FlexStep: Enabling Flexible Error Detection in Multi/Many-core Real-time Systems

Tinglue Wang[†], Yiming Li[†], Wei Tang[†], Jiapeng Guan[‡], Zhenghui Guo[†], Renshuang Jiang[§], Ran Wei[¶]*, Jing Li[‖]*, Zhe Jiang[†]*
[†]Southeast University, China. [‡]Dalian University of Technology, China. [§]National University of Defense Technology, China.
[¶]Lancaster University, UK. [‖]New Jersey Institute of Technology, US. *The corresponding authors.

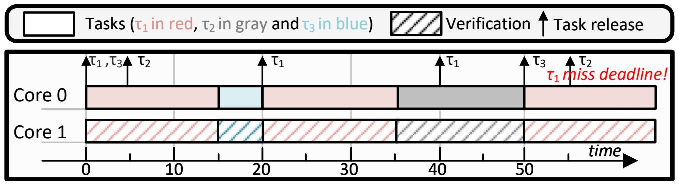
***Abstract*—Reliability and real-time responsiveness in safety-critical systems have traditionally been achieved using error detection mechanisms, such as LockStep, which require pre-configured checker cores, strict synchronisation, static error detection regions, or limited preemptions. However, these core-bound hardware mechanisms often lead to significant resource over-provisioning and diminished real-time performance in modern systems where tasks with varying reliability requirements are consolidated on shared processors for efficiency and cost reduction. To address these challenges, this work presents FlexStep, a systematic solution that integrates hardware and software across the SoC, ISA, and OS scheduling layers. FlexStep features a novel microarchitecture that supports dynamic core configuration and asynchronous, preemptive error detection. The FlexStep architecture naturally allows for flexible task scheduling and error detection, enabling new scheduling algorithms that enhance both resource efficiency and real-time schedulability.**

I. Introduction

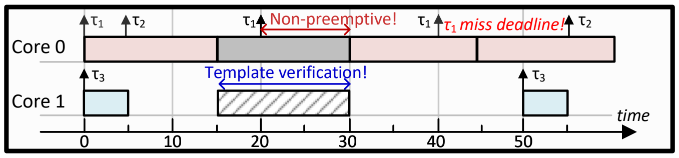
In safety-critical systems, such as automotives and avionics, processor cores must simultaneously detect and correct faults during execution [1]–[4] and guarantee task schedulability [5]–[7] as the foundational components to ensure reliable system. However, since reliability and schedulability address distinct safety dimensions, they are usually achieved through different development phases and across various architectural levels. Reliability is often ensured through hardware-level error detection mechanisms, such as LockStep used in ARM Cortex R series [8]–[11]. While software mechanisms exist [12]–[15], they offer limited coverage and degrade performance significantly. Conversely, schedulability is ensured through scheduling algorithms implemented in the Operating System (OS) [16]–[18].

Existing works. As hardware computational capabilities advance and software applications diversity expands, rising demand emerges to execute multiple tasks with varying reliability requirements on shared processor cores [19]–[23]. In this context, LockStep technology faces significant limitations due to its rigid hardware design. LockStep binds one or more identical cores, synchronously executing same program and comparing output results at each cycle. This design mandates that all tasks undergo the highest level of error detection, irrespective of their actual reliability needs, which leads to inefficient use of error detection capabilities, suboptimal resource utilisation, significant power and area overheads, and challenges to system schedulability. As Fig.1(a) demonstrates, LockStep's mandatory error checking for non-verification tasks like τ_1 and τ_3 causes unnecessary resource consumption and the third job of τ_1 to miss its deadline.

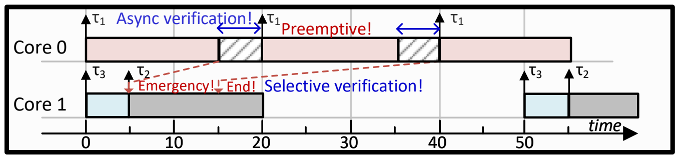
Methods supporting reconfiguration, such as LockStep with split-lock [24]–[26], have been developed to reduce resource overhead. For instance, the most recent Hybrid Modular Redundancy (HMR) approach [24] explicitly separates mission-critical and performance-critical code sections. It leverages hardware designs to enable runtime reconfiguration for split-lock, providing a degree of flexibility. However, the inherent limitations of the "core binding" still impose significant constraints. Specifically, when reconfigured into verification mode, checker cores must perform error detection synchronously with main core. Moreover, this synchronous error-checking execution



(a) LockStep configuration with fixed main core 0 & checker core 1.



(b) HMR [24] configuration with limited flexibility and synchronous checking.



(c) FlexStep allowing asynchronous, selective, and preemptive checking.

Fig. 1: Schedules on different dual-core architectures. τ_1, τ_2, τ_3 have implicit deadlines and worst-case execution time of 15, 15, 5. Non-verification tasks τ_1 and τ_3 do not require checking. An emergency occurs when τ_2's first job arrives and requires checking its first 10 units of work.

cannot be preempted by non-verification tasks, even if those tasks have higher priorities or earlier deadlines. Additionally, HMR can only statically perform checking for pre-determined tasks, lacking the capability to provide selective error checking based on dynamic system requirements. These constraints significantly reduce flexibility and impair schedulability. As shown in Fig. 1(b), although HMR with runtime split-lock reduces resource wastage by not performing error checking for non-verification tasks τ_1 and τ_3 (τ_3 can be executed on core 1), the "core binding" design still prevents τ_1 from preempting τ_2's error checking, resulting in τ_1 missing its second deadline.

Contributions. To guarantee reliability and schedulability for safety-critical systems while maintaining resource efficiency, developing more flexible hardware error detection architecture has become an industry necessity [26]–[28]. As illustrated in Fig. 1(c), such architecture should decouple cores from rigid LockStep and employ a flexible error detection mechanism. This mechanism, with OS support, should enable *error detection that is asynchronous, selective, and preemptable* by non-verification tasks. However, achieving this level of flexibility and efficiency presents significant technical challenges.

To address these challenges, we present FlexStep, a comprehensive full-stack framework that integrates hardware and software to deliver high flexibility and configurability building on three key technologies. First, we modified the Rocket [29] microarchitecture to enable

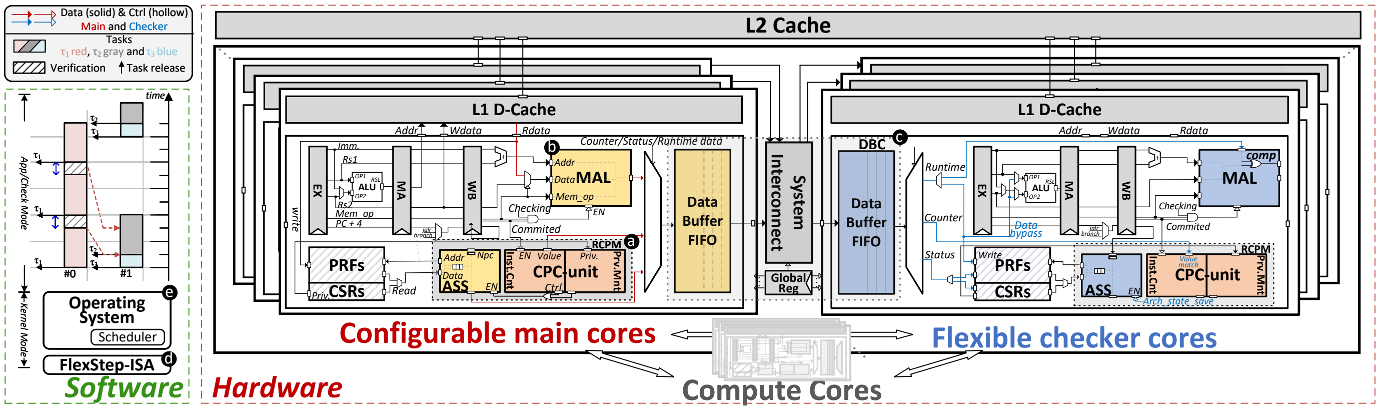


Fig. 2: FlexStep overview. At the hardware level, colored modules represent functional units added by FlexStep: orange hues denote identical functionality, while yellow and blue hues highlight variations in functionality for different cores. (a) Register Checkpoint Management (Sec. III-A). (b) Memory Access Log (Sec. III-B). (c) Data Buffering and Channelling (Sec. III-C). At the software level, FlexStep provides (d) customised ISA (Tab. I) and enables (e) a control flow to perform context switching between verification and non-verification tasks and flexible scheduling.

dynamic hardware reconfiguration and asynchronous error detection at thread level, establishing foundation for scheduling flexibility. Second, through abstraction of hardware configurations into a customized Instruction Set Architecture (ISA), we integrate runtime hardware control with OS context switching, allowing dynamic task preemption and resource reallocation. Third, the proposed full-stack framework systematically coordinates multicore architecture extensions with OS scheduling primitives, while our formalized theoretical model enables novel scheduling algorithms that exploit hardware flexibility to improve real-time schedulability. This approach overcomes the limitations of synchronous, core-binding methods through preemptable error detection and adaptive resource management for tasks with varying reliability requirements. We publicly release FlexStep's source code, at *https://anonymous.4open.science/r/FlexStep-DAC25-7B0C*.

We implemented FlexStep on AMD Alveo U280 FPGA and evaluated it using various metrics. Results demonstrate that FlexStep achieves microsecond-level detection latency with 1.07% slowdown, 2.21% area overhead, and 2.89% power consumption while delivering improved schedulability compared to LockStep and HMR.

II. FlexStep: Hardware/Software Co-design Framework

In FlexStep, any processor core can be configured as a main or checker core. The main core executes applications like a standard core, while checker core verifies the correctness of main core by re-running the same program. Fig. 2 illustrates the FlexStep framework.

Unlike the synchronised verification in LockStep, FlexStep employs error detection based on Register Checkpoints (RCPs) – architectural states at specific points – and memory accesses, similar to Paramedic [30]–[32]. Specifically, a user thread running on main core is divided into small checking segments, which can be reproduced on its associated checker core(s). Each segment is identified by ***Start RCPs*** (***SCPs***) and ***End RCPs*** (***ECPs***) stored in RCP Management units (Fig. 2.(a), Sec. III-A). The checker core halts memory access and sequentially replays the segments by initializing its architectural state to the segment's SCPs and compares its final state to the ECPs. During execution, memory access data (e.g., addresses and data for `LD/ST`, `LR/SC`, `AMO` instructions) recorded in Memory Access Log (Fig. 2.(b), Sec. III-B) are used for replay execution and verified at runtime. Main core execution is deemed correct if the ECPs of all checking segments and memory access data match the original traces.

The fundamental idea is that, as long as all data related to RCPs and memory accesses – required for execution replay and verification – are recorded and buffered, the checker thread can be executed *asynchronously* on any other core. Therefore, checker core can execute other tasks instead of verifying, as long as the necessary data of verification task have been extracted from main core. This design improves both utilisation and flexibility. FlexStep allows user threads running on any core to be duplicated and verified on different core(s) using a data buffer and ***configurable interconnected channel*** (Fig. 2.(c), Sec. III-C). Unlike LockStep and HMR, which supports specific modes such as Dual-Core LockStep (DCLS) and Triple-Core LockStep (TCLS) by binding cores via shared input path , FlexStep's interconnected channel can be configured to operate in one-to-one (similar to DCLS), one-to-two (similar to TCLS), or more modes. This accommodates scenarios with varying safety-criticality demands.

To manage cores and schedule error detection, FlexStep abstracts the configurable hardware control interfaces as a ***customised ISA*** (Fig. 2.(d), Tab. I) and integrates a control flow into OS. The ISA defines the instructions specific to main or checker core and global instructions, exposing all ***core attributes*** (main, checker, or compute) to OS for runtime configuration and scheduling facilitation (Fig. 2.(e)). Preemptive execution is supported, allowing any core to be interrupted and enabling more urgent tasks to be executed. Thus, the OS can dynamically configure core attributes and verification modes as needed, leveraging asynchronous verification and task preemption to optimise real-time scheduling while ensuring reliability.

III. Microarchitecture (μ-arch)

Implementing hardware-level FlexStep requires μ-arch modifications adding several key functional units, as illustrated in Fig. 2. We used the Rocket core as a case study to implement our framework and evaluate its effectiveness. Noted that incorporating identical units into each core is crucial for dynamic attribute switching, which requires maximising functional unit reuse to enhance utilisation and reduce area costs. This section discusses the details of the modified μ-arch.

A. Register Checkpoint Management Units (RCPM)

RCPM (Fig. 2.(a)) consists of ***Checkpoint Control*** (***CPC***) and ***Architectural State Snapshot*** (***ASS***). CPC controls the start and end of a checking segment in the main core, and identify segment boundaries and verify execution correctness in the checker core. ASS is a storage unit, which captures Register Checkpoint snapshots and releases them for transmitting or applying under the control of CPC.

TABLE I: FlexStep ISA, abstracting control interface for software.

Instruction	Description
`G.IDs.contain`	Return core attributes (Main/Checker)
`G.Configure`	Configure the main and checker cores'ID
`M.associate`	Allocate one or multiple checker core(s) to main
`M.check`	Enable/Disable the checking function
`C.check_state`	Switch the checking state (busy/idle)
`C.record`	Record the context to ASS
`C.apply`	Apply the SCP from data channel
`C.jal`	Jump to the next pc (npc) of SCP
`C.result`	Return the comparison result

Checkpoint Control (CPC) includes an instruction counter (Fig. 2. Inst.Cnt) and a privilege mode monitor (Fig. 2. Priv.Mntr). FlexStep only supports user mode checking and all cores could enter kernel during execution, resulting in premature extermination (Fig. 3.①) and temporary deviation (Fig. 3.②) of a checking segment.

CPC's Mechanism. For main core, a new checkpoint is generated when: a) privilege mode switches; b) instruction count limit is reached (default is 5000). At ECP, the counter stops and records the number of instructions of the segment. During the segment, main core sends SCP, `LD/ST` log, instruction count (IC), and ECP in order (Fig. 3) for checker core to receive. For checker core, upon applying SCP, it begins checking till the instruction count reaches the same value as main core before using its own architectural state to verify ECP.

Architectural State Snapshot (ASS) facilitates duplicate execution by temporarily storing Checkpoints and architectural states. It enables immediate access to architectural states and reduces the overhead of frequent memory access, achieving fast execution state switching.

B. Memory Access Log Unit (MAL)

For main core, information required for duplicate execution and verification needs to be recorded and transmitted during checking segments. MAL (Fig. 2.ⓑ) records data and addresses of a memory instruction, packaging them for transmission in the main core, or making comparisons to report potential errors in the checker core.

Handling instructions with multiple micro-ops. Regular `LD/ST` instructions are packaged into a single entry, while multi-micro-op instructions (`LR`, `SC` and `AMO`) use multiple entries to minimize data width. This introduces slight latency but reduces storage overhead, outweighing disadvantages due to their low frequency in workloads.

C. Data Buffering and Channelling Units (DBC)

DBC (Fig. 2.ⓒ) consists of ***Data Buffer FIFO*** to buffer data of checking segments for conflict resolution and asynchronous error checking and ***System Interconnect*** that establishes and alters links of the FIFOs between main cores and their associated checker cores.

Configurable Inter-core Channels. Inter-core communication channels between main and checker cores are achieved through the System Interconnect, a fully connected MUX-DEMUX network laid out between FIFO of each core. A global register, which can be modified by custom instructions, generates control signals for MUX/DEMUX to establish channels between the main core and one or more checker cores. This design eliminates communication conflicts and ensures minimal communication latency at a small scale. As the design scales up, the wiring complexity grows exponentially, and the interconnect could be replaced with a bus interconnect or NoC.

Buffering for Conflict Resolving and Asynchrony. In line with previous work [30]–[32], an SRAM-based FIFO is used for data buffering, enabling the checker core to retrieve information directly from the FIFO instead of main memory. This supports asynchronous verification; with larger capacity comes greater flexibility in scheduling. The main core's FIFO resolves conflicts when multiple main cores compete for a checker core, buffering data until the checker core is available. Additional main memory buffering, accessed via DMA, can further support conflict resolution and asynchrony.

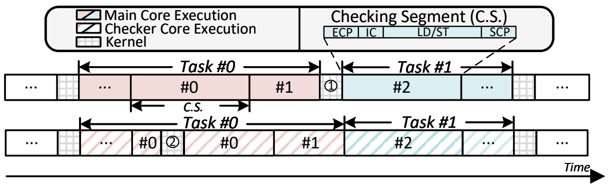


Fig. 3: Checking segments: their components, main and checker execution of them, and them of default length or interrupted by kernel.

IV. New ISA and OS Kernel Add-ons

In FlexStep, the OS plays a pivotal role in managing hardware resources and ensuring real-time schedulability. To enhance its functionality, we leverage a control interface within the ISA (Tab. I), enabling efficient core management with minimal modifications to the OS. This approach ensures compatibility with existing application programs while improving overall scheduling performance.

A. New ISA Support

As shown in Tab. I, the new ISA is classified into three categories for the main core (`M.x()`), checker core (`C.x()`) and both (`G.x()`). Global instructions include `IDs.contain` to identify core attributes and `Configure` to write the IDs of main and checker cores into global configuration registers. For main cores, `associate` allocates checker cores for redundant execution, and `check` controls the checking capability by managing the relevant units. For checker cores, we present `check_state` for changing the checking state of the core, and `record` for recording the architectural state into the ASS. In addition, a pair of "atomic" instructions `apply` and `jal` are presented to replay the checking segment on checker cores.

Algorithm 1: Each core's context switch.

```
Function Context_Switch(task *current, core core_id):
    if (G.Main_IDs.contain(core_id)) then
        M.check.disable();
    else if (G.Checker_IDs.contain(core_id)) then
        C.check_state(idle);
    Kernel.Intr(DISABLE);
    task *next = NULL;
    Kernel.Context.save(current);
    next = Kernel.Find_next();
    if (next→new_release) then
        /* Configure the main and checker core's ID */
        G.Configure(Main_IDs, Checker_IDs);
        Kernel.Context.init(next);
    else then Kernel.Context.restore(next);
    current = next;
    Kernel.Intr(ENABLE);
    if (G.Main_IDs.contain(core_id)) then
        /* Associate checker cores and enable checking */
        M.associate(Checker_ID(s));
        M.check.enable();
    else if (G.Checker_IDs.contain(core_id) and
      next→checker_thread) then
        C.check_state(busy);
    Kernel.Context.jalr(current→pc);
```

The `apply` is used to update the architectural registers, while the `jal` is used to jump to the main thread, which is modified from the standard jump instruction with a designated target from the main cores for accurate branch misprediction handling without necessitating any changes to the microarchitecture. Finally, `result` reports the comparison result at each ECP.

B. OS Kernel Add-ons and Customised Checker Thread

The modification of OS (Fig. 2.ⓔ) is shown in Al. 1, which requires minimal code additions to the scheduler's context switch function. During context switch, cores executes different instructions based on its attributes to switch off checking function (Al. 1: lines 2 - 5). Additionally, when a new task is released, `Configure` (re-)initializes the global registers (Al. 1: line 12). Finally, for main cores, `associate` equips it with the specific checker core(s), while `check_enable` activates the checking function (Al. 1: lines 19 - 20). For checker cores, it will switch to busy state upon receiving verification tasks(Al. 1: line 22) and enter a checker thread.

Besides the general modification of the context switch, we have also developed a dedicated thread for checker cores (Al. 2). Similar to the regular context switch, `record` is used to store current architectural state in ASS for state restoration (Al. 2: line 4). Next, the checker core will enter a loop and continuously request new SCP from the FIFO (Al. 2: line 6). Once a new SCP arrives, `apply` will deploy it to the architectural registers (Al. 2: line 7), followed by `jal` to control the directional jump of PC (Al. 2: line 8), thereby ensuring that the checker core executes the correct sequence of instructions. Lastly, the verification result is returned by `result` (Al. 2: lines 5), and the correction mechanism will be triggered if an error is detected.

Algorithm 2: Customised checker thread.

```
1 Function Checker_Thread():
2     /* launching checker thread with P-Thread */
3     ...
4     C.record(ASS); // return position after checking
5     if(!C.rslt()) then ReportErr();
6     while (C.NewSCP() != ready);
7     C.apply(C.NewSCP.data);
8     C.jal(C.NewSCP.npc);
```

V. System Model and Scheduling Analysis

We now describe formal model and analysis for scheduling problem with verification ensuring reliability and real-time guarantees.

Model. We seek to schedule a task set consisting of n sporadic tasks on m cores. Each task τ_i is characterised by its worst-case execution time C_i, period T_i, and implicit deadline $D_i = T_i$. Tasks can be classified into three types: (1) A task in $\mathcal{T}^N$ is a non-verification task that only needs to execute its normal work once every period and meet its deadline. (2) A task in $\mathcal{T}^{V2}$ may require double-check online. (3) A task in $\mathcal{T}^{V3}$ may require triple-check online. In the event of an emergency, the system dynamically triggers additional error checking for one or more jobs of specific verification tasks based on the nature of the emergency. Double-check (or triple-check) verification involves duplicating the computation under verification once (or twice) and reproducing it on one (or two) cores distinct from the core performing the original computation.

If asynchronous verification is supported, the duplicated computations can be executed after the original computation but must still be completed before the job deadline to ensure timely error detection and maintain system reliability. As discussed earlier, FlexStep supports both asynchronous verification and selective error checking, allowing error checking to be dynamically triggered for a task and performed on specific portions as needed. As an initial step in modeling of this flexibility, we limit our focus on asynchronous verification only, while assuming that all jobs of verification tasks require full error checking and must meet their deadlines. Without additional information, such as the probability of dynamic verification or assumptions allowing low-criticality tasks to miss deadlines during system emergencies, this formulation represents the system's worst-case behaviour.

Scheduling algorithm and analysis. For this scheduling problem, we propose a simple scheduling algorithm and corresponding schedulability analysis based on the partitioned Earliest Deadline First (partitioned EDF). The algorithm partitions all non-verification tasks, verification tasks, and their duplicated computations for double-check or triple-check verification on m available cores.

Algorithm 3: Scheduling with asynchronous verification.

```
1  Input: $\Gamma$: Task set $\tau_i \in \{\mathcal{T}^N, \mathcal{T}^{V2}, \mathcal{T}^{V3}\}$, $m$ available cores
2  Output: Deadline-compliant task partitioning
3  for each core $k \in \{1, \ldots, m\}$ do $\Delta[k] \leftarrow 0$;
4  for each $\tau_i \in \{\mathcal{T}^{V3}, \mathcal{T}^{V2}\}$ do
5      if $\tau_i \in \mathcal{T}^{V2}$ then $D'_i \leftarrow D_i/2$;
6      else if $\tau_i \in \mathcal{T}^{V3}$ then $D'_i \leftarrow (\sqrt{2}-1)D_i$;
7      $\delta^{\mathrm{o}}_i \leftarrow C_i/D'_i$; $\delta^{\mathrm{v}}_i \leftarrow C_i/(D_i - D'_i)$;
8      $k \leftarrow \mathrm{argmin}_{k \in \{1,\ldots,m\}} \Delta[k]$;
9      Assign $\tau^o_i$ to core $k$; $\Delta[k] \leftarrow \Delta[k] + \delta^{\mathrm{o}}_i$;
10     $k' \leftarrow \mathrm{argmin}_{k' \in \{1,\ldots,m\} \setminus \{k\}} \delta[k']$;
11     Assign $\tau^v_i$ to core $k'$; $\Delta[k'] \leftarrow \Delta[k'] + \delta^{\mathrm{v}}_i$;
12     if $\tau_i \in \mathcal{T}^{V3}$ then
13         $k'' \leftarrow \mathrm{argmin}_{k'' \in \{1,\ldots,m\} \setminus \{k,k'\}} \Delta[k'']$;
14         Assign $\tau^{v'}_i$ to core $k''$; $\Delta[k''] \leftarrow \Delta[k''] + \delta^{\mathrm{v}}_i$;
15 for each $\tau_i \in \mathcal{T}^N$ do
16     $\delta^{\mathrm{o}}_i \leftarrow C_i/D_i$;
17     $k \leftarrow \mathrm{argmin}_{k \in \{1,\ldots,m\}} \Delta[k]$;
18     Assign $\tau_i$ to core $k$; $\Delta[k] \leftarrow \Delta[k] + \delta^{\mathrm{o}}_i$;
19 for each core $k \in \{1, \ldots, m\}$ do
20     if $\Delta[k] > 1$ then return Fail;
21 return Success;
```

Since error-checking computations cannot begin before the original computation, our algorithm assigns a virtual deadline D'_i to each verification task to reserve enough time for verification. This virtual deadline is used for scheduling the original computation on its assigned core using EDF, while the original deadline is used for scheduling the duplicated computations. For a double-check task $\tau_i \in \mathcal{T}^{V2}$, $D'_i = D_i/2$, and for a triple-check task $\tau_i \in \mathcal{T}^{V3}$, $D'_i = (\sqrt{2}-1)D_i$. The virtual deadline is chosen to minimise the sum of the density of the original computation $\delta^o_i = C_i/D'_i$ and the density of the duplicated computation $\delta^v_i = C_i/(D_i - D'_i)$, optimising system schedulability. Note that, in practice, duplicated computations can start as soon as the original computation begins execution. However, our schedulability analysis consider the worst-case scenario — the error-checking starts only after the virtual deadline.

Our partitioning algorithm (Al. 3) partitions verification tasks with descending utilisation and ensures that their original and error-checking computations are allocated to different cores (Al. 3: lines 4-14). Their calculated densities δ^o_i, δ^v_i update the total density of allocated cores. Non-verification tasks are then partitioned with descending utilisations (Al. 3: lines 15-18). As EDF is used to execute tasks allocated to each core, core k is schedulable if its total density

($\Delta[k]$) of assigned tasks does not exceed one. The entire task set is guaranteed to be schedulable if all tasks (and their error-checking) are successfully assigned to cores and all cores remain schedulable (Al. 3: lines 19-20). Since our schedulability test is a sufficient test,when the test fails and hard real-time guarantees are not required, we can remove the virtual deadline and use the verification task's original deadline and utilisation for scheduling and partitioning.

VI. Evaluation

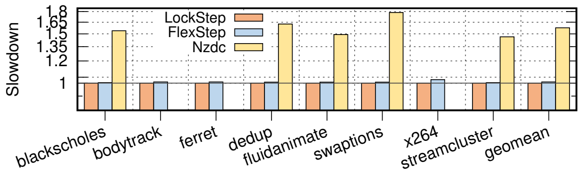
We built FlexStep featuring homogeneous SoC with multiple Rockets and implemented the microarchitecture upon an open-source platform Chipyard [33] (v1.8.0). With TSMC 28nm PDKs [34], we synthesised the RTL using Synopsys (v2019.12). We deployed the RTL design on the AMD Alveo U280 FPGA using FireSim [35] to simulate the settings in Tab II and boot Linux kernel (v5.7), executing full SPECint 06 [36] and Parsec V3 [37] with simmedium dataset.

TABLE II: Hardware configurations evaluated.

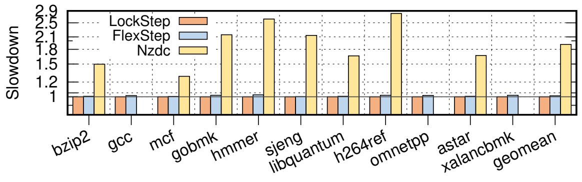
	Homogeneous Core
Core	In-order scalar Rocket, @1.6GHz
Pipeline	5-stage pipeline, 64 Int/FP Phy Registers, 1 ALU, 1 DIV, 1 FPU, 1 CSR
Branch Pred.	512-entry BHT, 28-entry BTB, 6-entry RAS
	Memory Hierarchy
L1 I-Cache	16 KB, 4-way, Blocking, 2 LatencyCycles
L1 D-Cache	16 KB, 4-way, Blocking, 2 LatencyCycles
L2 Cache	512 KB, 8-way, 8 MSHRs, 40 LatencyCycles

A. Performance Overhead

Experiment setup. We executed SPECint and Parsec with dual-core mode and compared the performance slowdown of Flexstep with the most widely used hardware scheme Lockstep, and the only available open-source software error-detection scheme Nzdc [38] which fails to compile on some workloads (e.g., bodytrack, ferret, gcc).



(a) Parsec (v3.0).



(b) Full SPECint CPU2006.

Fig. 4: Performance slowdown using LockStep, FlexStep and Nzdc.

Results. As shown in Fig. 4, FlexStep incurs a performance slowdown (geomean) of 1.07% when running Parsec and 1.24% when running SPEC, which is attributed to the extraction of RCPs and backpressure from Data Buffering resulting from cores undergoing different kernel mode switches and instruction executions. In contrast, the slowdown of Nzdc is almost 57.7% in Parsec and 91.5% in SPEC, roughly 1.56x and 1.89x slower than FlexStep. Nzdc experiences a significant performance degradation due to the redundant checking instructions. While LockStep incurs no performance drop, it requires an additional equivalent area. FlexStep achieves minimal slowdown with reasonable area overheads (Sec. VI-E), highlighting its advantages in both performance and resource efficiency.

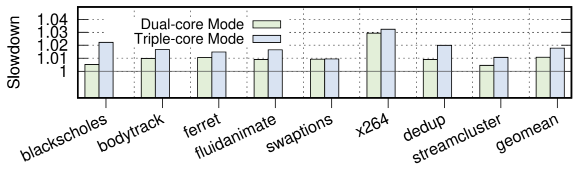


Fig. 5: Performance slowdown in dual-core and triple-core mode.

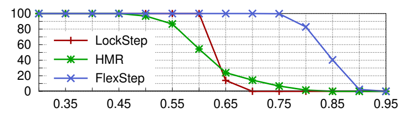
We further compare the slowdown of Parsec under different verification modes in Fig. 5. Results indicate that the slowdown (geomean) in triple-core mode increases slightly to 1.77%. This modest increase is attributed to having more checker cores, which exacerbates the execution inconsistency between cores, leading to more frequent backpressure on the main core. Overall, the performance overhead in both modes remain minimal, demonstrating that FlexStep's support for different verification modes is highly performance-efficient.

B. Percentage of Schedulable Task Sets

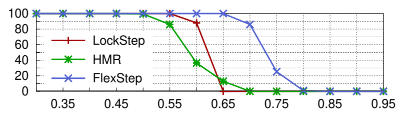
Experiment setup. To assess FlexStep's schedulability improvements, we conducted numerical simulations using randomly generated task sets across various configurations. Task sets were generated following the UUnifast algorithm [39]. We compared three error detection schemes: LockStep, HMR, and FlexStep, under partitioned EDF. For LockStep, tasks with different reliability levels were partitioned into separate queues and ordered by descending utilisation. Verification tasks were allocated first, minimising the number of checker cores by only allocating a new group of main and checker cores when the current group was full. Non-verification tasks were allocated last across all cores. For HMR, verification tasks were also allocated first. Then non-verification tasks were assigned, first filling cores without verification tasks and then cores with lowest utilisation. The partitioning of FlexStep is described in Sec. V.

Results. Fig. 6 shows that FlexStep consistently outperforms LockStep and HMR, especially at higher utilisation levels. As utilisation increases, FlexStep and HMR exhibit a more gradual performance decline, whereas LockStep experiences a sharp drop. This difference arises from LockStep's rigid verification approach. HMR encounters few deadline misses even under moderate utilisation, mainly due to the fact that non-verification tasks cannot preempt verification tasks allocated on the same core. FlexStep's flexibility allows for more efficient resource allocation, mitigating these issues.

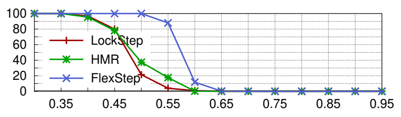
Comparing Figs. 6(a), 6(b), and 6(c), it is evident that FlexStep achieves greater performance improvements when there are fewer verification tasks, as more non-verification tasks are required to share cores with verification tasks. As FlexStep allows flexible switching between verification and non-verification tasks, without requiring binding between main and checker cores, it enhances resource utilisation and improves system schedulability. The improvement persists with an increased number of cores or higher utilisations of individual tasks, as shown in Figs. 6(e) and 6(f). Comparing Figs. 6(b) and 6(d), the addition of triple-check tasks significantly increases resource demand, leading to varying degrees of performance degradation across all three methods. However, LockStep and HMR exhibit a sharper decline in performance compared to FlexStep, as their core binding between main and checker cores limits flexibility in scheduling. In contrast, FlexStep enables more flexible verification and scheduling, optimising resource utilisation to meet deadlines, even in more demanding scenarios.



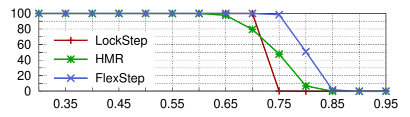
(a) $m = 8, n = 160, \alpha = 6.25\%, \beta = 6.25\%$.



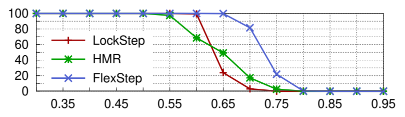
(b) $m = 8, n = 160, \alpha = 12.5\%, \beta = 12.5\%$.



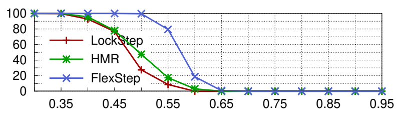
(c) $m = 8, n = 160, \alpha = 25\%, \beta = 25\%$.



(d) $m = 8, n = 160, \alpha = 25\%, \beta = 0\%$.



(e) $m = 16, n = 160, \alpha = 12.5\%, \beta = 12.5\%$.



(f) $m = 8, n = 80, \alpha = 25\%, \beta = 25\%$.

Fig. 6: Percentage of schedulable task sets (y-axis) under LockStep, HMR, and FlexStep with increasing task set utilisations (x-axis) and varying system configurations: m (number of cores), n (number of tasks), α (percentage of double-check tasks), and β (percentage of triple-check tasks).

C. Error Detection Latency

Experiment setup. To evaluate detection latency, we injected errors in the forwarded data from the main core, e.g., memory access data of MAL and architectural register data of ASS, simulating the hardware faults without disrupting the main core's normal execution. For each workload, 5,000 - 10,000 faults were randomly generated, resulting in over 100,000 sample points in total to ensure the validity of results.

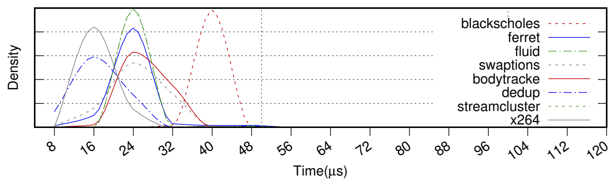
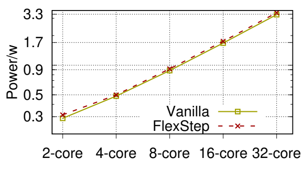


Fig. 7: Probability distribution of error detection latency of Parsec.

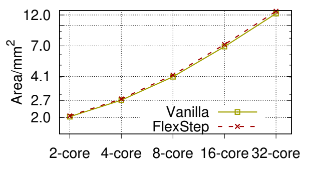
Results. Fig. 7 shows the density distribution of detection latency across different workloads. For most workloads, the majority of detection latencies are concentrated around 20 μs, while the maximum latency seen in blackscholes is 2 to 3 times higher, reaching up to 50 μs. Overall, FlexStep maintains an actual error detection latency of no more than 50μs in most cases, which is sufficient to cover over 99.9% of hardware faults and guarantee the system reliability.

D. Scalability

Experiment setup. To evaluate scalability, we increased the number of cores for both FlexStep and Vanilla (based on the original unmodified microarchitecture). We assessed their area and average power consumption using Design Compiler (v2019.12) for RTL synthesis and PrimeTime PX (v2019.12) for post-simulation analysis.



(a) Average Power.



(b) Area.

Fig. 8: The average power and area overheads on the SoC with different core numbers for Vanilla and FlexStep, including L1$ and L2$.

Results. Fig. 8 presents comparison of average power (Fig. 8(a)) and area (Fig. 8(b)) for Vanilla and FlexStep across varying core counts, indicating that the increase in average power and area for FlexStep relative to Vanilla remains nearly linear, rather than exponential, as the SoC scales from dual-core to 32-core. This demonstrates that FlexStep offers strong scalability in multi/many-core systems.

E. Hardware Overheads

Experiment setup. We used the same setup as in scalability to evaluate the hardware overheads of FlexStep compared to Vanilla.

TABLE III: Average power & area of Vanilla and FlexStep (4 cores).

	Vanilla	**FlexStep**	**Overhead**	
Core	Rocket	Rocket		×
Tech. (nm)	28	28		×
Power (w)	0.485	0.499		2.89%
Area (mm^2)	2.71	2.77		2.21%

Results. With the reuse of homogeneous checker cores, FlexStep introduces minimal hardware overhead. Compared to Vanilla, FlexStep incurs the storage overhead per core of only 1614 bytes: 8 bytes for CPC, 518 bytes for ASS, and 1088 bytes for DBC. For a 4-core SoC shown in Tab. III, FlexStep occupies 2.77 mm^2 of area and consumes 0.499w of power, representing just 2.21% and 2.89% overhead, respectively, compared to Vanilla. This validates that the hardware overhead for implementing FlexStep is well within reasonable limits.

VII. Conclusion

In this work, we developed FlexStep, a homogeneous-core error-detection framework. Using a hardware-software co-design approach, we implemented FlexStep by adding lightweight modifications to existing cores and integrating simple OS-level scheduling codes. Evaluated on an FPGA prototype, FlexStep demonstrates microsecond-level error detection capabilities with low performance and hardware overheads, high scalability, and dynamic verification mode switching, making it an efficient solution for practical applications. In future works, FlexStep enables flexible task scheduling and unlocks significant potential for developing new scheduling algorithms to enhance system efficiency, improve real-time responsiveness, and adapt to dynamic reliability requirements in safety-critical applications.

VIII. Acknowledgment

We would like to thank the anonymous reviewers for the helpful feedback. This work is supported by National Key Research and Development Program (Grant No.2024YFB4405600), U.S. National Science Foundation (Grant No.CNS-2340171), National Natural Science Foundation of China (Grant Nos. 62472086, 92264203 and 62204036), the Basic Research Program of Jiangsu (Grants No. BK20243042), Key Research and Development Program of Jiangsu Province(Grant No.BE2023020-1), and the Start-up Research Fund of Southeast University (Grant No. RF1028624005).

References

[1] V. Kumar, L. Singh, and A. K. Tripathi, “Reliability analysis of safety-critical and control systems: a state-of-the-art review,” *IET Software*, vol. 12, no. 1, pp. 1–18, 2018.

[2] A. Maurya and D. Kumar, “Reliability of safety-critical systems: A state-of-the-art review,” *Quality and Reliability Engineering International*, vol. 36, no. 7, pp. 2547–2568, 2020.

[3] P. Singh and L. K. Singh, “Reliability and safety engineering for safety critical systems: An interview study with industry practitioners,” *IEEE Transactions on Reliability*, vol. 70, no. 2, pp. 643–653, 2021.

[4] V. Kumar, K. C. Mishra, P. Singh, A. N. Hati, M. R. Mamdikar, L. K. Singh, and R. R. Parida, “Reliability analysis and safety model checking of safety-critical and control systems: A case study of npp control system,” *Annals of Nuclear Energy*, vol. 166, p. 108812, 2022.

[5] W. Ali and H. Yun, “Rt-gang: Real-time gang scheduling framework for safety-critical systems,” in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2019, pp. 143–155.

[6] V. P. Kumar and A. S. Pillai, “Dynamic scheduling algorithm for a utomotive safety critical systems,” in *2020 Fourth International Conference on Computing Methodologies and Communication (ICCMC)*. IEEE, 2020, pp. 815–820.

[7] D. Calvaresi, M. Marinoni, L. Lustrissimini, K. Appoggetti, P. Sernani, A. F. Dragoni, M. Schumacher, and G. Buttazzo, “Local scheduling in multi-agent systems: getting ready for safety-critical scenarios,” in *Multi-Agent Systems and Agreement Technologies: 15th European Conference, EUMAS 2017, and 5th International Conference, AT 2017, Evry, France, December 14-15, 2017, Revised Selected Papers 15*. Springer, 2018, pp. 96–111.

[8] A. Hanafi, M. Karim, and A. E. Hammami, “Dual-lockstep microblaze-based embedded system for error detection and recovery with reconfiguration technique,” in *2015 Third World Conference on Complex Systems (WCCS)*, 2015, pp. 1–6.

[9] 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)*, 2016, pp. 246–249.

[10] 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.

[11] X. Iturbe, B. Venu, J. Jagst, E. Ozer, P. Harrod, C. Turner, and J. Penton, “Addressing functional safety challenges in autonomous vehicles with the arm tcl s architecture,” *IEEE Design & Test*, vol. 35, no. 3, pp. 7–14, 2018.

[12] S. Hukerikar, K. Teranishi, P. C. Diniz, and R. F. Lucas, “Redthreads: An interface for application-level fault detection/correction through adaptive redundant multithreading,” *International Journal of Parallel Programming*, vol. 46, pp. 225–251, 2018.

[13] 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.

[14] D. S. Khudia and S. Mahlke, “Harnessing soft computations for low-budget fault tolerance,” in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014, pp. 319–330.

[15] A. Thomas and K. Pattabiraman, “Error detector placement for soft computation,” in *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2013, pp. 1–12.

[16] G. A. Ismael, A. A. Salih, A. AL-Zebari, N. Omar, K. J. Merceedi, A. J. Ahmed, N. O. Salim, S. S. Hasan, S. F. Kak, I. M. Ibrahim *et al.*, “Scheduling algorithms implementation for real time operating systems: A review,” *Asian Journal of Research in Computer Science*, vol. 11, no. 4, pp. 35–51, 2021.

[17] P. Agrawal, A. K. Gupta, and P. Mathur, “Cpu scheduling in operating system: a review,” in *Proceedings of the Second International Conference on Information Management and Machine Intelligence: ICIMMI 2020*. Springer, 2021, pp. 279–289.

[18] M. Hamayun and H. Khurshid, “An optimized shortest job first scheduling algorithm for cpu scheduling,” *J. Appl. Environ. Biol. Sci*, vol. 5, no. 12, pp. 42–46, 2015.

[19] F. Kluge, M. Gerdes, and T. Ungerer, “An operating system for safety-critical applications on manycore processors,” in *2014 IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, 2014, pp. 238–245.

[20] C. El Salloum, M. Elshuber, O. Höftberger, H. Isakovic, and A. Wasicek, “The across mpsoc–a new generation of multi-core processors designed for safety–critical embedded systems,” *Microprocessors and Microsystems*, vol. 37, no. 8, pp. 1020–1032, 2013.

[21] J. P. Cerrolaza, R. Obermaisser, J. Abella, F. J. Cazorla, K. Grüttner, I. Agirre, H. Ahmadian, and I. Allende, “Multi-core devices for safety-critical systems: A survey,” *ACM Computing Surveys (CSUR)*, vol. 53, no. 4, pp. 1–38, 2020.

[22] J. Perez-Cerrolaza, J. Abella, L. Kosmidis, A. J. Calderon, F. Cazorla, and J. L. Flores, “Gpu devices for safety-critical systems: A survey,” *ACM Computing Surveys*, vol. 55, no. 7, pp. 1–37, 2022.

[23] G. Durrieu, M. Faugère, S. Girbal, D. G. Pérez, C. Pagetti, and W. Puffitsch, “Predictable flight management system implementation on a multicore processor,” in *Embedded Real Time Software (ERTS'14)*, 2014.

[24] M. Rogenmoser, Y. Tortorella, D. Rossi, F. Conti, and L. Benini, “Hybrid modular redundancy: Exploring modular redundancy approaches in risc-v multi-core computing clusters for reliable processing in space,” *ACM Transactions on Cyber-Physical Systems*, 2023.

[25] F. Kempf, C. Kühbacher, C. Mellwig, S. Altmeyer, T. Ungerer, and J. Becker, “A holistic hardware-software approach for fault-aware embedded systems,” in *2022 25th Euromicro Conference on Digital System Design (DSD)*. IEEE, 2022, pp. 704–711.

[26] F. Kempf, T. Hartmann, S. Baehr, and J. Becker, “An adaptive lockstep architecture for mixed-criticality systems,” in *2021 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2021, pp. 7–12.

[27] F. Bas, S. Alcaide, R. Lorenzo, G. Cabo, G. Gil, O. Sala, F. Mazzocchetti, D. Trilla, and J. Abella, “Safede: a flexible diversity enforcement hardware module for light-lockstepping,” in *2021 IEEE 27th International Symposium on On-Line Testing and Robust System Design (IOLTS)*. IEEE, 2021, pp. 1–7.

[28] M. Barbirotta, F. Menichelli, A. Cheikh, A. Mastrandrea, M. Angioli, and M. Olivieri, “Dynamic triple modular redundancy in interleaved hardware threads: An alternative solution to lockstep multi-cores for fault-tolerant systems,” *IEEE Access*, 2024.

[29] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz *et al.*, “The rocket chip generator,” *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, vol. 4, pp. 6–2, 2016.

[30] 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.

[31] 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.

[32] S. Ainsworth and T. M. Jones, “Paramedic: Heterogeneous parallel error correction,” in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2019, pp. 201–213.

[33] A. Amid, D. Biancolin, A. Gonzalez *et al.*, “Chipyard: Integrated design, simulation, and implementation framework for custom socs,” *IEEE Micro*, vol. 40, pp. 10–21, 2020.

[34] “28nm PDKs,” https://www.tsmc.com/english/dedicatedFoundry/technology/logic/l_28nm.

[35] 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.

[36] J. L. Henning, “Spec cpu2006 benchmark descriptions,” *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[37] 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.

[38] 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.

[39] E. Bini and G. C. Buttazzo, “Measuring the performance of schedulability tests,” *Real-Time Systems (Real-Time Syst.)*, vol. 30, no. 1-2, pp. 129–154, 2005.