is a store:

$$\begin{aligned} C(tr') &= m * CWL + \\ &n * AHR(tr', Ld) * (CAS + Reordering + TTC) \end{aligned} \tag{10}$$

Where *Reordering* is the cost coming from the rescheduling forced by a higher priority command when entering the FIFO, e.g. read command. This value is estimated via measurements. *h* is a boolean whose value is one when tr'(Ld, DSP, DDR3) and zero when tr'(Ld, ARM, DDR3). This differentiation is due to the command stacking capacity of each core type resulting from the $\Delta(tr)$ difference. It is noticed that the controller behaves as if a read command from an ARM executes together with one from a DSP (if any) but not vice versa. This is appreciated when using the DDR3 controller performance counters.

7.1.3 *Row switching.*

The memory controller avoids switching between rows for performance purposes. This arbitration can lead to temporal transactions starvation from other rows which is avoided by the priority raise counter. The controller manages this counter in the PR_OLD_COUNT field of the VBUSM_CONFIG register. The number of cycles to wait until the priority is raised is given by the formula: Priority Elevation Cycles (PEC) = PR_OLD_COUNT * 16 clocks, resulting in 4096 in our case.

Each time there is a row switch, there are some associated delays (RD/WR to PRE, PRE to ACT and ACT to RD/WR). These are computed by the equation Row Switch Cost (RSC).

$$RSC = RDWR2PRE + PRE2ACT + ACT2RDWR$$

To compute the execution cycles of all the commands for a specific row, the actual commands in the Command FIFO and those that may enter during the execution these are considered. The equation that returns this value is the Row Transaction Exhausting Ratio (RTER), where the transaction whose length is to analyze is passed as an argument. This can be any *tr* from the Transactions set.

$$RTER(tr) = max\left(\sum_{n=1}^{N^{th}} \frac{CFS}{(GNCL(tr, Str, Ld) + 1)^n}, 1\right)$$

The higher the value used for N^{th} the higher the accuracy. In this work a value of 3 has been used.
The CAS and CWL latency from the interfering commands must be included as these impact $\Delta(tr')$. Thus, the RTER Cost (RTERC) is defined. This equation assumes the worst case and hence that the *tr'* is always facing the fastest interfering transaction from the set.

$$\begin{aligned} &RTERC = Lat * min(RTER(tr_{min}), LOOC) \\ &where\; tr_{min} = tr_i \; s.t. \; \Delta(tr_i) = min_{\,tr_j \neq tr'} \; (\Delta(tr_j)) \end{aligned}$$

LOOC stands for Load Out of Order Capacity, which indicates the maximum number of loads the processor can execute without stalling. The value of the latency *Lat* equals CWL (8 cycles) if tr_{min} belongs to a store or CAS (11 cycles) if it belongs to a load. When there is a single store interfering, it may happen that transactions with loads get starved until a priority elevation is triggered. If this happens, it is considered that $RTERC = 2 * PEC/LOOC$.

It is assumed a scenario where each transaction points to a different row of the same bank. Equations 11 and 12 explain the arbitration that rules row switching from a load and store perspective. Equation 11 first line describes the cost of waiting for the fastest transaction for a given row to be depleted and the row switching cost from interfering transactions and *tr'* itself. $RTER(tr')^{-1}$ transforms the row cost to a single *tr'* cost, i.e. cost per *tr'*. The second line returns the row switching cost caused by interfering loads, i.e. the row switch cost and operation latency. Equation 12 shows the write command perspective. The first line computes the cost of rows with write command bursts, i.e. the row switching cost per *tr'* and operation latency. The second line computes the row switching and operation delay caused by read commands, which are produced once per write transaction.

If the *tr'* instruction is a load:

$$\begin{aligned} C(tr') &= RSC + RTER(tr')^{-1} * (RTERC + m * RSC) + \\ &RTER(tr')^{-1} * n * (RSC + CAS) \end{aligned} \tag{11}$$

If the *tr'* instruction is a store:

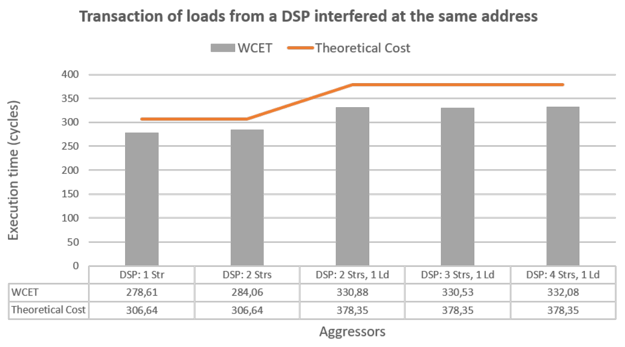
$$C(tr') = m * (RTER(tr')^{-1} * RSC + CWL) + \\ n * m * RTER(tr')^{-1} * (RSC + AHR(tr(Ld), Ld) * CAS) \tag{12}$$

Being n and m the number of interfering transactions with load and store instructions respectively.

7.2 Testing

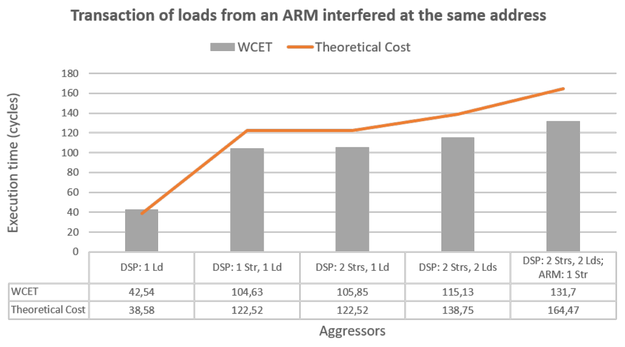
The validity of Expressions 7 to 12 are checked with the measurements obtained by applying in parallel the micro-benchmarks explained in Section 6. In order to have a direct comparison, the different interference cost function estimations must be multiplied by a frequency conversion factor as the SDRAM controller and the ARMs/DSPs clock frequency differs one from each other. The result is then added to the execution time of the instruction belonging to *tr'* in isolation conditions, i.e. the theoretical interference impact is added to the measured *tr'* in isolation. Hence, the test expression remains as follows: $Interfered\Delta(tr') = \Delta(tr') + frequency\,conversion * C(tr')$, where the conversion is 1.5 for the actual configuration. The validation tests consist of three sets of experiments where the impact to $\Delta(tr')$ is analyzed as function of the number of aggressors, i.e. the activation of interfering transactions. The experiments test the interference impact on transactions of type *tr'(LD, DSP, DDR)*, *tr'(LD, ARM, DDR)* and *tr'(STR, DSP, DDR)*. These scenarios are carried out for each of the three SDRAM memory situations so that the resultant behaviors can be directly compared. Subsequently, it is evaluated how well the interference cost functions perform.

7.2.1 Fixed block address. Figure 6 shows how the execution time per instruction (Y axis) of a transaction of loads from a DSP is affected when involving other transaction also coming from DSPs (X axis). In this specific scenario, the platform would behave as if it were homogeneous. The figure depicts how the interference cost, shown in orange, upper bounds the the worst case of the measured data in grey. This means that Equation 7 works for this scenario despite that it overestimates the results. The reason may be the pessimistic assumption for the *tr'* turnaround. It is assumed that the RD to WR transition latency affects the next *tr'* RD command which may not be necessarily true for the actual DSP configuration. In Figure 7, the loads from an ARM are analyzed. It can be seen that the increments in time are done correctly although the last experiment (DSP: 2 Strs, 2 Lds; ARM: 1 Str) is a bit more pessimistic than the other experiments. This fact, which should be checked as a future improvement, is maybe due to a not fully accurate interpretation of the platform heterogeneity. Besides, the first experiment (DSP: 1 Ld) theoretical value is optimistic. The interference impact caused by the DSP is well modeled with respect to the isolated scenario (increment of 2.88 cycles) but the experiment isolated value $\Delta(tr')$ and the one used for the theoretical output (see Table 2) data differ in 5.32 cycles. This difference might come from the higher number of measurements taken for these experiments. Figure 8 shows the case where the *tr'* is a DSP executing a store. The theoretical cost manages to upper bound the measurements WCET, being extra pessimistic in the increment from experiments 4 to 5 (DSP: 2 Lds; ARM: 3 Strs) and 5 to 6 (DSP: 3 Lds; ARM: 3 Strs).



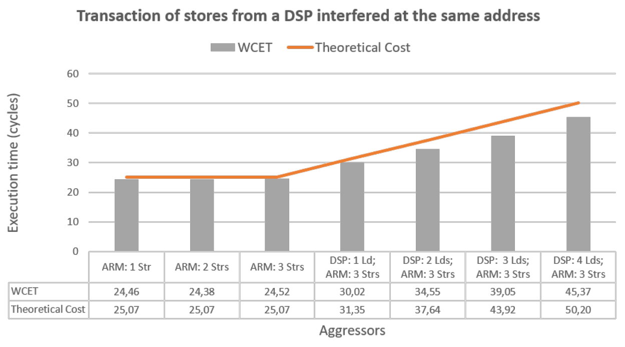
	DSP: 1 Str	DSP: 2 Strs	DSP: 2 Strs, 1 Ld	DSP: 3 Strs, 1 Ld	DSP: 4 Strs, 1 Ld
WCET	278,61	284,06	330,88	330,53	332,08
Theoretical Cost	306,64	306,64	378,35	378,35	378,35

Figure 6: tr'(LD, DSP, DDR) interfered at the same address



	DSP: 1 Ld	DSP: 1 Str, 1 Ld	DSP: 2 Strs, 1 Ld	DSP: 2 Strs, 2 Lds	DSP: 2 Strs, 2 Lds; ARM: 1 Str
WCET	42,54	104,63	105,85	115,13	131,7
Theoretical Cost	38,58	122,52	122,52	138,75	164,47

Figure 7: tr'(LD, ARM, DDR) interfered at the same address

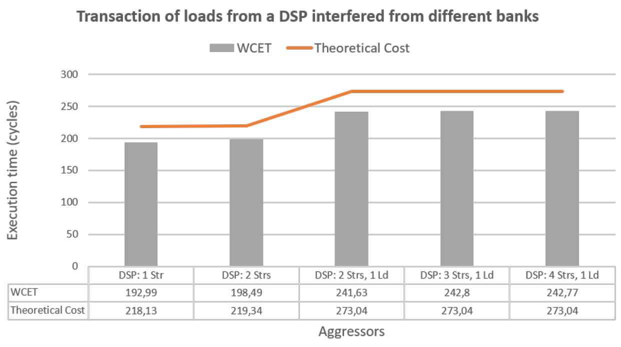


	ARM: 1 Str	ARM: 2 Strs	ARM: 3 Strs	DSP: 1 Ld; ARM: 3 Strs	DSP: 2 Lds; ARM: 3 Strs	DSP: 3 Lds; ARM: 3 Strs	DSP: 4 Lds; ARM: 3 Strs
WCET	24,46	24,38	24,52	30,02	34,55	39,05	45,37
Theoretical Cost	25,07	25,07	25,07	31,35	37,64	43,92	50,20

Figure 8: tr'(STR, DSP, DDR) interfered at the same address

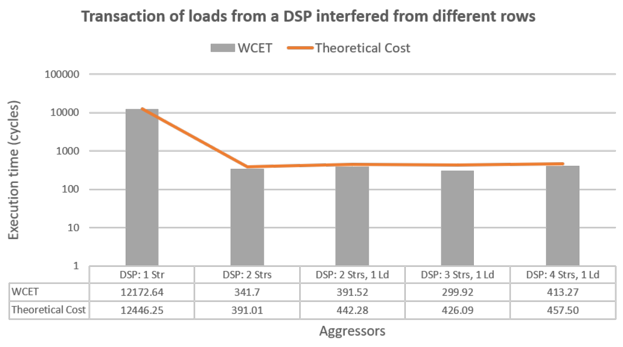
7.2.2 Bank switching. The experiments measurements shown in the three scenarios for the Fixed Block Address (Figures 6, 7 and 8) are retaken but using a different memory bank each. All the experiments in Figure 9 scene offer again the desirable response, upper bounding the WCET with a bit of overestimation. Figure 10, shows how Equation 9 is overall a bit pessimistic, what is not necessarily incorrect as we are aiming the worst case. However, in experiment 5 (DSP: 2 Strs, 2 Lds; ARM: 1 Str), the ARM addition interference is underestimated. Equation 10 apparently behaves correctly for the experiments found in Figure 11.

7.2.3 Row switching. Figure 12 depicts how row switching differs from the other two situations when it comes to impact. Note that



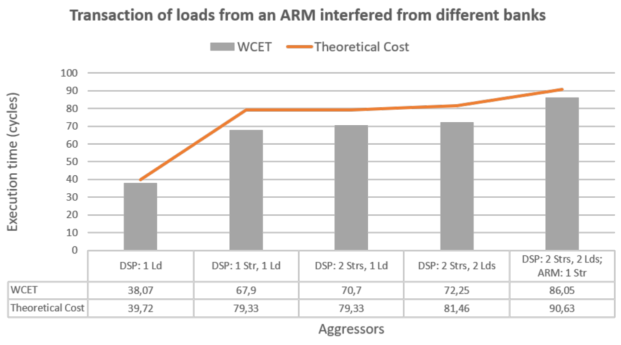
	DSP: 1 Str	DSP: 2 Strs	DSP: 2 Strs, 1 Ld	DSP: 3 Strs, 1 Ld	DSP: 4 Strs, 1 Ld
WCET	192,99	198,49	241,63	242,8	242,77
Theoretical Cost	218,13	219,34	273,04	273,04	273,04

Figure 9: tr'(LD, DSP, DDR) interfered from diff. banks



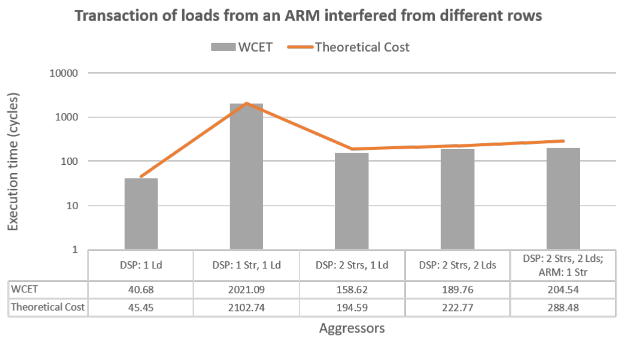
	DSP: 1 Str	DSP: 2 Strs	DSP: 2 Strs, 1 Ld	DSP: 3 Strs, 1 Ld	DSP: 4 Strs, 1 Ld
WCET	12172.64	341.7	391.52	299.92	413.27
Theoretical Cost	12446.25	391.01	442.28	426.09	457.50

Figure 12: tr'(LD, DSP, DDR) interfered from diff. rows



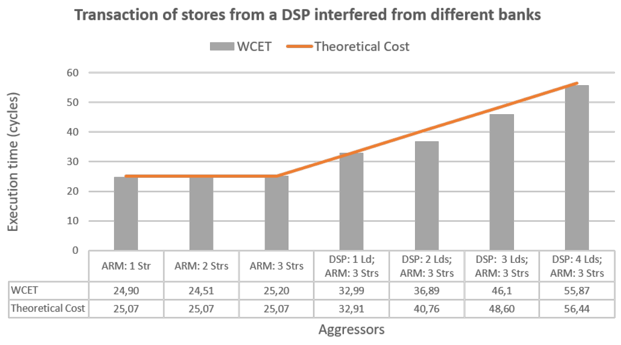
	DSP: 1 Ld	DSP: 1 Str, 1 Ld	DSP: 2 Strs, 1 Ld	DSP: 2 Strs, 2 Lds	DSP: 2 Strs, 2 Lds; ARM: 1 Str
WCET	38,07	67,9	70,7	72,25	86,05
Theoretical Cost	39,72	79,33	79,33	81,46	90,63

Figure 10: tr'(LD, ARM, DDR) interfered from diff. banks



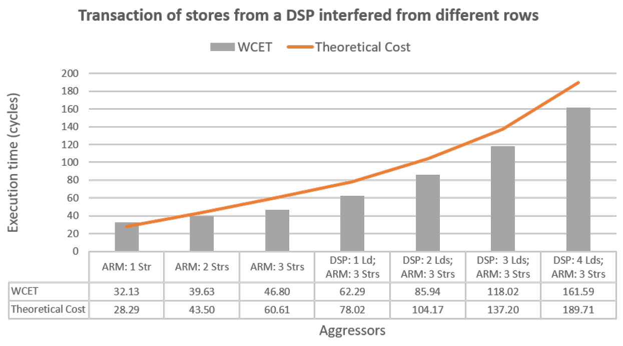
	DSP: 1 Ld	DSP: 1 Str, 1 Ld	DSP: 2 Strs, 1 Ld	DSP: 2 Strs, 2 Lds	DSP: 2 Strs, 2 Lds; ARM: 1 Str
WCET	40.68	2021.09	158.62	189.76	204.54
Theoretical Cost	45.45	2102.74	194.59	222.77	288.48

Figure 13: tr'(LD, ARM, DDR) interfered from diff. rows



	ARM: 1 Str	ARM: 2 Strs	ARM: 3 Strs	DSP: 1 Ld; ARM: 3 Strs	DSP: 2 Lds; ARM: 3 Strs	DSP: 3 Lds; ARM: 3 Strs	DSP: 4 Lds; ARM: 3 Strs
WCET	24,90	24,51	25,20	32,99	36,89	46,1	55,87
Theoretical Cost	25,07	25,07	25,07	32,91	40,76	48,60	56,44

Figure 11: tr'(STR, DSP, DDR) interfered from diff. banks



	ARM: 1 Str	ARM: 2 Strs	ARM: 3 Strs	DSP: 1 Ld; ARM: 3 Strs	DSP: 2 Lds; ARM: 3 Strs	DSP: 3 Lds; ARM: 3 Strs	DSP: 4 Lds; ARM: 3 Strs
WCET	32.13	39.63	46.80	62.29	85.94	118.02	161.59
Theoretical Cost	28.29	43.50	60.61	78.02	104.17	137.20	189.71

Figure 14: tr'(STR, DSP, DDR) interfered from diff. rows

the execution cycles (Y axis) is in logarithmic scale. The integrated table of the figures may result very useful. Equation 11 manages to adequately upper bound the results. The first experiment (DSP: 1 STR) depicts a prioritization starvation situation. By analyzing the DDR controller priority elevations, it could be appreciated that each *tr'* was starved. The fourth experiment (DSP: 3 Strs, 1 Ld) is pessimistic, despite that its behavior is correctly modeled (its value decreases with respect to the previous and posterior experiments). This amplitude reduction is due to the faster row switch provoked by the new transaction addition. This does not happen for the last experiment because the switches are produced before *tr'* is available. Figure 13 shows the theoretical cost evolving according to the measured cost. Nonetheless, there is a clear overestimation for the last experiment (65.71 theoretical increment against 14.78 with respect to the previous experiment(DSP: 2 Strs, 2 Lds)). Equation 12 describes correctly the experiments in Figure 14. The effect of loads on the ARMs stores provokes the exponential behavior.

7.2.4 *Validation.*

To assure the right applicability of the equations presented in this paper for its use in a task mapping evaluation, these should be directly compared in a figure. Therefore, the three situations outcome are merged into a single figure, resulting in three figures. In Figures 15 and 16, where the *tr'* instruction is a load, we can observe that from best to worst arbitration we have **Bank Switching**, **Fixed Block Address** and **Row Switching**. Comparing the theoretical output with the measurement based output and

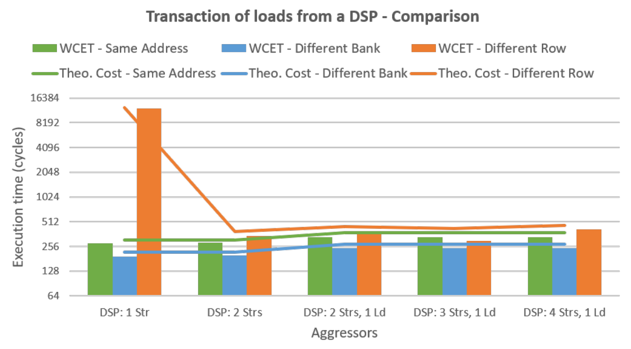


Figure 15: tr'(LD, DSP, DDR) WCET and theoretical cost

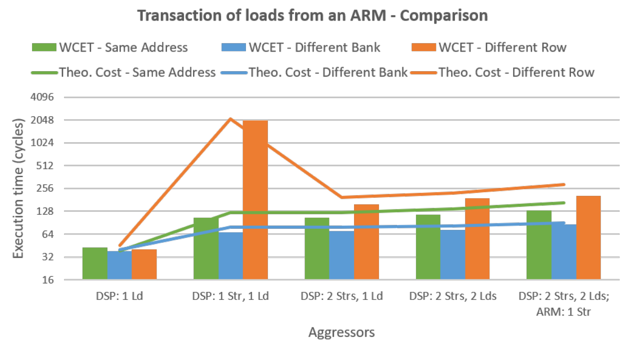


Figure 16: tr'(LD, ARM, DDR) WCET and theoretical cost

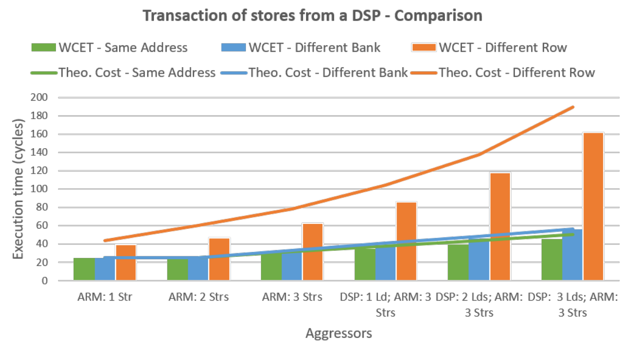


Figure 17: tr'(STR, DSP, DDR) WCET and theoretical cost

the theory, the equations seems correct in general. Nonetheless, it can be seen that the **Fixed Block Address** overestimates and it should be revised if the turnaround cost are unnecessarily pessimistic. Note that these figures Y axis scale is in base 2. Figure 17 ranks the arbitration from best to worst in the following way: **Fixed Block Address**, **Bank Switching** and **Row Switching**, indicating that, from the point of view of a store, the **Fixed Block Address** arbitration is the best option. This is rational as in this arbitration the stores don't suffer from loads higher priority blocking.

8 CONCLUSION AND FUTURE WORK

In this paper we study the interference overhead caused by the DDR3 accesses in heterogeneous multicore platforms which depends on the Command FIFO enqueuing and its reordering arbitration. To describe the resulting memory controller behavior, a set of equations, which are able to estimate the resulting impact, has been elaborated. These equations consider the command under analysis type, reordering arbitration, operational costs and enqueuing scheduling. As future work, these cost functions can be integrated into task/memory mapping tools as well as extending them to other memory controllers by generalizing them. In principle, to do this generalization, the SDRAM features specified by JEDEC will remain untouched but it may not be necessarily the case for the memory controller arbitration and platform heterogeneity. The memory controller may share the command burst technique, open row policy and load prioritization, but may have a different queuing policy or extra optimization techniques. The platform heterogeneity modeling is unique for each board, e.g. unified ARM slave port, processors. Furthermore, it should be researched about how to model situations where several Command FIFO arbitrations are applied at the same time, e.g. bank and row switching. Finally, there are some pending behaviors to be studied further, e.g. ARM single coherent slave port.

ACKNOWLEDGMENTS

This work was supported by the Defense Innovation Agency (AID) of the French Ministry of Defense (research project CONCORDE N° 2019 65 0090004707501) and the French Civil Aviation Authority (DGAC) (research project PHYLOG).

REFERENCES

[1] B. Akesson et al. 2007. Predator: A predictable SDRAM memory controller. In *2007 5th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*.
[2] ARM 2012. *Cortex-A15 MPCore - Technical Reference Manual.* ARM.
[3] Frédéric Boniol et al. 2020. PHYLOG certification methodology: a sane way to embed multi-core processors. In *10th European Congress on Embedded Real Time Software and Systems (ERTS 2020)*. Toulouse, France.
[4] Youcef Bouchebaba et al. 2010. MpAssign: A framework for solving the many-core platform mapping problem. 1 – 7.
[5] Francisco J. Cazorla, Roberto Gioiosa, Mikel Fernandez, and Eduardo Quiñones. 2012. *Multicore OS Benchmark.* Technical Report.
[6] Wei Chen. 2009. *Task Partitioning and Mapping Algorithms for Multi-core Packet Processing Systems.* Master's thesis. University of Massachusetts Amhers.
[7] Cédric Courtaud et al. 2019. Improving Prediction Accuracy of Memory Interferences for Multicore Platforms. In *RTSS 2019 - 40th IEEE Real-Time Systems Symposium*. IEEE.
[8] Hamid Reza Faragardi et al. 2014. An efficient scheduling of AUTOSAR runnables to minimize communication cost in multi-core systems. *2014 7th International Symposium on Telecommunications, IST 2014*, 41–48.
[9] G. Giannopoulou et al. 2014. Mapping mixed-criticality applications on multi-core architectures. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1–6.
[10] Danlu Guo et al. 2018. A Comparative Study of Predictable DRAM Controllers. 17, 2 (2018).
[11] M. Hassan. 2018. On the Off-Chip Memory Latency of Real-Time Systems: Is DDR DRAM Really the Best Option?. In *2018 IEEE Real-Time Systems Symposium (RTSS)*.
[12] D. Iorga et al. 2020. Slow and Steady: Measuring and Tuning Multicore Interference. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.
[13] Philipp Ittershagen et al. 2013. Hierarchical Real-Time Scheduling in the Multi-Core Era – An Overview. *16th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, ISORC 2013.*
[14] JEDEC ASSOCIATION 2008. *JEDEC STANDARD - DDR3 SDRAM - JESD79-3C.* JEDEC ASSOCIATION.

[15] Andreas Löfwenmark and Simin Nadjm-Tehrani. 2018. Fault and timing analysis in critical multi-core systems: A survey with an avionics perspective. *Journal of Systems Architecture* 87 (2018), 1 – 11.

[16] Claire Maiza et al. 2018. *A Survey of Timing Verification Techniques for Multi-Core Real-Time Systems.* Technical Report TR-2018-9. Verimag Research Report.

[17] Alfonso Mascareñas González et al. 2020. Multicore shared memory interference analysis through hardware performance counters. 10th European Congress on Embedded RealTime Software and Systems (ERTS 2020), Toulouse, France.

[18] Jan Nowotsch and Michael Paulitsch. 2012. Leveraging Multi-core Computing Architectures in Avionics. In *2012 Ninth European Dependable Computing Conference, Sibiu, Romania, May 8-11, 2012*, Cristian Constantinescu and Miguel P. Correia (Eds.). IEEE Computer Society, 132–143.

[19] Marco Paolieri et al. 2010. An analyzable memory controller for hard real-time CMPs. *IEEE embedded systems letters* (01 2010).

[20] Petar Radojkovic et al. 2012. On the evaluation of the impact of shared resources in multithreaded COTS processors in time-critical environments. *ACM Trans. Archit. Code Optim.* 8, 4 (2012), 34:1–34:25.

[21] Selma Saidi et al. 2015. The Shift to Multicores in Real-Time and Safety-Critical Systems.

[22] Jun Shao. 2006. *Reducing main memory access latency through SDRAM address mapping techniques and access reordering mechanisms.* Ph.D. Dissertation. Michigan Technological University.

[23] Texas Instruments 2010. *TMS320C66x DSP CPU and Instruction Set.* Texas Instruments.

[24] Texas Instruments 2012. *KeyStone II Multicore Shared Memory Controller.* Texas Instruments.

[25] Texas Instruments 2015. *Keystone II Architecture DDR3 Memory Controller.* Texas Instruments.

[26] Texas Instruments 2017. *66AK2Hxx Multicore KeyStone II System-on-Chip.* Texas Instruments.

[27] H. Yun and P. Valsan. 2015. Evaluating the Isolation Effect of Cache Partitioning on COTS Multicore Platforms.