Low-overhead Online Assessment of Timely Progress as a System Commodity

Weifan Chen
Boston University, U.S.A

Ivan Izhibirdeev
Boston University, U.S.A

Denis Hoornaert
Technical University of Munich, Germany

Shahin Roozkhosh
Boston University, U.S.A

Patrick Carpanedo
Boston University, U.S.A

Sanskriti Sharma
Boston University, U.S.A

Renato Mancuso
Boston University, U.S.A

Abstract

The correctness of safety-critical systems depends on both their logical and temporal behavior. Control-flow integrity (CFI) is a well-established and understood technique to safeguard the logical flow of safety-critical applications. But unfortunately, no established methodologies exist for the complementary problem of detecting violations of control flow timeliness. Worse yet, the latter dimension, which we term *Timely Progress Integrity* (TPI), is increasingly more jeopardized as the complexity of our embedded systems continues to soar. As key resources of the memory hierarchy become shared by several CPUs and accelerators, they become hard-to-analyze performance bottlenecks. And the precise interplay between software and hardware components becomes hard to predict and reason about. *How to restore control over timely progress integrity?* We postulate that the first stepping stone toward TPI is to develop methodologies for Timely Progress Assessment (TPA). TPA refers to the ability of a system to live-monitor the positive/negative slack—with respect to a known reference—at key milestones throughout an application's lifespan. In this paper, we propose one such methodology that goes under the name of *Milestone-Based Timely Progress Assessment* or MB-TPA, for short. Among the key design principles of MB-TPA is the ability to operate on black-box binary executables with near-zero time overhead and implementable on commercial platforms. To prove its feasibility and effectiveness, we propose and evaluate a full-stack implementation called *Timely Progress Assessment with 0 Overhead* (TPAw0v). We demonstrate its capability in providing live TPA for complex vision applications while introducing less than 0.6% time overhead for applications under test. Finally, we demonstrate one use case where TPA information is used to restore TPI in the presence of temporal interference over shared memory resources.

2012 ACM Subject Classification Computer systems organization → Real-time systems

Keywords and phrases progress-aware regulation, hardware assisted runtime monitoring, timing annotation, control flow graph

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2023.11

Funding *Denis Hoornaert*: Denis Hoornaert was supported by the Chair for Cyber-Physical Systems in Production Engineering at TUM and the Alexander von Humboldt Foundation.
Renato Mancuso: The material presented in this paper is based upon work supported by the National Science Foundation (NSF) under grants number CCF-2008799 and CNS-2238476.

1 Introduction

Prompted by the proliferation of cyber-physical, safety-critical, and human-in-the-loop systems, the notion of *timeliness* in computing has gained growing interest. The accompanying demand for complex, robust, and computationally demanding control algorithms has led the real-time community to shift its focus away from simpler hardware platforms to high-complexity and high-performance platforms. As the complexity increases in platforms, many challenges have surfaced at all the software/hardware stack layers. It is well understood that the logic of an application can be hardened against control-flow attacks via Control Flow Integrity (CFI) [39] methods. But no established methodologies exist for the dual problem in the temporal domain, for which we coin the name *Timely Progress Integrity* (TPI).

The introduction of heterogeneous multi-core System-on-Chip (SoC) along with complex memory subsystem mechanisms at the hardware level has complicated the problem of ensuring TPI. In particular, memory subsystem hierarchy such as shared [49], non-blocking caches [62], shared memory controller [67], and DRAM organization [66] are among noteworthy sources of interference. The interplay of each element mentioned above renders the task of guaranteeing timeliness an open challenge. In turn, the introduced complexity in SoCs and their ongoing proliferation have prompted the need for more complex operating systems and OS-level scheduling strategies, which exacerbate the problem.

The real-time community has achieved important milestones towards restoring predictability [45, 48]. But traditional methods—e.g. static WCET analysis, memory resource partitioning—have largely focused on respecting end-to-end constraints in the worst case, as opposed to reason on the current (timely) rate of progress of live applications. Solutions that leverage code instrumentation have been proposed to checkpoint the progress of applications at runtime [37, 38, 58], but a system-level solution that can operate on black-box binaries and *inform* a rich OS of the expected/detected progress of its applications for it to make informed management decisions has not been studied. We propose one such solution.

Timely progress assessment as a system commodity. Reasoning about, controlling, and reacting to changes in the progress of safety-critical applications is the goal. Thus, the ability to assess an application's progress must become a system commodity. In referring to this capability, we coin the term *Timely Progress Assessment* (TPA). With TPA, a system is capable of detecting deviations in the timely progress of an application well before a deadline is missed, providing the ability to enact corrective measures toward ensuring TPI early on. On the other hand, when faster-than-expected progress is detected, the accumulated slack can be redistributed to other workloads. Thus, TPA is an enabling capability towards Timely Progress Integrity (TPI).

This article presents a system design and methodology called Milestone-Based Timely Progress Assessment (MB-TPA) to perform TPA on live black-box applications. MB-TPA relies on binary analysis and widely available on-chip tracing subsystems to detect the timely completion of intermediate progress *milestones* for an application under analysis. We discuss a full-stack implementation of MB-TPA on commercial hardware. The implemented TPA subsystem was termed Timely Progress Assessment with 0 Overhead (TPAw0v), which we describe and evaluate. We show that MB-TPA (1) introduces negligible ($< 0.6\%$) overhead to the monitored applications under test. MB-TPA is able to provide live progress assessment even if a low-power CPU is used to monitor a high-performance CPU. In light of the discussion above, we make the following contributions:

1. We propose the concept of TPI as a requirement that is complementary to CFI to marry logical and temporal integrity.
2. We demonstrate for the first time that online progress assessment without source code instrumentation for black-box applications is feasible in commercial platforms.
3. We present a method called MB-TPA, that solves key challenges with offline milestone identification and online progress assessment.
4. Provide a full-stack proof-of-concept implementation and evaluation of MB-TPA for

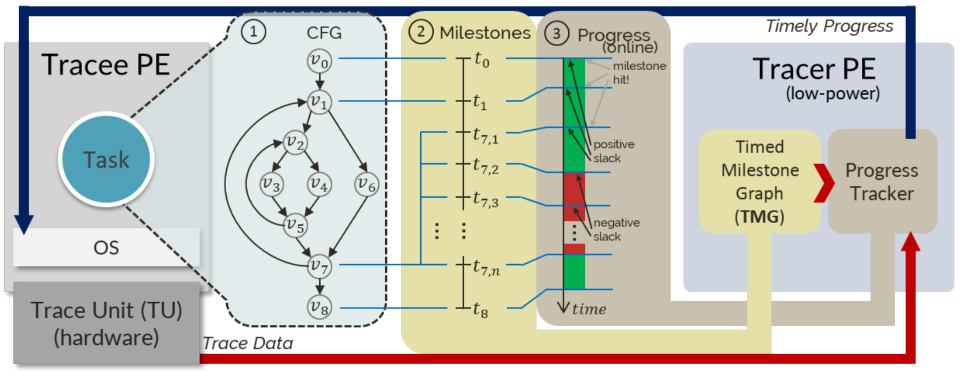


Figure 1 High-level overview of the proposed system design. The CFG of the target application is analyzed to produce a Timed Milestone Graph (TMG). Together with the online data produced by the Trace Unit (TU), a progress tracker assesses timely progress and reports to the OS. The OS can take corrective measures accordingly.

multi-core ARM AARCH64 SoCs. We refer to our implementation as TPAw0v.

5. Showcase three use cases focusing on real-world vision applications. We leverage TPA to (1) enforce the WCET of a target application; (2) achieve controlled performance degradation of the target application by modulating the degree of contention over shared memory resources; and (3) retrieve live progress-aware profiles of the microarchitectural resources used by the target application.

1.1 Overview of Proposed System Design for MB-TPA

The goal of making TPA a system commodity imposes two main design constraints. First and foremost, it must be possible for a system to enact TPA on potentially unknown (black-box) applications that cannot be recompiled from sources. At the same time, TPA shall be carried out with negligible temporal overhead. An overview of the proposed system design is provided in Figure 1. The design involves the use of a *Tracee* PE (Processing Element) where a target application (Task) runs unmodified. A second low-power/low-performance PE, the *Tracer*, controls the TU to generate trace data transparently to the application under analysis. Section 4 discusses the system assumptions that enable instantiating the proposed system.

Initially, the unmodified binary of the target application is analyzed to construct its Control Flow Graph (CFG)—(1) in Figure 1. Through a sequence of refinement steps, a Timed Milestone Graph (TMG) is derived from the original CFG. An in-depth description of the methodology proposed to produce a TMG from a CFG is provided in Section 5. The TMG is a graph of milestones, each corresponding to some vertex in the original CFG, with associated time information—(2) in Figure 1. At runtime, the tracer uses the input TMG and the data received from the TU and detects (un)timely completion of the milestones—(3) in Figure 1. The detected positive/negative progress slack is reported back to the OS to enact management decisions. The tracer was implemented as bare-metal firmware running on a low-power CPU. The details of our implementation are provided in Section 7.

2 Related Works

Our work finds context in the broad literature concerned with ensuring that the timeliness of a (set of) critical task(s) can be controlled. In modern platforms, the progress of application workload can be impacted by many factors. These include scheduling decisions, overheads introduced by preemptions and migrations [15, 40, 50] and I/O activity [16, 33, 55, 68], unpredictable cache effects such as self-eviction [17, 27], and contention over shared hardware resources [45, 48]. The set of solutions proposed by the real-time community to reason about the timeliness of an application can be placed on a spectrum. On one end are static analysis approaches; on the other are runtime monitoring solutions.

Timeliness (interpreted as the ability to meet a completion deadline) in static analysis approaches [5, 20, 31, 47] is ensured by computing an absolute worst-case execution time (WCET) which is then used to compute a worst-case response time (WCRT). The promise is that WCET/WCRT computation is done by considering the initial state(s) and sequences of system states that lead to the worst possible temporal application behavior. Given the sheer complexity of interactions between applications, system-level, and hardware-level components, static approaches seldomly scale to modern multicore processors [30, 35, 46].

Recently, approaches based on runtime monitoring have gained momentum. At a high level, these approaches select a *monitoring scheme* and a set of *system metrics*. By monitoring such metrics online—and taking management actions accordingly—the system detects and/or avoids undesired outcomes, e.g., uncontrolled contention over a shared resource or a deadline miss for a critical task. To properly contextualize our work with respect to related approaches, we categorize runtime monitoring solutions into *software-* and *hardware-based* approaches.

2.1 Software-based Monitoring and Progress Assessment

The vast majority of solutions for runtime monitoring and progress assessment introduce software mechanisms to enact monitoring and/or enact management decisions. We distinguish four main sub-categories discussed below.

(A) Memory Bandwidth Regulation: Memory bandwidth controllers [59, 62, 67] monitor the number of last-level data cache refills and/or writebacks against an allocation budget. Periodically, they stall the processor if the consumed budget is exceeded. Although bandwidth regulation aims to prevent the unbalanced progress of co-running applications sharing the same memory subsystem, no exact knowledge of application progress is constructed.

(B) Feedback Control Scheduling: Feedback control scheduling represents another form of runtime monitoring. In the context of real-time systems, this approach was pioneered in [60]. The key insight is that the knowledge of task parameters computed offline is refined via online observations performed at task completion. Task admission is geared accordingly to meet a target deadline miss ratio. Since the aforementioned original work, a broad literature on feedback control scheduling has surfaced [19, 44, 53].

(C) Early Deadline Detection: Early deadline detection is the runtime monitoring technique at the center of adaptive mixed-criticality scheduling (AMC) [14, 18]. The key insight is that multiple (at least two) runtime estimates are expressed for high-criticality tasks with varying degrees of pessimism. Initially, an optimistic execution time is assumed, and an early deadline (virtual deadline) is set accordingly. At runtime, the system detects if any early deadline is missed and takes corrective measures accordingly by dropping [13, 24, 29, 41, 54] or degrading low-criticality tasks [28, 42]. Like feedback control scheduling, runtime monitoring in AMC systems is limited to detecting an application's completion (or lack thereof) by a set (early) deadline. This is equivalent to detecting a single milestone at the application's end.

(D) Progress Detection: A handful of works attempt to provide a finer-grained understanding of the progress of target applications. For instance, the work in [26] periodically monitors the number of retired instructions to detect a sequence of phases in which the application's usage of hardware resources changes. This approach is inherently limited to applications with a single execution path. In a way that is more closely related to our work, the works in [36–38, 58] consider the full CFG of a target application. These works propose to instrument a target application's code via source-to-source translation and/or a modified compiler. The goal is to insert watchpoints at which progress is assessed in software. At runtime, when the execution reaches a watchpoint, an interrupt/syscall is issued to decide whether the system should raise the critical level and drop/suspend low-criticality jobs. In previous works, the overhead is a limiting factor. Kritikakou et al., in an extension [36] to [37, 38], propose an algorithm to ignore some checkpoints in order to reduce the overhead. The authors of PAStime [58] place watchpoints outside of loops to limit the overhead.

Compared to the works in the four categories surveyed above, this paper sets itself apart because we aim at precise progress assessment without the need to modify/recompile the application under analysis. Importantly, we are able to express a notion of timely progress even if the control flow is input dependent. Finally, for the first time, we demonstrate that leveraging widely available tracing hardware for progress assessment is possible and minimizes runtime overhead. Indeed, our system never interrupts the application under analysis while its progress is assessed asynchronously and, therefore, off the critical path.

2.2 Run-time Monitoring via Hardware

Comparatively, less work has explored progress monitoring via specialized hardware support. Most notably, Lo et al. proposed a customized hardware architecture for runtime monitoring of hard real-time tasks [43]. Apart from timely progress, the work aims to monitor other safety properties, such as the presence of uninitialized memory and the correctness of return addresses. Differently from [43], we focus on commercially available hardware.

Few works have also proposed to leverage trace unit at runtime to perform control flow integrity [25, 34], while FPGA-based trace decoders were proposed in [6, 32]. We are the first to utilize a trace unit online to perform timely progress assessment in real-time systems.

3 Background

All the aforementioned approaches for progress assessment [36–38, 43, 58] consider the CFG of critical tasks. Kritikakou et al. have constructed a formal grammar to extract the CFG from a wide range of binaries [37]. There are also a plethora of tools capable of such transformations [57]. The following section provides a brief overview of CFGs.

(A) Basic Block and Branch Instructions: A *basic-block* (`BB`) is a contiguous sequence of non-branching (assembly) instructions ending with a branching instruction. In other words, except for the last instruction, a basic block only contains instructions for which the *program counter* (`PC`) of the CPU—or more generally, processing element (PE)—is monotonously incremented. A branch instruction has one or more target `BB`s. For example, in ARM® AARCH32/64 [11], an unconditional branch instruction `b` would take `PC` to the operand address, the beginning of a `BB`. Conditional branch instructions `b.cond` have two target `BB`s. When `b.cond` is executed, if the condition is met, the `PC` is set to the operand address, otherwise to the instruction following the `b.cond` instruction. The return instruction `ret` can have more than two target `BB`s. It is possible to statically know its target(s) if the call sites can be fully enumerated.

(B) Control Flow Graph: A program's control flow transfer information can be expressed as a directed graph $\mathcal{G} = (V, E)$. A node $n \in \mathcal{V}$ represents a `BB`, and an edge $(n_p, n_s) \in \mathcal{E}$ indicates that the branch instruction in n_p has n_s as a target. We term this type of edge a *normal edge*. In practice, it is unnecessary to expand the complete CFG for runtime monitoring purposes. Instead, one can view the program as a collection of functions with the entry point at `main` [37]. Thus, if no watchpoints are to be placed inside a function `f`, all nodes and edges related to `f` can be removed, and an edge from the caller `BB` to the returning `BB` is added. We refer to this operation as the *folding* of function `f`, and to the newly added edge as the *folding edge*.

(C) Processor Trace: The *processor trace*, often called the *embedded trace*, is a highly compressed data stream generated by a PE when executing binary code. The trace contains the necessary information to reconstruct the history of the executed program. Trace generation is often used for debugging and performance evaluation purposes. As such, the on-chip hardware circuitry dedicated to processor trace generation, i.e., the *trace unit* (TU), is designed to introduce negligible overhead, if at all. The typical use of processor tracing capabilities is in conjunction with external trace probes. In this case, the system runs without modification while external hardware (probe) is connected to a physical trace port. The probe collects (portions of) the produced processor trace data for offline analysis. Two broadly used hardware probes are the Lauterbach® PowerTrace [1] and the Green Hills® Probe V4 [2].

Trace generation units are almost ubiquitous in embedded and general-purpose high-performance CPUs. Many embedded modern processors include more or less capable on-chip TU's. For example, ARM's lineup of hardware modules for tracing and debugging that fall under the CoreSight [7] umbrella includes TU modules such as the Embedded Trace Macrocell (ETM) and Program Trace Macrocell (PTM). The TU solution from Intel® is called Processor Trace (PT). The PT infrastructure has been introduced in 5th generation Intel processors, promising overheads below 5% [21, Chapter 32]. RISC-V also has its own embedded trace specification [4].

Since trace data is produced at the same (or comparable) timescale as instruction execution, the data bandwidth is usually considerably high, even after many lossless compression techniques are applied. A common compression technique only reports the progression of `BB`s instead of individual instructions. If the current `BB` is known, then a single bit of information is enough to encode whether the (conditional) branch at the end of the `BB` is taken or not. When this information is combined with static knowledge of the binary under analysis, the entire control flow can be recovered. If the current `BB` ends with an indirect branch such as a function return, the trace provides an explicit branching address.

Trace data include additional metadata about the processor state. For instance, in systems that support multiple tasks, the context ID of the process in execution (as determined by the OS) is also generated. The virtual machine ID is also included for systems with hardware virtualization extensions. Similarly, information that can identify an interrupt context (interrupt taken, interrupt type, interrupt return) is also provided. Other valuable meta-information for performance analysis can also be included, such as the cycle counter and the occurrence of other microarchitectural events.

A TU includes hardware resources that go beyond embedded trace generation to perform some degree of pre-processing. For instance, trace packet filters, counters, sequencers/formatters, external input selectors, or aggregators to combine trace data from multiple sources (e.g., multiple CPUs) can be included in the TU subsystem.

4 System Model and Assumptions

In this section, we describe the assumed system model upon which our MB-TPA is formulated. These assumptions also dictate the system requirements to implement the proposed MB-TPA, and ultimately introduce timely progress assessment as a commodity.

4.1 System-level Assumptions

(A) Tracee PE and Tracer PE: We assume that at least two PEs are present: (1) a main PE (or *tracee*) running the application under analysis and (2) the other PE serving as a *tracer*. Note that no assumption on the components' nature nor performance is made, meaning that the tracer and tracee can be implemented using various technologies. For instance, a system could have high-performance PEs as tracee and be monitored by a low-performance real-time core or specialized hardware implemented as an ASIC or on an FPGA.
(B) Address Range Filters: We assume that the tracee features a TU providing at least one range-programmable instruction address filter. That way, the TU can be programmed to trace specific address ranges corresponding to the immediate next milestones.
(C) On-chip Trace Data Path: We assume that an on-chip data path exists through which the TU-generated trace data stream can be forwarded to the tracer, as it is commonly the case for high-performance embedded systems. For instance, many ARM-based COTS platforms offer dedicated on-chip trace routing and storage within the CoreSight [7] infrastructure[1].

4.2 Application-level Assumption

(A) Single Binary: This work targets single-binary applications running on the tracee. No restrictions on the number of software layers used by the tracee are imposed, meaning that the target applications can equally run on top of a full-fledged OS, inside a virtual machine on a hypervisor, or as a bare-metal application. The binary is sufficient to apply the proposed MB-TPA: we place no assumption on the availability of the target's source code, nor that it can be recompiled and/or binary-instrumented. The goal is that MB-TPA can be automatically employed by a system.
(B) Single Entry/Exit: Without loss of generality, we assume that the entry `BB` address and the exit `BB` address are (1) known, (2) within the target's binary, and (3) they are linked by at least one valid control path. The entry and exit `BB` of a function generally[2] represent a valid selection. Otherwise, for applications implementing time- or event-triggered logic in an infinite loop, the first and last `BB`s of the loop iteration can be selected as the entry and exit `BB` points. If the debug symbols are part of the binary, the entry/exit `BB` selection can be automated (e.g., given a function name).
(C) Availability of Representative Inputs: Finally, for complex and input-dependent applications, we assume that a set of representative input vectors is available to experimentally produce (offline) a nominal progress reference to check against during the online phase.

5 Methodology for Milestone-Based Timely Progress Assessment

We hereby describe the proposed Milestone-Based Timely Progress Assessment in its different phases. With reference to Figure 1, this section details the design choices and steps involved in going from CFG creation to TMG generation. A bird's eye view of MB-TPA is depicted in Figure 2. The following sections cover the numbered steps (1) through (5) in detail.

[1] Trace data routing components include the Embedded Trace Router (ETR), Embedded Trace FIFO, and Funnel. Storage components include the Embedded Trace Buffer and Trace Memory Controller.
[2] If no infinite loops are present in the function nor in any other routine that can be called by it.

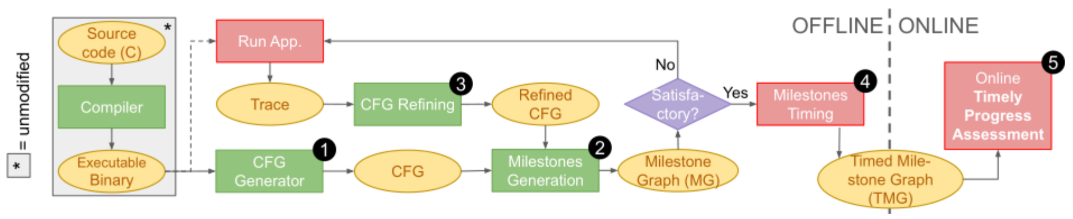


Figure 2 Abstract tool-chain proposed. Ovals represent the inputs and outputs, red rectangles represent timing-sensitive tools, and green rectangles represent timing-insensitive tools.

5.1 Intuition of Key Challenges and Solutions

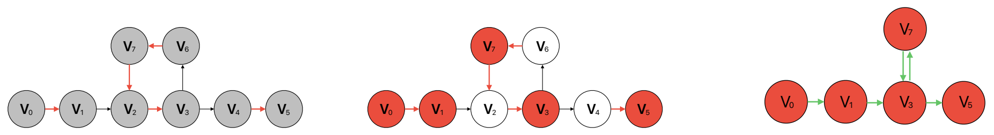
(A) Monotonic Progress in Black-Box Binaries: As discussed in Section 3, the execution of a binary implies control flow transfer over a graph. On the other hand, the idea that a target application must execute (and thus complete) on time implies a monotonic notion of progress. Therefore, the first challenge we face is to construct a notion of progress given black-box application binaries.

Our solution consists in identifying BBs that represent *progress milestones* (Section 5.3). Intuitively, a BB is a progress milestone (a.k.a. MBB) if, once reached, it is possible to conclude that a sizable amount of progress has been made by the application logic. Milestone identification is done through a combination of (1) *CFG extraction*, (2) *CFG refinement* by observing concrete runs of the target, and (3) applying the milestone placement algorithm. The output of the algorithm is a milestone graph (MG). The procedure is detailed in Section 5.3.

(B) Keeping up with Trace Data: Timely progress assessment has to be performed in a timely manner. Assuming that a valid set of MBBs has been identified, the goal is to detect the completion of milestones at the tracer as soon as they are reached on the tracee, or with negligible delay. This way, the tracer can promptly assess TPI violations and trigger any correction countermeasure if necessary. Conversely, if the tracer lags significantly behind the tracee, then it might be too late to act upon detected TPI violations—and one might as well detect TPI violations at target completion instead.

What makes this challenging? The first issue might reside in the **latency** for the propagation of TU-generated data to the tracer PE. As we evaluate in Section 8.1, it is not an issue if the tracer and tracee are different PEs on the same SoC. A second (and more problematic) issue is the limited **bandwidth** of the on-chip channels via which trace data is streamed. Despite aggressive trace compression, allowing the TU to stream trace data unrestrictedly leads to buffer overflows due to the performance gap between tracer and tracee PEs. These overflows can occur both within the TU or at the interface between the TU and the tracer, preventing any packet from reaching the tracer. Thus the naïve solution of constantly streaming data from the TU and matching against MBBs does not work.

(C) Dynamic TU Reconfiguration: To reliably ensure milestone detection, we propose to dynamically reconfigure the TU so that it is silent for most of the time and only emits bare minimum packets when the event of interest happens—i.e., one of the next MBBs is reached. At this point, a new set of MBBs to monitor is configured. The TU then becomes silent again, waiting for the next milestone. In this paradigm, the TU only emits sporadic and short-lived signals, thus consuming a fraction of the sustainable trace bandwidth. The information of which MBBs to monitor after a given MBB is reached is expressed in the TMG.



(a) The extracted CFG. Red edges are folding. **(b)** Nodes satisfying the constraint are colored red. **(c)** Remove white nodes, add corresponding edges.

Figure 3 Illustrative MG generation for the `main` of the `disparity` benchmark.

5.2 Trace Blackout Window

Two milestones cannot be placed arbitrarily close to one another. This is a consequence of the dynamic TU reconfiguration. Suppose $\texttt{MBB}_1$ and $\texttt{MBB}_2$ are adjacent, i.e., when the TU has detected that tracee's execution has reached $\texttt{MBB}_1$, then the TU should be reconfigured to detect tracee's execution on $\texttt{MBB}_2$. The reconfiguration typically consists of (1) disabling the TU to reprogram the relevant registers, (2) identifying the `MBB` that has been reached, (3) looking up in the TMG the next set of milestones to detect, and (4) resuming the TU.

Let t_1 and t_2 denote the time for tracee's execution reaching $\texttt{MBB}_1$ and $\texttt{MBB}_2$ respectively. From the time t_1 at which $\texttt{MBB}_1$ is reached and until the TU is brought back online to monitor $\texttt{MBB}_2$, there is a window of time during which milestones cannot be monitored. We call this the *trace blackout window* and indicate it with the symbol T_r. If the best-case path between $\texttt{MBB}_1$ and $\texttt{MBB}_2$ is such that $(t_2 - t_1) < T_r$, then detection of $\texttt{MBB}_2$ cannot be guaranteed. Our methodology avoids this issue by design.

Formally, call $D(\texttt{MBB}_i, \texttt{MBB}_j) \in \mathbb{R}^+$ the time-cost to reach $\texttt{MBB}_j$ starting from $\texttt{MBB}_i$. Clearly, this cost is a random variable that depends on the specific path taken and the progress at which the target executes. Moreover, $D(\texttt{MBB}_i, \texttt{MBB}_j) = \infty$ if $\texttt{MBB}_j$ cannot be reached from $\texttt{MBB}_i$. We show that a lower-bound of this cost can be computed and impose that, for any two valid $\texttt{MBB}_i, \texttt{MBB}_j$, it must hold that

$$\min_{i,j}\{D(\texttt{MBB}_i, \texttt{MBB}_j)\} > T_r. \tag{1}$$

It is worth noting that the blackout window and the sizable progress requirement discussed in the first challenge in Section 5.1 both require the distance between two milestones to be sufficiently large. In practice, the blackout window is generally smaller—we derive this parameter for our implementation in Section 8.1. Thus ensuring that enough progress occurs between milestones implies that the constraint imposed by the blackout window is also met.

5.3 Milestone Graph Construction (Step 1 and 2)

Figure 3 depicts the intuition behind the Milestone Graph (MG) construction procedure. First, the CFG of the target application is extracted (Figure 3a). The CFG is annotated by adding a weight on each edge that is indicative of the temporal distance between two nodes. Then a subset of nodes satisfying the constraint expressed in Eq. 1 is selected—the red nodes in Figure 3b. Finally, new edges are added to the red nodes to maintain reachability relationships, as per Figure 3c. The resultant digraph is a valid MG.

(A) CFG Notation: Given a target black-box binary, the CFG is extracted (Step 1 in Figure 2). This is a digraph $\mathcal{G}^{CFG} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ and $\mathcal{E}$ are the set of all the vertices and edges, respectively. Here a vertex $v_i \in \mathcal{V}$ is a `BB`. An edge $(v_i, v_j) \in \mathcal{E}$ is either normal or

folding (Section 3)[3]. For any edge $(v_i, v_j) \in \mathcal{E}$, we assign a per-edge weight w equal to the lower bound on the time to execute the instructions in v_i, including the folded function if its out-edge is folding. A safe albeit inaccurate lower bound can be obtained by dividing the number of instructions in v_i by the maximum clock frequency of the tracee[4]. We define $D(v_i, v_j)$ for any two vertices in $\mathcal{V}$ as the cost of the path (if any) from v_i to v_j with the minimum cost. This is used to lower-bound the minimum time needed to reach v_j from v_i.
(B) MG Notation: An MG $\mathcal{G}^{MG} = (\mathcal{M}, \mathcal{Q})$, is a digraph where $\mathcal{M} \subseteq \mathcal{V}$ is the set of `MBB`s. For each `MBB`$_i \in \mathcal{M}$, an edge $(\texttt{MBB}_i, \texttt{MBB}_j) \in \mathcal{Q}$ signifies that (1) `MBB`$_j$ is one of the next milestones to detect after `MBB`$_i$ has been reached, and (2) Eq. 1 holds. Note: the edge ($\texttt{MBB}_i$, $\texttt{MBB}_j$) might not exist in $\mathcal{E}$ because the corresponding `BB`s might not be in an immediate predecessor/successor relationship in $\mathcal{G}^{CFG}$.
(C) Milestone Selection: The milestone selection problem is the following: (1) given a blackout window T_r, color the vertices in $\mathcal{G}^{CFG}$ either red or white; (2) ensure that for any two red nodes, $r_i, r_j \in \mathcal{V}$, $D(r_i, r_j) > T_r$; and (3) find the maximal set of red nodes. Other optimization objectives and heuristics could also be used—e.g, minimizing the sum of distances among red nodes. Finding the optimal solution is not the focus of this work and left as future work; an algorithm that is guaranteed to find a solution (if one exists) is presented here.
(D) Graph Coloring Heuristic: The proposed strategy (Step 2 in Figure 2) is described in Algorithm 1. The algorithm first colors all of the vertices red (Line 6–8), then iterates over any non-visited remaining red vertex in DFS search order—thus, starting from the root `BB` (Line 9). Next, for each red vertex r_i we compute the path with the shortest total cost $D(r_i, r_j)$ to all other red vertices in $\mathcal{V}$ (Line 12). If for some r_j $D(r_i, r_j) > T_r$ does not hold (Line 14), color r_j white (Line 15). The full adjacency map D for r_i can be computed using Dijkstra's algorithm [22]. The only adaptation needed to the standard algorithm is to correctly compute $D(v_i, v_i)$, which is always 0 in the traditional algorithm. Instead, we must compute the cost to come back into v_i if v_i was reached, which can be computed as

$$D(v_i, v_i) = \begin{cases} w_i & \text{if } (v_i, v_i) \in \mathcal{E} \\ \min_{(v_i, v_j) \in \mathcal{E}} \{D(v_j, v_i) + w_i\} & \text{otherwise.} \end{cases} \qquad (2)$$

To finalize the MG $\mathcal{G}^{MG}$, we proceed as follows. $\mathcal{M}$ is created from the colored $\mathcal{G}^{CFG}$ by removing all the white vertices v_i. To compute $\mathcal{Q}$ from $\mathcal{E}$, we proceed as follows. For each white vertex v_i, remove any self-loop and say that incoming (resp., outgoing) edges are of the form (v_p, v_i) (resp., (v_i, v_s)). Then, for each direct predecessor v_p of an incoming edge, we add all the edges of the form (v_p, v_s) for any direct successor v_s of v_i in $\mathcal{Q}$.
(E) Degree Reduction: Recall that the number of address range registers available (noted M^*) at the TU is limited (Section 3). Intuitively, M^* constraint how many milestones can be monitored by the TU after (one of) the current milestone is hit. After the MG has been produced following the procedure described so far, there is no guarantee that the outdegree (number of outgoing edges) of all the $r_i \in \mathcal{M}$ is below M^*. Thus, a simple pruning strategy is adopted. That is, for each r_i with outdegree greater than M^*, randomly pick one of the outgoing edges and color the vertex pointed by that edge white; then repeat the procedure to remove white nodes. This is done until no vertex with outdegree greater than M^* is found.

[3] Folding all functions except for `main` can already produce meaningful milestone graphs for applications under test. In practice, if the execution time of a function is long, unfolding it to allow milestones to be placed inside can achieve better granularity.
[4] We assume the CPI is greater or equal to one. Notice this might not be true for multi-issue processors.

Algorithm 1 Constrained Directed Graph Coloring

```
input:
    $\mathcal{G}^{CFG} = (\mathcal{V}, \mathcal{E}), T_r$                ◁ CFG graph and blackout window
output:
    Colored $\mathcal{G}^{CFG}$                  ◁ CFG graph with red-colored marked MBB's
init:
    for each $v \in \mathcal{V}$ do
        $v$.color ← red                                      ◁ Color all nodes red
    end
    $R_{left}$ ← Topol($\mathcal{V}$)               ◁ Red vertices to visit, in DFS search order
algorithm:
    for each $r_i \in R_{left}$ do
        $D$ ← Dijkstra($r_i$, $\mathcal{G}^{CFG}$)            ◁ Get all shortest-paths from $r_i$
        for each $r_j \in \mathcal{V}$ s.t. $r_j$.color == red do
            if $D(r_i, r_j) \leq T_r$ then
                $r_j$.color ← white                        ◁ $r_j$ unsafe milestone from $r_i$
                $R_{left} \leftarrow R_{left} \setminus \{r_j\}$                  ◁ Remove $r_j$ from $R_{left}$
            end
        end
        $R_{left} \leftarrow R_{left} \setminus \{r_i\}$                          ◁ Mark $r_i$ as visited
    end
```

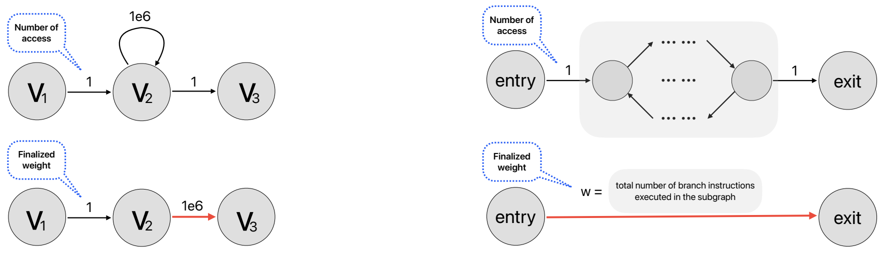
We call FINALIZEMG(Colored $\mathcal{G}^{CFG}$, M^*) the routine that takes in input a colored MG and performs edge construction plus MG pruning. Note that the selection of T_r and computation of the weights w can affect the pessimism of Algorithm 1. Moreover, in the presence of loops, the lack of static knowledge about the number of iterations that will be executed at runtime forces the algorithm to assume that only the iteration lower bound is taken. Finally, error-handling branches that are never taken during nominal execution create short-cut paths (e.g., from entry to exit in a routine) that prevent many intermediate BBs from being colored in red. Nonetheless, the important advantage of this first step is that an initial MG can be produced *without the need to execute the application.*

5.4 Milestone Graph Refinement with Concrete Runs (Step 3)

Refinement of the MG with concrete runs (Step 3 in Figure 2) mitigates the problems with static MG construction described in Section 5.3. During refinement, the target is executed on a set of representative inputs, potentially multiple times for each input. Techniques such as *symcretic execution* that combine symbolic execution and concrete runs can be used to automate the generation of representative inputs [23]. For the purpose of this work, we assume that a set of representative inputs has been identified for the target application.

By executing the target application using representative inputs, we are able to measure the temporal distance between two BBs in the CFG and gather additional information about the path(s) taken by the target for each input. Importantly, we can now compute the max/min number of times that each edge $(v_i, v_j) \in \mathcal{E}$ was taken, and thus the min/max number of iterations of each loop is discovered. We record both observed minimum $a_{i,j}$ and maximum $b_{i,j}$ number of times each edge is visited. We only preserve the number of visits, but not their order, despite the trace data does provide the full history of the visited BBs.

These runs are a way to collect extra information about the target and belong to the offline analysis phase of MB-TPA. In this phase, the TU is configured in a special mode where the TU can slow down the tracee. This is because the high-bandwidth nature of the trace data stream can overflow the internal buffer of the TU and cause information loss. Thus the slowdown ensures that a complete trace from entry to exit of the target is acquired. This is the *only* case in MB-TPA when the target is executed with a (possibly) heavy impact on its runtime due to the activity of the TU.



(a) The number of access are $a_{1,2} = 1$, $a_{2,3} = 1$, and $a_{2,2} = 10^6$. After applying the heuristic, the number of access for the self-loop becomes the weight for (v_2, v_3), i.e. $w_{2,3} = 10^6$

(b) No pair of nodes in the gray region satisfies the constraint. Thus the total number of branch instructions taken inside the region becomes the weight for $w_{\text{en,ex}}$.

Figure 4 Refinement by heuristics. The subgraphs before the heuristics applied are shown on top, in which the number on an edge indicates the number of access $a_{i,j}$. The subgraphs after the heuristics applied are below, in which the number indicates the assigned weight $w_{i,j}$.

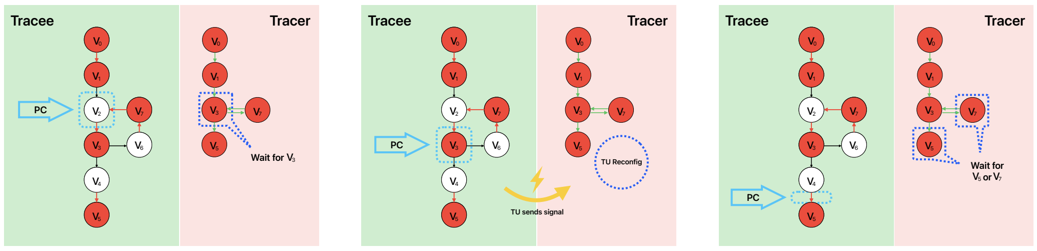
(A) Branches as a Proxy of Distance: Since the exact temporal progress has been impacted, we need a different metric that correlates (and lower-bounds) the temporal distance between MBBs. The metric must be available from the traces and preserved when the runtime of the application is impacted. Thus, we use the reported number of visited BBs—i.e., the number of executed branch instructions. The advantage is threefold: (1) can be computed directly from the acquired trace without interfacing with any other architectural unit—e.g., a performance measurement unit; (2) when execution flows within the known CFG of the target, one can always retrieve the number of instructions executed; (3) we can put a (conservative) weight on branches to the outside of the CFG under analysis, such as calls to dynamically linked libraries and system calls. From our experience, (2) is unnecessary since the newly acquired information about the min/max number of loop iterations and the presence of never-observed paths already enables much lower pessimism in the MG construction.

Under the new metric, the weight of every normal edge equals to one. The weight of a folding edge depends on the number of branch instructions executed in the folded function which can vary across different sample inputs. To ensure the blackout window condition holds (Eq.1), the weight of a folding edge is assigned to be the minimum across all inputs. Now the effective temporal distance $D(r_i, r_j)$ is the shortest path from r_i to r_j. The following two heuristics can further fold subgraphs with certain properties, so that extra milestones can be placed.

(B) Simplify Self-Loops: We identify any BB v_i having only (1) one incoming edge (v_{i-1}, v_i), (2) one outgoing edge (v_i, v_{i+1}), and (3) one self-loop (v_i, v_i). All edges are normal. If the incoming and outgoing edges are both accessed only once, then replace the temporal cost $w_{i,i+1}$ with the minimum number $a_{i,i}$ of self-edge accesses, and remove the self-loop, as shown in Figure 4a. Without this simplification, a suitable milestone candidate v_3 would not be considered due to $D(v_1, v_3) = 2$.

(C) Simplify Sub-graphs: Consider any sub-graph $\mathcal{G}_{sub}^{CFG}$ with a single entry vertex v_{en} and single-exit v_{ex}, in which all edges are normal. If it was unsafe to place any milestones within $\mathcal{G}_{sub}^{CFG}$, then (1) remove all the vertices that belong to $\mathcal{G}_{sub}^{CFG}$ except v_{en} and v_{ex}; (2) add the folding edge $(v_{\text{en}}, v_{\text{ex}})$; and (3) set the temporal cost $w_{\text{en,ex}} = W_{sub}$ to the minimum number of branches W_{sub} observed across all runs inside $\mathcal{G}_{sub}^{CFG}$, as shown in Figure 4b.

Besides the two heuristics above, the nodes/edges never accessed across all reference



(a) Initially, assume that tracee's program counter (PC) is inside v_2. The TU is programmed to monitor arrival at v_3. The TU is silent until then, and the tracer awaits a signal from the TU.

(b) As soon as the tracee starts executing instructions in v_3, the TU signals the tracer. The tracer reconfigures the TU to monitor the next milestones v_5 and v_7 during the blackout window.

(c) The TU reconfiguration is complete and the tracer is ready to wait for tracee's execution to enter either v_5 or v_7. By design, tracee's execution has not yet reached them.

Figure 5 Tracer-Tracee interaction for milestone detection and dynamic TU reconfiguration.

inputs are also removed. For this work, we only apply the above refinements, but a large space exists for more advanced heuristics.

5.5 Timed Milestone Graph Generation (Step 4)

By the end of Step 3 (Section 5.4), an MG refined using concrete runs is obtained. Recall that the goal is to monitor the target's progress online with negligible overhead. At this stage (Step 5 in Figure 2), the (refined) MG is decorated with timeliness information. The output of this step produces a Timed Milestone Graph (TMG) where each milestone is associated with a notion of *when the milestone should be completed* for satisfactory progress.

(A) Milestone Timing: To associate timing information to milestones, the TU is configured never to slow down the traced application. In this mode, allowing full trace generation might result in unpredictable trace overflows, as discussed in Section 5.1. Instead, the refined MG is used to wake up the TU and tracer only when a milestone is reached, as depicted in Figure 5. In the considered example, the tracee is initially (Figure 5a) executing code within v_2. The TU is configured to remain silent; its address range filter registers (see Section 3) are set to detect the arrival of execution into the next milestone (v_3). When v_3 is reached, the TU emits trace activity towards the tracer (Figure 5b). The TU uses the MG to dynamically reconfigure the TU to detect the next milestones, in this case, v_5 and v_7. Upon completion of the latter operation, the tracer goes back to waiting for an event from the TU (Figure 5c). Whenever a control transfer between two milestones is observed, the tracer measures the time—in terms of elapsed clock cycles—for the transfer.

(B) Milestone Timeliness Information: Using the measured milestone-to-milestone time, timeliness information is added to the MG in two parts. (1) Each node in the MG is given a *tail* time; (2) each edge in the MG is given a *nominal* time.

Tail time: The tail time $T_t(\texttt{MBB}_i)$ is the absolute time by which the target must hit $\texttt{MBB}_i$ for the last time. This value is the maximum taken across all the timed runs on the given set of representative inputs—worst-case in isolation. The tail time can be understood as the WCET till a specific milestone. However, loops and alternative paths make the tail time insufficient to assess a broader set of timeliness properties beyond WCET enforcement. Consider the case where we want to detect timely progress via loop iterations. Even if each iteration of the loop takes longer than usual, the tracer cannot detect per-iteration slowdowns by only using the tail time. The nominal time is designed to overcome such a limitation.

Nominal time: Given an edge $(\texttt{MBB}_i, \texttt{MBB}_j) \in \mathcal{Q}$, the nominal time $T_n(\texttt{MBB}_i, \texttt{MBB}_j)$ is a reference time the application is expected to spend to transfer from $\texttt{MBB}_i$ to $\texttt{MBB}_j$. Once again, the maximum is taken across all the timed runs. Even if the target runs in isolation (all other PEs idle), fluctuations in the value of T_n can occur due to microarchitectural noise. If $(\texttt{MBB}_i, \texttt{MBB}j)$ is part of a loop, nominal time is effective in detecting slower-than-expected transfer between $\texttt{MBB}_i$ and $\texttt{MBB}_j$. Thus the nominal time offers finer timeliness checking per iteration.

5.6 Online Timely Progress Assessment (Step 5)

Once a TMG has been obtained, online TPA is possible, which is the focus of Step 5 in Figure 2 and described below. The TMG is passed to the tracer when the target is launched. The $\texttt{MBB}_0$ that corresponds to the selected entry point for the target is programmed by the tracer on the TU. Live tracking of the application under analysis is performed by employing the same strategy described in Section 5.5 and illustrated in Figure 5.

At runtime, we track two times: (1) the *actual time* $\Theta(i)$ and (2) the *running nominal time* $N(i)$. Let $\texttt{MBB}_i$ be the i-th milestone for which a hit has been detected. $\Theta(i)$ is updated with the current time. Therefore, it tracks the time measured since $\texttt{MBB}_0$ was hit and until $\texttt{MBB}_i$ is reached. Conversely, $N(i)$ is updated as $N(i) = N(i-1) + T_n(\texttt{MBB}_{i-1}, \texttt{MBB}_i)$.

At this point, everything is set to assess the timely progress of the target. Whenever a milestone $\texttt{MBB}_i$ is hit, the tracer can check that $\Theta(i) \leq \min(T_t(\texttt{MBB}_i), N(i))$. If a controllable amount of degradation—compared to the reference timing acquired in isolation—is accepted, one can express the allowed slowdown as $\alpha > 1$ and check the following condition instead:

$$\Theta(i) \leq \alpha \min(T_t(\texttt{MBB}_i), N(i)). \tag{3}$$

Importantly, all the elements are in place not only for the detection of TPI violations but also to routinely report positive/negative current slack to the tracee PE. The slack at $\texttt{MBB}_i$ can be calculated as $\textit{slack}(i) = \min(\alpha T_t(\texttt{MBB}_i), \alpha N(i)) - \Theta(i)$.

6 Use Cases for MB-TPA

We hereby provide three use-cases enabled by the ability of MB-TPA to provide runtime timely progress assessment as a system commodity.

(A) Strict WCET Enforcement: Previous work has provided a methodology based on code-level instrumentation to insert progress checkpoints (milestones in our notations) with the goal of enforcing a target WCET for a high-criticality task under analysis [36–38,58]. The capabilities of MB-TPA seamlessly support one such use case. Consider a mixed-criticality system in which a critical task is scheduled exclusively on the main core, and low critical tasks are scheduled on other cores. Kritikakou et al. [37] have proved that the following regulation policy can guarantee the timeliness of the critical task[5]. Following their strategy, low-criticality tasks are suspended if a checkpoint is reached and the slack is not sufficient as indicated by the following condition:

$$\text{RWCET}_{\text{iso}}(x) + \text{RWCET}_{\text{max}} + t_{\text{RT}} > D_c - ET(x),$$

where $\text{RWCET}_{\text{iso}}(x)$ is the remaining WCET (measured in isolation) from the arrival at watchpoint x until completion. In our MB-TPA, this is equivalent to $T_t(\texttt{MBB}_{exit}) - T_t(\texttt{MBB}_x)$.

[5] Due to space constraint, the proof is omitted here. The work also includes a treatment to regulate loop components.

$\text{RWCET}_{\text{max}}$ is the WCET from watch-point x to the next watchpoint when other low critical tasks are present, which can be measured as $T_n(\texttt{MBB}_x, \texttt{MBB}_{x+1})$ according to Section 5.5 by adding interference. t_{RT} is the software interrupt overhead. Our MB-TPA does not use interrupts, but to remain safe, the delay in the milestone detection at the tracer must be considered. This term is evaluated in Section 8.1. D_c and $ET(x)$ are deadline and actual time at x. We refer to the latter as $\Theta(x)$. The required metrics for the regulation policy are offered by MB-TPA, thus our method can also achieve strict WCET enforcement.

(B) Progress-aware Profiling: In this use case, we demonstrate that it is possible to acquire application profiles about their interaction with the underlying hardware in a way that is progress aware. This can be done by performing online tracking according to what described in Section 5.6. In addition, the tracer is modified to interface with the performance monitoring unit of the tracee. By doing so, it is possible to measure the progression of architectural events (e.g. cache misses, branch mispredictions, bus stalls) at the reached milestones. This allows precise attribution of exhibited behaviors to specific code paths inside the target. More importantly, it enables correlating slowdowns on specific milestones to root causes in terms of platform behavior. And therefore, to identify hardware bottlenecks on a per-code-path basis. We evaluate this use case in Section 8.2.

(C) Progress-aware Controlled Degradation: Lastly, we consider TPA-driven detection of TPI violations due to contention over shared memory resources and perform regulation of interfering PEs with the goal of tracking a degraded performance setpoint for the target.

In a nutshell, TPI violation is triggered if the target suffers a slowdown greater than a selected α factor. At runtime, if Equation 3 does not hold, the tracer sends a signal to the tracee to pause the activity of all the other PEs. After the interfering cores have been stopped, the target might recover some slack. Thus, it might be possible to resume the paused PEs. To decide when the interfering PEs should be resumed, we use an *aggressiveness* parameter $\beta \in [0, 1]$. Whenever $slack(i) > \beta\alpha N(i)$, the interfering PEs are resumed. As β decreases, the tracer resumes the co-runners as early as possible. When β increases, the tracer becomes more conservative. We evaluate this use case in Section 8.2.

7 System Instantiation and Implementation Details

We performed a full-stack implementation of the proposed MB-TPA. We name our proof-of-concept system instantiation Timely Progress Assessment with 0 Overhead (TPAw0v). TPAw0v was implemented on the ZCU102 development board featuring a Xilinx UltraScale+ MPSoC. The target platform comprises two CPU clusters: (1) the APU cluster with four ARM Cortex-A53 CPUs operating at 1.3GHz, used as the tracee; (2) the RPU cluster with two ARM Cortex-R5 CPUs operating at 600MHz, used to implement the tracer. Following the platform assumptions described in Section 4, the target platform features an ARM Coresight infrastructure commonly with tracing capability.

Figure 6 illustrates the trace data path. Each tracee CPU has a TU, namely an ARM Embedded Trace Macrocell (ETM) [10]. The ETMs produce trace data for the respective core. The ETMs are capable of filtering the trace data by comparing the `PC` against a set of 4 range-address filters. Each filter uses two registers (`TRCACVRn`) for the address range's upper and lower ends. Trace data packets are generated whenever the `PC` falls within any of the defined ranges.

The trace packets traverse multiple on-chip CoreSight components. The bare-metal drivers used by the tracer to manage all these components were written from scratch. In TPAw0v, the ETR is configured to asynchronously store trace packets to the RPU cluster's

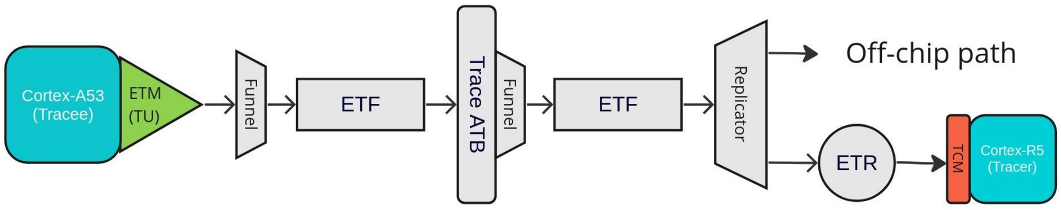


Figure 6 The Embedded Trace Macrocell (ETM) is the CPU-local device responsible for trace generation. The Trace Memory Controller [8] can be configured into an Embedded Trace FIFO (ETF) or Embedded Trace Router (ETR). The former serves as a buffer for the trace stream; the latter routes trace data to memory. ARM AMBA Advanced Trace Bus (ATB) [9] is adopted for trace data transmission. Funnels merge trace streams from potentially multiple ETMs and ATBs into a single ATB. The Replicator duplicates trace data from a single master to two slaves [12].

scratchpad (TCM), where a 2KB circular buffer is reserved. The TMG in binary format is also stored on the TCM. The tracer implements all the modes to carry out the full MB-TPA pipeline described in Section 5, including online tracking.

7.1 Constructing MB-TPA with ETM

To implement MB-TPA, the ETM is driven using a Finite State Machine (FSM) by the tracer and composed of three states (solid circles), two transition states (dashed circles), and several transitions as depicted in Figure 7. The controller starts in the *Inactive* state. This state is the only one in which reconfiguring the ETM (modifying the address filtering registers) is allowed, as the ETM is *idle*. Once reconfiguration is completed, the controller activates the ETM by asserting the `TRCPRGCTLR.EN` register (**A**), leading to a transition state to guarantee that the ETM is not idle. Here, the tracer waits for the `TRCSTATR.IDLE` register to be cleared before moving to the *Active-off* state (**B**). In *Active-off*, the ETM is monitoring the `PC`, but not generating informative packets[6], because the `PC` has not reached any addresses specified by the address filtering registers. I.e. the `PC` has not reached any milestones yet. When the `PC` reaches any of the specified addresses, Three packets are emitted in order by the ETM: a `synchronization`, a `trace-on`, and an `address` packet. This sequence signifies that a milestone was hit and the `address` packet includes the current value of the `PC`. Then, the controller moves to the *Active-on* state (**C**). Otherwise, the controller stays in Active-off (**!C**). Similar to its *"off"* counterpart, the Active-on state keeps the ETR actively waiting for the next packet (**!D**). Once the packet is finally captured, the controller (1) identifies the milestone hit via the `PC`, (2) computes the negative slack, and (3) propagates the latter to the tracee. The controller then moves back to the *Active-off* state (**D**). In both active states, the controller is allowed to request a change of address range to monitor. In such case, the ETM must be set to *idle* by clearing the `TRCPRGCTLR.EN` register (**D**). Then, the controller enters a transition state where it awaits for the `TRCSTATR.IDLE` register to be asserted, ensuring the ETM is *idle* (**E**).

8 Evaluation

First, we evaluate TPAw0v to understand its performance in terms of milestone detection delay, size of the trace blackout window, and overhead on the tracee. Next, we evaluate the

[6] In *Active-off* state the ETM still generates `synchronization` and `address` packet pairs at a very low rate. These packet pairs can be ignored for our purpose.

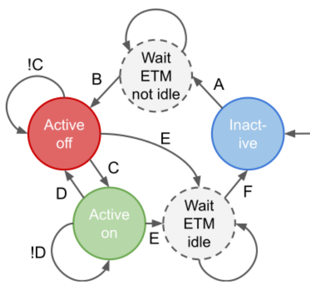


Figure 7 Tracer's controller as a finite state machine.

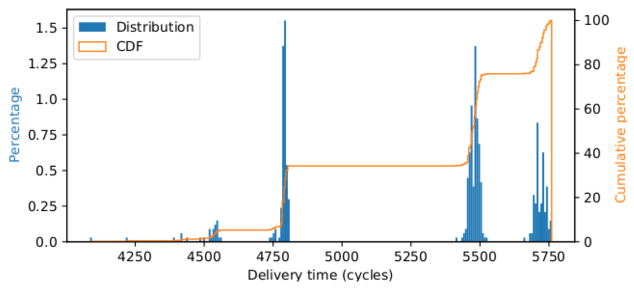


Figure 8 Delivery time (cumulative) distribution.

ability to enact progress-aware profiling and controlled performance degradation.

8.1 Progress Assessment Performance

(A) Delivery Time: Let t denote the time at which tracee executes the first instruction in the monitored MBB. The TU generates a trace packet toward the tracer via on-chip buses. Let t' denote the time at which the tracer receives it. The delivery time $\Delta t = t' - t$ has to be comparably small so that the TPAw0v can operate effectively. To measure Δt, we use a synthetic benchmark on the tracee in which the cycle counter is periodically read. MBBs are chosen as the BBs where the cycles are sampled. The tracer reads the same cycle counter upon receiving the signal. For a given MBB, the application's timestamp is t; the tracer's is t'. We sample 1500 data points, 50% in isolation and the rest with interference from memory-intensive applications. Figure 8 shows the overall (cumulative) distribution. The delivery time is upper-bounded by 5750 cycles, or $4.4\mu s$ on our 1.3GHz tracee.

Recall that software-based detection methods [38, 58] inevitably introduce overhead due to synchronous interrupt handling. In contrast, our method never interrupts the tracee. Due to our monitoring scheme's asynchronous nature, the delivery time is not an overhead term. Nonetheless, it is informative to contrast the overhead for software-based detection to the magnitude of our delivery time. A convenient way to obtain such measurement is to use a widely-adopted Linux support for dynamic binary instrumentation, namely UPROBEs [3]. They allow hooks to be registered at different locations of a user application. A software interrupt is issued when a hook is reached and time can be sampled. We measured the overhead of UPROBEs at about $4\mu s$.

(B) Blackout Window Size: The reconfiguration of the TU is solely handled by the function `reconfigure` residing in the control logic of the tracer. Thus by reading the cycle counter before/after the function call of `reconfigure`, the size of T_r can be measured. We conduct such measurements while running TPAw0v normally with target applications from the SD-VBS suite [64] which is a diverse collection of computer vision applications. The characteristics of these benchmarks have been extensively studied by the community [51,52,61]. Our measurements show that T_r is around $3\mu s$. Recall that we choose T_r in terms of number of executed branch instructions. In the (very unlikely) worst case, all the instructions executed during the blackout window are branch instructions. Thus, we conservatively set $T_r = 10000$ given the 1.3GHz tracee.

(C) Overhead on Tracee: When the tracer only performs TPA but takes no regulation actions, the target should only experience a negligible slowdown. Five SD-VBS benchmarks were evaluated: `disparity`, `texture_synthesis`, `mser`, `tracking`, and `sift`.

We run benchmarks with their respective default inputs in two configurations: (1) without

Table 1 Overhead (%) of tracer activity and TMG/trace size information.

Benchmark	`disparity`	`text.`	`mser`	`tracking`	`sift`
Mean(%)	0.512	-0.009	0.250	-0.072	0.168
Max(%)	0.585	0.033	0.263	-0.110	0.194
Min(%)	0.483	0.085	0.225	-0.059	0.173
# of MBBs in TMG	17	5	18	16	13
# of MBB hit in execution	143	1169	20	18	19
# of unfolding functions	1	1	1	1	2
TMG size (bytes)	340	108	408	320	324
Raw trace size (MB)	10	44.4	14	175.2	236.4
Filtered trace size (bytes)	1500	9400	210	350	300

TPAw0v, and (2) with TPAw0v but taking no regulation actions. Ten runs are conducted per benchmark and in each configuration. The top section of Table 1 reports the slowdown caused by TPAw0v on the benchmarks as a percentage of their runtime. Expectedly, the overhead is low ($< 0.6\%$). The low yet visible overhead in some applications might arise from interference on the main interconnect between the tracer and the tracee CPUs. Implementing the tracer on the on-chip FPGA might mitigate the issue [65] and further reduce the overhead. Negative entries indicate that the applications run faster when traced. H. Shah et al. [56] observed and theorized such counterintuitive timing anomalies.

(D) Application Considerations: The sum of delivery time and blackout window size ($\sim 7.4\mu s$) indicates the responsiveness of the tracer in detecting and reacting to milestone hits. Thus, TPAw0v is better suited for applications with execution times on the order of $10^3\mu s$ and above, e.g., data processing workloads. Approaches using software interrupts would incur overheads of at least $4\mu s$, as measured on our platform. Thus, for short-lived applications, the overhead introduced by software instrumentation would significantly degrade performance.

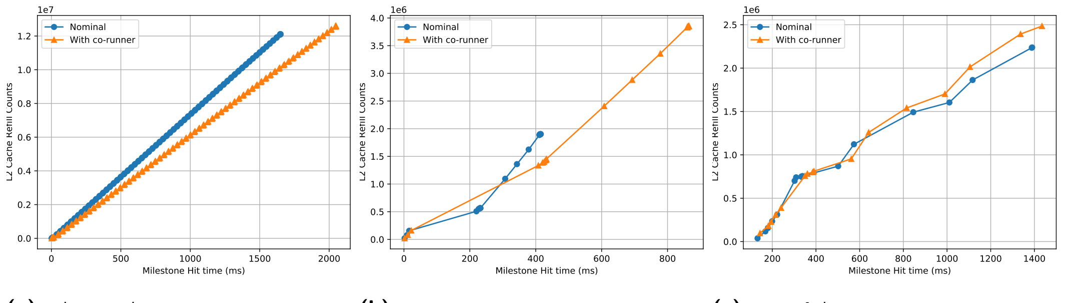
8.2 Evaluation of MB-TPA Use Cases

We hereby evaluate the last two use cases described in Section 6. For our evaluation, we consider the same five aforementioned SD-VBS benchmarks. The memory-intensive application `bandwidth` from IsolBench [63] is deployed on all the other cores to create interference in both main memory and shared cache.

(A) TMG Construction: First, we provide information regarding TMGs and trace data in the second section of Table 1. When a milestone is placed inside a loop, high granularity regulation can be achieved. `disparity` and `texture synthesis` demonstrate such granularity as the number of milestones hit is high. TMG size refers to the memory usage for the tracer to store the binary TMG; raw traces are only used during the offline MG refinement phase; the TU generates the filtered trace during online tracking.

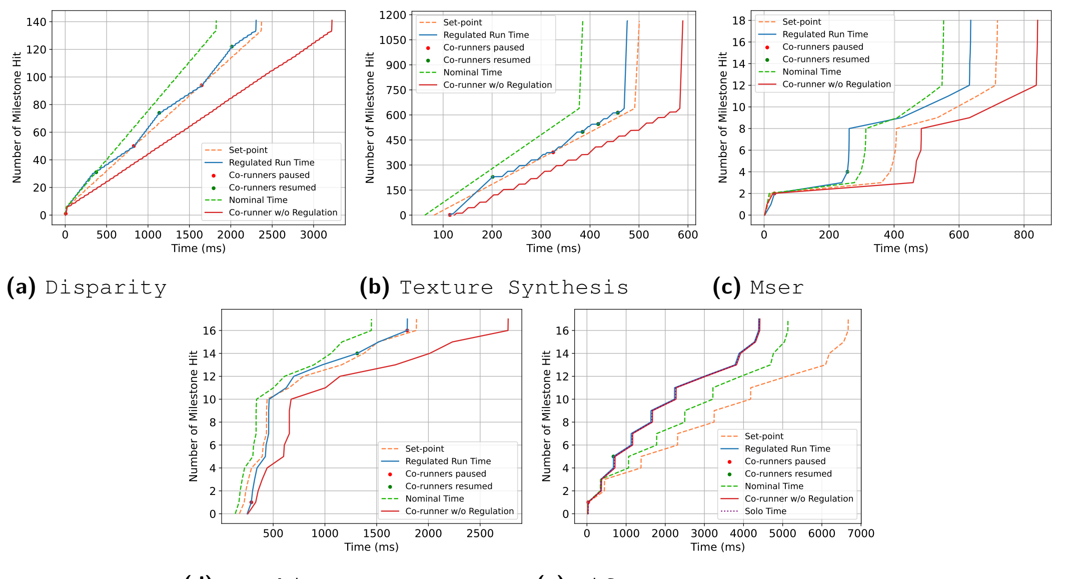
(B) Progress-aware Profiling: When the execution reaches a milestone, we collect architectural event statistics by directly reading the PMU event counters[7]. In this evaluation, the architectural event monitored is the L2 data cache refill, i.e. we track last-level cache misses. The benchmarks under evaluation run (1) in isolation and (2) with interference tasks.

[7] ETM can also report architectural events in the trace stream. ETM can optionally implement external inputs which connect to PMU event bus lines. Event packets can be inserted into the trace stream whenever the monitored events occur.



(a) `Disparity` **(b)** `Mser` **(c)** `Tracking`

Figure 9 Relationship between timeliness (x), L2 cache misses (y), and milestones (markers).



(a) `Disparity` **(b)** `Texture Synthesis` **(c)** `Mser`

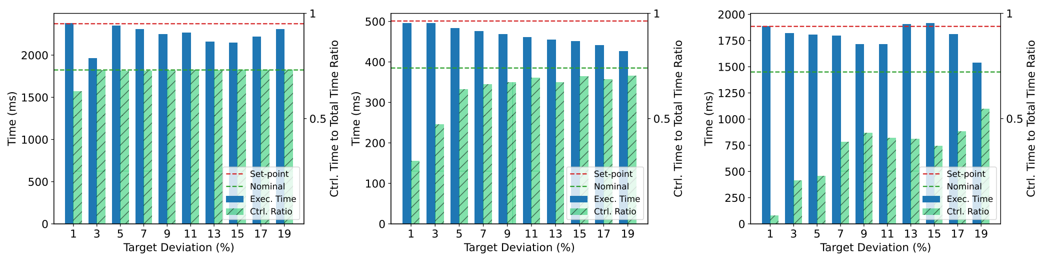
(d) `Tracking` **(e)** `Sift`

Figure 10 The TMG for `disparity` and `texture synthesis` captures appropriate loops, achieving fine granularity. Despite a coarser control for `mser`, TPI is maintained.

In each case, the benchmark runs 20 times. The tracer reports the time and cache refill statistics at each milestone hit. The relationship between elapsed time (x-axis), cumulative number of L2 misses (y-axis), and milestones hit (markers)—and therefore segments of executed code—as captured for three SD-VBS applications is reported in Figure 9. The figure highlights that `disparity` and `tracking` suffer only marginally from cache contention, while five milestones in `mser` are significantly impacted by contention in L2.

The significance of relating the consumption of hardware resources to progress is twofold. First, resource management decisions can be enacted proactively as opposed to reactively. Second, by comparing the expected profile at a milestone to what is observed online, a system can identify the root causes of performance degradation and enact appropriate corrective actions. The combination of progress tracking and progress-aware resource management requires extensive research.

(C) Controlled Performance Degradation: In this scenario, we evaluate the ability to set a degraded performance setpoint for the application under analysis and stop/resume interfering cores based on the online slack calculation reported by the tracer. The behavior of the five SD-VBS benchmarks is reported in Figure. 10. We compare the runtime under



(a) `Disparity` **(b)** `Texture Synthesis` **(c)** `Tracking`

Figure 11 As target deviation β increases, the tracer becomes more conservative, and only resumes the co-runner when a sufficient positive slack presents. Thus, the application follows the set-point more closely for small β.

tracer-enforced regulation ("Regulated Run Time") with two other cases: (1) the nominal case, i.e., the worst-case progress in isolation, and (2) the progress under unregulated interference ("Co-runner w/o Regulation"). We use $\alpha = 1.3$ and $\beta = 7\%$; the resulting progress reference is labeled "Set point." The history of accessed milestones in chronological order is reported on the y-axis; the time elapsed between milestones is reported on the x-axis; the binary decisions to suspend (red dot) or resume (green dot) the co-runners are reported.

In all the cases, the tracer was able to enforce a controllably degraded notion of TPI for the target. Corrective measures are taken as soon as the detected progress falls below the reference. The specific value of β we considered works well in most cases but becomes overly conservative in the case of `mser`. In this case, preventing a slowdown in the early stages (at milestones 2–4) is sufficient to ensure that the setpoint is met for the rest of the run. The behavior of `sift` (Figure 10e) is interestingly different. The solo, uncontrolled, and controlled progress nearly coincide. This indicates that `sift` is unaffected by the interference tasks. The nominal progress, however, is slower than the above three. Recall that the nominal time for each edge is taken as the maximum transfer time across all runs. But in a single run, not all transfers take the worst-case time.

To better understand the impact of β on the behavior of the applications, we sweep through values of $\beta \in [1\%, \ldots, 19\%]$ and present the results in Figure 11. The "Exec Time" bar captures the runtime under contention and regulation. The "Ctrl. Ratio" bar reports the fraction of time during which the real-time is below the set-point. As β increases, TPAw0v becomes more conservative, and the aggressiveness of the regulation increases. `sift` is not included since it does not suffer from performance degradation.

9 Conclusion

Prompted by the demand for high-performance embedded platforms, the design of modern system-on-chip has gained in complexity at the expense of software predictability and timeliness. We argue that reasoning on the progress of live applications must be a key requirement to achieve *Timely Progress Integrity*. In this paper, we propose a method called MB-TPA and present a prototype, TPAw0v, feasible on widely available commercial platforms featuring tracing capabilities. Our experiments show that our prototype is successful in tracking the progress of applications under test with near-zero overhead while operating on a lower-performance core! Moreover, through its prototype implementation, we demonstrate the capability of our model to detect execution anomalies and enforce corrective measures to preserve TPI. We envision that the contributions made by this work represent the first building blocks towards elaborated real-time policies with TPI at their core.

References

1 Powertrace iii. https://www.lauterbach.com/powertrace3.html. Accessed: 01-03-2023.

2 Technology overview. https://www.ghs.com/products/probe.html. Accessed: 01-03-2023.

3 Uprobe-tracer: Uprobe-based event tracing. https://docs.kernel.org/trace/uprobetracer.html.

4 Working draft of the risc-v processor trace specification. https://github.com/riscv-non-isa/riscv-trace-spec. Accessed: 01-03-2023.

5 J. Abella, C. Hernandez, E. Quiñones, F. J. Cazorla, P. R. Conmy, M. Azkarate-askasua, J. Perez, E. Mezzetti, and T. Vardanega. Wcet analysis methods: Pitfalls and challenges on their trustworthiness. In *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–10, 2015.

6 S. M. Ali Zeinolabedin, J. Partzsch, and C. Mayr. Analyzing arm coresight etmv4.x data trace stream with a real-time hardware accelerator. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1606–1609, 2021.

7 ARM. Coresight components technical reference manual, 2004.

8 ARM. CoreSight trace memory controller technical reference manual, 2010.

9 ARM. AMBA ATB Protocol Specification, 2012.

10 ARM. Embedded trace macrocell architecture specification etmv4.0 to etm4.6, 2012.

11 ARM. Arm architecture reference manual for a-profile architecture, 2013.

12 ARM. ARM CoreSight SoC-400 Technical Reference Manual, 2015.

13 S. Baruah, V. Bonifaci, G. DAngelo, H. Li, A. Marchetti-Spaccamela, S. van der Ster, and L. Stougie. The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems. In *24th Euromicro Conference on Real-Time Systems (ECRTS 2012)*, pages 145–154, Los Alamitos, CA, USA, Jul 2012. IEEE Computer Society.

14 S. Baruah, A. Burns, and R. Davis. Response-time analysis for mixed criticality systems. In *2011 IEEE 32nd Real-Time Systems Symposium*, pages 34–43, 2011.

15 A. Bastoni, B. B. Brandenburg, and J. H. Anderson. Cache-related preemption and migration delays : Empirical approximation and impact on schedulability. In *Proceedings of the 6th annual workshop on. Operating Systems Platforms for Embedded Real-Time Applications*, volume 10 of *OSPERT'10*, page 33–44, 2010.

16 E. Betti, S. Bak, R. Pellizzoni, M. Caccamo, and L. Sha. Real-time i/o management system with cots peripherals. *IEEE Transactions on Computers*, 62(1):45–58, 2013.

17 R. J. Bril, S. Altmeyer, M. M. H. P. van den Heuvel, R. I. Davis, and M. Behnam. Fixed priority scheduling with pre-emption thresholds and cache-related pre-emption delays: integrated analysis and evaluation. *Real-Time Systems*, 53(4):403–466, Jul 2017.

18 A. Burns and R. I. Davis. *Mixed Criticality Systems - A Review : (13th Edition, February 2022)*. February 2022.

19 M. Caccamo, G. Buttazzo, and L. Sha. Elastic feedback control. In *Proceedings 12th Euromicro Conference on Real-Time Systems. Euromicro RTS 2000*, pages 121–128, 2000.

20 H. Cassé and P. Sainrat. OTAWA, a Framework for Experimenting WCET Computations. In *Conference ERTS'06*, Toulouse, France, Jan. 2006.

21 I. Corp. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B, 3C & 3D): System Programming Guide, 2022.

22 E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

23 P. Dinges and G. Agha. Targeted test input generation using symbolic-concrete backward execution. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, page 31–36, New York, NY, USA, 2014. Association for Computing Machinery.

24 P. Ekberg and W. Yi. Outstanding paper award: Bounding and shaping the demand of mixed-criticality sporadic tasks. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 135–144, 2012.

25 L. Feng, J. Huang, J. Hu, and A. Reddy. Fastcfi: Real-time control-flow integrity using fpga without code instrumentation. *ACM Trans. Des. Autom. Electron. Syst.*, 26(5), jun 2021.

26 R. Gifford, N. Gandhi, L. T. X. Phan, and A. Haeberlen. DNA: Dynamic resource allocation for soft real-time multicore systems. In *27th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '21)*, May 2021.

27 G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni. A survey on cache management mechanisms for real-time embedded systems. *ACM Comput. Surv.*, 48(2), nov 2015.

28 X. Gu and A. Easwaran. Dynamic budget management with service guarantees for mixed-criticality systems. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 47–56, 2016.

29 X. Gu, A. Easwaran, K.-M. Phan, and I. Shin. Resource efficient isolation mechanisms in mixed-criticality scheduling. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 13–24, 2015.

30 J. Gustafsson. Usability aspects of WCET analysis. In *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 346–352, 2008.

31 D. Hardy, B. Rouxel, and I. Puaut. The Heptane Static Worst-Case Execution Time Estimation Tool. In J. Reineke, editor, *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, volume 57 of *OpenAccess Series in Informatics (OASIcs)*, pages 8:1–8:12, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

32 A. Hoppe, J. Becker, and F. L. Kastensmidt. High-speed hardware accelerator for trace decoding in real-time program monitoring. In *2021 IEEE 12th Latin America Symposium on Circuits and System (LASCAS)*, pages 1–4, 2021.

33 T.-Y. Huang, J.-S. Liu, and D. Hull. A method for bounding the effect of dma i/o interference on program execution time. In *17th IEEE Real-Time Systems Symposium*, pages 275–285, 1996.

34 M. Kadar, G. Fohler, D. Kuzhiyelil, and P. Gorski. Safety-aware integration of hardware-assisted program tracing in mixed-criticality systems for security monitoring. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 292–305, 2021.

35 R. Kirner and P. P. Puschner. Discussion of misconceptions about WCET analysis. In J. Gustafsson, editor, *Proceedings of the 3rd International Workshop on Worst-Case Execution Time Analysis, WCET 2003 - a Satellite Event to ECRTS 2003, Polytechnic Institute of Porto, Portugal, July 1, 2003*, volume MDH-MRTC-116/2003-1-SE, pages 61–64. Department of Computer Science and Engineering, Mälardalen University, Box 883, 721 23 Västerås, Sweden, 2003.

36 A. Kritikakou, T. Marty, and M. Roy. Dynascore: Dynamic software controller to increase resource utilization in mixed-critical systems. *ACM Trans. Des. Autom. Electron. Syst.*, 23(2), oct 2017.

37 A. Kritikakou, C. Pagetti, O. Baldellon, M. Roy, and C. Rochange. Run-time control to increase task parallelism in mixed-critical systems. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 119–128, 2014.

38 A. Kritikakou, C. Rochange, M. Faugère, C. Pagetti, M. Roy, S. Girbal, and D. G. Pérez. Distributed run-time WCET controller for concurrent critical tasks in mixed-critical systems. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, RTNS '14, page 139–148, New York, NY, USA, 2014. Association for Computing Machinery.

39 D. Kuzhiyelil, P. Zieris, M. Kadar, S. Tverdyshev, and G. Fohler. Towards transparent control-flow integrity in safety-critical systems. In *International Conference on Information Security*, pages 290–311. Springer, 2020.

40 C.-G. Lee, H. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 47(6):700–713, 1998.

41 J. Lee, H. S. Chwa, L. T. X. Phan, I. Shin, and I. Lee. Mc-adapt: Adaptive task dropping in mixed-criticality scheduling. 16(5s), sep 2017.

42 D. Liu, J. Spasic, N. Guan, G. Chen, S. Liu, T. Stefanov, and W. Yi. Edf-vd scheduling of mixed-criticality systems with degraded quality guarantees. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 35–46, 2016.

43 D. Lo, M. Ismail, T. Chen, and G. E. Suh. Slack-aware opportunistic monitoring for real-time systems. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 203–214, 2014.

44 C. Lu, J. A. Stankovic, S. H. Son, and G. Tao. Feedback control real-time scheduling: Framework, modeling, and algorithms*. *Real-Time Systems*, 23(1):85–126, Jul 2002.

45 T. Lugo, S. Lozano, J. Fernández, and J. Carretero. A survey of techniques for reducing interference in real-time applications on multicore platforms. *IEEE Access*, 10:21853–21882, 2022.

46 M. Lv, Z. Gu, N. Guan, Q. Deng, and G. Yu. Performance comparison of techniques on static path analysis of wcet. In *2008 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, volume 1, pages 104–111, 2008.

47 M. Lv, N. Guan, J. Reineke, R. Wilhelm, and W. Yi. A survey on static cache analysis for real-time systems. *Leibniz Transactions on Embedded Systems*, 3(1):05:1–05:48, Jun. 2016.

48 C. Maiza, H. Rihani, J. Rivas, J. Goossens, S. Altmeyer, and R. Davis. A Survey of Timing Verification Techniques for Multi-Core Real-Time Systems. *ACM Comput. Surv.*, 52(3), June 2019.

49 R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, page 45–54, 2013.

50 H. S. Negi, T. Mitra, and A. Roychoudhury. Accurate estimation of cache-related preemption delay. In *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '03, page 201–206, New York, NY, USA, 2003. Association for Computing Machinery.

51 M. Nicolella, D. Hoornaert, S. Roozkhosh, A. Bastoni, and R. Mancuso. Know your enemy: Benchmarking and experimenting with insight as a goal. In *2022 IEEE Real-Time Systems Symposium (RTSS)*, RTSS 2022, 2022.

52 M. Nicolella, S. Roozkhosh, D. Hoornaert, A. Bastoni, and R. Mancuso. Rt-bench: An extensible benchmark framework for the analysis and management of real-time applications. In *Proceedings of the 30th International Conference on Real-Time Networks and Systems*, RTNS 2022, page 184–195, New York, NY, USA, 2022. Association for Computing Machinery.

53 A. Papadopoulos, E. Bini, S. Baruah, and A. Burns. Adaptmc: A control-theoretic approach for achieving resilience in mixed-criticality systems. pages 14:1–14:22, Dagstuhl, July 2018. LIPICS.

54 J. Ren and L. X. Phan. Mixed-criticality scheduling on multiprocessors using task grouping. In *2015 27th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 25–34, Los Alamitos, CA, USA, Jul 2015. IEEE Computer Society.

55 G. Schwaricke, R. Tabish, R. Pellizzoni, R. Mancuso, A. Bastoni, A. Zuepke, and M. Caccamo. A real-time virtio-based framework for predictable inter-vm communication. In *2021 IEEE Real-Time Systems Symposium (RTSS)*, pages 27–40, 2021.

56 H. Shah, K. Huang, and A. Knoll. Timing anomalies in multi-core architectures due to the interference on the shared resources. In *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 708–713, 2014.

57 Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

58 S. Sinha, R. West, and A. Golchin. Pastime: Progress-aware scheduling for time-critical computing. *arXiv preprint arXiv:1908.06211*, 2019.

59 P. Sohal, R. Tabish, U. Drepper, and R. Mancuso. E-warp: A system-wide framework for memory bandwidth profiling and management. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 345–357, 2020.

60 J. Stankovic, C. Lu, S. Son, and G. Tao. The case for feedback control real-time scheduling. In *Proceedings of 11th Euromicro Conference on Real-Time Systems. Euromicro RTS'99*, pages 11–20, 1999.

61 D. Tarapore, S. Roozkhosh, S. Brzozowski, and R. Mancuso. Observing the invisible: Live cache inspection for high-performance embedded systems. *IEEE Transactions on Computers*, 71(3):559–572, 2022.

62 P. K. Valsan, H. Yun, and F. Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12, 2016.

63 P. K. Valsan, H. Yun, and F. Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12, 2016.

64 S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor. SD-VBS: The san diego vision benchmark suite. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 55–64, 2009.

65 Xilinx. Zynq UltraScale+ Device Technical Reference Manual, 2023.

66 H. Yun, R. Mancuso, Z. P. Wu, and R. Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, page 155–166, 2014.

67 H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, 2013.

68 M. Zini, G. Cicero, D. Casini, and A. Biondi. Profiling and controlling i/o-related memory contention in cots heterogeneous platforms. *Software: Practice and Experience*, 52(5):1095–1113, 2022.