

MEEK: Re-thinking Heterogeneous Parallel Error Detection Architecture for Real-World OoO Superscalar Processors

Zhe Jiang[†], Minli Liao[¶], Sam Ainsworth^{||}, Dean You[†], Timothy Jones[¶]

[†]National Center of Technology Innovation for EDA, School of Integrated Circuits, South East University, People's Republic of China, [¶]University of Cambridge, United Kingdom, ^{||}University of Edinburgh, United Kingdom

Abstract—Heterogeneous parallel error detection is an approach to achieving fault-tolerant processors, leveraging multiple power-efficient cores to re-execute software originally run on a high-performance core. Yet, its complex components, gathering data cross-chip from many parts of the core, raise questions of how to build it into commodity cores without heavy design invasion and extensive re-engineering.

We build the first full-RTL design, MEEK, into an open-source SoC, from microarchitecture and ISA to the OS and programming model. We identify and solve bottlenecks and bugs overlooked in previous work, and demonstrate that MEEK offers microsecond-level detection capacity with affordable overheads. By trading off architectural functionalities across codesigned hardware-software layers, MEEK features only light changes to a mature out-of-order superscalar core, simple coordinating software layers, and a few lines of operating-system code. The Repo. of MEEK's source code: <https://github.com/SEU-ACAL/reproduce-MEEK-DAC-25>

I. INTRODUCTION

Hardware faults, both permanent and transient, can induce system anomalies and execution errors, which become more common with the increasing number of transistors and lower operating voltages in modern processors [1], [2], [3], [4], [5]. To mitigate the errors caused by hardware faults, protection mechanisms exist, ranging from error codes to fault-tolerant architectures. Detection is always key: once an error is detected, the system can transition to a safe state, enabling corrective actions (e.g., system recovery or fault isolation). As mandated by global safety standards from industry, e.g., ISO26262 for automotive [6] and DO-178C [7] for avionics, hardware faults must be addressed before escalating into hazards, i.e., within the Fault Tolerance Time Interval (FTTI), often measured in milliseconds [8].

Software mechanisms (e.g., multithreading [4], [9] and software scanner [10], [11]) typically incur significant performance degradation or offer limited fault coverage [12], making them insufficient for processors that require stringent reliability standards (e.g., ASIL-D in ISO26262 [6]). Hardware mechanisms often employ a dedicated core to execute a program copy, enabling the comparison of the core's pins at each clock cycle (e.g., locksteps [13], [14], [15]). By replaying everything on a separate, synchronized core and performing run-time verification at the signal level, full coverage and real-time guarantees are achieved. Although dual- and triple-core locksteps have been successfully applied to microcontroller-sized processor cores in many life-critical application scenarios [16], [17], they have been shown impractical for Out-of-Order (OoO) superscalar cores due to prohibitive energy, area, and thermal costs [18], [19], [20].

Heterogeneous parallel error detection. As a promising alternative, heterogeneous parallel error detection [21], [22], [2] leverages strong induction to divide a software program running on an OoO superscalar high-performance core (big core) into multiple discrete segments using Register Check Points (RCPs) and re-execute them on sets of smaller, power-efficient cores (little cores) for verification.

To replay memory and other non-repeatable operations, the addresses and data of relevant instructions (e.g., load and store) are extracted from the program stream at the commit stage within the big core, generating a partitioned, distributed log of load and store operations. When the log of a segment is filled, or an instruction timeout is reached, a new RCP is triggered, and the corresponding little core begins verifying the segment between Start RCPs (SRCPs) and End RCPs (ERCPs). By overlapping verification jobs, little cores collectively offer sufficient computational capacity to keep pace with the big core, ensuring full coverage with low overheads [21].

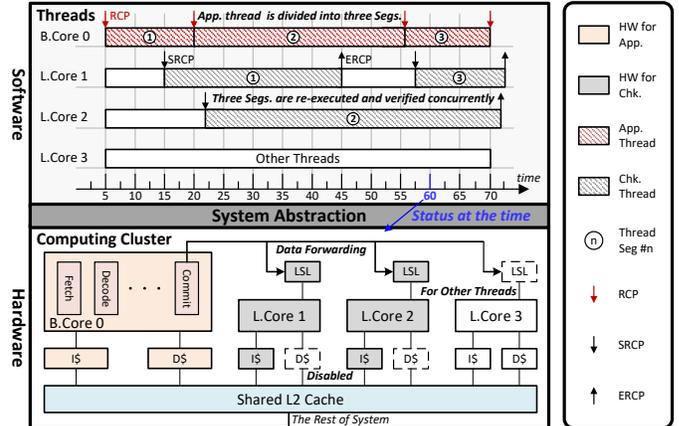


Fig. 1: Re-constructed heterogeneous parallel error detection architecture (RCP: Register Checkpoint; S/ERCP: Start/End RCP LSL: Load-Store Log): an application thread on big core 0 is divided into three Segs. using RCPs, replayed and verified on little core 1 and 2.

Challenges. Unlike conventional lockstep cores, which are typically implemented using identical processor cores with multiplexers and comparators [23], heterogeneous error detection relies on intricate asynchronous interactions between the big core and a sea of little cores. This involves continuously collecting and distributing loads, stores, and periodical register checkpoints from a big core and flexibly managing little cores. This data needs to be preserved from execution time until commit, before transmission in order, and handling superscalar commit. This also means that the collected data must be prioritized and routed to the corresponding little cores at high bandwidth, to prevent backpressure from stalling the big core.

Existing work [21], [22] on the architecture has been studied through abstract simulation, modeled upon idealistic assumptions in microarchitecture and without an OS, imposing barriers for practical deployment. For instance, it is unclear how to collect such massive data from different locations of a real core without heavy microarchitectural changes, route it cross-chip while handling contention, and provide a configurable framework for little cores' management.

Contributions. We show it is possible to build heterogeneous error detection into real cores with minimal changes, by re-constructing the concept via hardware/software codesign, trading off functionalities across system layers. This allows an application thread on a big core to be segmented using RCPs, replayed, and verified using any number of checker threads deployed on any little cores while still allowing other threads to be executed (Fig. 1). We build a full-stack framework, **Make Each Error Count (MEEK)**, upon an open-source SoC [24], demonstrating the architecture can be realized with minimal design invasion within mature cores, and a few line changes in a full Linux.

II. MEEK: A CPU/OS CODESIGNED APPROACH

To implement MEEK at minimal complexity while achieving high performance, we had to make careful design-choice partitions between hardware and software (figure 2). On one hand, we had little choice but to implement load- and store-logging, for replay of execution on little cores, in hardware: using software-based instrumentation [25] would have meant infeasibly high overheads. On the other hand, since little cores perform page-table walks asynchronously with

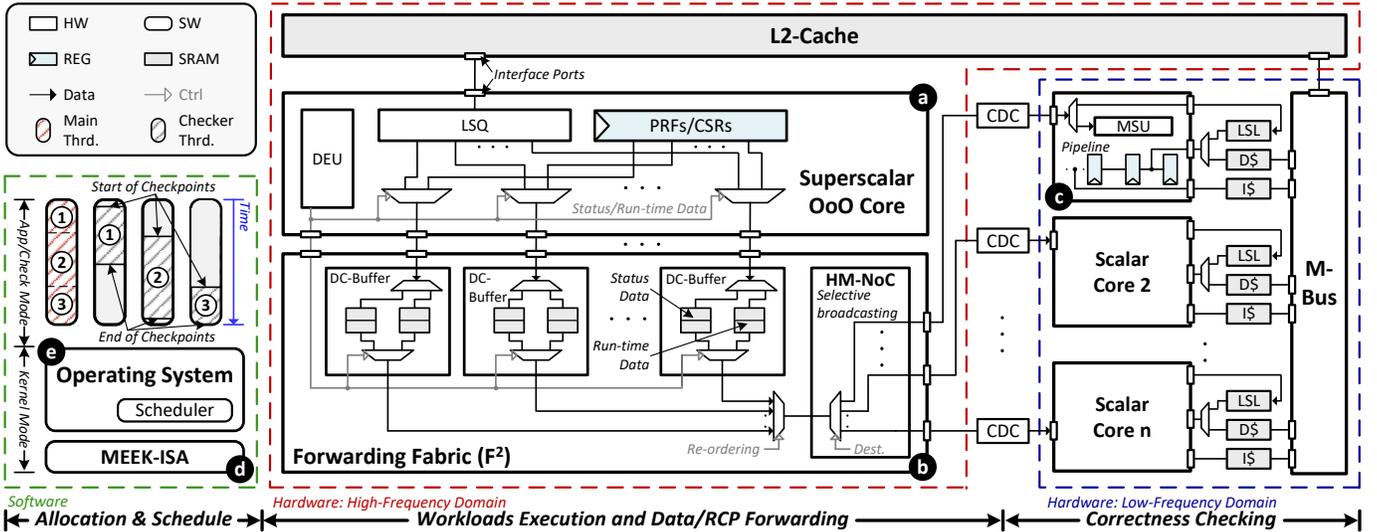


Fig. 2: An overview of MEEK. (DEU: Data Extraction Unit; LSQ: Load-Store Queue; PRFs: Physical Register Files; CSRs: Control and Status Registers; HM-NoC: Half-duplex Multicast Network-on-Chip; MSU: Mode Switch Unit; LSL: Load-Store Log; M-Bus: Memory Bus). At hardware: **a** a non-intrusive DEU is deployed in the big core, collecting data from various locations without disrupting the core’s execution; **b** a bespoke forwarding fabric F^2 is developed, prioritizing and distributing the data only to the relevant little cores; **c** dual-mode little cores are designed for correctness checking or workload execution. At software: **d** a customized ISA abstracts the control interface for **e**, the scheduler in the OS, enabling flexible management of verification and workload execution on the little cores.

the big core, the OS had to be aware of them in some form, and so we let the OS fully control their scheduling, avoiding implementing complex decisions that were not on the performance critical path in hardware, and allowing little cores to execute standard processes as well. Rather than a fully transparent interface, and to avoid expensive full error correction in hardware [22], programs interact with the MEEK-ISA by being wrapped with coordinator functions inserted before main, which request checker resources from the OS, verify checking outputs, and call fault-handling code if needed.

To do so, we slightly modified the big core’s microarchitecture to insert a read-only observation channel at the commit stage (Fig. 2 **a**), collecting the big core’s *status data* (i.e., architectural, control and status register files) at each RCP and the *run-time data* (addresses and data of memory and other non-repeatable operations) between RCPs. We built a dedicated data fabric (Fig. 2 **b**), selectively broadcasting/routing the extracted data to the little core(s), minimizing the backpressure from the data communications. In little cores, the received data is buffered in a Load-Store Log (LSL), replacing the L1 cache during program replay, allowing the little core to reset its architectural state to a given SRCP, replay the exact instructions between the RCPs, and verify execution correctness at the ERCP, using the different types of data (Fig. 2 **c**). Our ISA interface (Fig. 2 **d** and Tab. I) (re-)configures the little cores’ checking characteristics, i.e., the operational mode (application or check mode).

Detection approach. Error checking is parallelized using checker threads (Fig. 2 (b)): an application thread is segmented using RCPs, taken when the targeted LSL is full, an instruction timeout is reached, or the kernel mode is trapped. These segments are run in a second time by checker threads, assuming all previous segments are correct. After re-execution, the checker thread compares its architectural registers against the ones provided by the application thread at the same RCP. If the registers match, the segment is considered correct. If all segments pass check, the entire execution is deemed correct¹.

III. THE MICROARCHITECTURE

Even with the reduced hardware feature-set discussed in section II, MEEK required careful microarchitectural engineering to allow effi-

¹Memory operations, however, cannot be verified at ERCPs. In the re-execution, addresses and data of memory operations are compared directly in the LSL. A similar design method is used for Control and Status Registers (CSRs) to verify non-repeatable instructions.

TABLE I: MEEK ISA (*Priv 1/0: kernel/user modes*).

Instruction	Priv	Description
b.hook rs1, rs2	1	Hook big core rs1 with little core rs2.
b.check rs1	1	Enable/Disable checking capacity.
1.mode rs1, rs2	1	Switch little core rs1’s mode to rs2.
1.record rs1	0	Record arch. registers to address rs1.
1.apply rs1	0	Apply arch. registers from address rs1.
1.jal rs1	0	Jump to rs1 (PC of main thread).
1.rslt rd	0	Return the check results.

cient, low-overhead execution, particularly around data transmission. We identify and ameliorate many key bottlenecks that were missing from the analyses and high-level simulations presented by previous work [21], [22], [26] due to a lack of RTL implementation. We also identify inefficiencies in terms of redundant data storage, where much of the information required by the forwarding paths is already buffered inside the core until commit time, meaning dedicated structures are not needed; instead we can forward data from the existing structures. Despite bottlenecks, we demonstrate that it is possible to build MEEK’s microarchitecture from a mature heterogeneous SoC with very minor changes, avoiding heavy engineering efforts.

We build MEEK into an open-source heterogeneous SoC (Rocket Chip [24]), with both high-performance and energy-efficient cores (BOOM and Rocket). We leverage Rocket Chip to demonstrate the applicability of our methodology to the other heterogeneous SoCs, e.g., ARM’s big.LITTLE [27] and Intel’s P- and E-class cores [28].

A. OoO Superscalar Core (The Big Core)

Program replay requires the collection of both status data and run-time data from the big core. The data is temporarily stored within the core during the program executions but distributed across various locations. The status data is buffered in the Physical Register Files (PRFs) and Control and Status Registers (CSRs), while the run-time data is stored in the Load-Store Queue (LSQ)² and CSRs. All can be directly extracted but must occur in a narrow window, at commit

²By building MEEK, we noticed that Ainsworth et al. [21]’s policy, of storing a dedicated buffer of load data, indexed by Re-Order Buffer (ROB) entry, was unnecessary. All the need was already buffered in LSQ. Yet, light changes were needed to preserve redundancy and thus fault tolerance, as while we assume data is protected via parity in the cache [29], and via full duplication by the point of reaching the LSL, there is a small window in the LSQ where it is otherwise protected by neither. We copy the cache’s parity bits into the LSQ, double-checking them once the data is forwarded.

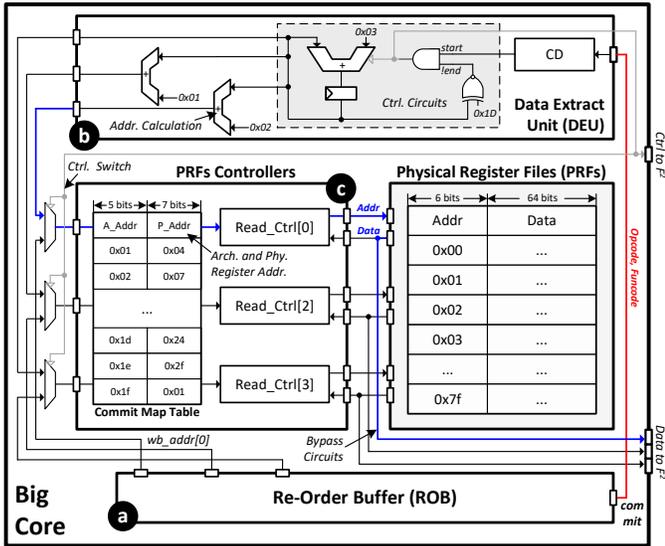


Fig. 3: Big core microarchitecture, extracting status data (red: ROB to DEU; blue: DEU to PRFs): **a** at commit time, opcode and function code are routed; **b** DEU determines whether to extract status data; **c** if so, a signal is routed and preempts the PRF controller for reading.

time rather than execution time to reflect ordering, and before being overwritten by next instructions. In light of this, we develop a *non-intrusive* DEU to establish an observation channel (Fig. 3), containing a Commit Detector (CD), control circuits and bypass circuits, injected at the PRFs, CSRs, and LDQ. The CD monitors instruction commits from the ROB and selects the bypass circuits to extract data at RCPs or run-time data between RCPs. This enables timely data extraction without bringing extra buffers or altering existing register paths.

PRF example. Fig. 3 reveals the microarchitecture using the PRFs (used to collect RCPs) as an example, where the DEU interfaces the ROB, PRFs, and F^2 . At each instruction’s commit, the opcode and function code are routed from the ROB (Fig. 3 **a**) to the CD, allowing the CD to determine whether to extract data (Fig. 3 **b**). If an RCP is reached, a signal is generated by the control circuits, preempting the PRF controller to read register files and forward them to the F^2 (Fig. 3 **c**). The PRF controllers are multiplexed between the ROB and the DEU, where the DEU has priority access when required, enabling immediate data reading and preventing data overwriting³.

B. Forwarding Fabric (F^2)

Previous simulations adopted a naive model for data forwarding, ignoring contention and critical paths [21], [22], [30], [31]. Our attempt to implement such a mechanism revealed a stumbling block impeding system performance, even with the deployment of a full-featured AXI interconnect [32], [33] (Sec. V-D). This is caused by the huge amount of run-time data generated by the big core’s parallel commits, in congestion with the frequent reaches of RCPs, which tends to occur in bursts, particularly when the core retires multiple load or store instructions at RCP boundaries, requiring many data transfers within a single cycle and high throughput.

We design F^2 with Dual-Channel Buffers (DC-Buffers) and a Half-duplex Multicast NoC (HM-NoC), storing and routing the extracted data (Fig. 2 **b**). A DC-Buffer is connected to each commit path, adding independent FIFOs for status and run-time data. This ensures that all run-time data can be stored at the same cycle of commit, even when status data is generated simultaneously, avoiding data needing to be stored inside the core’s own structures for longer than

³A similar microarchitecture is implemented for the CSRs, allowing the data extraction from arbitrary CSR addresses. In contrast, because the top of the LSQ consistently holds data from the most recently retired instructions, when the CD decides to forward the run-time data, the bypass circuits directly transmit from the queue top, minimizing the design complexity.

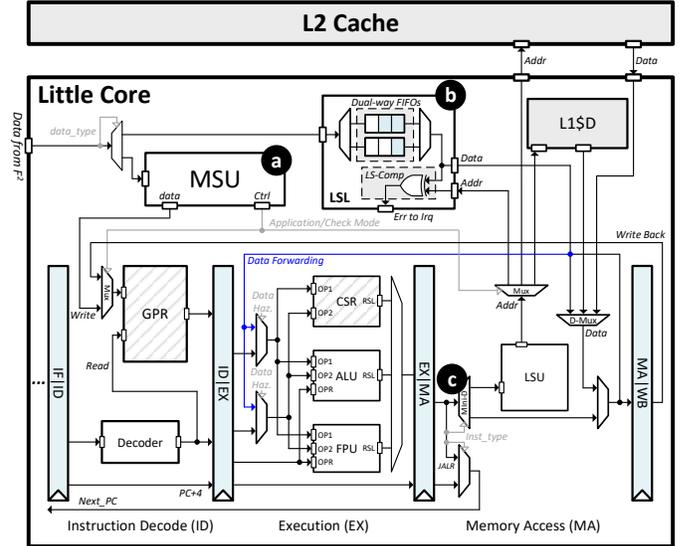


Fig. 4: Little core microarchitecture, upgraded with the MSU and LSL: the **a** MSU servers as the control engineer, and the **b** LSL buffers packets from F^2 . While running the checker thread, the MSU manages its status, and memory data is fetched from the LSL.

in the original core design. HM-NoC uses a half-duplex (1-to-N) Manhattan grid. To achieve high enough throughput, this NoC allows the transmission of two packets per cycle, while preserving ordering⁴.

C. In-order Scalar Core (The Little Core)

To enable thread-level error detection and allow the co-existence of the checker and application threads, the little core’s microarchitecture must support different operational modes and abstract an interface for software control. Hence, we upgrade the little core’s microarchitecture (Fig. 4) with a Mode Switch Unit (MSU, Fig. 4 **a**) and a Load-Store Log (LSL, Fig. 4 **b**). The MSU serves as the control engine, and the LSL buffers received packets. While replicating the big core’s states, the MSU records the little core’s architectural register files and replaces them with the status data from the LSL. The MSU also manages the little core’s operational mode by inspecting the Thread ID (TID) of the running thread with that of the checker thread. When the checker thread is scheduled, the MSU switches the operational mode, where the results of load and non-repeatable instructions are fetched from the LSL. Given that the little core accesses the LSL in an in-order manner, we implement the LSL bank using dual-way FIFOs, reducing complexity than conventional way-associative architectures.

Pipeline integrations. Fig. 4 shows the integration of LSL and MSU into a 5-stage pipelined little core. The LSL is added into the Memory Access (MA) stage (Fig. 4 **b**) by deploying a multiplexer and connecting it to the address port of the Load-Store Unit (LSU). The multiplexer selectively routes memory accesses to the LSL based on the operational mode (returned by the MSU) and combines the virtual index and physical tag (returned by the TLB) into the address port of the LSL. Also, a demultiplexer is integrated into the MA stage to direct the read data back to the next stages. A pair of multiplexer and demultiplexer are deployed into the Instruction Decode (ID) stage to allow the recording and updating of the architectural registers. Lastly, we deploy a Mini-Decoder (Mini-D, Fig. 4 **c**) at the MA stage to differentiate conventional RISC-V and MEEK-ISA.

Performance-gap mitigation. In previous work [21], [22], [26], a lack of microarchitectural detail meant bridging the performance gap

⁴Unlike previous work, where all data of a segment is buffered and forwarded collectively at an RCP, F^2 enables immediate data transmission and use upon collection time, allowing earlier re-execution by the little core. Also, as the same status data might be required by two little cores (respectively used as the SRCP and ERCP), the data is selectively broadcast to the little cores when they are capable of data receiving, eliminating redundant transactions.

Algorithm 1: Big core’s context switch (Blue: added code).

```

1 ▷ Scheduler
2 Function Context_Switch(task *current, core core_index) :
3   MEEK.b.check(DISABLE);
4   Kernel.Intr(DISABLE);
5   task *next = NULL;
6   /* switching current task to the next task */
7   Kernel.Context.save(current);
8   next = Kernel.Find_next();
9   if (next→new_release) then
10    for i = 0 to (size_of(next→checker_index) - 1) do
11      /* hook little cores to the big core */
12      MEEK.b.hook(core_index, next→checker_index[i]);
13    end
14    Kernel.Context.init(next);
15  else
16    Kernel.Context.restore(next);
17  end
18  current = next;
19  Kernel.Intr(ENABLE);
20  MEEK.b.check(ENABLE);
21  Kernel.Context.jalr(current→pc);
22 End Function

```

between the big and little cores involved merely scaling the core count of the little cores. Unfortunately, when we build MEEK using real RTL, the Rocket little cores are each a significant fraction of the size of the big core, and thus using twelve of them as in the original design [21] would be infeasible (section V-E). In particular, we discovered that there was a wide variety in the little cores’ ability to keep up with the big core depending on instruction distribution [34]. With this, we realized that the best way to minimize core overhead while achieving good performance was not to use the smallest Rocket cores available, but rather to *balance* their bottlenecks with respect to BOOM. To reduce the performance gap, we optimize little cores by increasing the size of bottlenecked components, e.g., increasing divider unrolling and extending FPU pipeline.

IV. THE ISA, OPERATING SYSTEM, AND PROGRAMMING MODEL

We detail ISA support needed to add to augment programs with fault-tolerance support. With just a few lines-of-code changes to the kernel, it can schedule and reserve resources for checker threads while allowing standard scheduling and context switching of other threads.

A. ISA Support

The new ISA is classified into two categories (Tab. I) for the big core ($b.x()$) and the little core ($l.x()$). We use $b.hook()$ to set the association between the big and little cores, followed by $b.check()$ to enable/disable the checking capacity via switching on/off the DEU. For little core, $l.mode()$ sets its operational mode, and a pair $l.record()$ and $l.apply()$ record and apply the architectural registers from a given source. To re-direct the PC to an application thread, we develop a $l.jalr()$, modified from the vanilla jump instruction with an alteration on the target. By treating a checkpoint-end as branch-like, the pipeline handles control hazards from the PC-change, without further changes. Lastly, $l.rslt()$ indicates whether an RCP mismatch is detected. As $b.hook()$ and $b.check()$ can lead to contention in use of the little cores, and the $l.mode()$ can cause erroneous execution from unintended memory accesses, they are privileged instructions, executed via OS syscall.

B. Checker Thread and Its Programming Model

The checker thread is initialized with the application thread by augmenting the application thread’s main function using constructor and destructor functions [35] (Al. 1: line 14). Since the checker thread relies on the data in LSL to replay memory operations and the LSL is designed using FIFOs, context switches of the log are undesirable. Hence, during the scheduling time, LSL is reserved for a single checker thread (Al. 1: line 12), even if multiple threads can be scheduled on the core. Once LSL is reserved, only data relevant

Algorithm 2: Little core’s context switch and checker thread.

```

1 ▷ Scheduler
2 Function Context_Switch(task *current, core core_index) :
3   MEEK.l.mode(MODE_APPLICATION);
4   /* switching current task to the next task */
5   ...
6   if (next→checker_thread) then
7     | MEEK.l.mode(MODE_CHECK);
8   end
9   Kernel.Context.jalr(current→pc);
10 End Function
11 ▷ Checker Thread, Newly Developed
12 Function Checker_Thread() :
13   /* initializing checker thread using P-Thread */
14   ...
15   MEEK.l.record(sp); // after checking, returns here
16   if (!MEEK.l.rslt()) then
17     | MEEK.ReportErr();
18   end
19   while (MEEK.NewSRCP()→invalid);
20   MEEK.l.apply(LSL);
21   MEEK.l.jal(MEEK.NewSRCP()→pc);
22 End Function

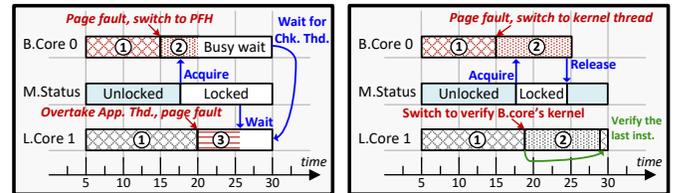
```

to the associated checker thread is forwarded until re-execution is complete. Likewise, a checker thread pinned to a specific application thread cannot be migrated before the re-execution completes. Since each checkpoint is finite in size (5000-instruction maximum), and since ownership returns to the OS after the end of each checkpoint for reallocation, this does not cause resource starvation.

Programming model. We develop the checker thread based on the new ISA, ensuring minimal coding efforts: initially, an $l.record()$ is employed to record the current architectural register state (Al. 2: line 15), allowing the core to return after verification. Then, a busy loop is created to await the arrival of status data in the LSL (Al. 2: line 19). Upon receiving it, $l.apply()$ is invoked to modify the core’s architectural state in accordance with the application segments (Al. 2: line 20), with a $l.jalr()$ being invoked to redirect PC to the replicated target (Al. 2: line 21). Lastly, a $l.rslt()$ is used to return the verification status. If an error is detected, an interrupt is triggered to notify the OS for corrective actions (Al. 2: line 16 - 18).

C. OS Kernel and Its Verification

With the new ISA, kernel modifications can be constrained to the context switch function within the scheduler, allowing the configuration of checking capabilities and the management of checker threads [36], [37]. In accordance with the distinct roles of the big and little cores, the context switches for application and checker threads are modified differently. While entering the context switch from an application thread, $b.check(DISABLE)$ is invoked to deactivate the checking functionality (Al. 1: line 3), and upon leaving the context switch, $b.check(ENABLE)$ is called to reactivate it (Al. 1: line 20). Furthermore, if a newly released thread is scheduled for execution, the $b.hook(core_index, next→checker_index[x])$ is used to associate it with the little cores (Al. 1: lines 10 - 13). For the



(a) Little core overtakes big core, and attempt to acquire the lock held by big core for PFH, causing a waiting chain. (b) By making the little core one inst. behind, page fault is always handled by the big core, avoiding dead lock.

Fig. 5: A deadlock occurs when the little core tries to acquire the lock held by the big core. By building synchronization between cores and I/Os, preventing the checker ever needing to claim locks, the deadlock is resolved (*M.Status*: Memory Status; *PFH*: Page Fault Handler).

context switch of application threads, the only required modification is using the `l.mode()` to switch the little core’s operational mode (Al. 1: lines 3 and 7), leaving the remainder unchanged.

Kernel verification and deadlock resolving. Since MEEK enables error checking at the thread level, the OS kernel can be treated as a specialized application thread and verified like the other application threads. Yet, during the development, we observed a *deadlock* missed in previous literature [21], [22], [26] due to a lack of evaluation with OS. Because a checker thread can block a main thread until the little core has finished, due to SRAM logs being finite, this behavior acts as a lock that the little core holds and the big core needs⁵.

This deadlock can be fixed by making it impossible for a checker thread to fault on instructions. Ensuring the checker thread is at least one instruction behind the main thread means the latter will reach faults first. Synchronizing on I/O makes it impossible to write out a page potentially used by an unfinished checker thread (Fig. 5 (b))⁶.

V. EVALUATION

We built MEEK upon an open-source heterogeneous SoC (Rocket Chip [24]), featuring both high-performance (BOOM [39]) and energy-efficient (Rocket [24]) cores. The BOOM was augmented with DEU and F² as the big core, and the Rocket was upgraded with LSL and MSU to support re-execution as the little core. We implemented the microarchitecture with Chisel (v3.4) and synthesized the RTL using Vivado toolchains (v2021.2). The generated netlist was deployed on AMD UltraScale+ FPGAs using FireSim [40], emulating the setup in Tab. II. We booted Linux kernel v5.7, executing full SPECint 06 [41] and Parsec V3 [42] with simmedium dataset.

A. Performance Overhead

Fig. 6 shows the slowdown experienced by the big core when running SPECint and Parsec in MEEK, compared to the software-based counterparts implemented in LLVM (Nzdc [43])⁷ and LockStep with Equivalent Area (EA-LockStep). Nzdc was chosen for comparison as it is the only open-source software mechanism available, while lockstep is the most widely used hardware mechanism. However, simply duplicating the core in lockstep would consume double the area of the big core while achieving the same performance as a vanilla core, leading to uninteresting comparisons. Therefore, we scaled down the big core’s configurations, through linear interpolation on each configurable BOOM component, to create a comparator with both cores combined matching the area overhead of MEEK. In all cases, MEEK is configured with four little cores.

Using four little cores is sufficient to execute SPEC with overheads of 1.4% and Parsec at 4.4% (geomean). For all workloads, except swaptions, the observed slowdown is below 5%. Swaptions, however, suffers the highest slowdown of 22%, due to the frequent divisions, where the Rocket core’s divider is significantly less performant than the BOOM core’s. For the comparators, Nzdc introduces geomean overheads of 60.2% on Parsec and 94.2% on SPEC, reflecting limitations associated with its software implementation. The hardware counterpart, EA-LockStep, incurred geomean overheads of 31.2% for Parsec and 48.7% for SPEC, approximately 6.1x and 33.7x higher than MEEK, evidencing the performance-area benefits of MEEK.

⁵If there are scenarios where the opposite occurs (the big core holds locks, e.g., software mutexes, required by the little core), then deadlock results. This is impossible during standard re-execution, as the checker thread itself does not access memory and thus cannot take locks; only replaying the memory previously read when the main thread took out locks. However, there are scenarios where the checker thread inadvertently requires real memory reads in order to continue: specifically, if an instruction fault on the little core (e.g., because it has overtaken the big core, or the instruction was paged out before the little core reaches it) [38], the page-fault handling may require a lock held by the main thread, causing deadlock (Fig. 5 (a)).

⁶More generally, care needs to be taken if kernel operations are inside the sphere of replication: similar effects could happen were the scheduler blocked via a lock held by a waiting main thread, in turn blocking checkers.

⁷For Nzdc, compilation fails in gcc, omnetpp, xalancbmk, and freqmine.

Big Core	
Core	4-Width, OoO superscalar SonicBoom, @3.2GHz 128-Entry ROB, 96-entry IQ, 32-entry LDQ/STQ, 128 Int/FP Phy Registers, 2 Int ALUs, 1 FP/Mult/Div
Pipeline	ALU, 2 MEM, 1 Jump Unit, 1 CSR Unit
Branch Pred.	TAGE algorithm, 256-entry BTB, 32-entry RAS, 6 TAGE table with 2 - 64 bits history
Memory Hierarchy	
L1 ICache	32 KB, 4-way, 8 MSHRs
L1 DCache	32 KB, 4-way, 8 MSHRs
L2 Cache	512 KB, 8-way, 12 MSHRs
LLC	4 MB, 8-way, 8 MSHRs
Memory	16 GB DDR3 @1066Mhz, max 32 requests
Little Cores	
Cores	4 × In-order Rocket, 5-stage pipeline, @1.6GHz, 8-Unroll DIV, 3-stage FPU
LSL	4 KB, 5000 instruction time-out
L1 Cache	4 KB, 2-way for both I- and D-Cache

TABLE II: Hardware configurations evaluated.

B. Detection Latency

To examine the detection latency, we inject errors in the forwarded data from the F² connected to the big core, e.g., data and address of memory operations and architectural register data, simulating the hardware faults without disrupting the big core’s normal execution. For each workload, 5,000 - 10,000 faults are randomly generated, and the density of detection latencies is shown in Fig. 7.

Each distribution features a long, but very thin, tail extending to the far right: the average detection latency is below 1 μ s, while the worst-case latencies are 5 to 10 times higher, reaching up to 2.7 μ s (in the ferret). Despite the randomness of the fault injection, the extremely high number of sample points (> 100, 000, in total) suggests empirically that 3 μ s is sufficient to cover over 99.9% of hardware faults, which is multiple orders of magnitude lower than the millisecond-level FTTI requirement for ASIL-D compliance.

C. Scalability

Fig. 8 presents the slowdown observed with varying numbers of little cores. Using two little cores to verify the Parsec execution results in a 54.9% slowdown (geomean), which remains significantly lower than that of Nzdc. With four little cores, the geomean overhead is reduced to 4.4%, with only the swaptions workload experiencing more than 5% overhead (as explained above). Scaling the system to six cores further decreases the geomean overhead to 0.3%, with all workloads having less than 1% overhead. The trend of slowdown exhibits a superlinear decline as the number of little cores increases, which suggests that adding little cores can effectively mitigate performance overhead even for more complex workloads in the future.

D. Microarchitectural Bottleneck Analysis

By detailed implementation, we identified the performance bottlenecks related to data collection, forwarding, and thread re-execution in the heterogeneous error detection process (Sec. III). Here, we provide an in-depth analysis of these bottlenecks and examine the effectiveness of our proposed microarchitecture.

Backpressure decomposition. Fig.9 illustrates the overhead breakdown in MEEK using 4 little cores while running Parsec, employing a full-featured AXI-Interconnect [44] and our F². As shown in Fig. 9, the AXI-Interconnect introduces a significant 16.7% overhead (geomean), evidencing that the 128-bit narrow bus, only handling one packet per cycle, is the primary bottleneck of the entire system, particularly when the little cores operating at a lower clock frequency. This also motivates the development of a dedicated data forwarding fabric with a wider data path and improved packets’ scheduling to mitigate these limitations (Sec. III-B). By replacing the AXI-Interconnect with a 256-bit fabric that supports multiple-packet transmissions per cycle while preserving ordering using FSMs, we effectively reduce the total overhead of data collection and forwarding to less than 5%. Our F² design alleviates this bottleneck by enabling parallel data transmission and improved clock-domain crossing efficiency, thereby minimizing contention. With F², MEEK shifts from being a system limited by data forwarding to one that is computation-bound, with performance depending on the combined compute power of its checker threads.

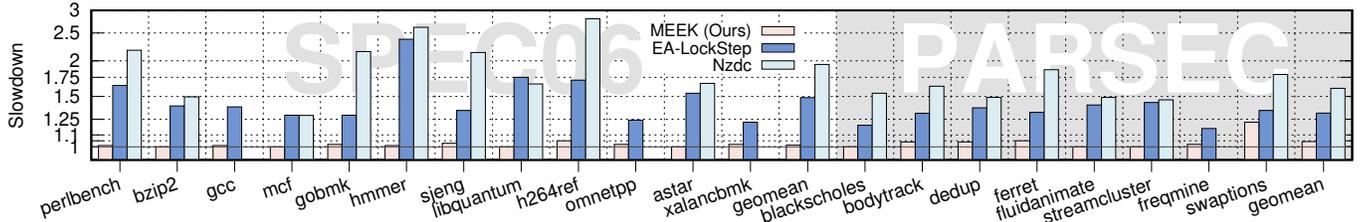


Fig. 6: Performance results for MEEK (4 little cores), Equivalent-Area Lockstep (EA-LockStep) and Nzdc [43], running SPEC06 and Parsec.

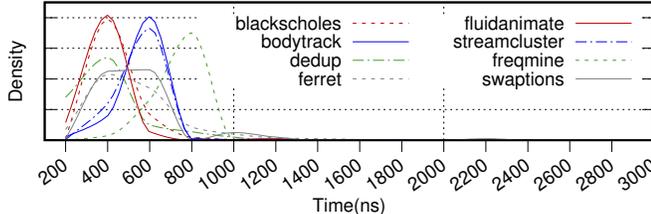


Fig. 7: Detection latency while using 4 little cores (unit: ns).

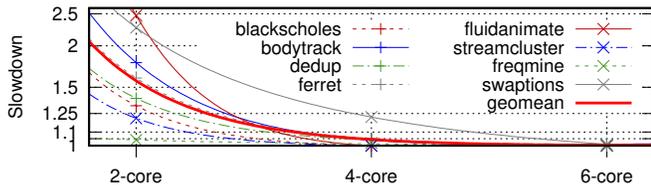


Fig. 8: Slowdown when using varying numbers of little cores.

Little core’s performance-area efficiency. As discussed in Sec.III-C, we analyzed to understand performance gaps between the little and big cores and increased the size of bottlenecked components in the little cores. The evaluation shows that four optimized little cores can match the performance of six default little cores for the verification job. Given that our configurations increase the area of the little cores, we assess them using a normalized performance/area metric for fair comparison (Fig.10), revealing a 15.2% improvement (geomean) and demonstrating the performance-area efficiency of our design choice.

E. Hardware Overhead

We synthesized a physical implementation of MEEK with four little cores using TSMC 28nm PDKs [45]. The RTL is synthesized using Design Compiler (v2022.12), and the netlist is placed and routed via IC Compiler 2 (v2022.12), see Tab. III.

The BOOM’s area is 2.811 mm², and each Rocket is 0.092 mm², excluding L1 D\$ (since it is not required for re-execution [21], [22]). The total area for BOOM’s data collecting and forwarding takes 0.122 mm², (4.3% of the BOOM): the DEU occupies 0.071 mm² and the F² consumes 0.051 mm². In summary, building MEEK with four Rocket upon a BOOM needs 0.726 mm² in total, i.e., 25.8% overhead.

F. Gap Analysis

We observe a discrepancy between the hardware overhead measured in MEEK and the estimation in previous high-level simulations [21]. While both work claims around 25% extra area is enough to enable heterogeneous parallel error detection for a Cortex-A57-level core⁸, this is concluded based on different configurations: twelve Rocket cores in [21] versus four Rockets at double the frequency here.

Upon further analysis (Tab. III), several key factors were identified. First, BOOM is only 72.1% the size of an A57, likely due to differences in ISA complexity. This means that it is misleading to use Rocket to match the performance of an A57 in [21], as a little ARM core with similar performance would be larger than the RISC-V core. Second, our implementation required 17.9% more area per core than the synthesis results used in [21], where their L1 instruction cache configuration was also insufficient for large benchmarks like SPEC (also noted in [22]). Lastly, wrapper logic was previously ignored.

⁸IPCs of BOOM and A57 are similar, 0.95 and 0.97, measured using Parsec.

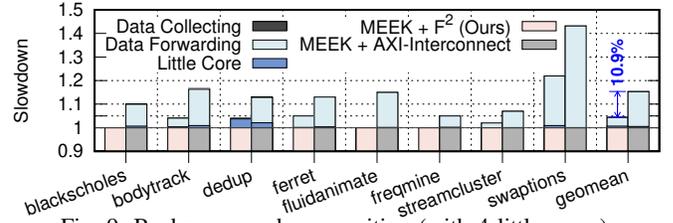


Fig. 9: Backpressure decomposition (with 4 little cores).

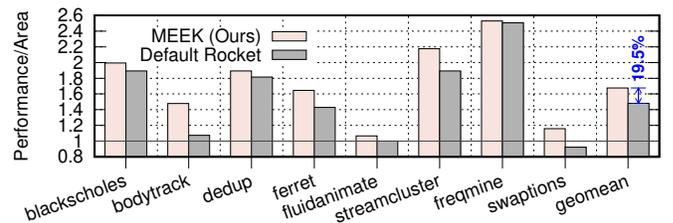


Fig. 10: Performance-area analysis of the little core.

	Big		Little	
	BOOM	Rocket	Cortex-A57	Rocket
Core Number	1	4	1	12
Freq. (GHz)	3.2	2	3.2	1
Tech. (nm)	28	28	20	40
Area (mm ²)	2.811	0.092	2.050	0.160
@28nm	2.811	0.092	3.905	0.078
Wrapper (mm ²)	0.122	0.059	x	x
Overhead	25.8%		24%	

TABLE III: Hardware overhead in MEEK and DSN’18 [21], excluding L1 D\$ in little cores (Grey shading indicates a key discrepancy).

Although the discrepancies in area estimation for individual modules may seem minor, the cumulative effect led to a significant overall mismatch between the simulated and real overhead, meaning we had the budget for only $\frac{1}{3}$ the little cores and so had to carefully optimize each one’s performance (section V-D) rather than scaling the number of cores as was the perspective of the original papers.

VI. CONCLUSION

We have presented MEEK, a systematic implementation of heterogeneous parallel error detection, from the microarchitecture and ISA to the OS and programming model. While the typical architectural research employs abstract modeling (e.g., GEM5 [46]) for rapid prototyping, which is significantly faster than real implementations, this work shows the practical challenges of implementing even a well-modeled architecture, identifying bugs, performance bottlenecks, and inaccurate estimations that are invisible in high-level simulations, as well as opportunities for heavy simplification via codesign. We believe there is much new insight to be gained at this level of analysis for the movement of complex research concepts into full production.

VII. ACKNOWLEDGEMENT

We appreciate the anonymous reviewers for their helpful feedback. This work is supported by the National Key Research and Development Program (Grant No. 2024YFB4405600), the National Natural Science Foundation of China (Grant No. 62472086, 62204036) and the Basic Research Program of Jiangsu (Grants No. BK20243042).

REFERENCES

- [1] C. Hernandez and J. Abella, "Timely error detection for effective recovery in light-lockstep automotive systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 11, pp. 1718–1729, 2015.
- [2] T. M. Austin, "Diva: A reliable substrate for deep submicron microarchitecture design," in *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, 1999, pp. 196–207.
- [3] B. F. Romanescu and D. J. Sorin, "Core cannibalization architecture: improving lifetime chip performance for multicore processors in the presence of hard faults," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 43–51.
- [4] S. K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," in *Proceedings of the 27th annual international symposium on Computer architecture*, 2000, pp. 25–36.
- [5] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson *et al.*, "Addressing failures in exascale computing," *The International Journal of High Performance Computing Applications*, vol. 28, no. 2, pp. 129–173, 2014.
- [6] I. ISO, "26262: Road vehicles-functional safety," 2018.
- [7] L. Rierson, *Developing safety-critical software: a practical guide for aviation software and DO-178C compliance*. CRC Press, 2017.
- [8] Z. Jiang, N. C. Audsley, and P. Dong, "Bluevisor: A scalable real-time hardware hypervisor for many-core embedded systems," in *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2018, pp. 75–84.
- [9] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed design and evaluation of redundant multithreading alternatives," *ACM SIGARCH Computer Architecture News*, vol. 30, no. 2, pp. 99–110, 2002.
- [10] P. Bernardi, R. Cantoro, S. De Luca, E. Sánchez, and A. Sansonetti, "Development flow for on-line core self-test of automotive microcontrollers," *IEEE Transactions on Computers*, vol. 65, no. 3, pp. 744–754, 2015.
- [11] K. Batchner and C. Papachristou, "Instruction randomization self test for processor cores," in *Proceedings 17th IEEE VLSI Test Symposium (Cat. No. PR00146)*. IEEE, 1999, pp. 34–40.
- [12] S. E. Michalak, K. W. Harris, N. W. Hengartner, B. E. Takala, and S. A. Wender, "Predicting the number of fatal soft errors in los alamos national laboratory's asc q supercomputer," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 329–335, 2005.
- [13] X. Iturbe, B. Venu, E. Ozer, and S. Das, "A triple core lock-step (tcls) arm@ cortex@-r5 processor for safety-critical and ultra-reliable applications," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*. IEEE, 2016, pp. 246–249.
- [14] X. Iturbe, B. Venu, E. Ozer, J.-L. Poupat, G. Gimenez, and H.-U. Zurek, "The arm triple core lock-step (tcls) processor," *ACM Transactions on Computer Systems (TOCS)*, vol. 36, no. 3, pp. 1–30, 2019.
- [15] B. Stolt, Y. Mittlefehldt, S. Dubey, G. Mittal, M. Lee, J. Friedrich, and E. Fluhr, "Design and implementation of the power6 microprocessor," *IEEE Journal of Solid-State Circuits*, vol. 43, no. 1, pp. 21–28, 2008.
- [16] "Renesas: Rh850 micro-controller family," <https://www.renesas.com/eu/en/products/microcontrollers-microprocessors/rh850.html>.
- [17] "Infineon: 32-bit TriCore Microcontroller," <https://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-microcontroller/>.
- [18] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy, "Doubleplay: Parallelizing sequential logging and replay," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 1, pp. 1–24, 2012.
- [19] S. Gupta, S. Feng, A. Ansari, J. Blome, and S. Mahlke, "The stagenet fabric for constructing resilient multicore systems," in *2008 41st IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2008, pp. 141–151.
- [20] Z. Gao, C. Cecati, and S. X. Ding, "A survey of fault diagnosis and fault-tolerant techniques—part i: Fault diagnosis with model-based and signal-based approaches," *IEEE transactions on industrial electronics*, vol. 62, no. 6, pp. 3757–3767, 2015.
- [21] S. Ainsworth and T. M. Jones, "Parallel error detection using heterogeneous cores," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2018, pp. 338–349.
- [22] —, "Paramedic: Heterogeneous parallel error correction," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2019, pp. 201–213.
- [23] Á. B. de Oliveira, G. S. Rodrigues, F. L. Kastensmidt, N. Added, E. L. Macchione, V. A. Aguiar, N. H. Medina, and M. A. Silveira, "Lockstep dual-core arm a9: Implementation and resilience analysis under heavy ion-induced soft errors," *IEEE Transactions on Nuclear Science*, vol. 65, no. 8, pp. 1783–1790, 2018.
- [24] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelievitz *et al.*, "The rocket chip generator," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, vol. 4, pp. 6–2, 2016.
- [25] K. Mitropoulou, V. Porpodas, and T. M. Jones, "Comet: communication-optimised multi-threaded error-detection technique," in *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ser. CASES '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: <https://doi.org/10.1145/2968455.2968508>
- [26] S. Ainsworth, L. Zoubritzky, A. Mycroft, and T. M. Jones, "Paradox: Eliminating voltage margins via heterogeneous fault tolerance," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 520–532.
- [27] ARM, "big.little architecture," in <https://www.arm.com/technologies/big-little>, 2024.
- [28] J. Doweck, W.-F. Kao, A. K.-y. Lu, J. Mandelblat, A. Rahatekar, L. Rappoport, E. Rotem, A. Yasin, and A. Yoaz, "Inside 6th-generation intel core: New microarchitecture code-named skylake," *IEEE Micro*, vol. 37, no. 2, pp. 52–62, 2017.
- [29] B. Schroeder, E. Pinheiro, and W.-D. Weber, "Dram errors in the wild: a large-scale field study," *ACM SIGMETRICS Performance Evaluation Review*, vol. 37, no. 1, pp. 193–204, 2009.
- [30] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *Proceedings of the 37th annual international symposium on Computer architecture*, 2010, pp. 37–47.
- [31] F. Restuccia, A. Biondi, M. Marinoni, G. Cicero, and G. Buttazzo, "Axi hyperconnect: A predictable, hypervisor-level interconnect for hardware accelerators in fpga soc," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.
- [32] Z. Jiang, K. Yang, N. Fisher, I. Gray, N. C. Audsley, and Z. Dong, "Axi-ic^{RT}: Towards a real-time axi-interconnect for highly integrated socs," *IEEE Transactions on Computers*, vol. 72, no. 3, pp. 786–799, 2022.
- [33] Z. Jiang, K. Yang, N. Audsley, N. Fisher, W. Shi, and Z. Dong, "Bluescale: a scalable memory architecture for predictable real-time computing on highly integrated socs," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 1261–1266.
- [34] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd annual ieee/acm international symposium on microarchitecture*, 2009, pp. 469–480.
- [35] "IBM: Overview of constructors and destructors (C++ only)," <https://www.ibm.com/docs/en/xl-c-and-cpp-aix/16.1?topic=only-overview-constructors-destructors-c>.
- [36] J. Aas, "Understanding the linux 2.6. 8.1 cpu scheduler," *Retrieved Oct*, vol. 16, pp. 1–38, 2005.
- [37] R. Love, *Linux kernel development*. Pearson Education, 2010.
- [38] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.
- [39] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, "Sonicboom: The 3rd generation berkeley out-of-order machine," in *Fourth Workshop on Computer Architecture Research with RISC-V*, vol. 5, 2020, pp. 1–7.
- [40] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra *et al.*, "Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 29–42.
- [41] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.
- [42] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 72–81.
- [43] M. Didehban and A. Shrivastava, "nzdc: A compiler technique for near zero silent data corruption," in *Proceedings of the 53rd Annual Design Automation Conference*, 2016, pp. 1–6.
- [44] "Xilinx: AXI-Interconnect," https://www.xilinx.com/products/intellectual-property/axi_interconnect.html.
- [45] "28nm PDKs," https://www.tsmc.com/english/dedicatedFoundry/technology/logic/l_28nm.
- [46] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashtu *et al.*, "The gem5 simulator," vol. 39, no. 2. ACM New York, NY, USA, 2011, pp. 1–7.