#HiPEAC 2026

January 26-28, 2026, Kraków, Poland

HIPEAC 2026 Conference

Schedule of

RAPIDO 2026 Workshop

Kraków, Poland

27th January 2026

#HiPEAC 2026

January 26-28, 2026, Kraków, Poland

Organizing committee

Gianluca Palermo, *Politecnico di Milano, Co-General Chair*
Daniel Chillet, *University of Rennes, IRISA, Co-General Chair*

Lilia Zaourar, *CEA List*
Morteza Biglari-Abhari, *University of Auckland*
Daniel Gracia-Pérez, *Thales Research & Technology*
Matthias Jung, *University of Würzburg, Fraunhofer IESE*
Reda Nouacer, *CEA List*

Program committee

Morteza Biglari-Abhari, *University of Auckland*
Pierre Boulet, *Univ Lille 1, CRIStAL*
Jeronimo Castrillon, *TU Dresden*
Daniel Chillet, *University of Rennes, IRISA*
Christian Haubelt, *University of Rostock*
Roberto Giorgi, *University of Siena*
Daniel Gracia-Pérez, *Thales Research & Technology*
Matthias Jung, *University of Würzburg, Fraunhofer IESE*
Tim Kogel, *Synopsys*
Reda Nouacer, *CEA List*
Gianluca Palermo, *Politecnico di Milano*
Mario Porrmann, *Bielefeld University*
Davide Quaglia, *University of Verona*
Sotirios Xydis, *National Technical University of Athens*
Lilia Zaourar, *CEA List*

Website

`https://rapidoworkshop.github.io/`

RAPIDO

Schedule

- **Workshop Introduction** 10 : 00 – 10 : 15
  - **Gianluca Palermo,** *Politecnico di Milano*
  - **Daniel Chillet,** *University of Rennes, IRISA*

- **Keynote Session 1** 10 : 10 – 11 : 10, **Chairman : Daniel Chillet**
  - **Jakob Engblom**, Product Marketing Director, Cadence (VLAB)
    Virtual Platforms for System Architecture, Development, and Test – an Overview

- **Paper Session 1** 11 : 10 – 12 : 50 **Chaiman : Gianluca Palermo**
  - Martin Kristien, Björn Franke, Nigel Topham, Igor Böhm, Harry Wagstaff and Tom Spink
    Scalable Decode Caching in Multi-Core Instruction Set Simulators
  - Nils Wilbert, Derek Christ, Timo Grundheber and Matthias Jung
    ARCADES : A RISC-V-Coupled Accelerator for Discrete Event Simulation
  - Maxime Gras-Chevalier, Christophe Jégo, Camille Leroux and Franck Guillemard
    A Simulation Methodology for Fast Verification of Cyber-Physical Systems
  - Wilfread Guillemé, Gaëtan Lounes, Angeliki Kritikakou, Youri Helen, Robin Gerzaguet, Matthieu Gautier, Cédric Killian and Daniel Chillet
    Selective Triplication for Fault-Tolerant Systolic Array Processing Elements

- **Closing Session** 12 : 50 – 13 : 00
  - **Gianluca Palermo,** *Politecnico di Milano*
  - **Daniel Chillet,** *University of Rennes, IRISA*
  - Best paper award

Keynotes

Keynote 1 10 : 10 – 11 : 10

Virtual Platforms for System Architecture, Development, and Test – an Overview

— **Jakob Engblom, Product Marketing Director, Cadence (VLAB).**

— **Abstract :** When architecting, designing, developing, manufacturing, and maintaining computer-based systems, engineers cannot rely on physical prototypes and real systems. Simulation and virtualization are necessary tools throughout the life cycle of a system, from early architecture work through detailed development to system validation and long-term software updates. Different types of simulation models are used for different purposes, including everything from simple traffic generators to cycle-accurate designs to implementation RTL to transaction-level models. Simulation can also be used to bring in the environment in which a system operates, at different levels of scope.

— **Biography :**

Return to Schedule

Scalable Decode Caching in Multi-Core Instruction Set Simulators

Martin Kristien, Björn Franke, Nigel Topham, Igor Böhm, Harry Wagstaff and Tom Spink

Return to Schedule

Scalable Decode Caching in Multi-Core Instruction Set Simulators

Björn Franke
Nigel Topham
bfranke@inf.ed.ac.uk
npt@inf.ed.ac.uk
University of Edinburgh
UK

Igor Böhm
Martin Kristien
Harry Wagstaff
iboehm@synopsys.com
Martin.Kristien@synopsys.com
wagstaff@synopsy.com
Synopsys
UK

Tom Spink
tcs6@st-andrews.ac.uk
University of St Andrews
UK

Abstract

Instruction set simulators (ISSs) play an important role in embedded software development. Integrated in virtual platforms, they enable coding, testing, and performance evaluation without the need for physical platforms. However, simulations incur a performance penalty over native execution, resulting in slow simulation speeds for complex applications. We realize that in interpreter-based ISS – developers' first choice when detailed processor pipeline and cache simulation are required – the simulator's own instruction fetch and decode stages substantially contribute to overall runtime. We propose a novel simulator instruction fetch and decode cache architecture: (a) We use instruction encodings for cache indexing instead of the program counter, (b) we introduce separate instruction fetch and decode caches instead of a single, unified cache, and (c) we introduce a tiered cache architecture, comprising private and global caches for multicore guest architectures. We have implemented our novel caching schemes in the commercial Synopsys ARC© nSIM ISS that provides an instruction accurate processor model for the Synopsys ARC processor families. We evaluated our new simulator cache architecture using complex real-world workloads and guest configurations with up to 128 simulated guest cores, where we demonstrate average speed-ups of 1.31× over a state-of-the-art baseline scheme, while requiring only 27% of the original cache memory.

1 Introduction

Instruction Set Simulators (ISSs) play a prominent role in the development of embedded hardware and software [11], much more so than in any other computing domain. They enable rapid prototyping of new instructions [18], developing and debugging applications before hardware exists [2], and architectural exploration [15]. In particular, ISSs decouple embedded software development from the availability of a physical device.

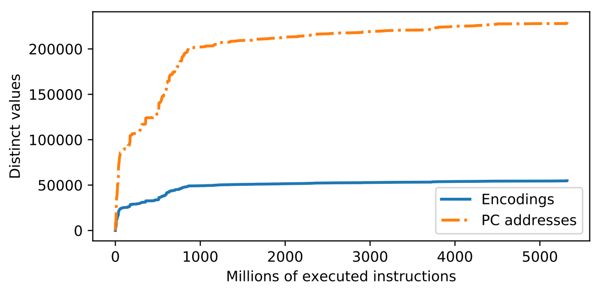
There are a range of strategies that can be used to implement an ISS, the most basic being a simple interpreter [19], and the more complex ones involving Dynamic Binary Translation (DBT) [13]. In general, interpreter-based ISSs cannot achieve the performance of native execution, however they still offer notable advantages that secure them a firm place in the developers' toolbox. Interpreters are relatively easy to implement, making rapid development or modifications of new or existing architectures feasible. More importantly, since interpretation happens inline with the execution of individual guest instructions, instrumentation, profiling, or debugging of the guest code becomes straightforward. Step-by-step interpreted execution enables interpreted ISSs to interface with detailed pipeline or cache models [10, 17].

Interpreted ISSs with their simple *fetch-decode-execute* execution strategy are faced with a fundamental performance challenge: the repeated fetching and decoding of instructions contributes substantially to overall simulator execution time [6]. To tackle this, ISSs typically employ a *decode cache*, to accelerate instruction decoding. Such caches are typically indexed by the emulated PC, so that pre-decoded instructions can be obtained quickly, and repetitive costs for bit-level instruction decoding can be avoided after an initial warm-up phase [23].

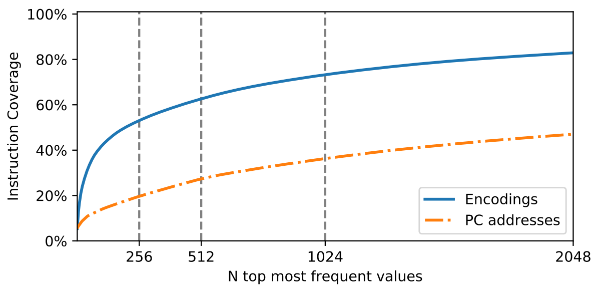
We observe that the commonly used PC-based caching scheme for decoded instructions is far from optimal: (1) The number of distinct PC values in a typical program is far greater than the number of distinct instruction encodings, leading to low cache utilization, and (2) as the number of simulated guest cores increases, the memory footprint for traditional decode caches also increases linearly, resulting in poor scalability.

In this paper we revisit the instruction fetch and decode cache architecture in ISSs, and introduce a strategy that combines a number of novelties: we (a) use instruction encodings for cache indexing instead of the PC for higher cache utilization, (b) introduce separate instruction fetch and decode

This work is licensed under a Creative Commons Attribution 4.0 International License.
RAPIDO '26, Krakow, Poland
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2287-5/2026/01
https://doi.org/10.1145/3787501.3787504



(a) Total size of discovered encoding and Program Counter (PC) spaces.



(b) Cumulative coverage of the most frequent encodings and PC values.

Figure 1. Encodings and PC distributions in gcc benchmark. The encoding space is significantly smaller than the PC space.

caches instead of the traditional unified cache, and (c) introduce a tiered cache architecture comprising private and global caches for use in simulation of multi-core guest architectures for greater scalability and increased efficiency.

Our evaluation, based on the commercial Synopsys ARC© nSIM ISS extended with our novel fetch and decode cache architecture, confirms that substantial gains in simulation performance and memory efficiency can be achieved. For complex, real-world workloads and industry-standard SPLASH-2 benchmarks average speed-ups of 1.31×, and up to 1.57× over the state-of-the-art baseline scheme, can be delivered, while operating on only 27% of the cache memory of the reference scheme.

1.1 Motivating Example

Consider the gcc benchmark shown in Figure 1. The number of distinct PC values encountered during the execution (the *PC space*) is far greater than the number of distinct instruction encodings (the *encoding space*). This is because the same few instructions are used repeatedly across multiple PCs. In particular, out of more than 5 billion executed instructions, we count approximately 229,000 distinct PCs, while only 54,000 distinct instruction encodings. Therefore, the PC space is about 4× greater than the encoding space. Furthermore, we observe that a few encodings account for a large proportion of the executed instructions (Figure 1b). In contrast, many more PC addresses are required to account for the same proportion of the dynamic instruction workload.

These observations indicate that many instructions are duplicated in the PC space, and thus we can design a more efficient instruction storage scheme for the decode cache. For example, using a fixed *PC*-indexed decode cache of the most frequent 256 PC values results in a hit rate of 19.62%, while using an *encoding*-indexed decode cache of the most frequent 256 instruction encodings results in a hit rate of 53.01%. In fact, the *encoding*-indexed cache achieves a greater hit rate vs. a 2× larger *PC*-indexed cache.

2 Decode Caching Schemes

Given these observations, we propose to remove decode object duplication by reorganizing the *decode cache* to be indexed by the raw instruction encoding, rather than the PC.

Whilst this strategy does eliminate *decode object* duplication, the cache can only be queried after a *fetch* operation has been performed, and the raw instruction encoding is available. As the *fetch* stage contributes to a significant portion of the interpreter execution pipeline, we introduce an additional PC-indexed *fetch cache*, which caches the result of the *fetch* operation (i.e. the instruction encoding), and other PC-specific information not encoded in the instruction's binary representation. Since the corresponding *fetch cache* entry is relatively small, we tolerate cache entry duplication for the benefit of improved overall performance.

2.1 The Indirect Scheme

We first introduce the *Indirect* scheme, where we store an index value next to the raw encoding in each *fetch* cache entry. If the lookup in the *fetch* cache hits, the index value is used to directly access the corresponding *decoded instruction object* without the need to perform a lookup in the *decode* cache. Figure 2 shows how the flow of operations work in this strategy: in the best case, a hit in the fetch cache results in a pointer directly to the valid *decoded instruction object.*

Since multiple *fetch* entries can point to the same *decode* entry, we have to introduce an invalidation mechanism in case the corresponding *decode* entry is evicted. This effectively defines an inclusive caching policy, where the *decode* cache is inclusive of the *fetch* cache.

We opt for a lazy invalidation approach, where a validity check is performed in the *decode* cache after hitting in the *fetch* cache. This is effectively a tag check, where we compare the raw encoding of the *fetch* entry with the raw encoding of the *decode* entry (which is stored as part of the *decoded instruction object*). The alternative is an eager invalidation approach, which would be triggered when a decode object is

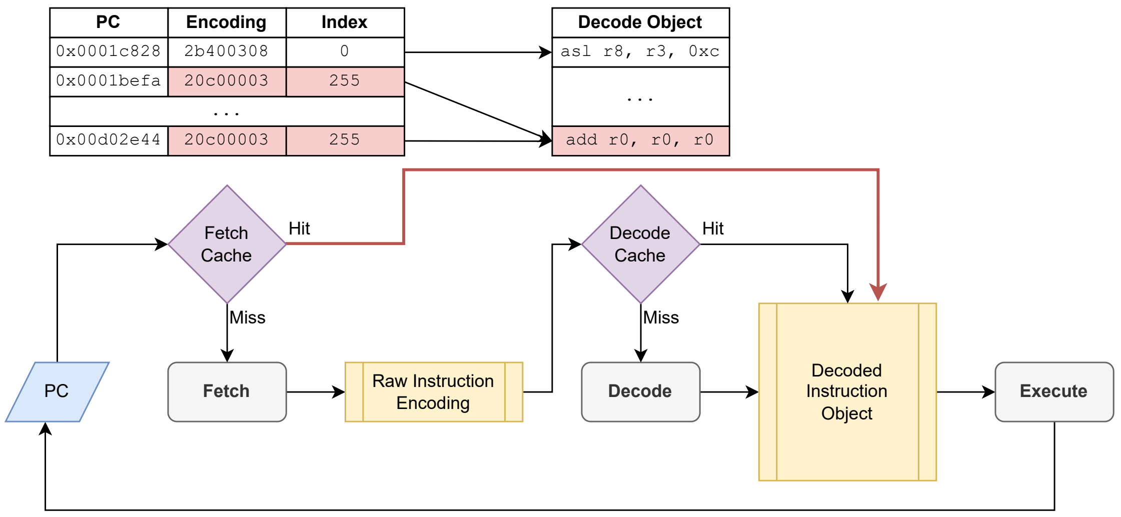


Figure 2. Operation of the separate *fetch* and *decode* caches: Instead of performing an additional lookup in the *decode* cache after a *fetch* cache hit, an index field is used as a pointer offset to bypass the *decode* cache lookup completely, and directly access the *decode object*.

evicted; however, this would require a full scan of the *fetch* cache to identify entries that need to be invalidated.

2.2 The Exclusive Scheme

We also observe significant *decode object* duplication in multi-core simulation. This duplication arises when private *decode* caches are instantiated for each simulated core. In the case of data parallel guest applications, the simulated cores execute the same code and thus contain the same instructions in their own private *decode* caches. In the case of task-parallel guest applications, even though the simulated cores might not execute the same code, they are still likely to share similar, if not identical, instructions (in terms of raw instruction encodings). Therefore, sharing *decode objects* among multiple simulated cores can reduce redundancy.

In this section, we introduce the *Exclusive* caching scheme suitable for multicore simulation that removes duplication of *decode objects* between simulated cores. This is achieved using a shared memory pool of *decoded instruction objects*, with additional low-overhead synchronization mechanisms to prevent data races.

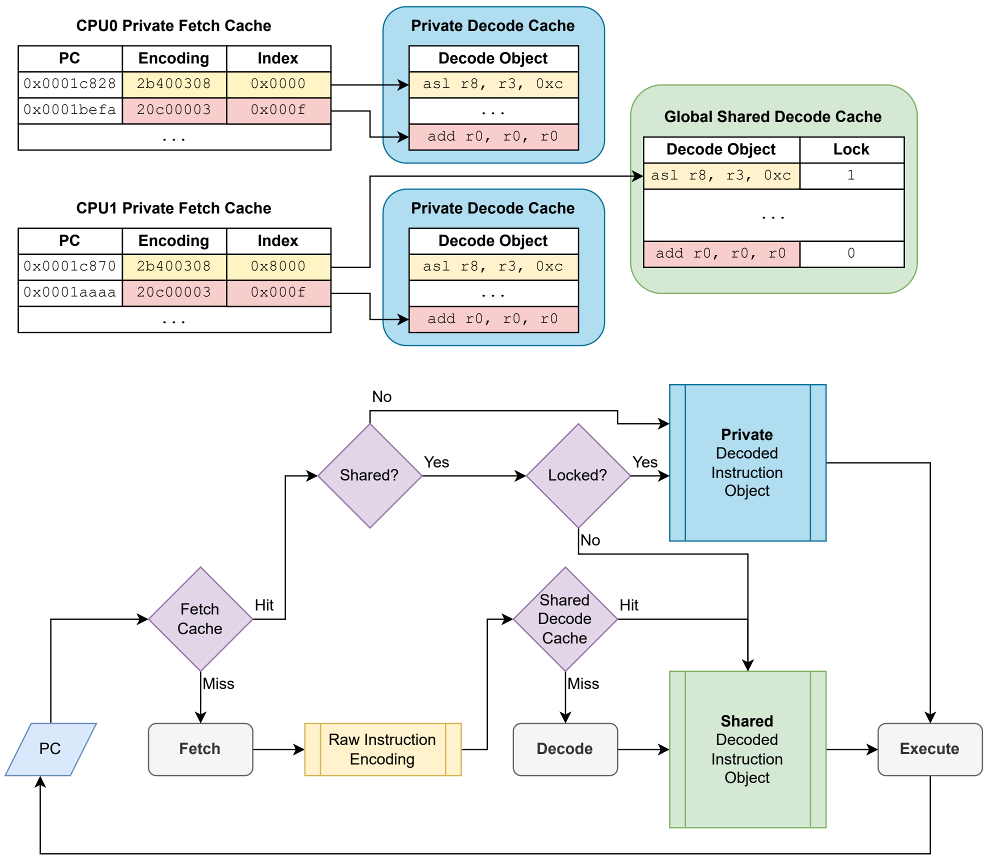
Sharing *decode objects* between multiple cores raises the possibility of concurrent modification, specifically when one core replaces a *decode* entry in the shared cache while another core has a pointer to the same entry in its private *fetch* cache. Since the replacement can happen concurrently, the invalidation mechanism from the *Indirect* scheme is not enough to guarantee correct execution. To protect against concurrent modifications, a lock must be acquired for each *decode* entry before the corresponding *decode object* is used in the *execute* stage.

We use a 1-byte spinlock to implement mutual exclusion. This reduces the memory footprint of the synchronization mechanism, as well as achieving the best performance relative to other synchronization options (e.g. a pthread mutex takes up 40 bytes of memory, and in our experiments was always slower than a spinlock, with up to 40% longer execution times). The outstanding performance of a spinlock is due to the small size of the critical section, i.e. the execution time of the *execute* stage is always shorter than the cost of two context switches, required to service a sleeping lock.

Unfortunately, other than protecting against concurrent modifications, the spinlock also forces mutual exclusion in the case of executing the same instruction. Serializing the execution of the same instruction becomes a bottleneck, especially for data-parallel applications with small workload kernels. For example, a 16-core simulation with a workload kernel of just sixteen distinct raw encodings is guaranteed to have at least two cores executing the same instruction encoding at any one time, and therefore needing to serialize, resulting in an overall slowdown of the application.

2.3 The Mixed Scheme

To address this, and prevent multiple cores from having to serialize their execution, we introduce a mixed private and shared *decode* cache as shown in Figure 3. This *Mixed* scheme builds on top of the *Exclusive* scheme by adding a small core-private *decode* cache. All *decode objects* first start in the shared cache. Then, if a core tries to acquire a lock during future *fetch* cache hits, and the lock variable is already acquired, the *decode object* is copied into the core's private


CPU0 Private Fetch Cache

PC	Encoding	Index
0x0001c828	2b400308	0x0000
0x0001befa	20c00003	0x000f
...		

Private Decode Cache

Decode Object
asl r8, r3, 0xc
...
add r0, r0, r0

CPU1 Private Fetch Cache

PC	Encoding	Index
0x0001c870	2b400308	0x8000
0x0001aaaa	20c00003	0x000f
...		

Private Decode Cache

Decode Object
asl r8, r3, 0xc
...
add r0, r0, r0

Global Shared Decode Cache

Decode Object	Lock
asl r8, r3, 0xc	1
...	
add r0, r0, r0	0

Figure 3. Cache organisation for the *Mixed* scheme. Private *fetch* entries reference either private or shared *decode* entries. Shared *decode* pointers are indicated by a bit tag in the index field of each *fetch* entry.

decode cache. By creating a private copy of the *decode object*, the core no longer needs to synchronize with others.

To suport this, the corresponding index in the *fetch* cache entry has to be patched to indicate that the corresponding *decode object* resides in the private cache. A shared *decode object* is indicated by setting the most significant bit of the 2 byte index field. We ensure the cache sizes are never big enough to require this bit, and thus we can use it for index/-pointer tagging. After the entry index has been patched, all future *fetch* cache hits can use the private *decode object* without any synchronization. Note, replacements in the private *decode* caches are still possible, therefore a validity check of the entry (using the *raw instruction encoding*) is still required.

2.4 The Parallel Scheme

Fully concurrent access to shared *decode objects*, with a negligible amount of synchronization, can be achieved if we design the system such that *decode objects* are never replaced during instruction emulation—concurrently or otherwise. Under this invariant, the validity check is no longer required, so we can remove the raw instruction encoding from the *fetch* cache entry. Furthermore, the lock variable in the *decode* cache entry can be removed, as no synchronization is required. This arrangement is shown in Figure 4.

The *shared decode* cache is only modified in the case of a compulsory miss. If any core tries to execute an instruction not already present in the *decode* cache, a global lock is acquired and the corresponding *decode object* inserted. Note: this only applies for compulsory misses to the *global shared decode cache*. If a core experiences a compulsory miss in its *private fetch cache*, but the corresponding *decode object* is found in the *shared decode* cache, no synchronization is necessary, because searching in the *decode* cache can be done without any data races, even in the case of a concurrent insertion of a *decode object*.

Since the *decode* cache used in this scheme is an append-only data structure with a statically known maximum size, we organize it as an array-allocated AVL binary search tree. This results in logarithmic look-up times, with respect to the size of the cache.

In case the *decode* cache becomes full, we discard the entire contents of the cache and start filling it anew. This approach,

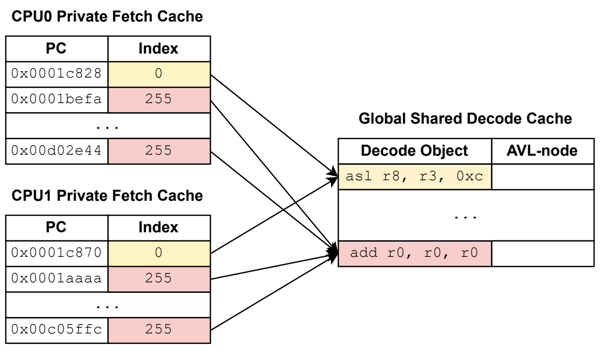


Figure 4. Cache organisation for multicore execution. Core-private fetch entries only need an index pointer, as no *decode* cache invalidations are allowed.

whilst quite brutal, is very similar to QEMU's handling of translated code caches [5]. To guarantee the aforementioned invariant, all cores must be stopped before the *decode* cache invalidation is performed. Furthermore, the *fetch* caches of all cores are also invalidated to eliminate any stale pointers. After the reset operation, cores experience compulsory misses that result from filling up the *decode* cache again.

The major drawback of this scheme is the high cost of the *Stop-the-World* cache reset. This can become a significant portion of the simulation if the memory allocated to the shared *decode* cache is small relative to the application workload size. However, as more and more cores are simulated, more and more memory is saved by having a shared *decode* cache, allowing for a larger *decode* cache size, resulting in less frequent resets.

3 Evaluation

To evaluate the effectiveness of our novel caching schemes, we extend the commercial Synopsys ARC© nSIM ISS simulator, and run our experiments in *multi-core* mode. All experiments are run on a 2.4 GHz 10-core Intel Xeon E5-2640, host machine, running CentOS 6.6. We refer to the vendor-provided caching scheme in the simulator as the *Original* scheme, which is used as our baseline throughout.

3.1 Experimental Setup

Multi-core evaluation is facilitated by a collection of benchmarks from the SPLASH-2 suite [22]. These benchmarks run as multi-threaded guest applications; the number of guest threads can be configured in powers of two.

In multi-core simulation, the *Original* and *Indirect* schemes duplicate their *decode* caches for each simulated core, resulting in memory consumption increasing linearly with the number of simulated cores. However, the *Exclusive*, *Mixed*, and *Parallel* schemes duplicate only the smaller *fetch* cache. Decode entries can be shared in a global memory pool, resulting in improved scalability for many-core simulation—the more cores that participate in cache memory sharing, the more memory is available to increase the size of both the fetch and decode caches. Alternatively, if the cache sizes are fixed, the multi-core schemes will require less memory than the *Original* scheme as more cores are simulated.

Scheme	Orig.	Indir.	Excl.	Mixed	Par.
# of fetch entries	512	1024	2048	2048	2048
Fetch entry size	188 B	14 B	14 B	14 B	10 B
# of private entrs.	-	445	-	32	-
Private entry size	-	184 B	-	184 B	-
# of shared entrs.	-	-	2900	2650	3150
Shared entry size	-	-	185 B	185 B	191 B
Total (8 cores)	752 KB	752 KB	748 KB	749 KB	748 KB
Total (128 cores)	11.8 MB	11.8 MB	4.0 MB	4.7 MB	3.1 MB
Reduct. over Orig.	-	0.1%	66%	60%	74%

Table 1. Memory budget for multi-core simulation.

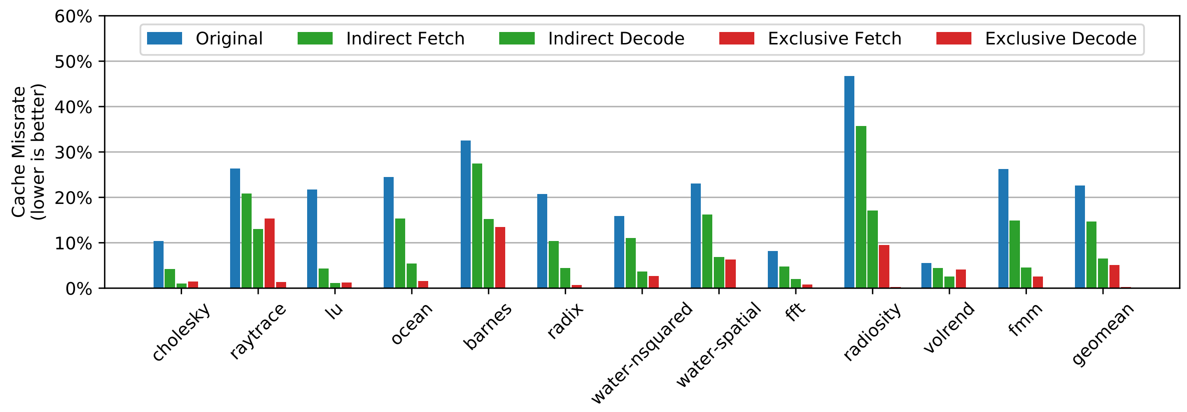
The multi-core schemes were evaluated using two configurations: 8-core, and 128-core. In the 8-core configuration, the multi-core schemes match the memory budget of the *Original* scheme. By sharing *decode objects* between 8 cores, the memory budget allowed multi-core schemes to allocate 4× more *fetch* entries, and more than 4× more *decode* entries before the memory budget was exhausted. This increased cache size significantly improved cache performance, resulting in most binaries in our evaluation achieving near zero miss rates. As increasing cache sizes further would not significantly improve performance, the 128-core configuration is evaluated using the same cache size configuration as the 8-core simulation. As a result, the multi-core schemes require significantly less memory than the *Original* scheme, whilst still achieving significantly better cache performance. The exact memory allocation is given in Table 1.

Furthermore, 128-core simulation evaluates the effect of host thread contention. Since the host machine used for experiments provides only 10 physical cores, simulating 128 virtual cores results in up to 13 host threads sharing a physical core.

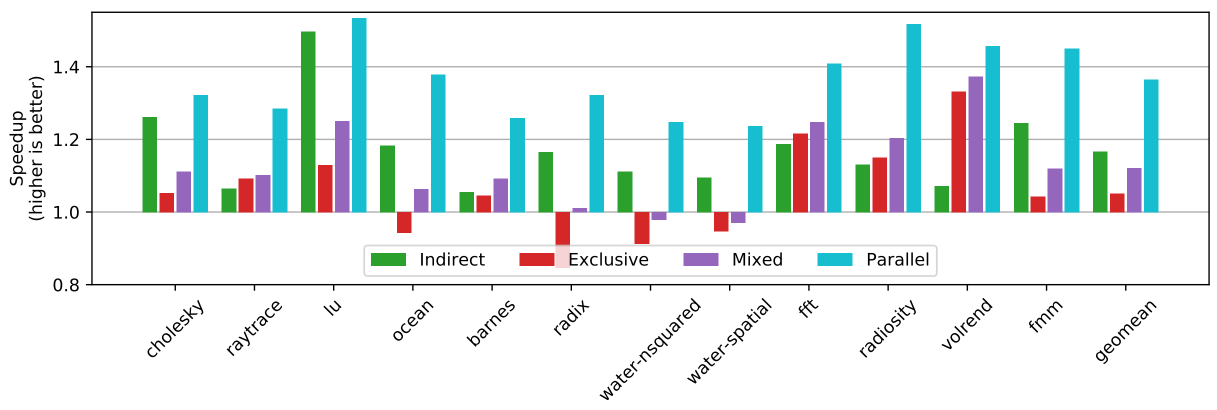
3.2 Key Results

The novel multi-core schemes vastly improve cache performance (Figure 5a), reaching average miss rates of 5.07% and 0.15% for fetch and decode caches respectively. In comparison, the *Indirect* scheme resulted in average miss rates of 14.66% and 6.51% for the fetch and decode caches, whilst the *Original* scheme's miss rate of its unified cache was 22.62%.

In terms of runtime performance, the *Exclusive* and *Mixed* schemes result in speedups of 1.05× and 1.12× respectively, for 8-core configuration. Although the schemes are sharing memory between cores, the synchronization overhead negates the benefits of improved cache performance. In fact, both schemes are often outperformed by the *Indirect* scheme, which does not provide any multi-core sharing but does not require any synchronization.



(a) Cache performance in the 8-core configuration. The multi-core *Exclusive* scheme reduces the miss rate more effectively than the best single-core *Indirect* scheme.



(b) Speedup of the novel multi-core schemes relative to the *original* scheme in the 8-core configuration. In this configuration, there are enough physical host cores to run every simulated core, resulting in no thread contention on the host.

Figure 5. Multi-core schemes evaluation.

The *Parallel* scheme achieves the best performance, with speed-ups of up to 1.53×, and an average speedup of 1.36× over the *Original* scheme. This outstanding performance is achieved due to the low synchronization overhead and high cache performance as a result of multi-core *decode object* sharing. In fact, the *Parallel* scheme is the best performing scheme for all benchmarks.

The *Mixed* scheme always outperforms the *Exclusive* scheme, because the *Exclusive* scheme acquires a lock to execute every instruction, causing sequential execution of the most contended instructions. On the other hand, the *Mixed* scheme promotes *decode objects* of contended instructions to a small core-private cache, improving execution parallelism. On average, the *Mixed* scheme results in about 35% of instructions executing directly from the private decode cache.

In the 128-core configuration, the *Parallel* scheme continues to dominate with a speed-up over the *Original* scheme of up to 1.57× and 1.31× on average (Figure 6). The `raytrace` benchmark is a notable exception to this trend, as the *Exclusive* scheme outperforms the *Parallel* scheme in this instance. Due to the large instruction footprint of `raytrace`, the *Parallel* scheme reaches the memory limit more than 600 times during the execution, and has to perform the costly *stop-the-world* reset. Simulating 128 virtual cores makes this reset particularly costly, resulting in diminished runtime performance. Note, the *Parallel* scheme still outperforms the *Original* scheme. Interestingly, the *Mixed* scheme results in worse runtime performance than the *Exclusive* scheme for simulated core counts that exceed the number of physical host cores. This degradation occurs due to increased pressure on the host operating system when scheduling the threads that run the virtual cores. The *Mixed* scheme is designed to keep the simulation running, by reducing contention on the locks that manage access to shared decoded instruction objects. However, in certain workloads guest worker threads use their own synchronization primitives to wait for the initialization thread to complete, resulting in those

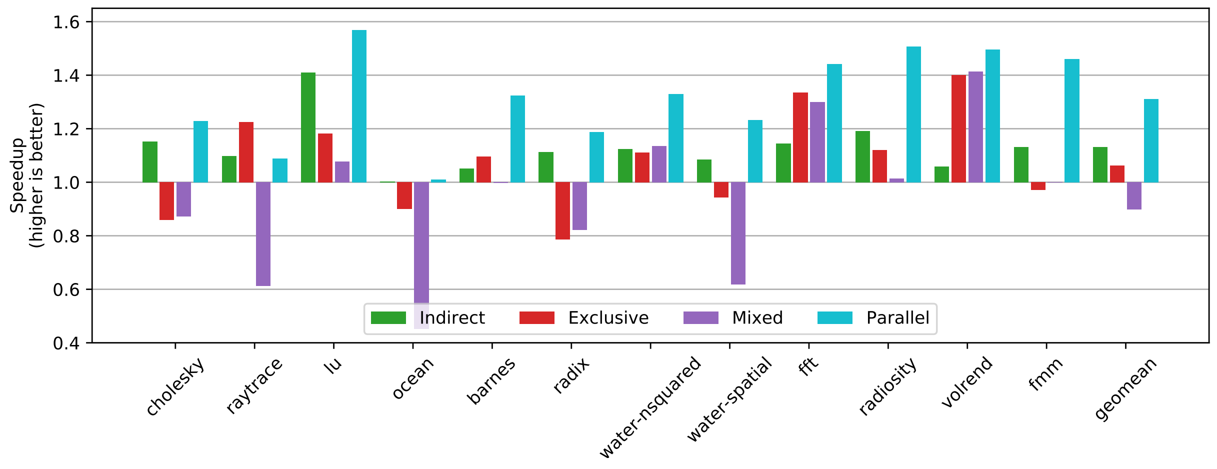


Figure 6. Speedup of the novel multi-core schemes relative to the *original* scheme, in the 128-core configuration. This configuration simulates guest cores using 128 threads on a 10-core machine, resulting in significant host thread contention.

worker threads running faster, and creating host Operating System (OS) thread scheduling contention that ultimately slows down the guest initialization thread.

3.3 Critical Evaluation

Our novel schemes have showed varied performance benefits for different benchmarks. For example, benchmarks with low instruction footprints do not benefit from improved cache performance, but rather suffer a slowdown due to the additional overheads incurred in the novel schemes. For this class of application, an adaptive scheme could monitor the current cache performance and switch between decode cache schemes dynamically to choose an optimal caching strategy, given a constrained memory budget. Dynamic scheme switching might be particularly beneficial to applications with phased behaviour, as different application phases are likely to exhibit different instruction pressure.

Similar adaptations can also be explored in multi-core simulation. For example, an application can be initialized using our low-synchronization *Parallel* scheme, and switch to the *Exclusive* scheme dynamically if too many *stop-the-world* resets are observed. Later, if increasing instruction contention is observed, the decode caching scheme can be changed again to the *Mixed* scheme.

Another limitation of the multi-core scheme is homogeneity, i.e. each virtual core is exposed to the same shared decode cache hierarchy. This might be a sub-optimal memory allocation for task-parallel applications as each task might have different instruction footprints. Even in data-parallel applications, execution often comprises a sequential initialization phase with a high-instruction footprint, and a parallel phase with a smaller kernel. Increasing the memory budget, or making the decode cache completely private for the application thread performing initialization, might significantly speed up the overall application.

4 Related Work

Whilst Instruction Set Simulator (ISS) and execution strategies have found ample attention among researchers and industrial developers, the specifics of caching of decoded instructions in ISSs are less well documented. It is known that interpreter-based ISSs have employed decode caching for a long time [8, 9, 14]. The OpenRISC 1000 repository [1] features a current example of an ISS that employs traditional decode caching. The Talisman ISS uses decoded instruction pages that contain slots for decoded instructions [4]. Each decoded instruction page corresponds to a physical page of a particular node's memory. Decoded instruction pages are allocated when a running program attempts to execute code on a physical page that does not yet have a corresponding decoded page. Both physical page structures and decoded instruction page structures are allocated lazily. This means, in principle, the Talisman decode cache relies on the address, i.e. the PC for indexing its decode cache. In Chen et al. [7] a hardware approach for reducing interpretation overhead has been developed. This work introduces a unified fetch and decode cache in hardware with up to 32 KiB size. It employs a simple decode object, where each entry only stores a pointer to a dispatch function, and two register/immediate fields with altogether 12 bytes. While effective, this hardware support for interpreted ISS is not available in any commercial processors. A design for a decode cache, which is based on the largest jump block, is developed in Xiao et al. [23]. Using this strategy, the ISS can effectively adapt its decode cache size at runtime. Stripf et al. [21] reduces the overhead of decode cache lookups by linking *decode instruction objects*, relying on the fact that for non-branch instructions, the following instruction is always identical. Lv et al. [16] only caches the results of the decode stage, requiring a fetch operation for every instruction. However, the decode cache is still indexed using PC, resulting in the same *instruction object* duplication as a unified cache. Trade-offs for decode caches in ISSs are discussed extensively in Balderas-Contreras [3], Jones [12], and a hybrid approach is shown in Reshadi et al. [20].

5 Conclusion

In this paper, we have demonstrated that traditional decode cache approaches to interpreter-based ISS under-utilize the memory allocated to them, due to duplication of *decoded instruction objects*. Our novel multi-core schemes improve memory utilization by introducing a novel cache data structure indexed by instruction encoding, and removing duplication using a globally shared decode cache. In an 8-core simulation, our novel schemes improved cache performance significantly, resulting in less that 0.5% of instructions requiring decoding, while using the same amount of memory as the traditional cache. Runtime speedup achieved up to 1.53× and 1.36× on average. In 128-core simulation, speed-ups of up to 1.57× and 1.31× on average were observed, while using only 27% of memory relative to a traditional decode cache.

References

[1] [n. d.]. OpenRISC 1000 Instruction Set Simulator (or1kiss). https://github.com/janweinstock/or1kiss.

[2] L. Albertsson. 2006. Holistic Debugging – Enabling Instruction Set Simulation for Software Quality Assurance. In *14th IEEE International Symposium on Modeling, Analysis, and Simulation*. 96–103. doi:10.1109/MASCOTS.2006.26

[3] Tomás Balderas-Contreras. 2000. Interpretive and Non-interpretive Techniques for Instruction-Set Simulation. In *Proceedings of the Sixth Conference on Electrical Engineering (CIE00)*.

[4] Robert C. Bedichek. 1995. Talisman: Fast and Accurate Multicomputer Simulation. In *Proceedings of the 1995 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems* (Ottawa, Ontario, Canada) *(SIGMETRICS '95/PERFORMANCE '95)*. Association for Computing Machinery, New York, NY, USA, 14–24. doi:10.1145/223587.223589

[5] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator.. In *USENIX annual technical conference, FREENIX Track*, Vol. 41. California, USA, 46.

[6] G. Braun, A. Nohl, A. Hoffmann, O. Schliebusch, R. Leupers, and H. Meyr. 2004. A universal technique for fast and flexible instruction-set architecture simulation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 23, 12 (2004), 1625–1639. doi:10.1109/TCAD.2004.836734

[7] Wei Chen, Zhiying Wang, Hongyi Lu, Li Shen, Nong Xiao, and Zhong Zheng. 2009. A Hardware Approach for Reducing Interpretation Overhead. In *2009 Ninth IEEE International Conference on Computer and Information Technology*, Vol. 1. 98–103. doi:10.1109/CIT.2009.104

[8] Bill Clarke, Adam Czezowski, and Peter Strazdins. 2002. Implementation Aspects of a SPARC V9 Complete Machine Simulator. In *Proceedings of the Twenty-Fifth Australasian Conference on Computer Science - Volume 4* (Melbourne, Victoria, Australia) *(ACSC '02)*. Australian Computer Society, Inc., AUS, 23–32.

[9] Bob Cmelik and David Keppel. 1994. Shade: A Fast Instruction-Set Simulator for Execution Profiling. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Nashville, Tennessee, USA) *(SIGMETRICS '94)*. Association for Computing Machinery, New York, NY, USA, 128–137. doi:10.1145/183018.183032

[10] Stefan Farfeleder, Andreas Krall, and Nigel Horspool. 2007. Ultra fast cycle-accurate compiled emulation of inorder pipelined architectures. *Journal of Systems Architecture* 53, 8 (2007), 501–510. doi:10.1016/j.sysarc.2006.11.003 Architectures, Modeling, and Simulation for Embedded Processors.

[11] A. Ghosh, M. Bershteyn, R. Casley, C. Chien, A. Jain, M. Lipsie, D. Tarrodaychik, and O. Yamamoto. 1995. A hardware-software co-simulator for embedded system design and debugging. In *Proceedings of ASP-DAC'95/CHDL'95/VLSI'95 with EDA Technofair*. 155–164. doi:10.1109/ASPDAC.1995.486217

[12] Daniel Jones. 2010. High speed simulation of microprocessor systems using LTU dynamic binary translation.

[13] Daniel Jones and Nigel Topham. 2009. High Speed CPU Simulation Using LTU Dynamic Binary Translation. In *High Performance Embedded Architectures and Compilers*, André Seznec, Joel Emer, Michael O'Boyle, Margaret Martonosi, and Theo Ungerer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 50–64.

[14] Rola Kassem, Mikaïl Briday, Jean-Luc Bêchennec, Yvon Trinquet, and Guillaume Savaton. 2010. Instruction Set Simulator Generation Using HARMLESS, a New Hardware Architecture Description Language. ICST. doi:10.4108/ICST.SIMUTOOLS2009.5643

[15] Taj Muhammad Khan. 2011. *Processor design-space exploration through fast simulation. (Exploration de l'espace de conception de processeurs via simulation accélérée)*. Ph. D. Dissertation. University of Paris-Sud, Orsay, France. https://tel.archives-ouvertes.fr/tel-00691175

[16] Mingsong Lv, Qingxu Deng, Nan Guan, Yaming Xie, and Ge Yu. 2008. ARMISS: An Instruction Set Simulator for the ARM Architecture. In *2008 International Conference on Embedded Software and Systems*. 548–555. doi:10.1109/ICESS.2008.73

[17] P.S. Magnusson. 1997. Efficient Instruction Cache Simulation And Execution Profiling With A Threaded-code Interpreter. In *Winter Simulation Conference Proceedings,*. 1093–1100. doi:10.1145/268437.268745

[18] Daniel Mueller-Gritschneder, Martin Dittrich, Marc Greim, Keerthikumara Devarajegowda, Wolfgang Ecker, and Ulf Schlichtmann. 2017. The Extendable Translating Instruction Set Simulator (ETISS) Interlinked with an MDA Framework for Fast RISC Prototyping. In *2017 International Symposium on Rapid System Prototyping (RSP)*. 79–84.

[19] Mehrdad Reshadi, Nikhil Bansal, Prabhat Mishra, and N. Dutt. 2003. An efficient retargetable framework for instruction-set simulation. In *First IEEE/ACM/IFIP International Conference on Hardware/ Software Codesign and Systems Synthesis (IEEE Cat. No.03TH8721)*. 13–18. doi:10.1109/CODESS.2003.1275249

[20] Mehrdad Reshadi, Prabhat Mishra, and Nikil Dutt. 2003. Instruction Set Compiled Simulation: A Technique for Fast and Flexible Instruction Set Simulation. In *Proceedings of the 40th Annual Design Automation Conference* (Anaheim, CA, USA) *(DAC '03)*. Association for Computing Machinery, New York, NY, USA, 758–763. doi:10.1145/775832.776026

[21] Timo Stripf, Ralf Koenig, and Juergen Becker. 2012. A cycle-approximate, mixed-ISA simulator for the KAHRISMA architecture. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 21–26.

[22] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 programs: Characterization and methodological considerations. *ACM SIGARCH computer architecture news* 23, 2 (1995), 24–36.

[23] Quan Xiao, Jian Wu, Zhuo Lin, Li Kong, Jing Liu, Lei Deng, Shuyu Li, Tao Zhang, and Fangquan Lin. 2012. The study of binary decoding cache for instruction-set simulation. In *International Conference on Automatic Control and Artificial Intelligence (ACAI 2012)*. 2187–2190. doi:10.1049/cp.2012.1433

ARCADES : A RISC-V-Coupled Accelerator for Discrete Event Simulation

Nils Wilbert, Derek Christ, Timo Grundheber and Matthias Jung

Return to Schedule

ARCADES: A RISC-V-Coupled Accelerator for Discrete Event Simulation

Nils Wilbert
University of Würzburg
Würzburg, Germany
nils.wilbert@uni-wuerzburg.de

Derek Christ
University of Würzburg
Würzburg, Germany
derek.christ@uni-wuerzburg.de

Timo Grundheber
University of Würzburg
Würzburg, Germany
timo.grundheber@stud-mail.uni-wuerzburg.de

Matthias Jung
University of Würzburg
Würzburg, Germany
Fraunhofer IESE
Kaiserslautern, Germany
m.jung@uni-wuerzburg.de

Abstract

Discrete Event Simulations (DES) are one of the most widespread approaches to model complex computer systems and are thus crucial for virtual prototyping. In this paper, we consequently propose **A** **R**ISC-V-**C**oupled **A**ccelerator for **DES** (ARCADES). We realize this by leveraging RISC-V extensions to allow for control of the accelerator, as well as implementing our custom event queue and process queue in hardware. We evaluate the accelerator using an FPGA implementation. Our experiments show that in event-heavy scenarios compared to a software implementation, speedups of up to 2.05× can be achieved. Owing to our accelerator being tightly coupled with a PicoRV32 RISC-V processor, we can achieve a high level of configurability and extendability, with ARCADES intended to serve as the base for efficient ASIC DES accelerators.

CCS Concepts

• **Hardware** → *Hardware accelerators*; • **Computing methodologies** → *Modeling and simulation.*

Keywords

DES, RISC-V, Hardware Accelerator, FPGA Implementation

ACM Reference Format:
Nils Wilbert, Derek Christ, Timo Grundheber, and Matthias Jung. 2026. ARCADES: A RISC-V-Coupled Accelerator for Discrete Event Simulation. In *Rapid Simulation and Performance Evaluation for Design (RAPIDO '26), January 27, 2026, Krakow, Poland.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3787501.3787502

1 Introduction

With transistor scaling facing physical and economical limits, single-core performance of general-purpose CPUs has also started to stagnate, marking the beginning of the post-Moore era [9, 41]. Moving forward, experts thus suggest the direction of domain-specific architectures to achieve further performance gains [14]. Simultaneously, with increasingly complex systems and higher time to markets, simulations are becoming increasingly important for virtual prototyping in order to, e.g., verify electronic circuits [15, 24]. Concerning the field of simulations, the Discrete Event Simulation (DES) [25, 38] is one of the most widespread simulation models, ranging far beyond the field of computer science [29, 39]. Here, updates to the system state are only performed at distinct points in time where an event is scheduled, thus skipping inactive periods until the next scheduled event. For example, gem5 [23] one of the most widely used full-system simulators, is considered to be a discrete-event simulator. Therefore, accelerators for DES are of natural interest. In this paper, we thus present ARCADES, a DES hardware accelerator by extending a RISC-V core with application-specific event queue, process queue and kernel implementations. The accelerator is implemented on an FPGA using a Verilog description and evaluated against a software implementation. In this paper we make the following contributions:

- To the best of our knowledge, we propose the first DES accelerator to use a tightly coupled RISC-V core, thus profiting from RISC-V's lightweightness and extendability, creating custom event queue implementations in the process.
- We implement ARCADES on an AMD/Xilinx XCZU5EV-SFVC784-1-E FPGA, providing a prototype and demonstrating its feasibility.
- We evaluate the speedup compared to a software-based DES as well as make area assessments for two different event queue variants in real hardware experiments.

The paper is structured as follows:

The necessary background for this paper is presented in Section 2 with related work being outlined in Section 2.3. The implementation of ARCADES is presented in Section 3 and evaluated in Section 4. An outlook with concluding notes is provided in Section 5.

2 Background

In this section, we outline a number of fundamentals on DES and RISC-V as well as presenting the related work.

This work is licensed under a Creative Commons Attribution 4.0 International License.
RAPIDO '26, Krakow, Poland
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2287-5/2026/01
https://doi.org/10.1145/3787501.3787502

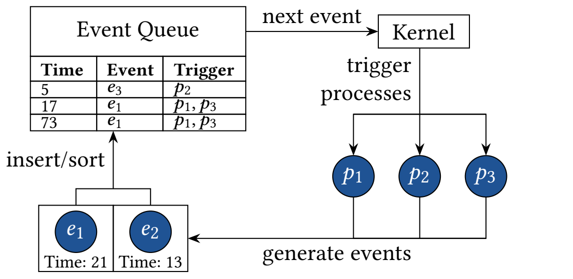


Figure 1: General concept of a discrete event simulation.

2.1 Discrete Event Simulation

The general concept of DES can be described as seen in Figure 1. A simulation starts with a set of initial events in an event queue. Events are scheduled at a specific point in time and, once triggered, can prompt processes that may change the system state as well as schedule new events. Processes are generally described as being sensitive to selected events. When multiple processes are sensitive to the same event, the execution is assumed to be performed concurrently. However, the DES serializes this concurrency using δ-cycles as an infinitesimally small unit of time between, i.e., signal assignments until a stable state is reached. The purpose of the kernel is to coordinate the simulation timeline by retrieving the next scheduled event from the event queue, advancing the simulation time to match the event's timestamp and triggering the execution of the processes sensitive to the event. Note that the order in which events are inserted into the event queue does not necessarily match the order in which they are scheduled. Therefore, the event queue requires a time-sorting mechanism to ensure the kernel retrieves the correct event. The simulation ends once the event queue is empty or a predefined maximum simulation time is exceeded.

By running multiple simulations that interact with one another through messages, the DES approach can be further extended to a Parallel Discrete Event Simulation (PDES) [13], requiring careful consideration to balance the cost and performance advantage when adding processors [36].

2.2 RISC-V

RISC-V is a free and open standard for an Instruction Set Architecture (ISA), particularly popular in the embedded area, as microprocessors can be developed and manufactured with no additional royalty fees [33]. Unlike, e.g., x86, RISC-V allows for developers to create their own custom ISA extensions [8].

2.2.1 ISA Extensions. Concerning the instruction set, RISC-V allows users to select a base instruction set with either 32-bit (RV32) or 64-bit (RV64) registers. RV32 is further split into RV32I and RV32E variants, featuring 32 and 16 registers, respectively. In this paper, we employ RV32I. Moreover, RISC-V offers a number of ratified ISA extensions for, e.g., floating-point operations (F) or atomic instructions (A). For this work, we make use of the M extension, which extends the RISC-V ISA with instructions for integer multiplication and division.

As seen in Figure 2, RISC-V base instructions follow one of six different instruction formats, for, e.g., J-Type jump operations or R-type instructions, where registers serve as both operands and destination. In the R-Type instructions, along with the *opcode*, the *funct7* and *funct3* fields are used to specify the operation to be performed.

To extend RISC-V, four opcodes are dedicated to serving as custom opcodes, with the *funct* fields allowing for further distinctions between custom instructions. After defining a new instruction that naturally has to adhere to one of RISC-V's instruction formats, the RISC-V toolchain [2] has to be recompiled in order to make the assembler aware of the new instruction.

A workaround method for RISC-V extensions while avoiding adjustments to the RISC-V toolchain can be realized by mapping custom instructions to the I-Type *ebreak* instruction. The *ebreak* instruction is conventionally used in order to receive control to a debugging environment. By modifying only the *imm* field, custom instructions can also be introduced, with the processor halting the current execution when encountering an *ebreak*.

2.2.2 PicoRV32 Core. The PicoRV32 is an open-source, compact RISC-V processor core that implements the RV32IM instruction set [42]. One standout feature of PicoRV32 is the Pico Co-Processor Interface (PCPI), which enables direct communication between custom modules and the CPU registers This makes it possible to extend processor functionality by offloading specialized tasks to external hardware accelerators without modifying the core [37]. Furthermore, works have shown that the PicoRV32 is possible to implement using only open-source design flows [22] For this paper, we thus chose a PicoRV32 as the host processor for our accelerator.

2.3 Related Work

The concept of creating dedicated machines to accelerate simulations ranges all the way back to 1982 with the Yorktown Simulation Engine accelerating gate-level simulations [31]. More recently, FPGA implementations to accelerate specific types of simulations, such as RTL [21], Monte Carlo [3] or accelerating neural networks [4, 35] are widely popular topics, with our paper contributing in the field of DES. Furthermore, approaches such as FAST [6] even suggest new simulation methodologies by using hardware accelerators for specific simulation tasks, such as handling the timing model.

Concerning the field of DES, most notably, in [32], the authors propose PDES-A, a parallelized DES accelerator implemented on an FPGA. In contrast to our approach, the authors make use of an x86 Intel Xeon-based coprocessor, while we opt for RISC-V, promising a more lightweight and open design. Additionally, since RISC-V is highly flexible, we thus propose an accelerator that is both easier to configure and extend. We further suggest a more area-efficient event queue implementation compared to PDES-A.

Moreover, in [20] the authors present DES acceleration by offloading calculations for simulation time advancement to network switches.

Using RISC-V has further proved to be a suitable base for the development of hardware accelerators, as seen in works such as [10, 28, 34]. Beyond DES, approaches such as MEG [43] present a RISC-V-based full-system emulation infrastructure using FPGAs.

One of the most popular methods to describe complex hardware software systems is to use SystemC [1]. The simulation of these

31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11|19:12]				rd	opcode	J-type

Figure 2: RISC-V base instruction formats [44].

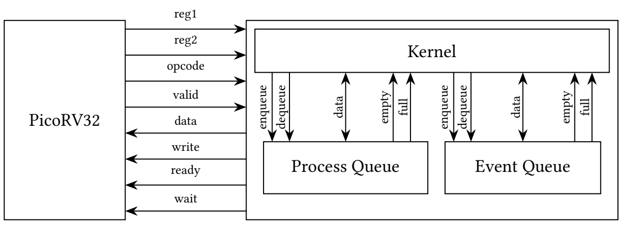


Figure 3: DES accelerator with PicoRV32 processor overview.

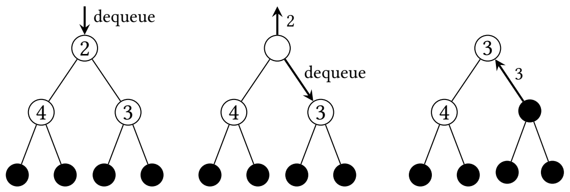


Figure 4: P-heap dequeue operation. Leaf nodes are colored in black.

components using SystemC's Transaction Level Modeling can be understood as a DES [11, 40]. Using SystemC for simulation further allows for temporal decoupling, thus no longer restricting simulation components to the global simulation time, requiring careful to maintain accuracy [19].

3 Accelerator Implementation

ARCADES can be described using the following components: an event queue, containing the ordered sequence of scheduled events; a process queue, containing the processes to be executed; and a kernel functioning as the control unit between the queues and the PicoRV32 processor. A schematic overview of our design is shown in Figure 3. In this section, we present thie mplementation of the accelerator components, including the coupling to the RISC-V coprocessor.

3.1 Event Queue

When retrieving the next event from the event queue, it is obligatory to keep the correct order in which events are scheduled on the timeline. While there are many types of sorting algorithms implemented in hardware [17], partial sorting is sufficient for our case, as it is solely required to correctly retrieve the event that is scheduled next instead of the entire sequence of events. Data-structure-wise, the event queue can be implemented using a priority queue representing a heap-like or a list-like structure. In the following, we will discuss both implementation options.

In [5], a binary heap called the pipelined-heap (P-Heap) is proposed. A P-heap implemented in hardware also serves as the event queue in [32]. At each node p_i in a P-Heap, the property that all children have a lower priority than p_i must always hold. Retrieving the element with the highest priority is thus as straightforward as taking the root of the tree. Nevertheless, when dequeuing the event at the root, the P-heap needs to keep its property regarding the priority ordering. The child node with the higher priority is thus moved up a level in the heap to fill in the gap left behind by the removed node. This process continues at the next lower level of the heap until a leaf node is reached. For n nodes, there are $O(log(n))$ levels in the P-heap, therefore, the complexity for the dequeue operation is $O(log(n))$, with the highest priority event being retrieved in $O(1)$. Vice versa, when enqueuing an event, starting at the root node, at each level the priority of the new node is compared with the existing node. The node with the lower priority is shifted down towards the appropriate child node until reaching a leaf node, leading to an $O(log(n))$ complexity for the enqueue operation. In our case, the time an event is scheduled serves as the priority. An example for dequeuing the highest priority event in a P-heap can be seen in Figure 4. In comparison, maintaining a fully sorted list of events can again guarantee $O(1)$ complexity for retrieving the highest priority event while leading to an $O(n)$ complexity for the entire dequeue operation, as all entries need to be shifted by one position. Enqueueing an event into an already sorted linked list results in $O(n)$ complexity, as it means potentially traversing over the entire list, starting at the highest priority entry.

For ASIC- or FPGA-based accelerators like ARCADES, these data structures also require a hardware implementation, with priority queues having been shown to be efficiently implementable in hardware [27, 30]. Notably, in [26] the authors compare the area impact of different priority queue implementations, namely binary trees, shift registers, FIFOs and systolic arrays. We build upon this by comparing a heap to a list-based priority queue implementation in the context of a full DES accelerator. Both a linked list and a priority heap structure are synthesized for an AMD/Xilinx XCZU5EV-SFVC784-1-E FPGA using SystemVerilog descriptions. Regarding the hardware implementation, the event queue resembles a systolic array of registers containing the queue entries, where upon enqueueing, entries are propagated through the queue by one position per cycle until reaching their intended position, shifting remaining entries in the process. A visualization of a list-structured hardware implementation can be seen in Figure 5.

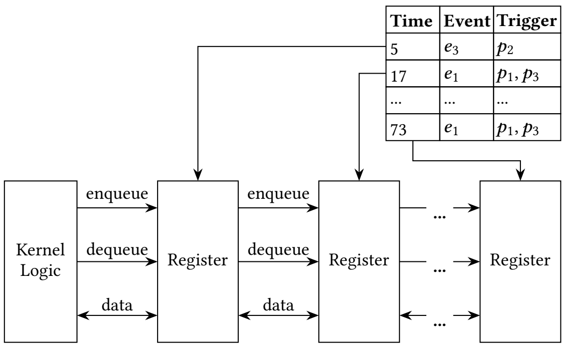


Figure 5: Hardware implementation of the event queue.

Naturally, enqueue and dequeue operations have to be synchronized in a way that we do not erroneously retrieve an event while an enqueue operation with a potentially higher priority is still in progress. However, as we start the enqueue operation at the highest priority node, any newly inserted event is guaranteed to pass the first queue entry in one clock cycle. For both the heap and the list, we can thus safely retrieve the next event in the following clock cycle, even if the new event has not yet reached its eventual position, requiring $O(log(n))$ steps in the heap or $O(n)$ steps in the list implementation. We analyze the impact of both queue implementations in Section 4.

3.2 Process Queue

Concerning the process queue, an implementation via a conventional First-In First-Out (FIFO) implementation is sufficient, as processes do not require ordering, thus allowing for a scalable, lightweight design. Naturally, *empty* and *full* flags are required to indicate the queue state, among the straightforward FIFO *enqueue* and *dequeue* operations.

The main challenge for a hardware-implemented process queue is the limited FIFO depth, which can result in overflows. Once the number of processes to be executed exceeds queue size, a process queue implemented in software can serve as a backup. The *full* flag indicates the overflow, signaling the DES algorithm to revert to the software queue.

3.3 Kernel

To allow the DES to enqueue events and processes into their corresponding queues as well as retrieve the next scheduled event, a kernel using custom RISC-V instruction is implemented. Using inline assembly, developers can make use of our RISC-V directly in C++ code via an interface provided by the kernel. For each of the functions from the C++ interface, one corresponding custom RISC-V instruction is introduced. To be precise, the following interface is provided:

- `static inline void resetKernel()`
- `static inline des::event* getNextEvent()`
- `static inline void putEvent(des::event* e, uint32_t t)`
- `static inline bool eventQueueIsEmpty()`
- `static inline des::proc* getNextProcess()`
- `static inline void putProcess(des::proc* p)`
- `static inline bool processQueueIsEmpty()`

Once one of the aforementioned functions is called by the simulation software, the PicoRV32 will send out a *valid* signal via the PCPI along with the opcode to the kernel, marking the offloading of the DES task to the hardware accelerator. Upon receiving the opcode, the kernel sets the corresponding signals to the hardware queues to trigger the desired operations. In Listing 1, the case for retrieving the next event from the queue is depicted.

```
if (pcpi_valid
 && KERNELCODE == OPCODE_KERNEL) begin
  pcpi_wait <= 1;
  case(OPCODE)
    getNextEvent: begin
      dequeue_event <= 1;
      pcpi_rd <= next_event_adress;
      pcpi_ready <= 1;
      pcpi_wr <= 1;
  end
    insertEvent: begin
      ...
  endcase
end
```

Listing 1: Snippet of SystemVerilog kernel.

Here, the kernel merely sets the `dequeue_event` signal connected to the event queue to active, returning the event queue's output `next_event_adress` via the PCPI back to the PicoRV32, thus completing the communication task.

4 Evaluation

In this section, we outline our evaluation setup regarding the benchmarks and FPGA implementation before presenting the results for our proposed accelerator.

4.1 Setup

Concerning the experiments to evaluate ARCADES, we make use of six different benchmarks. Furthermore, the benchmarks consist of both event-heavy and process-heavy simulations, as depicted in a breakdown of the software simulation time in Table 1. The observation that most of the simulation time is spent inside the simulation kernel aligns with results reported in works such as [12]. Using differently characterized benchmarks, we thus aim to provide a fair assessment of ARCADES in different scenarios. Namely, the benchmarks used are:

- **diningPhil:** A classic synchronization problem where 16 philosophers attempt to pick up forks at random intervals and release them after a random time.
- **phold:** A benchmark maintaining 5 logical processes with a high amount of event-based interprocess communication.
- **prodCons:** A producer generates tokens at each positive clock edge. The consumer consumes the token once it is produced. 5000 tokens are produced in total.
- **counter:** A basic counter that is incremented at every positive clock edge, counting up to 10,000.
- **risc16:** A simulation of a 16-bit RISC processor with 16 words of instruction memory and 8 words of data memory running

Benchmark	Event [%]	Process [%]	Kernel [%]
pipeline	2.63	48.06	49.31
risc16	1.37	39.05	59.58
counter	11.53	35.65	52.82
prodCons	11.09	35.08	53.83
phold	9.09	32.67	58.24
diningPhil	8.75	31.44	59.81

Event[%]: Percentage of simulation time spent handling events.
Process[%]: Percentage of simulation time spent handling processes.
Kernel[%]: Time spent inside kernel, including initialization overhead.

Table 1: Time distribution across specific simulation parts.

a small program with a mix of register, jump and load/store instructions.

- **pipeline:** A 3-stage pipeline where each stage performs simple single-cycle computations, such as addition or subtraction, depending on the previous stage's value. 5000 values are inserted into the pipeline.

For our evaluation, we implement ARCADES on a Genesys ZU-5EV development board, which features an AMD/Xilinx XCZU5EV-SFVC784-1-E FPGA with 5.1 Mbit embedded BRAM, 117,120 LUTs and 234,240 CLB flip-flops [16], with the program code for all benchmarks residing in the BRAM. Using the AMD/Xilinx Vivado design flow, we synthesized and implemented a variant of ARCADES with all DES functionality implemented in software through the PicoRV32 coprocessor, as well as two variants with the accelerator implemented in hardware. The two hardware variants are distinguished by the implementation of the event queue, one implementation resembling a P-heap and one a linked-list structure, as described in Section 3.1. The hardware process and event queues are implemented to hold up to 256 entries each. As a proof of concept for the software-based DES implementation, we resort to a generic lightweight DES description without using SystemC [18].

4.2 Results

Regarding the speedup of the hardware compared to the software implementation, both queue implementations behave equally, as the next event can always be retrieved in one clock cycle and are thus not distinguished in the following experiment. The simulation time using the ARCADES hardware accelerator compared to the software implementation can be seen in Figure 6. Considering the simulation breakdown from Table 1, the event-heavier *counter* and *prodCons* benchmarks achieve the highest speedups of 2.05× and 1.73×, respectively. This reflects the acceleration achievable thanks to the hardware implementation of the event queue.

For the *diningPhil*, *risc16* and *phold* benchmarks, very similar speedups ranging between 1.44× and 1.52× are recorded. As for these benchmarks, kernel time and thus communication overhead is the most dominant part of the simulation time, the FPGA hardware implementation cannot achieve the same speedup as a potential future ASIC implementation, with FPGA interconnects naturally providing a bottleneck. Our hardware queue implementations nevertheless lead to consistent speedups.

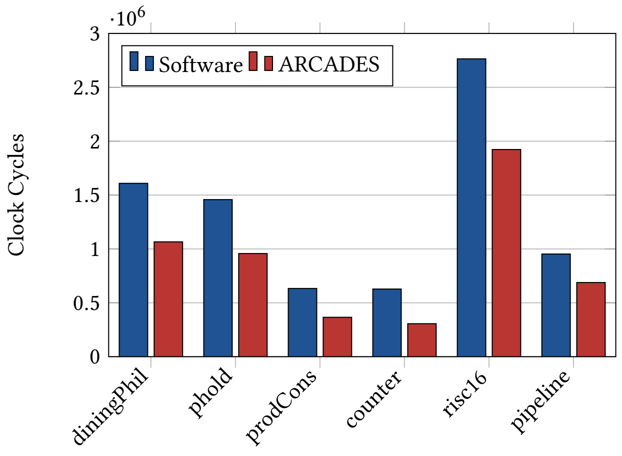


Figure 6: Performance comparison between a software and the ARCADES hardware-based approach.

	BRAM	DSP	IO
Utilization [%]	91.67	0.32	5.16

Table 2: FPGA utilization for the ARCADES implementation.

While still providing a fair 1.38× speedup, for the *pipeline* benchmark the lowest speedup is recorded. Here, kernel time is lower than in the other benchmarks, with event handling also playing a significantly low role. Therefore, for this more process-heavy benchmark, the accelerator can profit less from hardware queues with processes executed on the PicoRV32 in the hardware and software implementation alike.

As we exploit inserting entries into the event queue starting at the highest priority end, for both the P-heap as well as the simpler list implementation, we achieve the same simulation time. While both implementations utilize the same amount of BRAM, DSPs and IO blocks, as listed in Table 2, differences in the utilization of the configurable logic blocks, as depicted in Figure 7, arise, which can serve as an indication for ASIC area assessments.

Notably, for a capacity of 256 entries, the list implementation only requires 53% of the P-heap's LUTs. Although flip-flop utilization for the list queue also increases by 89%, flip-flops require a lot less area compared to LUTs, as for a 5-input LUT, 32 SRAM cells each consisting of 6 transistors are required, whereas D flip-flops are typically implemented using 16 transistors. Furthermore, considering the scalability of the queue implementation, for 512 entries, the state-of-the-art P-Heap implementation LUT utilization already exceeds what is provided by our prototype board, while the list implementations LUT usage remains at about 50%. The simpler list implementation can thus be implemented in hardware using fewer resources while performing better in terms of scalability compared to the P-heap implementation typically found in the literature, while being able to achieve the same speedups when inserting elements at the highest priority end of the queue.

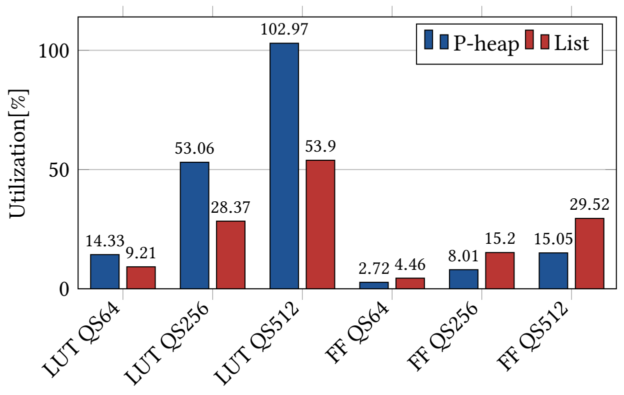


Figure 7: Comparison of the CLB utilization between a P-heap and the simplified list event queue for different Queue Sizes (QS).

5 Conclusion and Outlook

In this paper, we have presented ARCADES, a hardware accelerator for DES tightly coupled to a RISC-V host processor. Our results have shown speedups of up to 2.05× in event-heavy scenarios, whereas even in process-heavy scenarios, speedups of at least 1.39× were observed, demonstrating the accelerator's potential for differently characterized DES. We further outlined the resource utilization of ARCADES on an FPGA, resorting to a simple list-structured event queue, which can retrieve the next scheduled event at $O(1)$ complexity. ARCADES can thus serve as the base for accelerating more complex DES, such as full-system simulations, with the combination of software optimizations to simulators such as gem5 [7] and a hardware accelerator being a direction worth exploring. Furthermore, due to the RISC-V-based nature, ARCADES promises a more lightweight and performance-oriented implementation compared to, e.g., x86-based accelerators, as well as allowing for easier configuration with custom kernel implementations or additional kernel functionality extensions.

Naturally, in terms of execution speed, our FPGA implementation is not able to compete against a DES on high-end general-purpose CPUs. Instead, we envision ARCADES on an FPGA as a prototype for potential ASIC implementations of a DES accelerator. For future work, we further intend to parallelize ARCADES using multiple PicoRV32 coprocessors, promising greater speedups.

Acknowledgments

This work has been funded in part by the European Union via the European Defence Fund project ARCHYTAS under grant number 101167870. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Commission. Neither the European Union nor the granting authority can be held responsible for them.
This work has been funded in part by the German Federal Ministry of Research, Technology and Space (BMFTR) under grant number 16ME0934K (DI-DERAMSys)

References

[1] 2023. IEEE Standard for Standard SystemC® Language Reference Manual. *IEEE Std 1666-2023 (Revision of IEEE Std 1666-2011)* (2023), 1–618. https://doi.org/10.1109/IEEESTD.2023.10246125
[2] 2025. RISC-V GNU Compiler Toolchain. https://github.com/riscv-collab/riscv-gnu-toolchain [Online; accessed 10. Nov. 2025].
[3] Marco Barbone, A Howard, A Tapper, D Chen, M Novak, and W Luk. 2023. Demonstration of FPGA acceleration of monte carlo simulation. In *Journal of Physics: Conference Series*, Vol. 2438. IOP Publishing, 012023.
[4] Suhail Basalama, Atefeh Sohrabizadeh, Jie Wang, Licheng Guo, and Jason Cong. 2023. FlexCNN: An End-to-end Framework for Composing CNN Accelerators on FPGA. *ACM Trans. Reconfigurable Technol. Syst.* 16, 2, Article 23 (March 2023), 32 pages. https://doi.org/10.1145/3570928
[5] R. Bhagwan and B. Lin. 2000. Fast and scalable priority queue architecture for high-speed network switches. In *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No.00CH37064)*, Vol. 2. 538–547 vol.2. https://doi.org/10.1109/INFCOM.2000.832227
[6] Derek Chiou, Dam Sunwoo, Joonsoo Kim, Nikhil A. Patil, William Reinhart, Darrel Eric Johnson, Jebediah Keefe, and Hari Angepat. 2007. FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 40)*. IEEE Computer Society, USA, 249–261.
[7] José Cubero-Cascante, Niko Zurstraßen, Jörn Nöller, Rainer Leupers, and Jan Moritz Joseph. 2023. parti-gem5: gem5's Timing Mode Parallelised. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, Cristina Silvano, Christian Pilato, and Marc Reichenbach (Eds.). Springer Nature Switzerland, Cham, 177–192.
[8] Enfang Cui, Tianzheng Li, and Qian Wei. 2023. RISC-V Instruction Set Architecture Extensions: A Survey. *IEEE Access* 11 (2023), 24696–24711. https://doi.org/10.1109/ACCESS.2023.3246491
[9] Vincenzo De Maio, Atakan Aral, and Ivona Brandic. 2022. A Roadmap To Post-Moore Era for Distributed Systems. In *Proceedings of the 2022 Workshop on Advanced Tools, Programming Languages, and PLatforms for Implementing and Evaluating Algorithms for Distributed Systems* (Salerno, Italy) *(ApPLIED '22)*. Association for Computing Machinery, New York, NY, USA, 30–34. https://doi.org/10.1145/3524053.3542747
[10] Alessandra Dolmeta, Mattia Mirigaldi, Maurizio Martina, and Guido Masera. 2023. Implementation and integration of Keccak accelerator on RISC-V for CRYSTALS-Kyber. In *Proceedings of the 20th ACM International Conference on Computing Frontiers* (Bologna, Italy) *(CF '23)*. Association for Computing Machinery, New York, NY, USA, 381–382. https://doi.org/10.1145/3587135.3591432
[11] Rainer Dömer, Zhongqi Cheng, Daniel Mendoza, and Emad Arasteh. 2021. *Pushing the Limits of Parallel Discrete Event Simulation for SystemC.* Springer International Publishing, Cham, 97–105. https://doi.org/10.1007/978-3-030-47487-4_7
[12] Liana Duenha, Rodolfo Azevedo, and RJ de Azevedo. 2012. Profiling High Level Abstraction Simulators of Multiprocessor Systems. In *Proceedings of the Second Workshop on Circuits and Systems Design-WCAS*.
[13] Richard Fujimoto. 2015. Parallel and distributed simulation. In *Proceedings of the 2015 Winter Simulation Conference* (Huntington Beach, California) *(WSC '15)*. IEEE Press, 45–59.
[14] John L. Hennessy and David A. Patterson. 2019. A new golden age for computer architecture. *Commun. ACM* 62, 2 (Jan. 2019), 48–60. https://doi.org/10.1145/3282307
[15] Vladimir Herdt and Rolf Drechsler. 2022. Advanced virtual prototyping for cyber-physical systems using RISC-V: implementation, verification and challenges. *Science China Information Sciences* 65, 1 (2022), 110201. https://doi.org/10.1007/s11432-020-3308-4
[16] Advanced Micro Devices Inc. 2025. Zynq UltraScale+ MPSoC Data Sheet: Overview. https://docs.amd.com/v/u/en-US/ds891-zynq-ultrascale-plus-overview [Online; accessed 4. Nov. 2025].
[17] Amir Jalilvand, Faeze S. Banitaba, S. Newsha Estiri, Sercan Aygun, and M. Hassan Najafi. 2025. Sorting it out in Hardware: A State-of-the-Art Survey. *ACM Trans. Des. Autom. Electron. Syst.* 30, 4, Article 52 (June 2025), 31 pages. https://doi.org/10.1145/3734797
[18] Matthias Jung. 2021. Discrete Event Simulation Kernel using C++20 . https://github.com/myzinsky/des [Online; accessed 17. Oct. 2025].
[19] Lukas Jünger, Carmine Bianco, Kristof Niederholtmeyer, Dietmar Petras, and Rainer Leupers. 2021. Optimizing Temporal Decoupling using Event Relevance. In *Proceedings of the 26th Asia and South Pacific Design Automation Conference* (Tokyo, Japan) *(ASPDAC '21)*. Association for Computing Machinery, New York, NY, USA, 331–337. https://doi.org/10.1145/3394885.3431419
[20] Osman Volkan Karaca, Kayhan M İmre, and Ali Ziya Alkar. 2023. Network accelerator for parallel discrete event simulations. *The Journal of Supercomputing* 79, 16 (2023), 18728–18747.
[21] Sahand Kashani, Mahyar Emami, Keisuke Kamahori, Sepehr Pourghannad, Ritik Raj, and James R. Larus. 2024. A 475 MHz Manycore FPGA Accelerator for RTL

Simulation. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA '24)*. Association for Computing Machinery, New York, NY, USA, 78–84. https://doi.org/10.1145/3626202.3637579

[22] Mihailo Knežević and Lidija Paunović. 2025. Open-Source IC Design Tools: Implementation of PicoRV32 in 130nm Technology. In *2025 12th International Conference on Electrical, Electronic and Computing Engineering (IcETRAN)*. 1–4. https://doi.org/10.1109/IcETRAN66854.2025.11114230

[23] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jeronimo Castrillon, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Carlos Escuin, Marjan Fariborz, Amin Farmahini-Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Anthony Gutierrez, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Miquel Moreto, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, William Wang, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. 2020. The gem5 Simulator: Version 20.0+. arXiv:2007.03152 [cs.AR] https://arxiv.org/abs/2007.03152

[24] Ricardo Mejía-Gutiérrez and Ricardo Carvajal-Arango. 2017. Design verification through virtual prototyping techniques based on systems engineering. *Research in Engineering Design* 28, 4 (2017), 477–494. https://doi.org/10.1007/s00163-016-0247-y

[25] Jayadev Misra. 1986. Distributed discrete-event simulation. *ACM Computing Surveys (CSUR)* 18, 1 (1986), 39–65.

[26] Sung-Whan Moon, J. Rexford, and K.G. Shin. 2000. Scalable hardware priority queue architectures for high-speed packet switches. *IEEE Trans. Comput.* 49, 11 (2000), 1215–1227. https://doi.org/10.1109/12.895938

[27] Andrew Morton, Jeffrey Liu, and Insop Song. 2007. Efficient Priority-Queue Data Structure for Hardware Implementation. In *2007 International Conference on Field Programmable Logic and Applications*. 476–479. https://doi.org/10.1109/FPL.2007.4380693

[28] Seyed Kian Mousavikia, Erfan Gholizadehazari, Morteza Mousazadeh, and Siddika Berna Ors Yalcin. 2022. Instruction Set Extension of a RiscV Based SoC for Driver Drowsiness Detection. *IEEE Access* 10 (2022), 58151–58162. https://doi.org/10.1109/ACCESS.2022.3177743

[29] Seyed-Esmaeil Moussavi, Evren Sahin, and Fouad Riane. 2024. A discrete event simulation model assessing the impact of using new packaging in an agri-food supply chain. *International Journal of Systems Science: Operations & Logistics* 11, 1 (2024), 2305816. https://doi.org/10.1080/23302674.2024.2305816

[30] Antti Nurmi, Per Lindgren, Tom Szymkowiak, and Timo D. Hämäläinen. 2023. AnTiQ: A Hardware-Accelerated Priority Queue Design with Constant Time Arbitrary Element Removal. In *2023 26th Euromicro Conference on Digital System Design (DSD)*. 462–469. https://doi.org/10.1109/DSD60849.2023.00070

[31] G.F. Pfister. 1982. The Yorktown Simulation Engine: Introduction. In *19th Design Automation Conference*. 51–54. https://doi.org/10.1109/DAC.1982.1585479

[32] Shafiur Rahman, Nael Abu-Ghazaleh, and Walid Najjar. 2017. PDES-A: a Parallel Discrete Event Simulation Accelerator for FPGAs. In *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (Singapore, Republic of Singapore) *(SIGSIM-PADS '17)*. Association for Computing Machinery, New York, NY, USA, 133–144. https://doi.org/10.1145/3064911.3064930

[33] RISC-V International. 2025. The RISC-V Instruction Set Manual Volume I. https://docs.riscv.org/reference/isa/_attachments/riscv-unprivileged.pdf [Online; accessed 22. Oct. 2025].

[34] Alejandra Sanchez-Flores, Lluc Alvarez, and Bartomeu Alorda-Ladaria. 2022. A review of CNN accelerators for embedded systems based on RISC-V. In *2022 IEEE International Conference on Omni-layer Intelligent Systems (COINS)*. 1–6. https://doi.org/10.1109/COINS54846.2022.9855006

[35] Jan Sommer, M. Akif Özkan, Oliver Keszocze, and Jürgen Teich. 2022. Efficient Hardware Acceleration of Sparsely Active Convolutional Spiking Neural Networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 11 (2022), 3767–3778. https://doi.org/10.1109/TCAD.2022.3197512

[36] Yentl Van Tendeloo and Hans Vangheluwe. 2018. An Introduction to Classic DEVS. arXiv:1701.07697 [cs.OH] https://arxiv.org/abs/1701.07697

[37] Dillon Wesley Todd. 2021. *Tightly coupling the PicoRV32 RISC-V processor with custom logic accelerators via a generic interface.* Master's thesis. Clemson University.

[38] Yentl Van Tendeloo and Hans Vangheluwe. 2020. *DEVS: Discrete-Event Modelling and Simulation for Performance Analysis of Resource-Constrained Systems*. Springer International Publishing, Cham, 127–153. https://doi.org/10.1007/978-3-030-43946-0_5

[39] Jesús Isaac Vázquez-Serrano, Rodrigo E. Peimbert-García, and Leopoldo Eduardo Cárdenas-Barrón. 2021. Discrete-Event Simulation Modeling in Healthcare: A Comprehensive Review. *International Journal of Environmental Research and Public Health* 18, 22 (2021). https://doi.org/10.3390/ijerph182212262

[40] Jan Henrik Weinstock, Luis Gabriel Murillo, Rainer Leupers, and Gerd Ascheid. 2016. Parallel SystemC Simulation for ESL Design. *ACM Trans. Embed. Comput. Syst.* 16, 1, Article 27 (Oct. 2016), 25 pages. https://doi.org/10.1145/2987374

[41] Jiahao Xie, Yansong Zhou, Muhammad Faizan, Zewei Li, Tianshu Li, Yuhao Fu, Xinjiang Wang, and Lijun Zhang. 2024. Designing semiconductor materials and devices in the post-Moore era by tackling computational challenges with data-driven strategies. *Nature Computational Science* 4, 5 (2024), 322–333.

[42] YosysHQ. 2024. PicoRV32 - A Size-Optimized RISC-V CPU. https://github.com/YosysHQ/picorv32 [Online; accessed 22. Oct. 2025].

[43] Jialiang Zhang, Yue Zha, Nicholas Beckwith, Bangya Liu, and Jing Li. 2020. MEG: A RISCV-based System Emulation Infrastructure for Near-data Processing Using FPGAs and High-bandwidth Memory. *ACM Trans. Reconfigurable Technol. Syst.* 13, 4, Article 19 (Sept. 2020), 24 pages. https://doi.org/10.1145/3409114

[44] James Zhu. 2024. riscv-card. https://github.com/jameslzhu/riscv-card [Online; accessed 4. Nov. 2025].

A Simulation Methodology for Fast Verification of Cyber-Physical Systems

Maxime Gras-Chevalier, Christophe Jégo, Camille Leroux and Franck Guillemard

Return to Schedule

A Simulation Methodology for Fast Verification of Cyber-Physical Systems

Maxime Gras--Chevalier
University of Bordeaux, Bordeaux INP, IMS Laboratory, UMR CNRS 5218
Bordeaux, France
Stellantis
Poissy, France
maxime.graschevalier@stellantis.com

Christophe Jégo
University of Bordeaux, Bordeaux INP, IMS Laboratory, UMR CNRS 5218
Bordeaux, France
christophe.jego@ims-bordeaux.fr

Camille Leroux
University of Bordeaux, Bordeaux INP, IMS Laboratory, UMR CNRS 5218
Bordeaux, France
camille.leroux@ims-bordeaux.fr

Franck Guillemard
Stellantis
Poissy, France
franck.guillemard@stellantis.com

ABSTRACT

The increasing complexity of cyber-physical systems makes fast simulators essential in the early phases of development. In this paper, a novel approach is proposed to take advantage of a C++-based embedded language (StreamPU) combined with a high speed hardware simulation framework (Verilator). This new methodology is evaluated on three different scenarios featuring control systems. Compared to existing solutions, speed-up factors from ×2.25 to more than ×3000 are achieved with small error margins. The proposed methodology leverages free and open-source tools, making it easily available to the community. Moreover, this approach can be extended to support hardware-in-the-loop simulations.

ACM Reference Format:
Maxime Gras--Chevalier, Christophe Jégo, Camille Leroux, and Franck Guillemard. 2026. A Simulation Methodology for Fast Verification of Cyber-Physical Systems. In *Proceedings of Workshop on Rapid Simulation and Performance Evaluation for Design Optimization: Methods and Tools (RAPIDO'26).* ACM, New York, NY, USA, 6 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Cyber-physical systems combine interfaces, discrete, analogic and physical subsystems, all interacting with each other [1]. They are present all around us [2], as their number increased drastically in a short time, along the innovations in the digital domain. They find extensive usage in the Internet of Things, Smart Cars, Smart Homes, Smart "Anything" [3]. Furthermore, these systems' complexity and heterogeneity rose sharply, and as a consequence, a critical requirement to simulate interactions across many domains appeared. As discussed by Fernández-Mesa et al. in [4], some subsystems require discrete simulators, with fixed time step solvers. Other ones need continuous time equation solvers. Exchanging data between the two domains is no trivial matter, as synchronization problems arise [5], slowing down simulations.

To address this issue, multiple solutions exist. The first one is to add an Analog/Mixed-Signal (AMS) extension to an existing digital system description language. A second approach involves using two different simulators, and exchange information between the tools to co-simulate the discrete and continuous time subsystems. A third solution is to translate models into a different language to ensure that it can be effectively executed by a more efficient simulator.

Current methodologies do not permit high levels of model abstraction, present low simulation speeds because based on interpreted languages, or see their execution performance degraded by data exchanges between multiple tools. Consequently, the necessity becomes evident for a faster simulation alternative. Moreover, during early development cycles and while performing coarse-grain tuning of the studied systems, simulation speed becomes a critical factor. This process often involves a large number of iterations. In addition, it can be necessary to test several sets of parameters with various systems' topologies, once more showcasing the importance of efficient simulation reconfigurability and fast execution.

Borrowing from the fact that SystemC provides good simulation speed gains, the proposed solution makes use of the C++ programming language across the whole simulation environment, in order to leverage the performance benefits of compiled languages. However, for reasons explained in this paper, the proposed approach employs the StreamPU [6] C++ library to benefit from its execution speed and block-based modelling capabilities. This Domain Specific Embedded Language (DSEL) has been chosen despite being initially designed for processing feed-forward data streams of many values. The proposed approach is based on the modules/tasks paradigm from StreamPU to model the complete system. Therefore, continuous time subsystems are implemented in pure C++ tasks. The digital discrete subsystems C++ equivalent code is then generated by Verilator [7] and wrapped in a task in the same manner. It should

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

RAPIDO'26, January 27, 2026, Kraków, Poland
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/XXXXXXX.XXXXXXX

be noted that, both StreamPU and Verilator are open-source, and free.

This study is done in the context of the automotive industry. For this reason, the various simulation scenarios showcased represent systems found in the automotive domain, where cyber-physical systems are pervasive.

In this paper, existing methodologies for the simulation of cyber-physical systems are first presented. Then, a new approach to improve simulation execution time is described. Finally, experiments are detailed, and their results analyzed, to compare the proposed methodology to existing solutions.

2 PREVIOUS WORKS

In this section, the existing solutions to conduct cyber-physical [1] simulations (i.e. part discrete and part continuous time) are introduced. Their limitations are discussed, in order to identify their weaknesses, and propose an adapted alternative approach.

2.1 Verilog-AMS / VHDL-AMS

Verilog-AMS [8] and VHDL-AMS [9] language extensions are aimed at bridging the gap between logic and analog circuit simulations. They are based on the original Verilog and VHDL Hardware Description Languages. As such, most simulators supporting them are initially designed for the base Verilog and VHDL languages. Despite the fact that those extensions are not recent, very few simulators support them, and even less to a good level. Verilator has very limited support for some keywords, and to the author's knowledge, Xilinx's Vivado does not support them at all.

2.2 SystemC / SystemC-AMS

SystemC [10] was introduced to enhance the modelling of whole systems. Verilog and VHDL are restricting and have a level of abstraction too close to the hardware to effectively simulate software / hardware interactions. Moreover, as SystemC is implemented as a set of C++ classes and macros, it is much faster to simulate than traditional HDLs. Like the Verilog and VHDL HDLs, SystemC's specification was later improved with an AMS extension, named SystemC-AMS [11]. This extends the SystemC language capabilities to simulate cyber-physical systems, with both discrete, analog and physical subsystems. This improvement requires a different simulation environment than just compiling a C++ application. Solvers are required for continuous-time equations and can be implemented using a variety of techniques. The SystemC-AMS standard places no restrictions on the choice of method. In practice, few simulation tools support the SystemC's AMS extension. COSEDA Technologies introduced a proof of concept for a C++ SystemC-AMS library [12]. As it is easily available, its performance has been evaluated and its results are compared to the proposed solution's ones in this paper. However, the tool's development seems to have stopped a few years ago. As pointed in [13], ease of extendability also seems to be a weakness of SystemC-AMS. All those points prompted the search for a different solution, leading to the choices presented in this paper.

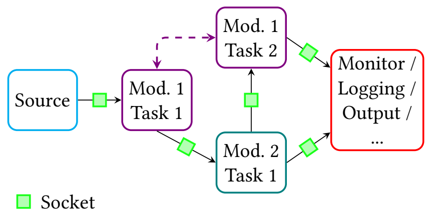


Figure 1: StreamPU typical sequence example

2.3 MATLAB Simulink

MATLAB Simulink is a well-known block-based simulation environment. It features a furnished catalog of extensions, among which the *HDL Verifier* one adds the possibility to co-simulate HDL languages along a traditional Simulink model. To achieve this, the hardware model simulator can be either Cadence's Xcelium, Siemens' Questa Sim / ModelSim, or Xilinx's Vivado. Usually, MATLAB Simulink can leverage its variable time step solvers to greatly speed up simulations. However, in coupled simulations, because of data exchanges between the HDL simulator's discrete domain and Simulink, the time step can rarely be larger than the dynamic of the fastest logic signal, which is often the clock period. It is important to note that obtaining licenses for both MATLAB and the HDL simulator can be a significant financial burden, even more so that all the compatible simulators do not present the same performance levels.

2.4 C++ Code Generation

In parallel, to achieve faster simulation execution, research has developed around translating HDL languages and physical models descriptions into pure C++ code, such as presented by Fraccaroli et al. in [14]. This approach enables good performance gains, by eliminating most of the execution overhead, and mainly keeping only the operations described in the system's model. Moreover, as the target language is compiled, the resulting simulations would tend to be more resource efficient at execution time than would be possible by using an interpreted language.

3 PROPOSED OPEN-SOURCE CO-SIMULATION METHODOLOGY

3.1 StreamPU Project

StreamPU [6] is *a Domain Specific Embedded Language (DSEL) for streaming applications.* It functions by connecting *tasks* together through their *sockets*. The resulting execution graph is called a *sequence*. *Tasks* are contained in *modules*. *Modules* have at least one *task*, or multiple ones sharing the same context, as *modules* are implemented as C++ classes, and *tasks* as C++ methods of those. The tool excels at exchanging data vectors between tasks (i.e. when the sockets' size tends to be higher) in a feed-forward manner. The proposed method, though, must deal with dependencies between each scalar data and feedback loops. The main challenge is to preserve good simulation performance in the particular case of StreamPU. StreamPU simulations are compiled, and the *tasks* execution graph is built only once. This static execution avoids the traditional scheduling overhead of dynamic execution. It is also important to note

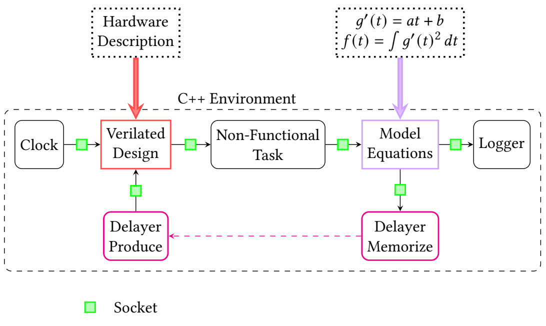


Figure 2: Proposed co-simulation environment

that *sockets* are implemented as a shared memory structure between the connected *tasks*. This means that the overhead introduced by data exchanges is thus minimized.

Designers familiar with block-based modeling tools may find that StreamPU is not that different in its block / *tasks* philosophy, as illustrated in Figure 1. *Blocks* become StreamPU *modules* and *tasks*, and *interconnections* become *sockets*. This approach makes it easy to re-use previously developed functions in various simulation scenarios. StreamPU is for instance used in AFF3CT [15] and as part of the Fast Meteor Detection Toolbox (FMDT) project [16]. Simulation scenarios are described in what is referred as a testbench. This testbench, like the rest of the simulation, is written in the C++ language. Each *task* and its corresponding *module* are implemented in a separate pair of C++ source and header files. Models follow a simple template, which facilitates their reuse.

3.2 Verilator Simulation Tool

Verilator [7] is a fast Verilog simulator. It achieves higher speeds than typical tools by translating the Verilog code into pure C++ instead of involving a simulation engine. Moreover, this lets the compiler optimize the result to the best for the execution environment. Such translated hardware descriptions are said to be *Verilated*. This *Verilated* design is then directly integrated in a StreamPU task, with few additions to interface the model's inputs and outputs to StreamPU's socket logic. As the proposed method imposes a C++ simulation environment, it appears straightforward to adopt Verilator as simulator. Additionally, this enables tight coupling of the hardware design simulation with the C++ environment, limiting costly data exchanges to the strict minimum. Another point is that Verilator is free and open-source. To the authors' knowledge, this tool is accurate in regard to the descriptions when translating them to C++ code, making it a good alternative to commercial computer-aided design tools.

3.3 Proposed Co-Simulation Environment

As presented in Figure 2, the proposed method enables to connect multiple tasks together. This facilitates simulation reconfigurability, and encourages task reuse. The model's mathematical equations are simplified and implemented in C++ in a StreamPU task manually. The task is then connected with other already or newly designed ones. The hardware description is translated to C++ code thanks to Verilator and wrapped into another task. As illustrated in Figure 2, it is connected to other models in the same way, completing the cyber-physical simulation. Notably, this implementation method does not feature an equation solver, enabling performance improvements. This might be restricting for certain use cases. Despite this, no system has been encountered by the authors that cannot be modeled with the proposed approach. It is important to note that all the blocks in the simulation are built on StreamPU's modules and tasks. Tasks directly modeling the scenario (i.e. the verilated design or model equations) are said to be functional tasks. Other tasks added around for convenience or to help run the simulation (i.e. the Clock or Logger blocks for example) are said to be non-functional. Non-functional blocks can be freely inserted in the simulation graph to fulfill various needs.

4 DEVELOPMENT AND EXECUTION OF THE CYBER-PHYSICAL SIMULATION

4.1 Time reference

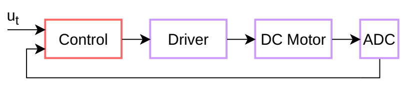
As there is no absolute time concept built into StreamPU and verilated designs, it is necessary to find a way to define it for simulating time-continuous models and real-world scenarios. This is done by specifying a fixed time step as a common time reference across the simulation, also named Δt in this paper. The time step is mainly employed for integration operations in models requiring it. Each module receives it as one of its attributes. In simulations with a discrete subsystem, the time step's value is fixed to the digital clock period. Each simulation step corresponds to the execution of each task of the sequence. Meaning the various models' states are updated for the current time, before advancing to the next step and starting this process again. Applying a single time step across the whole simulation avoids synchronization problems.

4.2 Additional Utility Modules and Functions

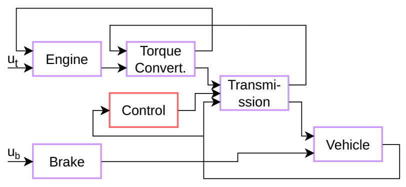
Much like AFF3CT builds on top of StreamPU by adding a set of modules specialized for error correcting codes for digital communications, some utility modules are added to enable cyber-physical systems simulation. A *Logger* module has been added, whose purpose is writing in a file the data passed to its input. The logged data can then be plotted and analyzed by another software. This module is also in charge of data decimation at the logging stage. It means that of all the data passed to it, only some will be written to the file. In a simulation environment, this permits calculation at every step without recording many unnecessary values. Moreover, the module can run the file writing operation on a separate thread, minimizing the impact of logging on simulation speed. A *Clock* module is also included. It provides a point of entry for the simulation graph, and stops execution at the designated time. Despite its name, it does not provide time for the rest of the simulation.

4.3 Discrete Integration

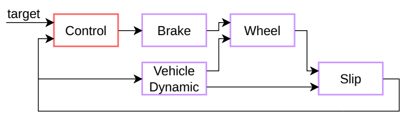
Many models, such as electric motors or electronic capacitors, contain an integral function. As the proposed method does not feature a traditional solver, it becomes mandatory to find a good integration approximation. A simple, fast, yet accurate enough in most cases solution, is the trapezoidal rule [17] to approximate the integral in a discrete manner. This rule states that the integral of a function



(a) DC Motor Speed Control



(b) Automatic Gearbox



(c) Anti Blocking System

Figure 3: Control loop models simulation scenarios

can be estimated by approximating each region under the graph of a function by a trapezoid. Thus, the considered function's discrete integral, with a fixed sampling time Δt and N samples, is expressed as:

$$\int f(t)dt \approx \Delta t \times \left(\frac{f(t_N) - f(t_0)}{2} + \sum_{k=1}^{N-1} f(t_k) \right)$$

This approximation is inexpensive to execute on a computer, as it essentially consists in simple additions, one multiplication and an accumulation. Moreover, it is more accurate than a simpler Euler approximation for a very limited increase in terms of complexity.

5 EXPERIMENTAL EVALUATION

5.1 Simulation Setup

Three simulation use case were investigated. They all feature a discrete time control subsystem alongside other models. For comparison purposes, all simulations are implemented both in the MATLAB Simulink environment, and with the proposed approach. The control subsystems are also implemented in hardware, plain C++ and MATLAB Simulink blocks. The *HDL Verifier* MATLAB toolbox is used to co-simulate the hardware implementations with Xilinx Vivado or Siemens' Questa Sim simulators in Simulink. Hardware implementations are described using the Chisel HDL [18], providing higher-level abstractions than traditional HDLs. The designs are then compiled to Verilog for final implementation. This language also features good parametrization and design configuration capabilities. It enables saving time during implementation thanks to a clear syntax and convenient functions. The experimental setup operated to run the presented simulations is common to the three simulation scenarios. For reading convenience, a few acronyms are defined as follows: [SlD] Simulink with Discrete solver, [SlC] Simulink with Continuous solver, [SlVi] Simulink with Vivado simulator, [SlQs] Simulink with Questa Sim simulator, [Spu] StreamPU, [SpuVe] StreamPU with Verilator. While [SlD], [SlC] and [Spu] are based on equivalent models of the control systems, [SlVi], [SlQs] and [SpuVe] directly simulate the actual hardware implementations through co-simulation (with Vivado or Questa Sim simulators), or HDL translation (with Verilator).

5.2 DC Motor Simulation

As detailed in Figure 3a, the DC motor scenario considers the closed-loop speed control of brushed DC motor. The time step parameter's value is set at 100 ns for the controller's equivalent model simulations, and 10 ns for the hardware implementation. This effectively defines a clock frequency of 100 MHz.

5.2.1 *Proportional Controller.*

The control function is implemented by a proportional controller whose output is calculated with the following equation :

$$cmd = K_p \times (u_t - \omega_{adc}) + 0.5$$

where K_p is the proportional gain value, u_t the target speed, and ω_{adc} the measured speed image through the ADC. Note that the output is raised by 0.5 as the "driver" stage provides a negative voltage under 0.5 and a positive one over it.

5.2.2 *Electric Driver Model.*

The PWM logic signal is converted to a voltage value by the "driver". The mathematical formula to calculate the output voltage v_{drv} from a logical signal with value cmd is expressed as:

$$v_{drv} = cmd \times (V_{max} - V_{min}) + V_{min}$$

with V_{min} and V_{max} being the maximal and minimal voltages to produce depending on the command signal's value.

5.2.3 *Brushed DC Motor Model.*

The selected brushed DC motor model is detailed in [19]. The motor's model parameters were set close to real-life measurements [20], made by the company *Portescap*, to stay in a realistic simulation environment.

5.2.4 *ADC Model.*

The Analog to Digital Converter's model used is the so-called "perfect" one. As such, its equation is as follows:

$$\omega_{adc} = floor\left(\frac{\omega - \omega_{min}}{\omega_{max} - \omega_{min}} \times (2^N - 1) \right)$$

with ω the current speed, ω_{min} and ω_{max} the minimal and maximal speeds supported by the ADC, and N the number of output bits of the ADC. Under MATLAB Simulink, this ideal ADC block is used. The C++ model is equivalent as it is based on the abovementioned equation.

5.3 Gearbox Simulation

As showcased in Figure 3b, this scenario simulates an automatic gearbox model coupled to a vehicle, shifting gears as the speed evolves. The equations are based on the models provided in [21]. The time step parameter's value is set at 1 ms for the controller's equivalent model simulations, and 100 ns for the hardware implementation. This effectively defines a clock frequency of 10 MHz.

Table 1: Simulation error (%)

Scenario	SlD / Spu	SlVi / SpuVe
DC Motor★	1.3	1.0
Gearbox♦	0.79	0.78
ABS▲	0.07	0.03

★: $t \in [1\ ms; 50\ ms]$; ♦: $t \in [10\ ms; 20\ s]$;

▲: $t \in [0\ s; 13\ s]$

Table 2: Simulation duration for each configuration

	Wall-clock time (s)					
Scenario	SlD	SlC*	Spu	SlVi	SlQs	SpuVe
DC Motor	0.69★	0.55	0.06★	19.9♦	54.6♦	0.018♦
Gearbox	0.022▲	0.009	0.004▲	614★	2016★	45★
ABS	0.025▲	-	0.004▲	0.066▲	0.22▲	0.004▲

* Variable Δt; ★: $\Delta t = 100\ ns$; ♦: $\Delta t = 10\ ns$; ▲: $\Delta t = 1\ ms$

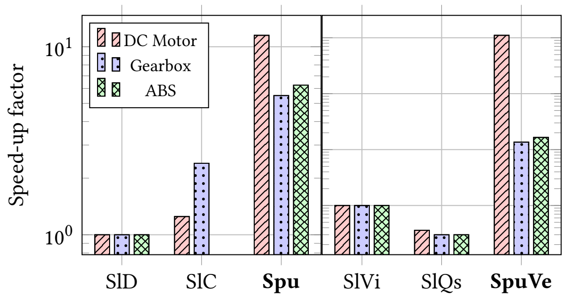


Figure 4: Speed-up factors w.r.t. SlD (left) / SlVi (right)

Table 3: Execution time required to simulate 50 ms

SystemC-AMS	StreamPU
4.0 s	1.2 s

5.4 Anti Blocking System Simulation

As presented in Figure 3c, this scenario simulates a car brake Anti Blocking System (ABS), preventing the wheel from losing adherence and reducing the vehicle's stopping distance. The equations are based on the models provided as an example by MathWorks in [22]. The time step parameter's value is set at 1 ms. This effectively defines a clock frequency of 1 kHz for the simulated hardware implementation.

5.5 Comparison Between SystemC-AMS and StreamPU

In order to compare SystemC-AMS and StreamPU's performance, an experiment was conducted on both, as presented in this section. The setup of this simulation's environment is different from the other experiments previously detailed. In this study, the previously mentioned brushed DC motor model is implemented both in StreamPU and SystemC-AMS. Both environments are set to only compute the simulation, without logging the results. The time step is fixed to 5 ns and the simulations run for a duration of 50 ms. As it is easily available, the SystemC and SystemC-AMS libraries version 2.3.4 provided by Accellera and CODESA Technologies were chosen. Both environments are executed inside the same virtual machine, as the SystemC-AMS library had compatibility issues with the previous experiment's operating system. The virtual machine executes the Ubuntu 20.04.6 LTS operating system, on the Virtualbox 7.1.4 software.

6 EXPERIMENTAL RESULTS ANALYSIS

6.1 Error Analysis

Table 1 presents the absolute error average values with regard to MATLAB Simulink-based solutions for each simulation scenario. Those values are obtained with the following formula:

$$error = mean(|f(t) - g(t)|/f(t))$$

with $f(t)$ the values obtained with the proposed method, and $g(t)$ those obtained with Simulink. We have evaluated the average over a smaller range than the whole simulation, as it is impossible to normalize by 0. It is apparent that average error is small, ranging from around 1.3% to as low as 0.03%. Error values are mainly linked to the scenario, i.e. the models, rather than the choice of using an equivalent C++ model or Verilator for simulating the controller. In the case of the DC motor speed control scenario, we attribute part of the higher error metric to potential differences in the Simulink implementation of the models, which relies on available blocks rather than the original C++ description. Nonetheless, the average absolute error is relatively low in all scenarios tested, effectively showcasing the proposed method's correctness.

6.2 Simulation Speed Gain

As shown in Table 2 and Figure 4, which present the times taken to simulate and the speed-up factors with regard to SlD and SlVi, for each scenario, StreamPU paired with Verilator or not, brings a significant increase in simulation speed. The proposed method achieves speed-up factors ranging from 2.25 in the worst case, to factors as high as 3033 in the best case.

It is apparent that the speed-up factors are much lower when only MATLAB Simulink is considered with the equivalent hardware models, compared to using co-simulation. This means using Siemens' Questa Sim or Xilinx's Vivado greatly slows down the simulation. The deep integration of the verilated hardware models in the simulation environment avoids the highly time-consuming synchronization operations present in MATLAB Simulink's co-simulation environment. The use of Verilator, which is comparatively faster than Siemens' Questa Sim or Xilinx's Vivado, and its deep integration into the simulation environment contribute the most to the simulation speeds achieved with the proposed methodology.

The DC Motor simulation appears to be more challenging for MATLAB Simulink's solver, explaining the big difference in speed-up factor between this simulation scenario and the two other presented in this paper.

As showcased by the results in Table 3, this technique also presents a simulation speed gain compared to SystemC-AMS, despite both being fully implemented in C++. This behavior is mostly due to the lower overhead involved in StreamPU's tasks scheduling compared to SystemC-AMS.

6.3 Practical Considerations

As the proposed methodology's implementation only supports a single fixed time step across the whole simulation, the time step value is set by the user to the smallest value acceptable for each simulated model. This means some parts of the simulation are called more often than needed. Consequently, using the proposed methodology to simulate a system without a discrete model would probably provide less speed-up or even be slower than other simulators capable of using variable time steps, such as some in MATLAB Simulink. Further speed gains to the already good results could be obtained by extending the implementation to support variable time steps. In the proposed method, the integration operation is an approximation of its mathematically complete version. As such, some models present more error accumulation than others in particular simulation conditions. The presented experiments already showcase good error results, and other integration methods could be added to provide more control over the simulation to the user.

7 CONCLUSION

In this paper, a simulation methodology for fast verification of cyber-physical systems is detailed. The proposed approach, by following an efficient method, and adding some utility functions, leverages the speed of C++ simulations thanks to Verilator and StreamPU. By this way, fast simulations of cyber-physical systems are executed. Experimental results indicate that the proposed environment significantly reduces simulation time. Consequently, it enables shorter design iterations compared to alternative solutions. Moreover, the proposed approach exclusively leverages free and open-source tools, making it affordable to anyone.

Adding the possibility to execute multiple steps of the discrete time domain simulation, for a single step in the rest of the system, could also improve simulation speed for physical models with slow reaction times. One can push the capabilities of the proposed approach to extend it for hardware-in-the-loop simulations, as the use of a fully C++ environment brings a lot of implementation flexibility. The proposed methodology leaves a lot of extensions possibility to users, for them to adapt it to their particular use case.

8 ACKNOWLEDGEMENT

This work took place in the framework of the 'Electronic and Systems for Automotive' OpenLab, combining the IMS CNRS Laboratory and the Stellantis company.

REFERENCES

[1] *Cyber-Physical Systems - a Concept Map.* https://ptolemy.berkeley.edu/projects/cps/.
[2] Siddhartha Kumar Khaitan et al. "Design Techniques and Applications of Cyberphysical Systems: A Survey". In: *IEEE Systems Journal* 9.2 (June 2015), pp. 350–365.
[3] "IoT Devices and Infrastructures Group". In: *NIST* (June 2014).
[4] Breytner Fernandez-Mesa et al. "Synchronization of Continuous Time and Discrete Events Simulation in SystemC". In: *IEEE TCAD* 40.7 (July 2021), pp. 1450–1463.
[5] Liliana Andrade et al. "Pre-Simulation Symbolic Analysis of Synchronization Issues between Discrete Event and Timed Data Flow Models of Computation". In: *DATE*. IEEE Conference Publications, 2015, pp. 1671–1676.
[6] Adrien Cassagne et al. "StreamPU: A DSEL for High Throughput and Low Latency Software-defined Radio on Multicore CPUs". In: *CCPE* 35.23 (Oct. 2023), e7820.
[7] Wilson Snyder et al. *Verilator.* https://github.com/verilator/verilator. Oct. 2024.
[8] *Verilog-AMS Language Reference Manual.* Feb. 2024.
[9] IEEE Computer Society et al., eds. *IEEE Standard VHDL Analog and Mixed-Signal Extensions.* IEEE, 1999.
[10] "IEEE Standard for Standard SystemC® Language Reference Manual". In: *IEEE Std 1666-2023 (Revision of IEEE Std 1666-2011)* (Sept. 2023), pp. 1–618.
[11] "IEEE Standard for Standard SystemC(R) Analog/Mixed-Signal Extensions Language Reference Manual". In: *IEEE Std 1666.1-2016* (Apr. 2016), pp. 1–236.
[12] *SystemC AMS Proof-of-Concept.* https://www.coseda-tech.com/systemc-ams-proof-of-concept.
[13] Cedric Ben Aoun et al. "Pre-Simulation Elaboration of Heterogeneous Systems: The SystemC Multi-Disciplinary Virtual Prototyping Approach". In: *SAMOS*. IEEE, July 2015, pp. 278–285.
[14] Enrico Fraccaroli et al. "Automatic Generation of Analog/Mixed Signal Virtual Platforms for Smart Systems". In: *IEEE Transactions on Computers* 69.9 (Sept. 2020), pp. 1263–1278.
[15] Adrien Cassagne et al. "AFF3CT: A Fast Forward Error Correction Toolbox!" In: *SoftwareX* 10 (July 2019).
[16] Mathuran Kandeepan et al. *Fast Meteor Detection Toolbox.* https://github.com/alsoc/fmdt.
[17] Kendall E. Atkinson. *An Introduction to Numerical Analysis.* 2. ed. Wiley, 1989.
[18] Jonathan Bachrach et al. "Chisel: Constructing Hardware in a Scala Embedded Language". In: *Proceedings of the 49th Annual Design Automation Conference.* DAC '12. ACM, June 2012, pp. 1216–1225.
[19] William Bolton. *Mechatronics: Electronic Control Systems in Mechanical and Electrical Engineering.* 3rd ed. Pearson/Prentice Hall, 2003.
[20] *Miniature DC Motors Parameters.* https://www.portescap.com/de-de/produkte/bürsten-dc-motoren/miniature-dc-motors.
[21] Payman Shakouri et al. "Simulation Validation of Three Nonlinear Model-Based Controllers in the Adaptive Cruise Control System". In: *Journal of Intelligent & Robotic Systems* 80.2 (Nov. 2015), pp. 207–229.
[22] *Model an Anti-Lock Braking System - MATLAB Simulink.* https://www.mathworks.com/help/simulink/slref/modeling-an-anti-lock-braking-system.html.

Selective Triplication for Fault-Tolerant Systolic Array Processing Elements

Wilfread Guillemé, Gaëtan Lounes, Angeliki Kritikakou, Youri Helen, Robin Gerzaguet, Matthieu Gautier, Cédric Killian and Daniel Chillet

Return to Schedule

Selective Triplication for Fault-Tolerant Systolic Array Processing Elements

Wilfread Guillemé[*]
wilfread.guilleme@irisa.fr
Univ Rennes, CNRS, Inria, IRISA
Lannion, France

Gaëtan Lounes[*]
gaetan.lounes@irisa.fr
Univ Rennes, CNRS, IRISA
Lannion, France

Angeliki Kritikakou
angeliki.kritikakou@irisa.fr
Univ Rennes, CNRS, Inria, IRISA
Rennes, France

Youri Helen
youri.helen@intradef.gouv.fr
DGA MI
Bruz, France

Cédric Killian
cedric.killian@univ-st-etienne.fr
Université Jean Monnet Saint-Etienne, CNRS,
Institut d'Optique Graduate School,
Laboratoire Hubert Curien UMR 5516
Saint-Étienne, France

Robin Gerzaguet
robin.gerzaguet@irisa.fr
Univ Rennes, CNRS, IRISA
Lannion, France

Matthieu Gautier
matthieu.gautier@irisa.fr
Univ Rennes, CNRS, IRISA
Lannion, France

Daniel Chillet
daniel.chillet@irisa.fr
Univ Rennes, CNRS, Inria, IRISA
Lannion, France

Abstract

Systolic arrays are widely used for high-performance digital computations, enabling efficient and parallel execution of operations such as matrix multiplication and convolution. They are deployed in digital signal processing and embedded systems, and they have been more recently adopted to accelerate deep neural networks. However, due to manufacturing defects, aging, and harsh environments, systolic arrays are vulnerable to faults, which can compromise computation reliability. These vulnerabilities call for reliability analysis and hardening methods. Since full triplication induces a high hardware overhead, this work introduces an automatic method that applies redundancy only to the most impactful flip-flops in the processing elements. FPGA simulations show a wide Pareto trade-off between reliability and hardware resources, offering flexible protection levels for fault-resilient systems.

[*] Equal contribution.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
RAPIDO'26, Krakow, Poland
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN XXX-X-XXXX-XXXX-X/2026/01
https://doi.org/XXXXXXX.XXXXXXX

CCS Concepts: • **Hardware** → **Redundancy**; **Hardware description languages and compilation**; **Test-pattern generation and fault simulation**.

Keywords: Fault Tolerance, Selective Redundancy, Systolic Arrays, Compiled Simulation

ACM Reference Format:
Wilfread Guillemé[*], Gaëtan Lounes[*], Angeliki Kritikakou, Youri Helen, Cédric Killian, Robin Gerzaguet, Matthieu Gautier, and Daniel Chillet. 2026. Selective Triplication for Fault-Tolerant Systolic Array Processing Elements. In *Proceedings of 17th Workshop on Rapid Simulation and Performance Evaluation for Design Optimization: Methods and Tools (RAPIDO'26).* ACM, New York, NY, USA, 7 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Efficient parallel computation is a fundamental requirement in modern digital systems, and specialized architectures have been developed to meet high-performance and energy-efficiency demands. Systolic arrays (SAs) [1] are one such architecture, consisting of regularly arranged Processing Elements (PEs) that operate in a synchronized, pipelined manner to perform parallel computations efficiently. This structure is particularly well suited for repetitive operations, such as matrix multiplication and convolution, which are widely used in digital signal processing, scientific computing, and embedded systems [2]. More recently, SAs have been employed in modern deep learning accelerators, including Convolutional Neural Networks (CNNs) and Google's TPU [3], to efficiently perform large-scale neural computations. Their regular and modular design enables high throughput, predictable timing, and energy-efficient execution, with scalable and reconfigurable architectures.

While high-performance architectures achieve impressive computational efficiency, their reliability remains a critical concern. Indeed, digital circuits are vulnerable to various faults arising from manufacturing defects, process variations, device aging mechanisms, as well as environmental factors. Among these, radiation-induced Single-Event Effects (SEEs) [4], such as Single-Event Upsets (SEUs), are particularly important because they can alter values stored in registers. In systolic architectures, where computations are distributed across multiple PEs and deeply pipelined, even a single corrupted value can propagate across many stages and significantly affect the final result. Ensuring reliability against such events remains a key challenge, motivating fault mitigation and hardening techniques that protect the architecture while maintaining computational efficiency.

Robust fault-tolerance in digital architectures is commonly achieved with full Triple Modular Redundancy (TMR) [5]. While effective, this approach incurs significant hardware overhead. However, faults do not have the same impact on the computational accuracy of Deep Neural Networks (DNNs) [6]. In particular, errors affecting Most Significant Bits (MSBs) of registers tend to degrade DNN outputs more severely than faults in Least Significant Bits (LSBs). These observations motivate a selective hardening strategy, where only D Flip-Flops (DFFs) storing the most impactful bits are triplicated, maintaining high fault tolerance while reducing resource usage.

This work proposes a selective triplication methodology for PEs, associated with a register sensitivity analysis, enabling the identification of optimal protection configurations and their generalization to other SA architectures.

The remainder of this article is organized as follows. Section 2 reviews related work and presents background on SAs. Section 3 details the proposed selective hardening methodology. Section 4 describes the experimental setup, while Section 5 reports the experimental results. Section 6 discusses the generalization of the approach to other architectures. Finally, Section 7 concludes the article, summarizes the main findings, and outlines future perspectives.

2 Background and Related Work

2.1 Background on Systolic Arrays

An example 4×4 SA with a detailed view of a PE is illustrated in Figure 1. Without lacking of generality, and to simplify the presentation, we detail our methodology on a small size of systolic array. SAs can be organized according to different dataflow strategies, which define how data are reused and propagated across the array.

Each PE is the fundamental computational unit of a SA. It typically performs a Multiply-and-Accumulate (MAC) operation between an input 'A' activation and a 'W' weight, while also propagating the partial sum 'S' corresponding to the accumulation to its neighboring elements. The PE architecture includes local registers for inputs, weights, and partial sums, supporting pipelined MAC operations.

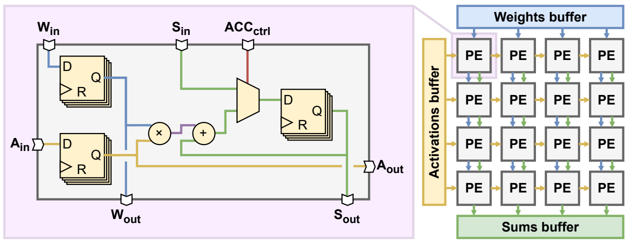


Figure 1. 4×4 SA with a detailed PE.

2.2 Related Work

Among the research efforts that address the problem of fault tolerance in SA architectures, various approaches have been proposed to enhance their robustness.

FORTALESA [7] is a runtime-reconfigurable SA designed to improve the reliability of DNN inference. It supports three execution modes (no redundancy, Double Modular Redundancy (DMR), and TMR), offering different protection levels depending on reliability and performance requirements. In DMR mode, neighboring PEs operate in main and shadow pairs to detect and mask faults through averaging or zeroing mismatched bits, while TMR mode employs groups of three PEs to provide fault correction. Another approach, FSA [8], addresses fault tolerance in SA-based DNN accelerators using a different strategy. This design employs a unified re-computing module to recover computations affected by faulty PEs, preserving inference accuracy under permanent faults. Algorithm-Based Fault Tolerance (ABFT) is a classical approach for detecting errors in matrix operations [9]. Recent work has adapted ABFT for modern hardware. In [10], the authors propose a lightweight ABFT technique for SAs on FPGAs that performs block-level algorithmic checks during matrix multiplication, achieving a high fault detection rate while maintaining low hardware and performance overhead.

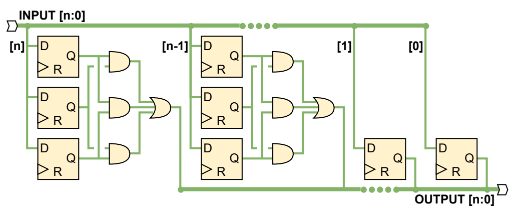
Unlike conventional redundancy schemes that replicate entire PEs, our approach provides fine-grained protection at the register level within each PE. This reduces hardware overhead while maintaining sufficient fault tolerance for inherently error-tolerant workloads such as DNN inference.

3 Proposed Method for Selective Hardening

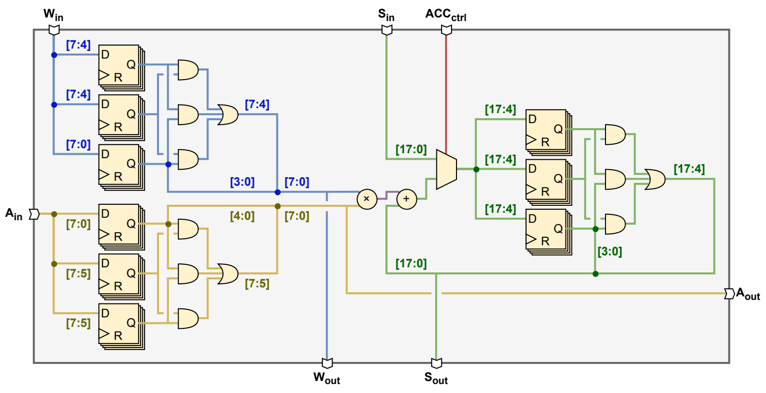
This section presents the selective approach to enhance fault tolerance. First, the method is applied to a single data word, then to the registers within a PE.

3.1 Principle of Selective TMR

To enhance fault tolerance, the proposed hardening is performed selectively, focusing on protecting only the MSBs of each register. Figure 2a illustrates the principle of selective triplication applied to a N-bits data word. In this approach,



(a) Selective TMR applied to a N-bits data word with $N = n + 1$.



(b) Example of Hardened PE with the A:T3U5-W:T4U4-S:T14U4 configuration.

Figure 2. Selective TMR and hardened PE example.

the M MSBs out of a total of N bits in a data word are triplicated, while the remaining $N - M$ LSBs are left unprotected.

A bit-flip occurring in an integer or fixed-point encoded value affects its numerical magnitude according to the position of the flipped bit. More precisely, an error on bit position n changes the value by $\pm 2^n$ in integer representation, or by $\pm 2^{n-f}$ in fixed-point formats, where f denotes the number of fractional bits. As expected, faults occurring in the MSBs induce larger deviations in the computation, whereas those affecting the LSBs have a negligible impact on the final result.

The resulting hardware overhead in DFFs for a single register is given by $O_{\text{DFF}} = \frac{2M}{N}$. This represents the percentage of additional DFFs relative to the unprotected implementation.

3.2 Selective Hardening applied on a PE

In this work, fault-tolerance is applied selectively to the sequential elements within each PE. Specifically, only the DFFs storing the MSBs of activations (A), weights (W), and the accumulator (S) are protected using selective TMR. The level of protection can be adjusted by choosing the number of bits to triplicate in each register of the PE.

A PE configuration is described using the following notation `A:T(M)U(N-M)-W:T(M)U(N-M)-S:T(M)U(N-M)`, where `A:T(M)U(N-M)` indicates that the M MSBs of the A register are protected by TMR, while the remaining $N - M$ LSBs are unprotected. The notations `W:T(M)U(N-M)` and `S:T(M)U(N-M)` apply similarly to the W and S registers, respectively. For instance, Figure 2b shows a PE hardened using the `A:T3U5-W:T4U4-S:T14U4` configuration, illustrating how the methodology is applied. In this example, the accumulator register is 18 bits wide to safely store the sum of four 8-bit multiplication results, preventing any overflow.

4 Experimental Setup

This section presents the experimental setup used to evaluate the robustness of the proposed architecture. It outlines the fault model and the compilation flow used to perform accurate Register Transfer Level (RTL) simulation and analysis.

4.1 SEU-based Fault Model

To assess the robustness of the proposed architecture, we considered an SEU-based fault model targeting its internal registers. Each PE contains 34 registers (16 for the 8-bit input operands A and W, and 18 for the accumulator S), leading to a total of 544 registers across the 4×4 PE SA. To evaluate the design exhaustively, we consider all possible SEU faults affecting the registers during a matrix multiplication. Since the multiplication of two 4×4 matrices completes in 15 clock cycles [1], this results in $544 \times 15 = 8{,}160$ distinct fault scenarios for a single input matrix pair. These scenarios must be evaluated at the register level with cycle-accurate precision to capture the fault behavior at the hardware abstraction level.

Each experiment consists of multiplying a pair of randomly generated 4×4 integer matrices. The total number of possible input matrix pairs is extremely large ($2^{8 \times 16 \times 2}$), making exhaustive evaluation impractical. Therefore, a statistical estimation equation [11] with a maximum error margin set as 5% and at 90% confidence level was employed, resulting in 271 randomly generated matrix pairs. Combining all fault locations with all input matrices leads to $8,160 \times 271 = 2{,}211{,}360$ individual faults injected. This approach enables a reliable evaluation of the system behavior across all feasible fault locations while keeping the computational effort manageable. The scale of the events being simulated motivates the need for a efficient simulation infrastructure, compiler-based simulations has proven to provide state of the art [12] performances for RTL simulations.

4.2 Compilation Flow

The compilation flow, illustrated in Figure 3, transforms the modural SA design defined using the Chisel [13] Hardware Description Language (HDL) into a form suitable for

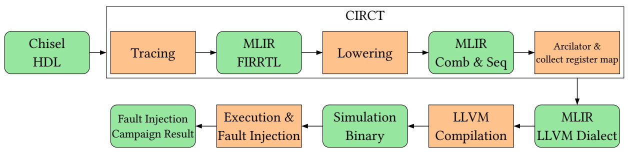


Figure 3. Modified toolchain to support fault injection.

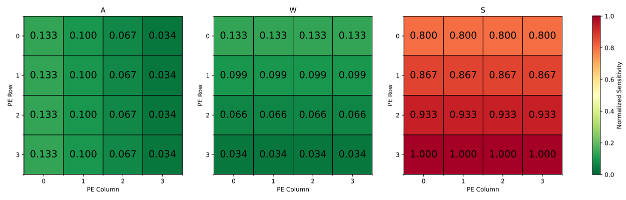


Figure 4. Heatmap of registers (A, W and S) sensitivity.

cycle-accurate RTL simulation, fault injection, and hardware synthesis. By leveraging a toolchain using established compiler and hardware infrastructure tools, we took benefice of existing infrastructure and enabling our custom simulation flow. The following subsections describe each tool and its role in the workflow.

LLVM

LLVM [14] is a compiler framework designed to facilitate the development of programming languages by defining an Intermediate Representation (IR) aimed to handle compiler back-end. This representation is kind of hardware and language agnostic, and is used to apply a large range of optimization to target hardware architecture such as x86.

MLIR

Multi-Layer Intermediate Representation (MLIR) [15] is a compiler framework and IR that extend LLVM and that aims to provide a rich and flexible infrastructure to build compilers that feature different levels of abstraction. Dialect represents a domain specific semantics defined using Operations, Types and Passes (transformation). MLIR is designed to create, optimize, and transform an IR across various dialects.

CIRCT

Circuit Intermediate Representation Compiler and Tools (CIRCT) [16] is a compiler infrastructure built with MLIR and design to handle hardware flows such as High Level Synthesis, Hardware Description Language (HDL) front-end such as Chisel or simulation. Notably, these flows are defined using several dialects, linked to specific hardware abstraction. For instance, Flexible IR for RTL (*FIRRTL*) integrated into Chisel describes an RTL dialect. More specific dialect exists: *comb* defined combinational logic while *seq* adds sequential logic.

ARCILATOR

Arcilator [17] is a cycle-accurate RTL simulator part of CIRCT project. It leverages abstractions defined in the *Arc* dialect and, from hardware descriptions defined in the *comb* and *seq* dialects, generates an LLVM dialect program that accurately models the behavior of the circuit. Notably, all circuit registers are mapped to a LLVM memory buffer.

4.3 Fault Injection Simulator

An analysis during the simulator compilation allows for the extraction of RTL registers position in the generated LLVM. This allows the use of Arcilator for error injection simulations. This approach has several advantages: it incurs no time overhead compared to standard simulation, as faults are applied as bit flips inside a buffer, and it requires no modification to the original design.

4.4 Fault-aware RTL

After completing the simulation campaign, the circuit must be protected according to the sensitivity analysis. To accomplish this, we leverage the Verilog front-end integration with CIRCT to define custom MLIR passes that generate the hardware required for fault mitigation. Verilog annotations guide HDL synthesis, notably for redundant registers. To ensure that duplicated DFFs are not optimized away during synthesis, a *KEEP* attribute is applied to these registers so that the reported DFF overhead reflects the intended protections. The IR is then translated into a Verilog representation and synthesized to obtain more fine hardware information.

5 Experimental Results

This section presents the evaluation of the proposed selective hardening methodology. We analyze fault propagation in SAs, the resulting reliability improvements, the associated hardware overhead, and identify Pareto-optimal protection configurations.

5.1 Fault Propagation in Systolic Arrays

In this study, we consider an Output Stationary (OS) SA, where partial sums remain in each PE while input activations and weights flow through the array.

Single bit-flip faults are simulated in the internal registers storing input activations (horizontal flow), weights (vertical flow), and partial sums (accumulation followed by vertical flow). Let R denote a register in the SA. For each injection experiment i targeting R, we define the deviation from the fault-free reference (golden inference) as d_i. The aggregated deviation for register R is obtained by summing over N experiments, $D_R = \sum_{i=1}^{N} d_i$, and the normalized fault sensitivity is defined as $S_R = D_R / \max_{R'} D_{R'} \in [0, 1]$, where the maximum is taken over all registers R in the SA. For each register type, the fault sensitivity was measured and visualized by heatmap as presented in Figure 4, offering a clear view of how errors propagate through the array depending on their position and their impact on the final computation.

Distinct sensitivity patterns are observed for each register type. For input registers (A), the leftmost column, $PE(*, 0)$, shows the highest sensitivity, as activations propagate horizontally and are reused by subsequent PEs along the same row, allowing early faults to propagate further. For weight registers (W), the most sensitive elements are located in the top row, $PE(0,*)$, where vertical data reuse amplifies the impact of early bit-flips along the column. In contrast, the accumulation registers (S) exhibit almost uniform sensitivity during computation, since all PEs perform similar accumulation operations. However, during the output phase, the last active row becomes more critical because it stores the final accumulated results. Any bit-flip at this stage directly corrupts the output. Accumulators in other PEs no longer contribute to the matrix result and thus have no effect once their computations are completed.

Furthermore, accumulation registers are inherently more sensitive due to their larger bit-width. As already explained, input and weight registers are encoded on 8 bits, whereas accumulation registers use 18 bits. As a result, a single bit-flip in an accumulator can have a significantly larger impact on the output value. The maximum normalized sensitivity of an accumulation register is approximately 7.5 times higher than that of an input register, indicating a corresponding increase in potential sensitivity.

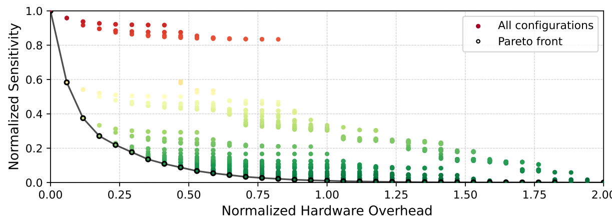
These findings demonstrate that fault propagation in SAs is directional, reflecting the inherent dataflow of the architecture, and that the behavior would differ under alternative dataflows such as Weight Stationary or Input Stationary.

5.2 Reliability Improvement

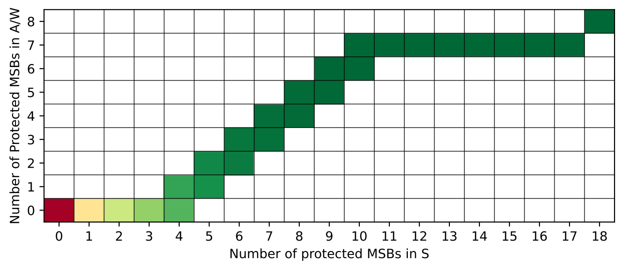
Based on the selective PE protection strategy described earlier, we generate the simulation setups for all possible configurations of PEs within a 4×4 SA. For each configuration, we evaluate the effect of the protection mechanisms by comparing the fault behavior against a baseline design without protection. The reliability is quantified by measuring the Manhattan distance between the output resulting from a single bit-flip and the reference matrix output, which is then normalized. For instance, if the MSBs of an accumulator are triplicated, any fault affecting these bits can be corrected and will no longer impact the computation, effectively reducing the register's sensitivity. Increasing the degree of triplication further improves the protection of the design. This approach allows a systematic analysis of how each configuration influences the overall reliability of the array.

5.3 Pareto-optimal Configurations

Each PE in our study contains three registers: A, W, and S. For each register, we consider protection strategies ranging from no bits protected to all bits protected, prioritizing the MSBs. This gives 9 possible configurations for A and W (0 to 8 bits protected) and 19 possible configurations for S (0 to 18 bits protected). By enumerating all combinations of these three registers, we obtain $9 \times 9 \times 19 = 1{,}539$ possible PE configurations, each exhibiting a distinct trade-off between reliability and hardware overhead.



(a) Sensitivity vs. Hardware overhead for OS PEs.



(b) Pareto-optimal OS PEs (A/W vs. S).

Figure 5. Comparison of Pareto-optimal configurations.

The results obtained for all configurations are shown in Figure 5a. Among these, only 33 configurations belong to the Pareto front with respect to the two criteria, representing the best trade-offs between reliability and hardware overhead. These configurations range from no protection with minimal hardware cost to full protection with the maximum overhead, providing guidance for selecting efficient protection strategies in SAs. Figure 5b illustrates the evolution of register protection across the 26 Pareto-optimal hardened PE configurations that are displayed. Initially, 33 configurations were identified as Pareto-optimal. However, due to the presence of zeros in the 271-input matrix, some configurations show small differences, since a bit-flip may be nullified when multiplied by zero, even though the sensitivity is effectively identical for the A and W registers. To provide a clearer view, only configurations where A and W are protected equally are shown. The degree of protection applied to accumulation registers (S) and input/weight registers (A and W) is presented for each configuration. It can be observed that the accumulation registers are consistently prioritized for protection compared to the input and weight registers. This behavior aligns with the previous fault sensitivity analysis, where accumulation registers exhibited higher vulnerability due to their larger bit-width and critical role in storing intermediate results. The figure highlights how the selective protection strategy naturally focuses on the most critical registers, providing efficient reliability improvement across the SA, as guided by the sensitivity metric.

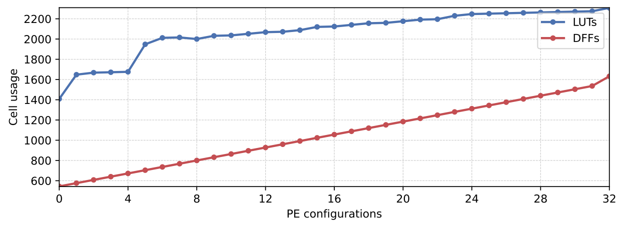


Figure 6. LUTs and DFFs usage per OS PE configuration.

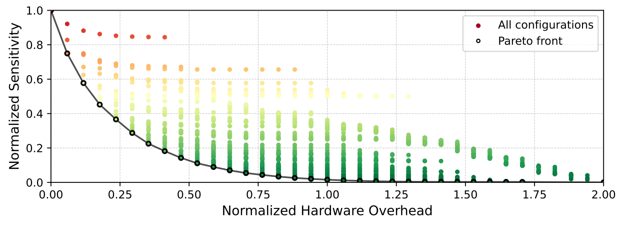


Figure 7. Sensitivity vs. Hardware overhead for WS PEs.

5.4 Hardware Overhead

To evaluate the hardware overhead induced by the proposed selective protection strategy, we synthesized all 33 Pareto-optimal hardened PE configurations within the 4×4 SA. From the synthesis *report_utilization* provided by Vivado 2025.1 and targeting a Zynq-7000 FPGA, we extracted the number of Look-Up Tables (LUTs) and DFFs used. The results, shown in Figure 6, are presented in ascending order of overhead. The baseline PE contains 34 DFFs, resulting in a total of 544 DFFs for the 4×4 SA. For the most protected PE design, the total number of DFFs reaches 1,632, corresponding to a 200% increase compared to the baseline. Interpreting LUT utilization is more complex, as the synthesis tool applies optimizations that may reduce or share logic resources. In practice, the LUT count ranges from 1,408 to 2,308 across all configurations, reflecting the impact of these optimizations rather than a strictly increasing trend with additional protection.

6 Generalization to Other SA Architectures

This section explores the applicability of the proposed selective hardening methodology beyond the baseline SA. We first analyze its impact under a weight-stationary dataflow and then discuss its scaling to larger array sizes.

6.1 Weight Stationary Analysis

Although the Weight Stationary (WS) and OS dataflows share similar PE architectures, differences arise in the number of registers involved and in the way data is managed within the array. In the WS dataflow, the W registers are first loaded and then remain stationary. This leads to a different exposure pattern compared to the A registers, which are continuously streamed through the array. The S registers also behave differently, since they store values encoded with a larger bit width. These characteristics affect the relative sensitivity of the registers and result in changes to the Pareto front compared to the OS dataflow. Figure 7 illustrates the Pareto front and the performance of all PE configurations for the WS dataflow, which reflects these distinctions.

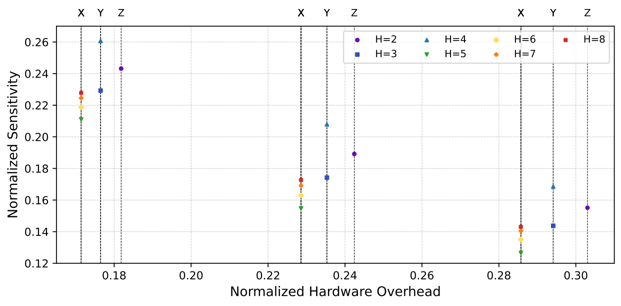


Figure 8. Partial Pareto-optimal OS PEs for varying H×H matrices size (H = 2–8).

6.2 Scaling to Larger Arrays

To assess the scalability of the proposed compilation flow and selective hardening methodology for design-space exploration, we extended the analysis to SAs performing matrix multiplications of size $H \times H$, with H ranging from 2 to 8. Figure 8 reports partial sets of Pareto-optimal configurations, similar to Figure 5a, and illustrates how the trade-off changes with matrix size. The hardware overhead varies because the accumulator S requires a larger bit width as H increases to avoid overflow, resulting in three register sizes ($X = 17$ bits for $H = 2$, $Y = 18$ bits for $H = 3$–4, and $Z = 19$ bits for $H = 5$–8). Regarding normalized sensitivity, protection is slightly less effective for larger H when using the same hardened PE configuration and S register size. This behavior is consistent with the longer propagation paths in larger SAs, which increase the impact of error effects.

7 Conclusion and Perspectives

This paper presented a methodology for selectively hardening processing elements of systolic arrays to improve fault tolerance. The approach was evaluated through exhaustive RTL cycle-accurate simulations, analyzing fault propagation, reliability improvement, hardware overhead, and Pareto-optimal protection configurations. Results demonstrate that selective hardening can significantly enhance reliability while limiting additional hardware cost. Although the study focused on a 4×4 systolic array, the methodology and proposed simulation toolchain generalize to other dataflows and larger arrays. Future work may explore heterogeneous or additional protection within a single systolic array and extend the fault model to multiple bit upsets.

Acknowledgments

This work is supported by the Direction Générale de l'Armement (DGA), funded by the Defense Innovation Agency (AID) and the National Institute for Research in Digital Science and Technology (Inria). In collaboration with the French National Research Agency (ANR) under the grant number ANR-22-CE25-0007-01 (RedInBlack project).

References

[1] Hsiang Tsung Kung and Charles E Leiserson. Systolic arrays (for vlsi). In *Sparse Matrix Proceedings 1978*, volume 1, pages 256–282. Society for industrial and applied mathematics Philadelphia, PA, USA, 1979.

[2] Rui Xu, Sheng Ma, Yang Guo, and Dongsheng Li. A survey of design and optimization for systolic array-based dnn accelerators. *ACM Computing Surveys*, 56(1):1–37, 2023.

[3] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

[4] Vitor AP Aguiar, Saulo G Alberton, and Matheus S Pereira. Radiation-induced effects on semiconductor devices: A brief review on single-event effects, their dynamics, and reliability impacts. *Chips*, 4(1):12, 2025.

[5] F Lima Kastensmidt, Luca Sterpone, Luigi Carro, and M Sonza Reorda. On the optimal design of triple modular redundancy logic for sram-based fpgas. In *Design, Automation and Test in Europe*, pages 1290–1295. IEEE, 2005.

[6] Guanpeng Li, Siva Kumar Sastry Hari, Michael Sullivan, Timothy Tsai, Karthik Pattabiraman, Joel Emer, and Stephen W Keckler. Understanding error propagation in deep learning neural network (dnn) accelerators and applications. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*, pages 1–12, 2017.

[7] Natalia Cherezova, Artur Jutman, and Maksim Jenihhin. Fortalesa: Fault-tolerant reconfigurable systolic array for dnn inference. *arXiv preprint arXiv:2503.04426*, 2025.

[8] Yingnan Zhao, Ke Wang, and Ahmed Louri. Fsa: An efficient fault-tolerant systolic array-based dnn accelerator architecture. In *2022 IEEE 40th International Conference on Computer Design (ICCD)*, pages 545–552. IEEE, 2022.

[9] Kuang-Hua Huang and Jacob A Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE transactions on computers*, 100(6):518–528, 1984.

[10] Fabiano Libano, Paolo Rech, and John Brunhaver. Efficient error detection for matrix multiplication with systolic arrays on fpgas. *IEEE Transactions on Computers*, 72(8):2390–2403, 2023.

[11] Régis Leveugle, A Calvez, Paolo Maistri, and Pierre Vanhauwaert. Statistical fault injection: Quantified error and confidence. In *2009 Design, Automation & Test in Europe Conference & Exhibition*, pages 502–506. IEEE, 2009.

[12] Wilson Snyder and Verilator Contributors. Verilator simulator. https://github.com/verilator/verilator, 2003.

[13] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the 49th annual design automation conference*, pages 1216–1225, 2012.

[14] Chris Lattner. *LLVM and Clang: Next Generation Compiler Technology*, volume 5. 2008.

[15] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, Seoul, Korea (South), February 2021. IEEE.

[16] CIRCT Developers. Circt: Circuit Intermediate Representation for Compilers and Tools. https://circt.llvm.org/, 2021.

[17] Martin Erhart, Fabian Schuiki, Zachary Yedidia, Bea Healy, Tobias Grosser. Arcilator simulator: Fast and cycle-accurate hardware simulation in circt. https://github.com/llvm/circt, 2022.