SafeTI: a Hardware Traffic Injector for MPSoC Functional and Timing Validation

Oriol Sala[†,‡], Sergi Alcaide[†], Guillem Cabo[†,‡], Francisco Bas[†,‡], Ruben Lorenzo[†], Pedro Benedicte[†], David Trilla[†], Guillermo Gil[†], Fabio Mazzocchetti[†], Jaume Abella[†]
[†]Barcelona Supercomputing Center (BSC)
[‡]Universitat Politècnica de Catalunya (UPC)

***Abstract*—Functional and timing validation of safety-related MPSoCs requires testing specific traffic patterns in the on-chip interconnects. Generally, testing needs to be performed by using software tests whose degree of control on the traffic generated is indirect, and limited to behavior that can be triggered by software, thus often unable to produce traffic generated by peripherals. Therefore, untested traffic scenarios can be abundant and, to a large extent, it is hard to know what traffic scenarios have been effectively tested.**

This paper presents the *safe traffic injector*, SafeTI, which allows injecting programmable traffic in AMBA AHB interconnects with high flexibility and degree of control, thus easing achieving high coverage in terms of traffic scenarios tested, and mitigating the uncertainty due to the difficulties to relate software tests with actual traffic scenarios tested. We also integrate successfully the SafeTI in an industrial MPSoC for the space domain proving the effectiveness of the proposed traffic injector.

I. Introduction

Increasing performance demands in safety-related systems imposes the adoption of MultiProcessor Systems-on-Chip (MPSoCs) to reach the required performance levels within limited power envelopes [17]. Examples of those MPSoCs can be found in the space domain (e.g. Cobham Gaisler's LEON [7] and NOEL-V families [8]), in the automotive domain (e.g. Infineon AURIX family), and in the avionics domain (e.g. NXP P/T/LS and Xilinx Zynq UltraScale families).

Safety-related MPSoC devices and systems built on top must comply with specific safety requirements dictated by the corresponding domain-specific safety standards and guidelines (e.g. DO254 [20] and DO178C [19] for avionics, and ISO26262 [13] for automotive). Such compliance imposes strict verification and validation (V&V) processes to obtain evidence that the system behaves according to its specifications by construction, and stressful tests cannot spot any misbehavior.

In the context of MPSoCs, on-chip traffic across components must go through V&V to prove that on-chip components behave correctly in functional terms, and multicore timing interference due to contending traffic in on-chip interconnects does not exceed estimated bounds. Generally, such V&V processes resort to specific software benchmarks intended to trigger the behavior to be verified and/or validated [10]–[12]. Unfortunately, software-only solutions suffer from two main limitations:

1) Some behavior, such as, for instance, burst accesses, may not be directly triggered by software running locally in the MPSoC, and may only be generated by incoming traffic through peripherals (e.g. Ethernet interfaces). Therefore, complex solutions are needed to generate such traffic with multiple systems connected or resorting to loopback modes in peripherals, which ultimately lead to limited traffic patterns.
2) Controllability to exercise specific interference scenarios is too low to have certainty on whether the desired scenarios have been effectively tested. For instance, scenarios where core access patterns to DRAM memory need to collide with incoming Ethernet traffic or DMA transactions in the interconnect or memory require an unrealistic degree of control by software only means due to the asynchronous nature of Ethernet and DMA controllers.

This paper tackles these challenges by providing a new *safe traffic injector*, SafeTI[1], which allows evaluating arbitrary traffic patterns in on-chip interconnects with high degree of flexibility and control on the traffic injected. In particular, the contributions of this work are as follows:

- We architect and implement SafeTI, which allows injecting programmable traffic in on-chip interconnects so that several parameters can be configured, including: traffic duration (e.g. a finite or infinite sequence of transactions), transaction type (read or write), burstiness (length of the transaction), and sequence of transactions to produce any pattern desired.
- We tailor SafeTI to ease its portability to different communication interfaces (e.g. AXI, ACE), and integrate it to work with the AMBA Advanced High-performance Bus (AHB) interface.
- We integrate SafeTI in a commercial space MPSoC based on RISC-V compliant cores from Cobham Gaisler's NOEL-V on an FPGA. We evaluate SafeTI showing its capabilities and flexibility, and compare it against software-only traffic injector counterparts, which are intrinsically limited by their software nature.

The rest of the paper is organized as follows. Section II provides some background on MPSoC V&V, and introduces relevant related work. Section III presents SafeTI architecture. Section IV describes how SafeTI has been integrated in Gaisler's NOEL-V MPSoC with an AMBA AHB interface. SafeTI is evaluated in Section V. Section VI draws the main conclusions of this work.

II. Background & Related Work

V&V activities for safety-related MPSoCs and systems built on top must consider those aspects related to the concurrent access of multiple components to the interconnect. Those aspects include both purely functional behavior of the components and the interconnect itself, as well as timing behavior.

Different methods and tools exist applicable to inject traffic for functional and timing V&V: (i) performance simulators, (ii) circuit simulators, (iii) hardware emulated injectors, and (iv) software stress tests. This section provides some background on each of those approaches.

[1]Available as an open-source component at https://bsccaos.github.io [5].

A. *Traffic Injection with Performance Simulators*

Gem5 [4] is a performance simulator targeting multiple Instruction Set Architectures (ISAs) and CPU models. Gem5 also models the memory system and cache coherence protocols. Garnet [16] is an interconnect model part of Gem5, which includes a network-on-chip (NoC)-only configurable synthetic traffic injector. Garnet allows to configure the duration of the simulation, the injection rate, senders and receivers for the traffic, amongst others. Gem5 and Garnet have been used as part of some NoC proposals for their evaluation, such as SMART [14] and OpenSMART [15].

GNoCSim [9] is a NoC simulator with embedded synthetic traffic generation capabilities. Much in the line of Garnet, GNoCSim allows injecting synthetic traffic patterns, but it also allows injecting traffic patterns traced a priori.

Synfull [3] is also a synthetic traffic generator targeting cache coherence traffic. Synfull is based on tracing traffic in a performance simulator and smartly compressing patterns to inject them in a NoC simulator in the form of synthetic traffic.

While those tools have some similarities with SafeTI in terms of features offered, they are software models, thus targeting much higher abstraction levels for MPSoCs than RTL designs (e.g. simplified performance simulators) and/or with a much higher performance cost by modelling the MPSoC RTL at software level. Instead, SafeTI is a hardware module that is synthesized in an FPGA along with the MPSoC, and has already been integrated with a commercial MPSoC for the space domain as part of this paper. Hence, SafeTI can inject traffic *at speed* (e.g. at the operating frequency of the FPGA, 100 MHz in our paper), whereas performance simulators would typically inject it at rate several orders of magnitude slower depending on whether a simplified model or RTL implementation of the MPSoC is used.

B. *Traffic Injection with Circuit Simulators*

Traffic injection can be performed using circuit simulators such as, for instance, Verilator [22] or Questasim [21], where test vectors can be injected in the desired signals to produce specific behavior. Unfortunately, this type of tools operates at RTL, Verilog, SystemVerilog or VHDL abstractions, thus being extremely slow (i.e. even slower than performance simulators), and thus limiting drastically the scenarios that can be practically simulated. This is particularly true when large parts of the MPSoC need to be simulated (e.g. multiple cores, caches, a bus and a memory controller). Instead, using SafeTI, the traffic injector is synthesized along with the MPSoC and injection occurs at speed in the FPGA, and also allows fabricating an ASIC with SafeTI embedded to perform periodic tests, as needed in safety-related MPSoCs.

C. *Traffic Injection with Hardware Emulated Injectors*

Xilinx AXI Traffic Generator (ATG) [23] is an IP for traffic generation in AXI4 interfaces. Our work has many similarities with the ATG since both are highly flexible and programmable traffic injectors. However, there are some key differences related to their usability and their capabilities. In terms of usability, Xilinx's ATG has a restrictive licensing model that, among other things, imposes the use of the ATG on Xilinx devices only. Moreover, it is intended for AXI4 protocols. Instead, SafeTI is based on a MIT license, hence allowing any type of modification and use, with a target release date – after completing validation and documentation – of Q2 2021. On the technical side, since SafeTI is intended not only for high traffic generation, but also for highly controllable traffic generation, it allows scheduling pauses in between transactions and loading multiple traffic patterns so that a single write operation into the SafeTI allows choosing across different patterns easily.

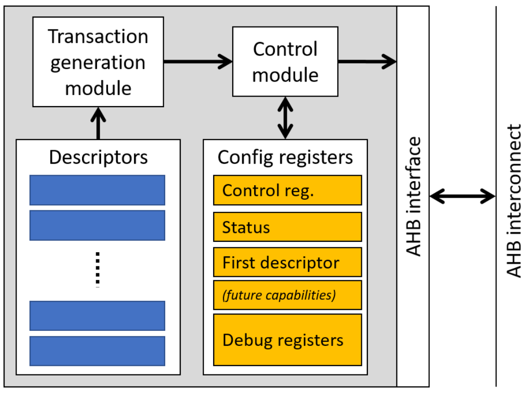

Fig. 1. High level schematic of SafeTI.

In the context of the H2020 Mont-Blanc 2020 project [2], IP for traffic generation is being developed. Differently to SafeTI, Mont-Blanc 2020's traffic generator is based on traffic traces that need to be collected a priori and fed into the generator. To the best of our knowledge, such traffic injector is not yet available, its licensing model – if any – is unknown, and lacks the flexibility of ATG and SafeTI to exercise tight control on the traffic injected, as required for the validation of safety-related systems.

D. *Traffic Injection with Software Stress Tests*

Some solutions have been proposed to generate on-chip traffic on COTS devices by means of specific software programs (a.k.a. stressing benchmarks) that produce repeated memory access patterns inducing indirectly traffic in on-chip interconnects [10]–[12]. Those approaches have some advantages, such as allowing to test patterns at speed and working on top of COTS MPSoCs. However, disadvantages relate to (a) lack of flexibility to inject some traffic patterns (e.g. burst sizes caused only by peripheral devices), (b) lack of controllability due to the fact that interconnect accesses need to traverse some core pipeline stages and cache memories before reaching the interconnect, and (c) lack of observability to tell whether the desired pattern has been effectively generated.

SafeTI overrides those limitations by providing full controllability of the traffic patterns injected and flexibility to generate any traffic patterns allowed by the protocol of the interconnect.

III. SafeTI Features and Operation

This section presents SafeTI. First, we present its general structure and operation, then its main features, and finally the design of the traffic descriptors used for traffic injection. Details on the internal design and integration of SafeTI are presented later in Section IV.

A. *SafeTI General Structure and Operation*

SafeTI's principal structure is a buffer holding descriptors, where each descriptor provides all the information needed to generate a transaction in the interconnect where SafeTI is attached. Those descriptors need to be preprogrammed before traffic starts being injected. For that purpose, they are mapped to a specific address range in the MPSoC, so that they can be easily set with a simple API.

The process followed to inject traffic consists of:

1) Preprogramming SafeTI by writing as many descriptors as needed. As described later, operation can be set to loop over a set of descriptors, so that programming

even a single descriptor makes it possible to generate an unbounded number of transactions.
2) Activating SafeTI so that it starts generating transactions in accordance with the descriptors stored.
3) Deactivating SafeTI to stop generating transactions. Note that if SafeTI is not set to work in loop mode, SafeTI will deactivate itself whenever transactions for all descriptors have been generated. Pausing traffic injection and resuming it later from the point it was stopped is also allowed.

Figure 1 shows a schematic of the SafeTI. Note that this schematic does not fully match the actual modules used for its implementation, but eases the explanation. The actual modules of SafeTI are described in next section. As shown, SafeTI includes a control module orchestrating traffic injection. Such traffic injection is regulated by the configuration registers, which indicate the starting descriptor and whether traffic injection is enabled, among other settings. The actual transactions to be injected are stored in the descriptors list. Such descriptors are converted into transactions in the transaction generation module. Finally, the AHB interface sends those transactions through the interconnect when indicated by the control module. As shown, porting SafeTI to other interfaces (e.g. AMBA AXI) is expected to require only adapting the interface module.

B. Features

The main features of SafeTI are the following:

- **Enable/disable**. The status register in the control module that allows to enable or disable traffic generation. SafeTI stores its status on registers so that, whenever traffic generation is enabled, it either starts from the first descriptor or resumes injection with the following descriptor to the last one processed.
- **Descriptors**. The descriptors module provides a simple way of configuring AHB transactions so that it is possible to customize the type of transaction, address, and repetitions. Details about the descriptors are provided later in Section III-C. A descriptor is a set of configuration registers that encapsulates this functionality. The module is capable of decoding descriptors as requested by the control module. Note that each descriptor embeds information on what the next descriptor is – if any. Hence, by properly setting that field of the descriptors, one can make them loop and produce an unbounded number of requests.
- **Status monitoring**. SafeTI includes a status register reflecting the current state of the injector.
- **Error handling**. SafeTI includes error handling features such as reporting whether a descriptor contains invalid information, and different types of errors that can occur in the injection interface, such as receiving an error signal when performing read or write transaction, or when performing no transaction.
- **Interrupts**. SafeTI generates interrupts whenever the last transaction descriptor is reached, and whenever an error occurs. Note that, if descriptors are programmed to describe a loop, there is not such a concept of "last descriptor" and hence, interrupts are only possible upon an error.
- **Debug support**. SafeTI includes a number of registers exposing its internal state (e.g. what descriptor is being processed) usable for debug purposes. They can be removed without affecting its functionality, but since SafeTI is mainly conceived for V&V purposes and for online periodic tests, exposing debug information is a plus.

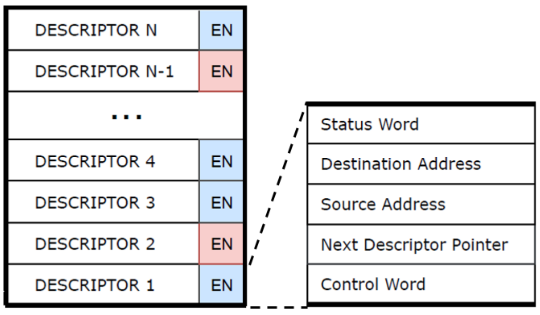

Fig. 2. SafeTI descriptors structure.

C. Descriptors specification

Descriptors are used to define and control transactions in SafeTI. Descriptor types supported by this module can be classified as read and write descriptors. A single descriptor uses 20 Bytes of memory. Their structure and contents are illustrated in Figure 2. Each of the 5 registers in the descriptor is a 4-bytes word. The specification of those registers is as follows.

- **Control word**. The control word is used to configure the main parameters of the descriptor. Due to its relevance, it is fully specificed in Register III.1.

Register III.1: DATA DESCRIPTOR CONTROL WORD (ctrl - 0x00)

	size	count	intrv	dstfix	srcfix	irqe	type	en
	31 13	12 10	9 7	6	5	4	3 1	0
Reset	0	0	0	0	0	0	0	0

size	Size of data to be transferred.
count	Number of transaction repetitions
intrv	Number of clock cycles interval between repetitions of the same transaction.
dstfix	Bit reset (set) for write burst (non-burst) transfer.
srcfix	Bit reset (set) for read burst (non-burst) transfer.
irqe	Enable interrupt on descriptor completion.
type	Descriptor type (read or write).
en	Enable data descriptor (enabled or disabled).

- **Next descriptor pointer**. It indicates the address for the next descriptor and/or whether it is the last descriptor.
- **Destination address**. It indicates the destination address for the descriptor transaction (only relevant for write transactions).
- **Source address**. It indicates the source address for the descriptor transaction (only relevant for read transactions).
- **Status word**. The descriptor status shows if an error has occurred or if the transaction has been done. It includes reserved bits for future extensions.

IV. SafeTI Architecture and Integration

This section provides details on the actual implementation of SafeTI and about its integration in a Gaisler's NOEL-V 4-core MPSoC for the space domain. Hence, we first introduce the MPSoC and then the SafeTI architecture.

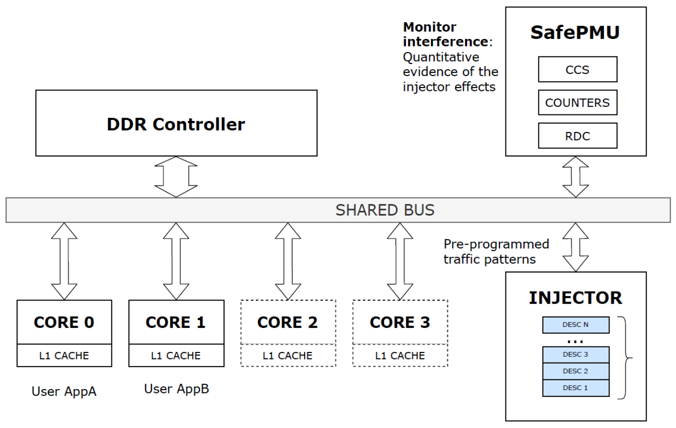

Fig. 3. MPSoC schematic with SafeTI

A. *Gaisler's NOEL-V MPSoC*

The MPSoC where we have integrated SafeTI is based on the NOEL-V RV64 synthesizable VHDL model of a 64-bit processor that implements the RISC-V architecture [18]. The processor is the first released model in Cobham Gaisler's RISC-V line of processors that complement the LEON [7] line of processors. The 4 NOEL-V cores in the MPSoC are interfaced using the AMBA 2.0 AHB [1] bus with the memory and peripheral subsystems. SafeTI integration in the MPSoC is illustrated in Figure 3, where we refer to SafeTI as *injector*. As shown, the injector is connected to the AHB bus analogously to the cores. For evaluation purposes, we have also integrated a multicore interference aware Performance Monitoring Unit (SafePMU) [6], which allows measuring accurately the interference experienced by the bus masters. Hence, the SafePMU provides us with accurate measurements of the delays caused by SafeTI traffic on specific cores.

B. *SafeTI Architecture*

SafeTI architecture is shown in Figure 4, where we can see the modules used for its implementation. The current incarnation of SafeTI acts as an AHB master IP connected to the main AMBA bus of the aforementioned Gaisler's multicore. To some extent, SafeTI acts as a core with limited capabilities, issuing read and write transactions to the slave devices attached to the bus. For its configuration and control, SafeTI implements AMBA Advanced Peripheral Bus (APB) registers, which are driven through the AHB interface as shown in the figure.

The top-level module implements the APB and AHB-master bus ports. The SafeTI top-level module can be realized varying its design parameters that allow varying the AHB and APB indexes where SafeTI is mapped, the address range where its control registers are mapped, the particular interrupt line used, the size of the descriptor list, and the maximum burst length allowed in the AHB interconnect. The top-level module includes the 5 following submodules.

1) AHB Master Interface (`INJECTOR_AHB`): The AHB master interface front end data width can be configured to 32, 64, or 128 bits. While burst accesses are limited to up to 1KB, as specified in the AMBA protocol specification [1], they are limited to 512B in the NOEL-V MPSoC. Upon termination of a burst, an idle cycle is inserted thus allowing re-arbitration in the AHB bus. If, eventually, the intended transaction exceeds the maximum burst length allowed in the MPSoC, the burst is interrupted whenever the limit is reached, and an idle cycle is inserted to guarantee re-arbitration before resuming the descriptor operation with a new burst transaction.

2) APB Slave Interface (`INJECTOR_APB`): The traffic injector is controlled and monitored through registers mapped into a defined APB address space. The complete set of registers is shown in Table I. Note that registers from `0x10` to `0x24` are used for debug capabilities. The control register allows enabling/disabling the injector, configuring whether interrupts can be raised or resetting the injector, among other features. The status register provides information about any error that may have occurred, the number of repetitions elapsed of the descriptor being processed, whether all descriptors have been already processed, and other details of the descriptor processing state. The first descriptor pointer points to the descriptor from where generation of transactions can start. Note that it may not be necessarily the one in a fixed position (e.g. position 0) since we may preload multiple traffic patterns and switch from one to another just changing this register. To ease the extension of the SafeTI features, a reserved register (future capabilities) has been added. Finally, debug registers allow reading the registers of the descriptor being processed, and what the next descriptor is.

TABLE I
APB REGISTER MAP

APB Address Offset	Register
0x00	Control Register
0x04	Status Register
0x08	First descriptor pointer
0x0C	Future capabilities register
0x10	**Debug** Descriptor control word
0x14	**Debug** Next descriptor pointer
0x18	**Debug** Destination address
0x1C	**Debug** Source address
0x20	**Debug** Descriptor status
0x24	**Debug** Current descriptor pointer

3) Control Unit (`INJECTOR_CONTROL`): The control unit is in charge of decoding descriptors and request the read or write interface module to generate the corresponding transaction for such descriptor. The control unit also monitors the status and errors of the transactions, and reports errors – if any – back to the APB Interface.

4) Read Interface Module (`INJECTOR_READ_IF`): The Injector Read Interface deals with descriptors for read transactions. In particular, this module implements a Finite State Machine (FSM) that creates the corresponding read transaction(s). Note that multiple transactions may be needed if data size exceeds the data bus size in a non-burst transaction, or data size exceeds maximum burst size in a burst transaction.

5) Write interface module (`INJECTOR_WRITE_IF`): The Write Interface module is analogous to the Read Interface one, but instead or read transactions, it deals with descriptors for write transactions building on an FSM to manage descriptors requiring multiple transactions.

V. EVALUATION

A. *Evaluation Framework*

The MPSoC has been synthesized for a Xilinx Kintex-7 FPGA KC705 FPGA [24], which is particularly suitable to evaluate space MPSoCs. Results have been collected in a simulation environment and further validated on the FPGA. Results in terms of contention cycles have been collected with the SafePMU.

For the sake of controllability and interpretability of the results, we have used 3 benchmarks:

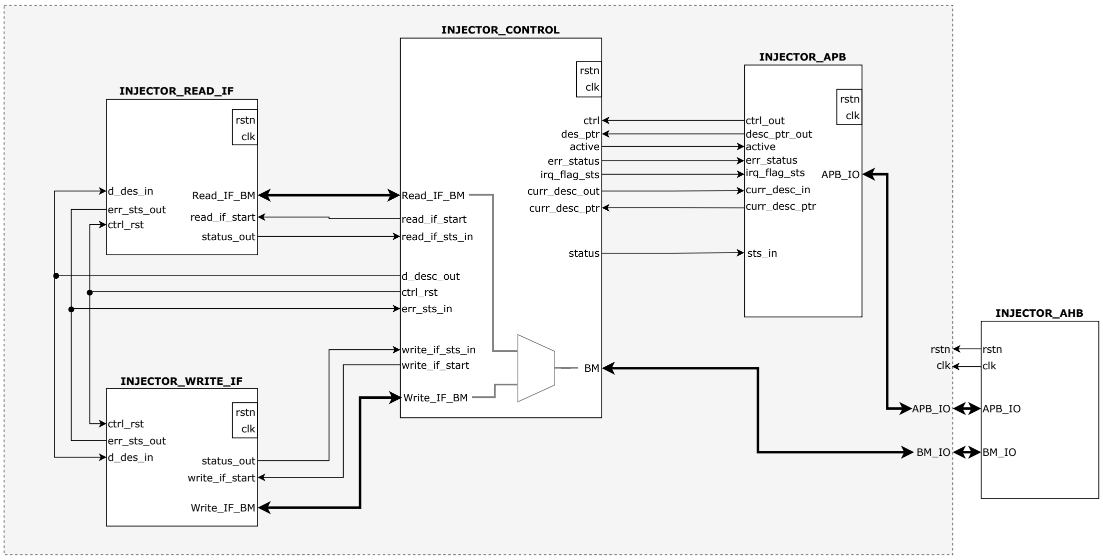

Fig. 4. Module internal block diagram and entity ports

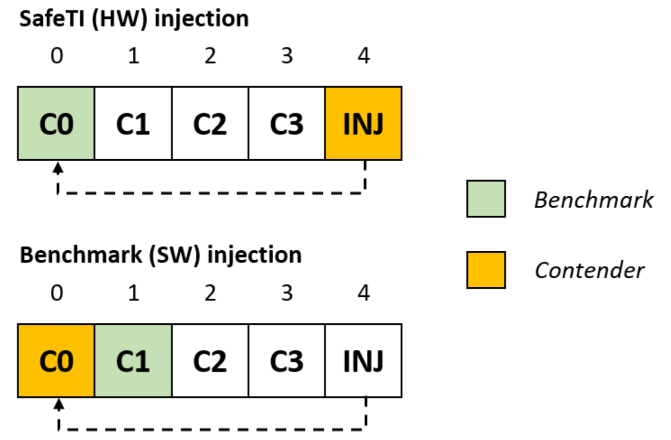

Fig. 5. Mapping of masters in the AMBA AHB arbitration. *Benchmark* refers to the TUA and INJ to SafeTI.

- LD_{MISS}: This benchmark performs sustained 4-byte load accesses that miss in cache and have to fetch data from DRAM, accessing it through the AMBA AHB bus.
- ST_{MISS}: This benchmark is analogous to previous one, but with 4-byte stores.
- SW_{MISS}: This benchmark is analogous to previous one, but with 8-byte stores.

Note that the AMBA AHB bus transfers up to 4 bytes per transaction *beat*. Hence, any request of up to 4 bytes is served in just one beat. Instead, longer requests perform as many beats as needed to transfer all data in beats of 4 bytes (or up to 4 bytes). Those multi-beat requests can be performed in the form of burst requests, so that all beats are transferred in just one transaction (as long as the maximum transaction length is not exceeded), or across multiple independent transactions.

Since full cache lines are fetched upon a cache miss regardless of the data size requested (e.g. 1, 2, 4 or 8 bytes), we use only LD_{MISS} for loads since 16 bytes are fetched per load miss to get the full cache line. Those 16 bytes are fetched in the form of 4 individual 4-byte requests. Instead, store operations are forwarded to memory without allocating or replacing cache lines upon a miss, and they send exactly the data stored in a single burst transactions with either 1 beat (if data stored is up to 4 bytes) or 2 beats (if data stored is 8 bytes). Hence, we use two different store benchmarks to illustrate both cases, namely ST_{MISS} for 4-byte stores and SW_{MISS} for wider 8-byte stores.

Regarding workloads, we assess the impact caused by interference on one of the benchmarks, which we refer to as Task Under Analysis (TUA for short). Since we have 5 masters active in the AMBA AHB bus, namely the 4 cores and SafeTI, in order to fairly compare SafeTI with the traffic generated by benchmarks, we must set its relative position in the round-robin arbitration identically w.r.t. the TUA. Hence, as shown in Figure 5, the TUA is run in core 0 ($C0$) when SafeTI injects traffic. Instead, if contending traffic is injected by another benchmark, then the contending benchmark is run in $C0$ and the TUA in $C1$, so that, in both cases, the contender, either hardware or software, is arbitrated right before the TUA. Note that contention is injected sustainedly meaning that SafeTI never stops and the contenting benchmark is restarted whenever it finishes. Measurements are performed when the TUA finishes its execution.

B. Results

Figure 6 shows the total contention cycles experienced for several scenarios. The magenta triangle (at 7.3 millions of cycles) correspond to the case where LD_{MISS} is the TUA and also the contender. The dark magenta flat line (also at 7.3 millions of cycles) shows the contention caused by SafeTI when injecting increasingly large transactions sustainedly, but without burst transactions. As shown, contention remains constant and identical to the case of the LD_{MISS} benchmark as contender since TUA transactions are only made to wait a 4-byte transaction before being granted access to the bus. In both cases, contention is injected at maximum rate with 4-byte transactions regardless of the transfer size since no burst transactions are used.

If instead we use burst transactions with SafeTI (magenta raising line), we observe how contention increases noticeably (note the logarithmic scale of both axes). In fact, contention roughly doubles as we double transaction length until reaching 512 bytes, which is the maximum transaction size. From that point onwards, the AMBA AHB interface enforces transactions of about 512 bytes being split into up to 512-byte bursts. Contention cycles double because each TUA transaction has

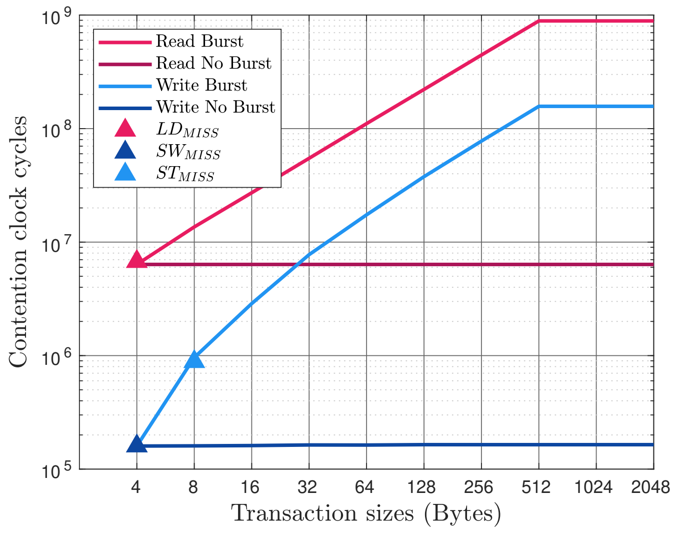

Fig. 6. Total contention cycles in logarithmic scale for different transaction sizes for SafeTI.

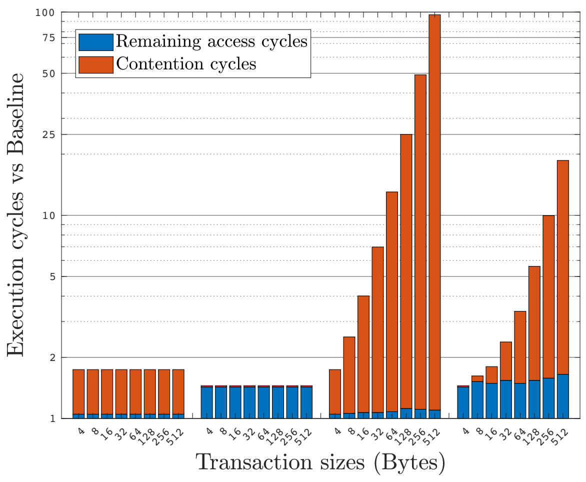

Fig. 7. SafeTI contention cycles for for non-burst and burst cases, for both LD_{MISS} and ST_{MISS} as TUA.

to wait for one SafeTI transaction to be served before being granted access to the bus, but SafeTI transactions double the data sent in a single burst as we move along the x-axis.

Blue triangles and lines correspond to the case where ST_{MISS} is used as TUA. The dark blue triangle corresponds to the case where ST_{MISS} is also used as contender. The light blue triangle corresponds to the SW_{MISS} used as contender. Dark blue (no burst) and light blue (burst) lines correspond to SafeTI traffic injection. As shown, SafeTI behavior when injecting write operations in the AMBA AHB is analogous to that of read operations. In particular, non-burst transactions keep contention flat whereas burst transactions roughly double contention as we double transaction size until reaching the maximum burst size (512 bytes). We also observe that results for SafeTI and benchmarks as contenders roughly match whenever the burst size matches. However, SafeTI is much more flexible and allows, for instance, using any burst size, thus mimicking the impact that long bursts potentially caused by peripherals could cause without needing to build complex tests with I/O operations and challenging synchronization of asynchronous peripheral activity with the TUA execution.

We have further analyzed the impact in a per transaction basis using the SafePMU [6], which allows measuring total access cycles, total contention cycles, and the number of accesses per master. Results are shown in Figure 7. The first set of columns corresponds to LD_{MISS} as TUA and read non-burst traffic injection with SafeTI. The second set of columns is analogous but using ST_{MISS} as TUA and write traffic from SafeTI. The third and fourth sets of columns are analogous to the first two sets, but using bursts transactions. Orange columns correspond to actual contention cycles whereas blue columns correspond to access cycles as reported by the SafePMU. Note that y-axis starts at 1 (using the TUA in isolation as baseline) and is shown in logarithmic scale.

First, we note that pipelining effects may introduce 1-cycle variations in some cases. For instance, in the first set of columns (LD_{MISS} as TUA vs non-burst read traffic), injected traffic causes alternative cases with 0 or 1 interference cycles, thus leading to an average of 0.5 interference cycles per request. If we use ST_{MISS} as TUA and write traffic from SafeTI (second set of columns), then pipeline alignment makes transactions take 1.5 cycles on average and experience no contention. If we use SafeTI with burst transactions, then contention per transaction doubles as we double transaction size as noted before. Note that pipelining effects may make transaction duration oscillate slightly (around 0.1 cycles on average). We also observe that read requests experience noticeable contention even with small burst sizes since load operations stop the pipeline. However, write operations can be delayed to some extent without impacting performance since they do not block the pipeline given that cores have a 2-entry store buffer. Hence, SafeTI write burst transactions of up to 16 bytes cause less than 1 cycle of contention per TUA transaction on average. However, once the store buffer is saturated (with 32-byte transactions and beyond), doubling SafeTI transaction size doubles the contention cycles per TUA transaction.

Overall, our results show how SafeTI allows injecting traffic with high flexibility and degree of control, thus easing V&V of the SoC, as well as implementing safety measures (e.g. periodic testing during operation).

VI. Conclusions and Future Work

The ability to inject traffic in MPSoC's shared resources is instrumental for the verification and validation of safety-related MPSoCs, as well as for the deployment of appropriate safety measures. AMBA interconnects are central elements in MPSoCs since, generally, all traffic traverses them. However, injected controlled traffic patterns at specific times in those interfaces is a challenge in existing MPSoCs either due to the characteristics of the (limited) hardware support, or due to the limitations to achieve that goal by software-only means.

This paper presents SafeTI, a flexible and programmable hardware traffic injector tailored to produce traffic for AMBA AHB interfaces. We prove its effectiveness integrating it in a space MPSoC, and showing how traffic can be effectively tailored providing much more coverage than software-only solutions. Furthermore, SafeTI has been release as an open-source component [5].

While SafeTI is already a complete component, a number of tasks are in front of us: (1) extending its evaluation to more complex traffic patterns and testing scenarios; (2) tailoring it to other interfaces such as AMBA AXI and ACE; and (3) investigating applications of SafeTI for the validation of distributed interconnects and coherence protocols, and to evaluate the effectiveness of means against security attacks (e.g. denial-of-service attacks).

Acknowledgements

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement no. 871467. This work has also been partially supported by the Spanish Ministry of Science and Innovation under grant PID2019-107255GB.

References

[1] ARM. *AMBA 2.0 Specification*, 1999.

[2] Adrià Armejac, Bine Brank, Jordi Cortina, François Dolique, Timothy Hayes, Nam Ho, Pierre-Axel Lagadec, Romain Lemaire, Guillem López-Paradís, Laurent Marliac, Miquel Moretó, Pedro Marcuello, Dirk Pleiter, Xubin Tan, and Said Derradji. Mont-blanc 2020 towards scalable and power efficient european hpc processors. pages 1–6, 2021.

[3] M. Badr and N. E. Jerger. Synfull: Synthetic traffic models capturing cache coherent behaviour. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 109–120, 2014.

[4] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[5] BSC - CAOS. SafeTI. https://bsccaos.github.io.

[6] Guillem Cabo, Francisco Bas, Ruben Lorenzo, David Trilla, Sergi Alcaide, Miquel Moreto, Carles Hernandez, and Jaume Abella. SafeSU: an extended statistics unit for multicore timing interference. https://people.ac.upc.edu/jabella/ets21.pdf. In *IEEE European Test Symposium (ETS)*, 2021.

[7] Cobham Gaisler. LEON5 processor. https://www.gaisler.com/index.php/products/processors/leon5.

[8] Cobham Gaisler. NOEL-V Processor. https://www.gaisler.com/index.php/products/processors/noel-v.

[9] Universitat Politècnica de València. Gnocsim. a cycle-accurate wormhole-based network simulator. http://www.gap.upv.es/index.php?option=com_content&view=article&id=72&Itemid=108.

[10] G. Fernandez, F. J. Cazorla, J. Abella, and S. Girbal. Assessing time predictability features of Arm Big. LITTLE multicores. In *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 258–261, 2018.

[11] Mikel Fernández, Roberto Gioiosa, Eduardo Quiñones, Luca Fossati, Marco Zulianello, and Francisco J. Cazorla. Assessing the suitability of the ngmp multi-core processor in the space domain. In *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '12, page 175–184, New York, NY, USA, 2012. Association for Computing Machinery.

[12] Sylvain Girbal, Jimmy Le Rhun, and Hadi Saoud. METrICS: a Measurement Environment For Multi-Core Time Critical Systems. In *ERTS 2018*, 9th European Congress on Embedded Real Time Software and Systems (ERTS 2018), Toulouse, France, January 2018.

[13] International Standards Organization. *ISO/DIS 26262. Road Vehicles – Functional Safety*, 2009.

[14] T. Krishna, C. O. Chen, W. C. Kwon, and L. Peh. Breaking the on-chip latency barrier using smart. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 378–389, 2013.

[15] H. Kwon and T. Krishna. Opensmart: Single-cycle multi-hop noc generator in bsv and chisel. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 195–204, 2017.

[16] Synergy Lab. Garnet. An on-chip Network Model for Diverse Interconnect Systems. https://synergy.ece.gatech.edu/tools/garnet/.

[17] J. Perez-Cerrolaza et al. Multi-core devices for safety-critical systems: A survey. *ACM Comput. Surv.*, 53(4), 2020.

[18] RISC-V International. RISC-V International website. https://riscv.org/.

[19] RTCA and EUROCAE. *DO-178B / ED-12B, Software Considerations in Airborne Systems and Equipment Certification*, 1992.

[20] RTCA and EUROCAE. *DO-254 / ED-80, Design Assurance Guidance for Airborne Electronic Hardware*, 2000.

[21] Siemens. Questa Advanced Simulator, 2020.

[22] Veripool.org. Verilator, 2020.

[23] Xilinx. *AXI Traffic Generator v3.0. LogiCORE IP Product Guide*, 2019.

[24] Xilinx Kintex-7 FPGA. KC705 evaluation kit. https://www.xilinx.com/products/silicon-devices/fpga/rt-kintex-ultrascale.html.