PROCEEDINGS OF

OSPERT 2022

The 15th Annual Workshop on
Operating Systems Platforms for
Embedded Real-Time Applications

July 5th, 2022 in Modena, Italy

in conjunction with

The 34th Euromicro Conference on Real-Time Systems
July 5–8, 2022, Modena, Italy

Editors:
Daniel LOHMANN
Renato MANCUSO

Contents

Message from the Chairs

Welcome to OSPERT'22, the 16th annual workshop on Operating Systems Platforms for Embedded Real-Time Applications. After two years of pandemic silence, we invite you to join us in participating in a workshop of lively discussions, exchanging ideas about systems issues related to real-time and embedded systems.

The workshop will open with a keynote by Konrad Schwarz, discussing hardware partitioning options and issues on RISC-V and telling from his decade-long experience working on real-time and mixed-criticality system software in industrial settings.

OSPERT'22 received 12 submissions from which 8 were selected by the program commitee to be presented at the workshop. Each paper received three individual reviews. Our special thanks go to the program committee, a team of ten experts for volunteering their time and effort to provide useful feedback to the authors, and of course to all the authors for their contributions and hard work.

OSPERT'22 would not have been possible without the support of many people. The first thanks are due to Sebastian Altmeyer, Benjamin Rouxel, Marko Bertogna and the whole ECRTS organizing team for entrusting us with organizing OSPERT, and for their continued support of the workshop. We would also like to thank the chairs of prior editions of the workshop who shaped OSPERT and let it grow into the successful event that it is today.

Last, but not least, we thank you, the audience, for your participation. Through your stimulating questions and lively interest you help to define and improve OSPERT. We hope you will enjoy this day.

The Workshop Chairs,

Daniel Lohmann
Leibniz Universität Hannover
Germany

Renato Mancuso
Boston University
USA

Program Committee

Wolfgang Mauerer, *OTH Regensburg*
Richard West, *Boston University*
Hyoseung Kim, *University of California Riverside*
Mohamed Hassan, *McMaster University*
Rudolfo Pellizzoni, *University of Waterloo*
Michal Sojka, *Czech Technical University in Prague*
Bryan Ward, *MIT Lincoln Lab*

Keynote Talk

Mixed Criticality on RISC-V: Experiences from Porting a Partitioning Hypervisor

Konrad Schwarz
Senior Engineer, Siemens Corporate Technology

RISC-V is an emerging open-source and greenfield ISA designed to be minimalistic and modular, yet scalable from embedded to data center. One of the first extensions added to the base architecture is support for hypervisors. Jailhouse is a small hypervisor designed to statically partition embedded multi-core systems to enable consolidation of mixed-criticality systems onto a single hardware platform.

In the talk, I describe my experiences with porting Jailhouse, originally designed for x86 and later to ARM, to RISC-V. This proved to be surprisingly long-winded; some of the traps and pitfalls will be discussed, as well as some of the shortcomings of the current hardware state of the art and how RISC-V plans to address these in future.

Konrad Schwarz has degrees in Computer Science and Mathematics from the Technical University of Vienna. After short stints at Accenture and Motorola, he has been at Siemens Corporate Technology for over 20 years, working with business units such as Semiconductors (now Infineon), Automotive (now Continental), Mobility (train control systems), and industrial automation and motion control.

RTOS-Independent Interaction Analysis in ARA

Gerion Entrup, Jan Neugebauer, Daniel Lohmann
Leibniz Universität Hannover
{entrup, lohmann}@sra.uni-hannover.de, jan.neugebauer@stud.uni-hannover.de

***Abstract*—ARA is an RTOS-aware whole-system compiler for embedded applications that takes RTOS semantics into account for interprocedural analysis and optimization. To be applicable for a multitude of RTOS interfaces and semantics, ARAs analysis steps shall operate on an abstract RTOS model as far as possible, while still providing means to exploit OS-specific particularities. In this paper, we describe the design of such a model and its utilization with two static analysis algorithms for AUTOSAR, FreeRTOS, Zephyr and a subset of POSIX.**

I. Introduction

Embedded systems typically come as whole systems: All code that will eventually run on the device is known in advance. For the compilation process, this lays the foundation for interprocedural whole-system optimization, which is well-explored on the language level [15], [23], [18]. Taking also the real-time operating system (OS) into account [21], [2], [8] enables further aggressive optimizations by tailoring the OS to the actual application implementation.

One approach to achieve whole-system optimization is model-based generation, that is, generating the system including the application from an abstract language description [29], [1]. However, in practice, embedded applications are written against a classical system-call interface, employing OSs such as FreeRTOS or Zephyr mainly as a helper library or markup-language to describe event- and control-flow interactions at run time.

With the Automatic Real-time System Analyzer (ARA)[1], we are building a whole-system compiler (based on LLVM) for embedded systems written against such common OS interfaces. ARA is then able to compile the application with additional optimizations based on the actual interactions between the OS and the application code. Examples include the transformation of dynamic into static initialization [13], folding of pre-known scheduling decisions [9], or elision of never taken locks in multi-core settings (not yet published).

For genericity, ARA shall support multiple OSs without having to change the underlying analysis algorithms. However, these analyses need, by design, OS-specific knowledge in some parts. We, therefore, split them into an OS-agnostic core and summarize all OS-specific parts in an analysis independent *OS model* that serves as a unified interface between all algorithms and OSs. Figure 1 visualizes the separation and an overview of the ARA toolchain (we present details in Section III).

This work was partly supported by the German Research Foundation (DFG) under grant no. LO 1719/4-1

[1]https://github.com/luhsra/ara

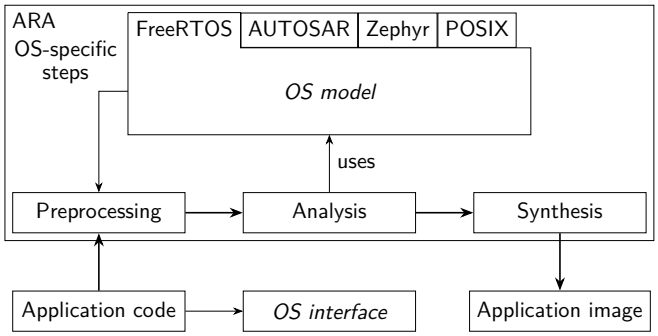


Fig. 1: Overview over the ARA toolchain. ARA processes the application code with various steps and finally emits an image. The model encapsulate all OS specific knowledge. It (optionally) triggers OS-specific preprocessing steps and is used by the main analyses for syscall interpretation.

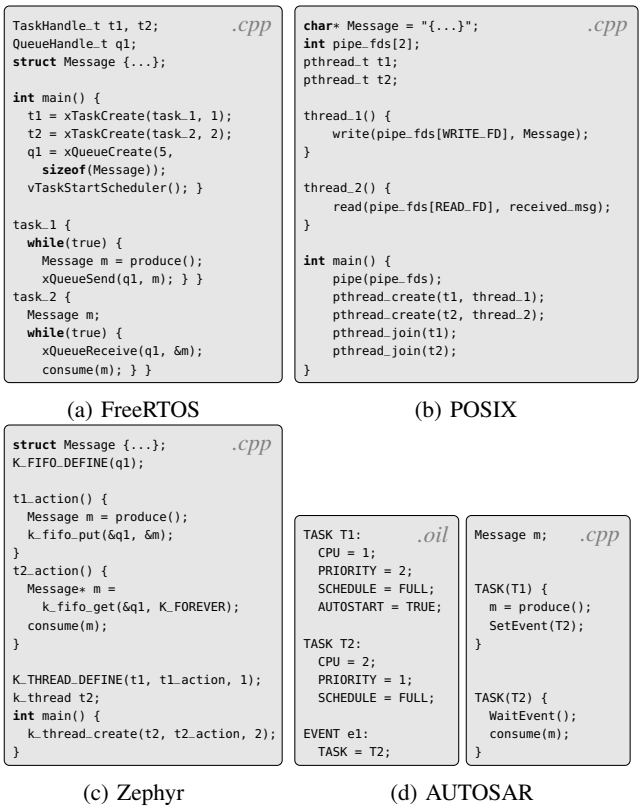
In this paper, we describe our findings from designing this model and implementing it for four very different syscall interfaces: AUTOSAR, Zephyr, FreeRTOS, and a subset of POSIX. In particular, we claim the following contributions:

- An abstract OS model with implementations for FreeRTOS, AUTOSAR, Zephyr, and POSIX.
- The possibility to analyze and (partly) optimize real-world systems for all these OSs.

II. System Model and Implementation

For our OS model, we target embedded real-time systems. The concrete OS semantics and types of system objects have no further constraints. However, the model requires that all communication between application and OS takes place via explicit syscalls or interrupts.

The current implementation in ARA imposes some further constraints: As a toolset for static analysis and optimization, its algorithms rely on a closed-world assumption, that is, all code is known in advance. Late binding via function pointers is supported and sound, but excessive use may impact the strictness of analysis results. We furthermore assume a defined application starting point, which, however, can also be given by the OS model according to the OS-defined scheduling strategy. Technically, ARA operates on the LLVM intermediate representation (IR) and expects a single file in this format. We implemented the model for FreeRTOS, AUTOSAR, Zephyr, and POSIX; Figure 2 shows a minimal application example for each of them. While semantically equivalent, the system mostly differs in the way system objects (threads, events, ...) are instantiated: AUTOSAR is completely static, all instances of OS



(a) FreeRTOS (b) POSIX

(c) Zephyr (d) AUTOSAR

Fig. 2: Examples for OS interfaces: Two threads implement a producer–consumer scheme. In FreeRTOS, POSIX, and Zephyr via a queue; in AUTOSAR, which misses a queue abstraction, an event (condition variable) is employed. AUTOSAR specifies its OS objects in an extra configuration file (*.oil*).

Listing 1: The SIA algorithm (sketched)

```
def SIA(entry) -> InstanceGraph:
  instance_graph = InstanceGraph()
  for call in CFG:
    if model.is_syscall(call):
      for call_context in all_call_contexts(entry, call)
        instance_graph.update(model.interpret(call,
                                              call_context))
  return instance_graph
```

objects are specified in a configuration file and typically created at compile time. In contrast, FreeRTOS and POSIX (except static mutexes) require dynamic OS object creation by syscalls at run time. Zephyr supports both, static (via preprocessor macros) and dynamic (via syscalls) instantiation.

III. OS Model Design

Our model is based on two fundamental ideas: (1) The least common ground of all operating systems are syscalls, which modify the state of OS objects (such as threads or mutexes). (2) The model shall always serve the most detailed information possible about a specific syscall and its resulting state changes. Thereby, the model supports the most detailed analysis while others can just throw away the unneeded details.

For the initial design of the OS model, we target mainly two analyses, the static instance analysis (SIA) [13], which is a flow-insensitive analysis, and the system-state enumeration (SSE) [7], as an example of a flow-sensitive analysis. To better understand the underlying requirements for the model design, we briefly introduce them here.

Listing 2: The SSE algorithm (sketched)

```
def system_semantic(state) -> List[State]:
  if model.is_syscall(state.abb):
    new_states = model.interpret(state)
    return model.schedule(new_states)
  else:
    return follow_control_flow(state)

def SSE(entry) -> SSTG:
  sstg = SSTG()
  stack = model.get_initial_os_state()
  while stack:
    state = stack.pop()
    new_states = system_semantic(state)
    sstg.add_nodes(new_states)
    sstg.connect(new_states, state)
    stack.push(new_states)
```

A. SIA

The SIA retrieves all system-object instances (and interactions) that are created over the whole lifetime of the system and captures them in the instance graph. Its nodes represent the instances, its edges the interactions. Listing 1 sketches the algorithm. First, it iterates all syscalls. After that, the analysis calculates the call context of each syscall to enable a call-context–aware analysis of the argument values. The call together with its context is then given to the model which calculates the OS-specific effect on the instance graph. From the model point of view, the analysis mainly needs this information:

- Which call is a syscall and what is its category?
- What is the effect of the syscall on the instance graph?

B. SSE

The SSE at its core is designed as a symbolic execution on the OS level. It defines an abstract system state (the OS relevant state of the whole system), extracts the starting state of the system, and traverses the control flow from this point while capturing the effect of each instruction as a new state. Listing 2 sketches the SSE algorithm. The analysis starts by retrieving an OS-specific initial state that it pushes onto a stack. Then, for each state, it first retrieves the effects of the current control flow onto the state (it interprets the semantics of the system), which it captures in a set of new states. After that, it connects the new states with the old one thus forming a graph, the static state-transition graph (SSTG). The *system semantic* function is divided into two parts: If the state represents a syscall, the model needs to interpret and schedule it. Otherwise, the analysis calculates the new states by following the normal control flow. As part of this, it also triggers all currently active interrupts whose handling is part of the model again (not sketched).

Listing 3: The model interface

```
class OSBase:
  public:
    get_special_steps() -> List[Step]
    get_initial_state(cfg, instances: Graph) -> State

    get_interrupts(instances: Graph) -> List[int]
    handle_irq(state, cpu_id: int, irq: int) -> State
    handle_exit(state, cpu_id: int) -> List[State]

    interpret(state, cpu_id: int,
              categories=All) -> List[State]
    schedule(state, cpus=None) -> List[State]

  private:
    List[Syscall] syscalls
```

From the model point of view, the SSE needs the following information:

- Which call is a syscall?
- What is the effect of the syscall on the abstract state?
- In which abstract state does the system start?
- The possibility to schedule an abstract state.
- Which interrupts can occur in which state and how they are handled?

C. A generic OS model

Additionally, to be more generic, our model should fulfill also the following requirements: (1) It should be able to support multi-core applications. In particular, this means, that the model must be able to calculate the effect of a syscall on a specific CPU. (2) Furthermore, it should not restrict the OS initialization and setup process. The presented OS all have different configuration mechanisms, which shall be supported. (3) Finally, the model should allow other future analyses of different precision.

All this results in the definition of an OS interpreter that acts on abstract system states (AbSSs), that is, the model implements a function for each syscall that takes an AbSS, interprets the effect of a syscall on this state, and outputs one or multiple follow-up states:

$$\text{AbSS}_{a,n+1}, \text{AbSS}_{b,n+1}, \cdots = \text{interpret}(\text{AbSS}_n)$$

Conceptually, this is pretty close to the SSE algorithm. The AbSS, however, is extended. Figure 3 presents such an AbSS. First, it includes a reference to the instance graph, a data structure whose elements are immutable and to which only can be added. Furthermore, it holds all system-object instance contexts, which represents all changeable parts of an OS object instantiation, for example, the current thread status. The exact content of the context is OS specific and therefore not part of the generic interface. Finally, it holds the current execution context for each CPU, that is, each execution unit in the system. This consists of the current instruction pointer, a call path to specify the calling context, the current interrupt state, and the currently executed instance. To summarize, the state consists of OS-specific parts, the objects and their contexts, and hardware-specific parts, the execution contexts.

With that, each OS model implements a generic interface that uses the AbSSs. Listing 3 shows the (simplified) interface. The list of syscalls contains most of the information. Each syscall is an object with four properties: the name, its signature, a category and an interpret function. The name serves as unique identifier to dispatch to the correct syscall interpretation function. The category is used to fasten the analysis when used as a filter. The signature is necessary for the extraction of the syscall arguments, which in turn is necessary for the correct interpretation of the syscall. Finally, the interpret function gets an abstract state as input and outputs a list of new states which represents the effects of this specific syscall.

To ease the development, we use a Python decorator that turns a function into a syscall object and adds the necessary value-analysis code to each interpret function. Each argument in the signature can be annotated with extra information for the value analysis. The interpret function gets the results of the value analysis via the `args` argument. The syscall name is extracted from the function name. With that, for example, the implementation for *SetEvent* in AUTOSAR looks as follows:

```
@syscall(categories={SyscallCategory.comm},
         signature=(Arg("task", ty=Task, hint=SigType.instance),
                    Arg("event_mask")))
def SetEvent(cfg, state, cpu_id, args, va):
  task_ctx = state.context[args.task]
  # set the task ready if it already waits
  if task_ctx.status == TaskStatus.blocked and \
      event_mask & task_ctx.waited_events != 0:
    task_ctx.status = TaskStatus.ready
    task_ctx.waited_events = 0

  # set the event
  if task_ctx.status != TaskStatus.suspended:
    task_ctx.received_events |= event_mask

  # update the instance graph
  cur_task = state.cpus[cpu_id].instance
  for event in get_events(args.event_mask):
    state.instances.add_edge(cur_task, event)

  return state
```

All the effects of *SetEvent* are captured: First, it wakes up the potentially waiting task. Then, it updates the task's event mask and finally marks the interaction within the instance graph.

The rest of the interface provides the necessary functions for initialization, interrupt handling, and interpretation:

- `get_special_steps` gives the OS-specific preprocessing steps and `get_initial_state` returns the first abstract system state.
- `get_interrupts` returns a list of interrupts that are triggered by the analysis if needed and can be handled via `handle_irq` on which the `irq` argument describes the interrupt to be handled. `handle_exit` outputs a new state which represents all effects that result from an interrupt exit.
- `interpret` and `schedule` are the actual transition functions for abstract states to simulate a syscall interpretation and a reschedule. Both functions return a list of follow-up states.

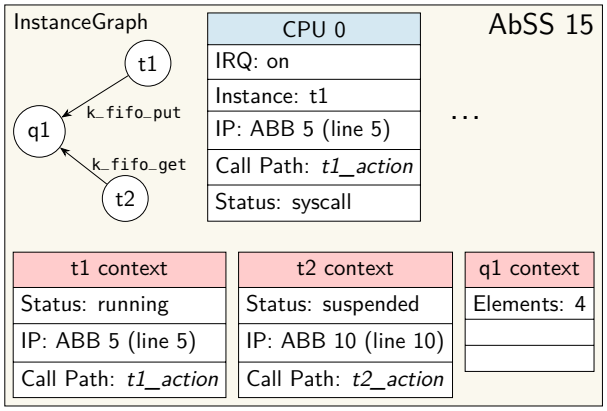


Fig. 3: Representation of the AbSS. It contains a reference to the instance graph, a list of OS-object-specific contexts, and a list of CPUs. The values match the Zephyr example application (Figure 2c).

To support multi-core systems, most functions also get an additional `cpu_id` argument to specify the (abstract) CPU on which the action should take place. The analysis has to take care of invoking the in reality happening parallel actions in a sequential manner.

Figure 1 gives an overview of embedding the model into ARA. The application code that is written against a specific OS interface is preprocessed by ARA to extract control flow and data flow. At this stage, the model can request additional steps, like the parsing of extra system configuration files. After that, the main analyses run, which use the model interface for all OS-specific parts.

Mapping the SSE onto the generic model is trivial. The model interface basically provides all necessary functions for direct SSE support.

For using the model with the SIA, we have to slightly modify the algorithm. Since the model applies the instance-graph specific effects only as part of an overall state change, the SIA has to craft a fake state to fit the model functions. For that, it combines the current instance graph, constructs a fake CPU and empty OS-object contexts. The fake CPU contains the current instruction pointer, call context and active OS-object. The model interprets this state and returns an updated state. From that, the SIA can extract the updated instance graph and continue.

IV. Experimental Validation

To validate the model, we apply the SIA and the SSE to several applications (see Table I for details). The SSE, which enumerates *all* system states, depends on a strictly bounded set of system objects and interactions. Currently, only AUTOSAR ensures this. On the other OSs, the application might, for instance, create system objects in an unbounded loop.

A. FreeRTOS

For FreeRTOS, we verified the model with the GPSLogger[2], an embedded application for logging GPS positional data on an handheld device and the LibrePilot CopterControl[3] firmware as a safety-critical real-time application for the flight controller of a quadcopter.

Doing so for the GPSLogger results in 6 tasks, 3 queues and 1 mutex which a manual verification proofs as complete. The SIA fails to resolve 3 interactions between tasks and mutexes (out of 15 interactions in total) due to restrictions in the value analysis, which fails the correct syscall argument retrieval to get the involved mutex instance. Especially for mutexes, the GPSLogger uses a C++ wrapper class which forces the value analyzer to resolve two indirections.

For the LibrePilot the SIA finds 17 tasks, 15 queues and 9 mutexes which are correct. Additionally, it can determine 12 interactions, while it ignores 22 invocations of interaction syscalls due to restrictions in the value analyzer.

B. Zephyr

For Zephyr,[4] we were not able to obtain any implemented real-world application. Hence, we decided to choose two of the bigger benchmarks from their test suite as applications: sys_kernel and app_kernel.

The app_kernel application creates all OS objects statically, for which we use a special preprocessing step. It detects 2 threads, 6 kernel semaphores, 4 message queues, 3 pipes and 1 mutex, which a manual check verifies as correct.

The OS objects are connected with 60 interactions. ARA fails to determine the arguments of 2 interactions. They belong to a pipe interaction in which the pipes are stored dynamically in an array and therefore are not found by the value analyzer.

The sys_kernel benchmark creates its instances dynamically of which ARA detects 22 threads, 14 queues, 6 kernel semaphores, and 6 stacks. Additionally, ARA finds 274 interactions. A manual check confirms these results.

The sys_kernel application reassigns its OS objects to the same memory location. This makes it impossible for a flow-insensitive analysis like the SIA to retrieve a correct mapping between OS object creation and its usage. Our model, therefore, marks these objects as duplicated and cannot distinguish interactions that lead to them. We plan to extend the SIA in the future to become flow-sensitive for exactly those parts.

C. AUTOSAR

For AUTOSAR, we use the *I4Copter* [25], a safety-critical embedded real-time control system (quadrotor helicopter), as a test application. Due to its static nature, both the SIA and SSE work with AUTOSAR.

Applying the SIA to the *I4Copter* results in 78 interactions. They happen on 30 OS objects which are defined statically in the configuration file that ARA parses in a preprocessing step.

Applying the SSE to the *I4Copter* results in an SSTG with 648 479 states and 2 032 326 transitions. We also ported the SSE unit tests from dOSEK [9] to ARA and manually verified all resulting SSTGs.

[2]https://github.com/grafalex82/GPSLogger, Git commit: `8808b922`

[3]https://www.librepilot.org/, Version 16.0.9

[4]https://www.zephyrproject.org/ Commit: c2a0b0f50b

	FreeRTOS		Zephyr		AUTOSAR	POSIX
	GPSLogger	LibrePilot	app_kernel	sys_kernel	i4copter	libmicrohttpd
Lines of code	79 573	78 787	1603	1206	591	45 322
Number of basic blocks	11 268	19 974	1 152	688	148	41 698
Number of functions	1 311	3 028	212	95	30	2 755
Number of calls	118	919	49	34	0	129
Number of syscalls	37	187	56	91	48	88
Maximal call path depth	16	5	5	3	1	8

TABLE I: Code statistics of the benchmark applications.

D. POSIX

POSIX defines more than a thousand syscalls. A huge part of them (e.g. `strcmp`) does not need the operating system or defines concepts that are unusual in embedded systems (e.g. `fork`). We therefore restricted our POSIX model to the subset of calls[5] that is likely to be employed in an embedded context.

We evaluated the model with libmicrohttpd,[6] an HTTP server library, which is well suited for embedded controllers due to its small memory footprint. To make use of the library, we decided for `fileserver_example_dirs` as an application, which libmicrohttpd includes as an example. We built and analyzed the library in conjunction with the musl libc[7] as implementation of the POSIX standard for the user space.

We found that libmicrohttpd on its own was too dynamic to be useful for ARA. Therefore, we modified libmicrohttpd to reduce its complexity, make the data more static, and focus on the features that are supported by the model. The demo application is runnable on our modified version of libmicrohttpd.

The generated instance graph consists of 4 threads, 1 pipe, 19 files, and 64 mutexes. ARA was able to find 123 interactions. For 46 invocations of interaction syscalls, it fails to determine the belonging OS object due to dynamic calculations or a complicated data flow. While ARA detects nearly all created objects correctly one static mutex is missing since libmicrohttpd `typedef` the standard mutex type which our preprocessing step cannot resolve.

ARA finds 18 different call contexts for `fopen` and one call to `opendir` which it tracks as files. However, this does not reflect all loaded files at runtime since libmicrohttpd opens all files in the current directory, which is inherently dynamic information.

Overall, while we were not able to always find all OS object instantiations and interactions this is not a restriction of the model but of the value analyzer and the analysis algorithms.

V. Discussion

Implementing the model for the four OSs shows the general applicability of the approach, but also uncovers some practical challenges:

Value analysis. The semantics of a concrete syscall is also determined via its arguments. To statically extract their values, we leverage a sophisticated value analyzer based on the Static Value-Flow (SVF) [24] framework, which, however, is still not able to find all value-flows in real-world C/C++ code. Especially constant values that are passed around via (nested) structs appear to be difficult to resolve.

Dynamic object creation. A general problem of static analysis is the possibility to create new system objects at run time. While most real-time applications behave well in this respect in that they create and initialize all system objects before entering the application's main loop, the OS interface does not enforce this. Luckily, the SIA is able to detect this in a reliable manner [13].

The most analysis friendly system in these respects is AUTOSAR, as all system-object instances and also some of their possible interactions (e.g., which task may take which resource) must be declared ahead of time in the configuration file. FreeRTOS, Zephyr, and POSIX are less nicely. While Zephyr, at least, supports static system object definition, POSIX and FreeRTOS basically rely on dynamic object creation only.

Scheduling determinism. For the state change, the scheduling policy is taken into account. Again, AUTOSAR specifies a fixed priority scheduling (partitioned on multi core) which requires no additional information in the state. FreeRTOS allows tasks with the same priority that are scheduled in a round-robin fashion via a timer interrupt which has to be tracked in the state. Zephyr supports multi core but does not really specify how tasks are scheduled on different cores.

Semantically uncompleted syscalls. Especially POSIX defines syscalls that are not conclusive with respect to their effect on the system state. Thread creation is a good example here: To specify the exact behavior of threads, the developer may optionally provide thread attributes, which need to be created and modified by a sequence of syscalls before the actual `pthread_create` call that puts the new thread into existence. This may result in complex dependency chains, defined by the control flow, rendering a flow-insensitive analysis like the SIA impossible. In the future, we plan to extend the SIA to switch to a flow-sensitive analysis for exactly these parts of the code.

OS/library interface ambiguities. The POSIX standard does not distinguish between syscalls (like `read()`) and utility library functions (e. g. `strcmp()`). Both of them are implemented within the same libc. We tackle this by splitting the libc conceptually in syscalls and user functions and analyzing only the former. This can also be implementation-defined, so it might be required to adopt the POSIX model for the concrete OS.

[5] pthread_create, pthread_mutex_init, PTHREAD_MUTEX_INITIALIZER, sem_init, pthread_cond_init, PTHREAD_COND_INITIALIZER, pipe, pause, nanosleep, read, readv, sigaction, open, pthread_join, pthread_detach, pthread_cancel, pthread_mutex_lock, pthread_mutex_unlock, sem_wait, sem_post, pthread_cond_wait, pthread_cond_signal, pthread_cond_broadcast, write, writev

[6] https://www.gnu.org/software/libmicrohttpd/ v0.9.73, Commit: 64e91ef6

[7] http://musl.libc.org/

Overall, our findings show that ARA is able to extract OS-interaction knowledge regardless of the specific OS. The remaining implementation challenges mostly result from idiomatic anachronisms of the respective OS interfaces and programming models (e.g., POSIX, FreeRTOS) and underspecification of its semantics (e.g., multi-core scheduling), which naturally limits static analysis. Note, however, that ARA still behaves sound in these cases, even though it yields less tight analysis results.

VI. Related Work

To the best of our knowledge, no other compiler exists that tailors OSs while supporting multiple of them. However, there do exist other usages of operating system models and compilers for different purposes.

First, dOSEK [16] is a whole system compiler and able to tailor applications written for the OSEK OS standard, the predecessor of AUTOSAR. As part of its implementation, it contains an abstract OSEK interpreter. In contrast to the OS model of ARA, this interpreter is for OSEK only which includes the restriction to single-core applications. However, ARA is influenced by dOSEK and shares concepts and code with it.

SWAN [22] is a whole-system WCET analyzer that also builds an SSTG to calculate tighter bounds for a better WCET. While SWAN also operates on FreeRTOS, it does not use a model of it but analyzes the internal syscall implementation to calculate a tighter WCET bound.

With the RTSC, a whole system compiler exists that was written to automatically convert event-driven real-time systems (written for OSEK) into time-driven real-time systems (written for OSEKTime) [20]. As an extension, it supports the mapping on multi core and into a POSIX program [14]. The RTSC, therefore, uses a system model for OSEKTime and POSIX but aims for system generation only.

Another common use of operating system models is formal verification or conformance checking. Brekling et. al. define a precise operating system specification based on timed automata [3]. With the help of UPAAL, the automata can both be converted to running code and theoretically verified. The model here, however, can be seen as another OS implementation and does not try to unify and abstract existing OSs.

For the OSEK and AUTOSAR standard, several works exist to convert the specification into a formal model to verify the system like checking for schedulability or generating conformance tests [4], [17], [27], [28], [26], [10], [11]. Similar works try to formalize parts of FreeRTOS [5], [19], [6] or Zephyr [12]. All of these works use models that are not used to optimize the system but proof the real-time capabilities. Furthermore, in contrast to ARA, no unifying model is formulated.

VII. Conclusion

In this paper, we presented a generic interface to model OS behavior for usage in static analysis. Thereby, we were able to reduce the constraints for an OS to a minimum: The system must communicate with a syscall interface.

We implemented the interface for the four OSs FreeRTOS, AUTOSAR, Zephyr, and POSIX and evaluated all targets with at least one test application. The results prove the working of the model. While experiencing limitations, we can assign them either to the analysis algorithm or inaccuracies in the value analysis, not the OS model itself. We discussed the characteristics of different OSs and their potential for optimization.

References

[1] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. TIMES: A tool for schedulability analysis and code generation of real-time systems. In *Formal Modeling and Analysis of Timed Systems*. Springer Berlin Heidelberg, 2004.

[2] Ramon Bertran, Marisa Gil, Javier Cabezas, Victor Jimenez, Lluis Vilanova, Enric Morancho, and Nacho Navarro. Building a global system view for optimization purposes. In *2nd Work. on the Interaction between Operating Systems and Computer Architecture (WIOSCA '06)*. IEEE Computer Society Press, 2006.

[3] Aske Brekling, Michael R. Hansen, and Jan Madsen. Models and formal verification of multiprocessor system-on-chips. *The Journal of Logic and Algebraic Programming*, 77(1), 2008. The 16th Nordic Work. on the Prgramming Theory (NWPT 2006).

[4] Jiang Chen and Toshiaki Aoki. Conformance testing for OSEK/VDX operating system using model checking. In *18th Asia-Pacific Software Engineering Conf. (APSEC 2011)*. IEEE Computer Society Press, 2011.

[5] Nathan Chong and Bart Jacobs. Formally verifying FreeRTOS' interprocess communication mechanism. In *Embedded World Exhibition and Conf.*, 2021.

[6] David Déharbe, Stephenson Galvao, and Anamaria Martins Moreira. Formalizing FreeRTOS: First steps. In *Brazilian Symp. on Formal Methods*. Springer, 2009.

[7] Christian Dietrich, Martin Hoffmann, and Daniel Lohmann. Cross-kernel control-flow-graph analysis for event-driven real-time systems. In *2015 ACM SIGPLAN/SIGBED Conf. on Languages, Compilers and Tools for Embedded Systems (LCTES '15)*. ACM Press, 2015.

[8] Christian Dietrich, Martin Hoffmann, and Daniel Lohmann. Global optimization of fixed-priority real-time systems by RTOS-aware control-flow analysis. *ACM Trans. on Embedded Computing Systems*, 16(2), 2017.

[9] Christian Dietrich and Daniel Lohmann. OSEK-V: Application-specific RTOS instantiation in hardware. In *2017 ACM SIGPLAN/SIGBED Conf. on Languages, Compilers and Tools for Embedded Systems (LCTES '17)*. ACM Press, 2017.

[10] Timothee Durand, Katalin Fazekas, Georg Weissenbacher, and Jakob Zwirchmayr. Model checking AUTOSAR components with CBMC. In *2021 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2021.

[11] Ling Fang, Takashi Kitamura, Thi Bich Ngoc Do, and Hitoshi Ohsaki. Formal model-based test for AUTOSAR multicore RTOS. In *2012 IEEE Fifth Intl. Conf. on Software Testing, Verification and Validation*. IEEE, 2012.

[12] Zhang Feng, Zhao Yongwang, Ma Dianfu, and Niu Wensheng. Fine-grained formal specification and analysis of buddy memory allocation in Zephyr RTOS. In *2019 IEEE 22nd Intl. Symp. on Real-Time Distributed Computing (ISORC)*, 2019.

[13] Björn Fiedler, Gerion Entrup, Christian Dietrich, and Daniel Lohmann. ARA: Static initialization of dynamically-created system objects. In *27th IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS'21)*, 2021.

[14] Florian Franzmann, Tobias Klaus, Peter Ulbrich, Patrick Deinhardt, Benjamin Steffes, Fabian Scheler, and Wolfgang Schröder-Preikschat. From intent to effect: Tool-based generation of time-triggered real-time systems on multi-core processors. In *19th IEEE Intl. Symp. on Object-Oriented Real-Time Distributed Computing (ISORC '16)*. IEEE Computer Society Press, 2016.

[15] T. Glek and Jan Hubicka. Optimizing real world applications with GCC link time optimization. *CoRR*, abs/1010.2196, 2010.

[16] Martin Hoffmann, Christian Dietrich, and Daniel Lohmann. dOSEK: A dependable RTOS for automotive applications. In *19th Intl. Symp. on Dependable Computing (PRDC '13)*. IEEE Computer Society Press, 2013. Fast abstract.

[17] Yanhong Huang, Yongxin Zhao, Longfei Zhu, Qin Li, Huibiao Zhu, and Jianqi Shi. Modeling and verifying the code-level osek/vdx operating system with csp. In *5th Intl. Symp. on Theoretical Aspects of Software Engineering (TASE'11)*. IEEE Computer Society Press, 2011.

[18] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. Bolt: a practical binary optimizer for data centers and beyond. In *2019 IEEE/ACM Intl. Symp. on Code Generation and Optimization (CGO)*. IEEE, 2019.

[19] David Sanán, Liu Yang, Zhao Yongwang, Xing Zhenchang, and Mike Hinchey. Verifying FreeRTOS' cyclic doubly linked list implementation: From abstract specification to machine code. In *2015 20th Intl. Conf. on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 2015.

[20] Fabian Scheler and Wolfgang Schröder-Preikschat. The RTSC: Leveraging the migration from event-triggered to time-triggered systems. In *13th IEEE Intl. Symp. on Object-Oriented Real-Time Distributed Computing (ISORC '10)*. IEEE Computer Society Press, 2010.

[21] Horst Schirmeier, Matthias Bahne, Jochen Streicher, and Olaf Spinczyk. Towards eCos autoconfiguration by static application analysis. In *1st Intl. Work. on Automated Configuration and Tailoring of Applications (ACoTA '10)*, CEUR Work. Proceedings. CEUR-WS.org, 2010.

[22] Simon Schuster, Peter Wägemann, Peter Ulbrich, and Wolfgang Schröder-Preikschat. Proving real-time capability of generic operating systems by system-aware timing analysis. In *2019 IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS)*, 2019.

[23] Amitabh Srivastava and David W. Wall. Link-time optimization of address calculation on a 64-bit architecture. 29(6), 1994.

[24] Yulei Sui and Jingling Xue. SVF: Interprocedural static value-flow analysis in LLVM. In *25th Intl. Conf. on Compiler Construction*, CC 2016. Association for Computing Machinery, 2016.

[25] Peter Ulbrich, Rüdiger Kapitza, Christian Harkort, Reiner Schmid, and Wolfgang Schröder-Preikschat. I4Copter: An adaptable and modular quadrotor platform. In *26th ACM Symp. on Applied Computing (SAC '11)*. ACM Press, 2011.

[26] Dieu-Huong Vu, Yuki Chiba, Kenro Yatake, and Toshiaki Aoki. Verifying OSEK/VDX OS design using its formal specification. In *Proc. TASE'16*. IEEE Computer Society, 2016.

[27] Haitao Zhang, Toshiaki Aoki, and Yuki Chiba. Yes! you can use your model checker to verify OSEK/VDX applications. In *8th IEEE Intl. Conf. on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*, 2015.

[28] Min Zhang, Yunja Choi, and Kazuhiro Ogata. A formal semantics of the OSEK/VDX standard in K framework and its applications. In *Proc. WRLA'14*. Springer, 2014.

[29] Ming-Yuan Zhu, Lei Luo, and Guang-Ze Xiong. The minimal model of operating systems. *SIGOPS Oper. Syst. Rev.*, 2001.

Supporting Multiprocessor Resource Synchronization Protocols in RTEMS

Junjie Shi*, Jan Duy Thien Pham*, Malte Münch*, Jan Viktor Hafemeister*, Jian-Jia Chen* and Kuan-Hsun Chen†
*Department of Computer Science, Technische Universitat Dortmund, Germany
†Department of Electrical Engineering, Mathematics and Computer Science, University of Twente, the Netherlands
E-mail: {junjie.shi, jan.pham, malte.muench, jan.hafemeister, jian-jia.chen}@tu-dortmund.de, k.h.chen@utwente.nl

***Abstract*—When considering recurrent tasks in real-time systems, concurrent access to shared resources can cause race conditions or data corruption. Such a problem has been extensively studied since the 1990s, and numerous resource synchronization protocols have been developed for both uni-processor and multiprocessor real-time systems, with the assumption that the operating overheads are negligible. However, such overheads may also impact the performance of different protocols depending on the practical implementation, e.g., resources are accessed locally or remotely, and tasks spin or suspend themselves when the requested resources are not available. In this paper, to show the applicability of different protocols in real-world systems, we detail the implementation of several state-of-the-art multiprocessor resource synchronization protocols in RTEMS. To study the impact of the implementation overheads, we deploy these implemented protocols on a real platform with synthetic task sets. The measured results illustrate that the developed resource synchronization protocols in RTEMS are comparable to the officially supported protocol, i.e., MrsP.**

I. Introduction

In multi-tasking real-time systems, the accesses to shared resources, e.g., file, memory cell, etc., are mutually exclusive, to prevent race conditions or data corruptions. A code segment that a task accesses to the shared resource(s) is called a *critical section*, which is protected by using binary semaphores or mutex locks. That is, a task must finish its execution of the critical section before another task can access the same resource. However, the mutually exclusive executions of critical sections may cause other problems, i.e., priority inversion and deadlock, which could jeopardize the predictability of the real-time system. In order to guarantee the timeliness of a real-time system, a lot of resource synchronization protocols have been developed and analyzed since 1990s for both uni-processor and multiprocessor real-time systems.

In uni-processor real-time systems, the Priority Inheritance Protocol (PIP) and the Priority Ceiling Protocol (PCP) by Sha et al. [20], as well as the Stack Resource Policy (SRP) by Baker [4] have been widely studied. Since PIP may potentially lead to a deadlock requiring additional verification to avoid [13], PCP has been relatively common and its performance has been widely accepted. Specifically, a variant of PCP has been implemented in Ada (named Ceiling locking) and in POSIX (named Priority Protect Protocol).

Because of the increasing demand of computational power of real-time systems, multiprocessor platforms have been widely used. A lot of multiprocessor resource synchronization protocols have been proposed and extensively studied in the domain, such as the Distributed Priority Ceiling Protocol (DPCP) [19], the Multiprocessor Priority Ceiling Protocol (MPCP) [18], the Multiprocessor Stack Resource Policy (MSRP) [14], the Flexible Multiprocessor Locking Protocol (FMLP) [5], the $O(m)$ Locking Protocol (OMLP) [7], the Multiprocessor Bandwidth Inheritance (M-BWI) [12], gEDF-vpr [2], LP-EE-vpr [3], the Multiprocessor resource sharing Protocol (MrsP) [8], the Resource-Oriented Partitioned PCP (ROP-PCP) [17], the Dependency Graph Approach (DGA) for frame-based task set [11], and its extension for periodic task set (HDGA) [25].

Although the protocols above provide the timing guarantees by bounding the worst-case response time of tasks, most of them rely on the assumption that the overheads invoked by the implementation are negligible. However, rethinking of the assumption is in fact needed. Depending on their settings, e.g., local or remote execution of critical sections, multiprocessor scheduling paradigm, and the tasks' waiting semantics, the performance of different protocols is highly relevant to the implementation. For example, under a suspension-based synchronization protocol, tasks that are waiting for access to a shared resource (i.e., the resource is locked by another task) are suspended. This strategy frees the processor so that it can be used by other ready tasks, which exploits the utilization of processor, but also increases the context switch overhead due to extra en-queue and de-queue operations for each suspension. In contract, under a spin-based synchronization protocol, the task does not give up its privilege on the processor and has to wait by spinning on the processor until it can access the requested resource and starts its critical section, which is efficient when the critical sections are short [16].

In fact, there are only a few of the protocols have been officially supported, and there are two real-time operating systems popular in the domain: the Linux Testbed for Multiprocessor Scheduling in Real-Time Systems (LITMUS^{RT}) [9], and Real-Time Executive for Multiprocessor Systems (RTEMS) [1]. LITMUS^{RT}is an experimental platform for timing analysis mainly for academic usages. Brandenburg et al. implemented DPCP, MPCP, and FMLP [6], Catellani et al. implemented MrsP [10], and Shi et al. solidate the implementation of MrsP [23]. In addition, the recently developed DGA and its

extension for periodic tasks HDGA have been implemented by Shi et al. in [21], [22]. Alternatively, RTEMS is an open-source real-time operating system which is popular for industrial applications. RTEMS has been widely used in many fields, e.g., space flight, medical, networking, etc. However, in RTEMS, only MrsP implemented by Catellani et al. in [10], is officially supported in the upstream repository.

Therefore, we believe it is beneficial to provide comprehensively support on RTEMS with resource synchronization protocols for the related researches. Afterwards, the performance of resource synchronization protocols might be clarified by system designers, and the optimizations of implementation can also be discussed. In this work, we focus on the resource synchronization protocols which are based on (semi-) partitioned scheduling, detailed as follows:

- **Partitioned Schedule:** Each task is assigned on a dedicated processor, each processor maintains its own ready queue and scheduler. Tasks are not allowed to migrate among processors, e.g., MPCP.
- **Semi-partitioned Schedule:** Unlike the pure partitioned schedule, semi-partitioned schedule allows tasks to migrate to other processors under certain conditions. For example, in DPCP and ROP-PCP, shared resources are assigned on processors, the critical sections have to be executed on the corresponding processors, where may not be the same as the original partition of a task.

Our Contribution in a nutshell: We enhance the RTEMS with the aforementioned multiprocessor resource synchronization protocols and discuss how to revise the kernel with RTEMS Symmetric Multiprocessing (SMP) support.

- To harden the open source development, we review the SMP support of RTEMS and point out the potential pitfalls during the implementation, so that the insights can be reused on any other platforms (see Section III).
- We detail the development of three multiprocessor resource synchronization protocols, i.e., MPCP, DPCP, and FMLP, and their variants in RTEMS (see Section IV).
- To study the impact of the implementation overheads, we deploy our implementations on a real platform with synthetic task sets (see Section V). The measured overheads show that our implementation overheads are comparable to the existed implementation of MrsP, in RTEMS, which illustrates the applicability of our implementations.

The patches have been released under MIT license in [24] for RTEMS 4.12. Please note that this release branch was planned to be the latest release, but significant changes warranted to bump the major number from 4 to 5. To apply our patches to RTEMS 5, a certain adaption is additionally needed.

II. System Model

We consider a task set $\mathbf{T}$ consists of n recurrent tasks to be scheduled on M symmetric and identical (homogeneous) processors. All tasks can have multiple (non-nested) critical sections, each critical section accesses one of the Z shared resources, denoted as s_z Each task τ_i is described by a tuple $(C_i,\ \mu_i,\ T_i,\ D_i,\ q_i)$, where:

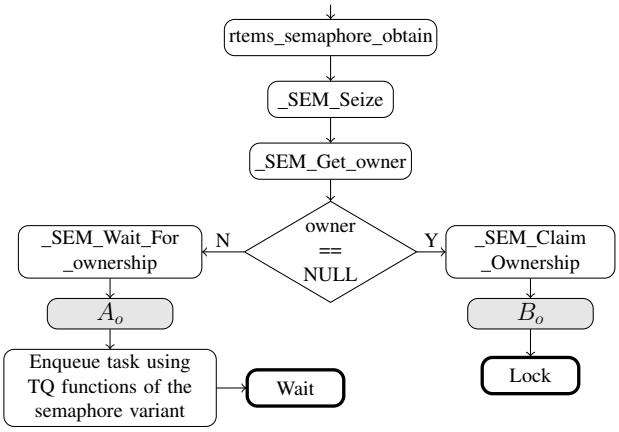


Fig. 1. Workflow of the lock directive. Block A_o and B_o are specified according to the adopted protocols.

- C_i is the worst-case execution time (WCET) of task τ_i, i.e., $C_i > 0$.
- μ_i is the set of resource(s) that τ_i requests.
- T_i is the period of task τ_i, i.e., $T_i > 0$.
- D_i is the relative deadline of the task τ_i. To fulfill its timing requirements a job of τ_i released at time t must finish its execution before its absolute deadline $t+D_i$. We consider constrained-deadline task systems, i.e., $D_i \leq T_i$ for every task $\tau_i \in \mathbf{T}$.
- q_i is the priority of task τ_i.

III. Symmetric Multiprocessing Support in RTEMS

RTEMS allows users to implement new resource synchronization protocols by strictly following the RTEMS API. To create a new semaphore, `SEM_Initialize` function is called to define the specified attributes for each resource synchronization protocol. Besides the creation of semaphore, which is defined by different protocols, some common components that are similar for all the protocols, i.e., lock and unlock directives, configuration for applications, and migration mechanism, are introduced in this section.

A. Lock and Unlock Directives

The workflow of the lock directive is shown in Fig. 1. Once a task τ_i requests a shared resource, it will try to lock the corresponding semaphore. After selecting the right semaphore, denoted as `SEM`, τ_i calls the `_SEM_Seize` function. Then, the ownership of the semaphore is checked by getting the owner of the Thread queue Control. If the semaphore is locked by another task, τ_i has to wait for the owner to release the semaphore. The detailed operations in block A_o are specified according to the design of different protocols. If there is no owner yet, τ_i is set as the owner of the semaphore, and starts the execution of its critical section. The operations in block B_o can be different depending on the specified design of protocols.

The workflow of the unlock directive is shown in Fig. 2. It will be called when task τ_i has finished the execution of its critical section and releases the lock of the semaphore. The unlock directive selects the right `_SEM_Surrender` function

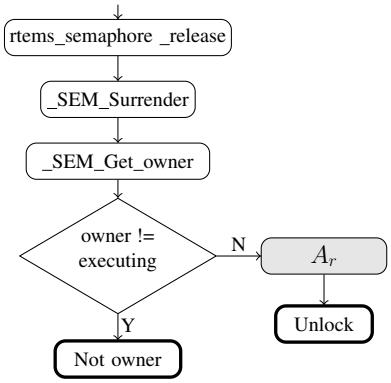


Fig. 2. Workflow of the unlock directive. Block A_r is specified according to the adopted protocols.

to check whether the τ_i is the current owner of the semaphore. If τ_i is not the owner, the semaphore cannot be unlocked. Otherwise, τ_i can unlock the semaphore by executing the commands in block A_r. The main function in A_r is to find the next owner for the semaphore if (at least) one task that is waiting for the semaphore. If there is no waiting task, the owner will be set to `NULL` accordingly. The details of the functions in A_r will be discussed in the corresponding sections for different protocols.

B. Application Configuration

In order to support semi-partitioned schedule in RTEMS, the flow for configuration in Fig. 3 has to be followed. Firstly, processors have to be bound to specific scheduler instances by using macro `_RTEMS_SCHEDULER_ASSIGN` supported in RTEMS by default. After that, each task is partitioned to a scheduler instance by using the `rtems_task_set_scheduler` directive. Each task can only be executed on the processor of the corresponding scheduler instance.

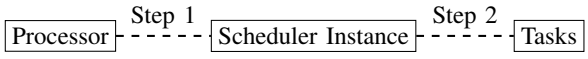


Fig. 3. The steps to configure

When a RTEMS application is configured with SMP support by following the work flow in Fig. 3, some new functions have to be implemented. In Step 1, an initial task has to be defined, which is executed in the beginning of the RTEMS application. The binding of scheduler instances to processor is based of the guide in the official c-user guide. The dedicated schedule algorithm for the scheduler instances has to be selected at first. In this paper, the *Deterministic Priority SMP Scheduler* supported in RTEMS by default is selected for all the protocols, which is the same as Fixed-Priority (FP) scheduler in the literature. Please note that, the instances have to be defined for all the available processors in the system, in order to support the semi-partitioned schedule, i.e., tasks may migrate to other processors by changing their scheduler nodes, details can be found in next subsection.

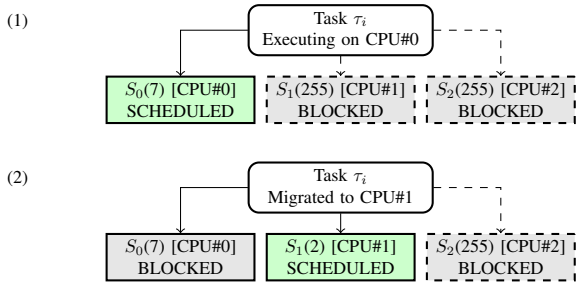


Fig. 4. Scheduler Node management: (1) Before migration, (2) After migration. Dashed blocks and lines represent that τ_i has no access to the respective scheduler instances, whereas green block is the currently used one.

C. Migration Mechanism

The migration mechanism by using arbitrary processor affinity in [15] is not supported in the current version of RTEMS. Therefore, a new migration mechanism has to be applied for those distributed-based protocols, e.g., DPCP. In our implementation, the scheduler node is modified during the run time in order to realize the task migration. When a task needs to migrate to another processor, the scheduler node of the task in its original scheduler instance is blocked, and the scheduler node of the task in its destination processor is unblocked. An additional function named `_Scheduler_Migrate_To` is implemented in `schedulerimpl.h`, which contains the task information block, the target processor, and the priority of the task in the target processor. In addition, in order to guarantee the correctness of the migration, `thread-dispatch` is disabled during the migration operation.

Fig. 4 demonstrates an example of the implemented task migration. In Fig. 4 (1), task τ_i has a scheduler node for every scheduler instance in the system. τ_i is currently executing on CPU#0 with a priority of 7 by using scheduler node S_0, which is indicated by the the node with green background. Other two nodes with grey background are blocked, since τ_i has no access their respective scheduler instances, denoted as dashed line. In Fig. 4 (2), task τ_i performs migration to CPU#1. τ_i blocks itself on its original scheduler by using the block function of the scheduler instance on S_0. After that, it adds S_1 to the list of its active scheduler nodes and modifies the priority of S_1 accordingly. It unblocks S_1 by using the unblock function of the corresponding scheduler instance. Migrating back to the original processor works similarly, i.e., Fig. 4 (1) is restored by using the same unblock/block function of the scheduler instances.

IV. Multiprocessor Resource Synchronization

In this section, the implementation details of three protocols and corresponding variants are explained and discussed. Please note, we only consider non-nested resource accesses in our implementation, i.e., only one shared resource is requested during the execution of one critical section.

A. Multiprocessor Priority Ceiling Protocol

The Multiprocessor Priority Ceiling Protocol (MPCP) is a typical protocol that is based on a partitioned fixed priority (P-

Algorithm 1 MPCP implementation

Input: Task τ_i, and `ceiling_priority` of related semaphore;

Function mpcp_lock():
1: **if** `semaphore_owner` is NULL **then**
2: `semaphore_owner` $\leftarrow \tau_i$;
3: $\tau_i.priority \leftarrow$ `ceiling_priority`;
4: τ_i starts the execution of its critical section;
5: **else**
6: Add τ_i to the corresponding `wait_queue`;
7: **end if**

Function mpcp_unlock():
8: τ_i releases the semaphore lock;
9: Next task $\tau_{next} \leftarrow$ the head of the `wait_queue`;
10: **if** τ_{next} is NULL **then**
11: `semaphore_owner` $\leftarrow$ NULL;
12: **else**
13: `semaphore_owner` $\leftarrow \tau_{next}$;
14: τ_{next} starts the execution of its critical section;
15: **end if**

FP) scheduler. That is, each task has a pre-defined priority, and the execution of a task is bound on a pre-defined processor, i.e., no migration is allowed. The main features of MPCP are: 1) a task will suspend itself if the resource is not available. 2) if a task is granted to access a shared resource, the priority of the task will be boosted to the ceiling priority, which equals to the highest priority of these tasks that request that resource.

The self-suspension feature is supported in RTEMS by default. In order to implement the ceiling priority boosting, one new semaphore structure is created. Besides these normal components, e.g., semaphore lock, wait queue, and current semaphore owner, one variable named `ceiling_priority` is added. Please note that, in our implementation the ceiling priority is defined by users instead of being calculated by the system dynamically. The pseudo code provided in Algo. 1 shows two main functions in our implementation, which fits the lock and unlock directive in Section III-A. The details are as follows: Once a task τ_i requests a shared resource, the ownership of the shared resource (semaphore) will be checked. If the owner of the requested shared resource is `NULL`, τ_i becomes the owner, and the priority of τ_i is boosted to the ceiling priority on the corresponding scheduler instance (operations in block B_o in Fig. 1). Otherwise, τ_i will be added into a wait queue, which is sorted by tasks' original priorities, i.e., task with higher priority will get earlier position (operations in block A_o in Fig. 1). Once the task τ_i finishes the execution of critical section, it will release the semaphore. The first task of the wait queue is checked, i.e., the task with the highest priority in the wait queue. If there is no task in the wait queue, the semaphore owner will be set to `NULL`. Otherwise, the first task of the wait queue will be set as the semaphore owner (operations in block A_r in Fig. 2).

B. Distributed Priority Ceiling Protocol

The Distributed Priority Ceiling Protocol (DPCP) is based on semi-partitioned fixed priority schedule. In DPCP, tasks and shared resources are assigned on different processors separately, i.e., these processors that are assigned for the execution of non-critical sections are called application processors, and processors for the execution of critical sections are called synchronization processors. Once a task τ_i tries to access a shared resource, it will migrate to the corresponding synchronization processor where the shared resource is assigned on, before trying to lock the corresponding semaphore. Afterwards, these tasks on the same synchronization processor operate follow the uni-processor PCP, which been supported in RTEMS by default, i.e., Immediate Ceiling Priority Protocol (ICPP). When a task τ_i finished its execution of critical section, it will migrate back to the original application processor to continue the execution of its non-critical section, if it exists.

Hence, the main challenge of the implementation of DPCP is to allow task migrations among processors. In RTEMS, task partitioning is realized by the scheduler node in the scheduler function, i.e., scheduler node defines the original partition for each task before the execution, and stays the same during the run time. Details have been explained in Section III-C.

C. Flexible Multiprocessor Locking Protocol

In Flexible Multiprocessor Locking Protocol (FMLP), requests of shared resources are divided into two groups, i.e., long and short, according to the length of the execution time of corresponding critical section. When the requested resource is not available, a task will suspend itself if it is a long request, and a task will spin on the correspond processor if it is a short request. However, there is no conclusion regarding to how to divide requests to obtain a better schedulability. Therefore, we divided our implementation into FMLP-L which only supports long requests, and FMLP-S which only supports short requests. Please note, to simplify the implementation, all the tasks in one task set all belong to either long group or short group, no mixed division of these two groups is allowed.

In both FMLP-L and FMLP-S, the wait queue in the semaphore structure is in a FIFO order, rather than sorting by priorities like MPCP and DPCP. The operations in block B_o in Fig. 1 are as follows: In FMLP-L, we maintain a ceiling priority dynamically for each resource, which equals to the highest priority of these tasks that are currently waiting for the resource, i.e., tasks in the corresponding wait queue. The priority of the semaphore owner will be boosted to the ceiling priority if the original priority is lower than the ceiling priority, when it starts the execution of its critical section. In FMLP-S, the owner of the semaphore gets priority boosted to the highest possible priority in the system, so that the execution of its critical section is the non-preemptive. The operations in block A_o in Fig. 1 are the same for both FMLP-L and FMLP-S, i.e., add task τ_i in the end of the corresponding wait queue. The unlock operations in block A_r in Fig. 2 are also the same, i.e., try to find the next owner for the semaphore by checking the first task in the wait queue, if it exists.

Additionally, we implemented a distributed version of FMLP, denoted as DFLP, where all the requests are treated as long requests. The main difference between FMLP and DFLP

TABLE I
PROCESSOR ALLOCATION OF THE TEST APPLICATION.

CPU#0 Application	CPU#1 Application	CPU#2 Application	CPU#3 Synchronization
L (s_1)	L (s_2)	L (s_3)	-
ML (s_2)	ML (s_3)	ML (s_1)	-
M (s_3)	M (s_1)	M (s_2)	-
MH (s_2)	MH (s_3)	MH (s_1)	-
H (s_3)	H (s_1)	H (s_2)	-

is when a task requests a shared resource, it will migrate to the corresponding synchronization processor, which is similar to DPCP. The mechanism how we implement the migration has been explained in Section III-C. After the migration, critical sections are executed by following the FMLP-L on the corresponding synchronization processor(s).

V. EVALUATION AND DISCUSSION

In this section, we introduce the setup of experiments for overheads evaluation at first. Afterwards, the measured overheads are reported and analyzed. At the end, we discuss the need of formal verification over the implementation generally.

A. Experimental Setup

We evaluated the overheads of our implementations on the following platform: a NXP QorIQ T4240 RDB reference design board, which is the same as used in [10]. It has 6 GB DDR3 memory with 1866 MT/s data rate, 128 MB NOR flash(16-bit), and 2 GB SLC NAND flash. The processor T4240 contains 24-virtual-core (12 physical cores) with the PowerPC Architecture, and is running on 1.67 GHz.

To measure the overheads of our implemented protocols, timestamps are added before and after the function of our implementations. The obtain and release functions of the semaphore are measured, denoted as lock and unlock respectively. We consider a multi-processor system consists of four processors, i.e., $M = 4$, including three application processors and one synchronization processor. The total number of tasks $n = 15$, and the number of available shared resources $Z = 3$, i.e., $\mu_i \in \{s_1, s_2, s_3\}$. On each application processor, there are five tasks with five different priority levels, i.e., $q_i \in \{$High (H), Medium-High (MH), Medium (M), Medium-Low (ML) and Low (L)$\}$. Each task requests one of these three shared resources. Details can be found in Table I.

B. Overheads Evaluation

The overheads for different protocols are reported in Figure 5, based on more than 9,000 instances of lock and unlock operations. These distributed-based protocols, i.e., DPCP and DFLP have higher overheads than others, due to the task migrations. DFLP has the highest average overheads, since it also maintains the dynamic ceiling priority update. MrsP also has relative high overheads, since it has the help mechanism requiring task migration (however, help mechanism may not be activated all the time). Our results related to MrsP are similar as reported in [10], i.e., 5376 ns for lock and 5514 ns for unlock on average. FMLP-L has the lowest overheads, due to the simplest mechanism. Overall, the overheads for all the protocols are relatively low and acceptable. For distributed-based protocols, we can observe that there are quite a few outliers. In fact, a similar observation has been reported in [23]. One reason could be that the behavior of cache memories kicks in to introduced operation overheads, but we have no sufficient data to pinpoint the exact cause here.

The migration overheads are measured separately, and reported in the left side of Figure 5. The results show that the overheads of task migration are significant, which might substantially affect those distributed-base protocols, i.e., DPCP and DFLP. Interestingly, we also notice that the overhead of a task to migrate to the synchronization processor is faster than migrating back to the application processor. The reason is that, normally there are more tasks running on the application processors than synchronization processors, which causes a task has to wait for longer time to obtain the scheduler instance lock on average. That is why the unlock overheads of DPCP and DFLP are higher than the lock overheads.

Although our evaluated overheads on RTEMS are similar to these protocols that are implemented on LITMUS^{RT} [10], [25], implementations of protocols on RTEMS and LITMUS^{RT}are not directly comparable due to the difference of purposes and architectures in two operating systems. Please note that RTEMS is a self-contained RTOS for real-world applications, whilst LITMUS^{RT}is a Linux-based testbed, which is mainly used for functional validation. It might be interesting to investigate which protocol is preferable on which operating systems, but it is considered out of scope here.

C. Validation and Formal Verification

To validate the correctness of our implementation, at first we test over the official coverage tests provided by RTEMS, i.e., the SMP test suites (https://github.com/RTEMS/rtems/tree/master/testsuites/smptests) especially, on the PowerPC device and also the QEMU emulator for ARM RealView Platform `realview-pbx-a9`, and conclude that the SMP related peripheries in RTEMS are not affected at all. Moreover, we further design several dedicated corner cases for each protocol and ensure that the designated tasks execute as the expected behaviors, which are treated as the additional coverage test for the future integration.

We note that such case-based validation may not be sufficient, since it is not possible to test over every case exhaustively. One possible way is to adopt software model checkers as proposed in [13] to detect potential data races and deadlocks in the implementation of PIP with nested locks in RTEMS. However, such searching approaches may not scale well for multiprocessor protocols unless an effective pruning strategy can be found beforehand. How to validate or formally verify an existing implementation of synchronization protocols is still an unsolved problem but out of the scope.

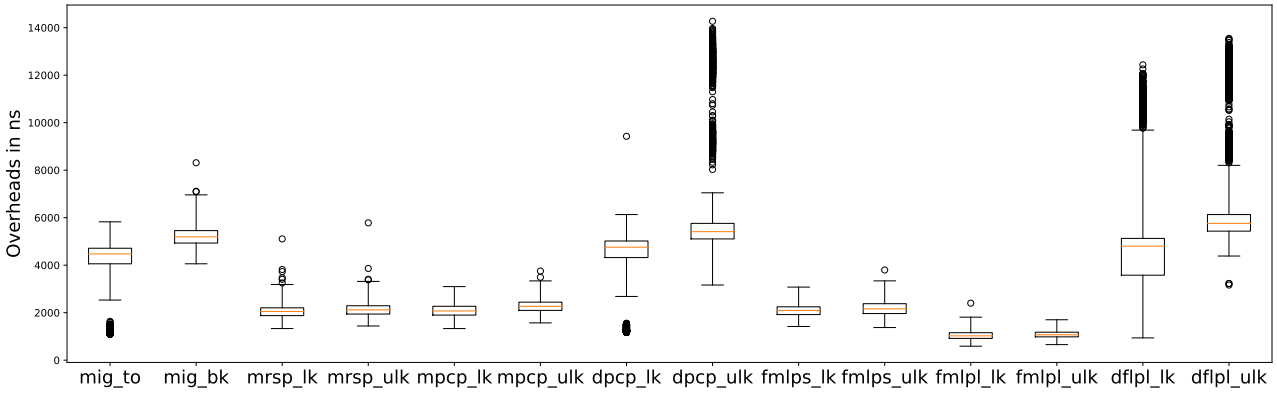


Fig. 5. Overheads of protocols in RTEMS (lock operation is ended by `_lk` and unlock operation is ended by `_ulkl`)). The measurement of migrating a task to the synchronization processor (denoted as `mig_to`) and back to the application processor (denoted as `mig_bk`).

VI. Conclusion

Over the decades, quite a few number of resource synchronization protocols have been extensively studied for uniprocessor and especially multiprocessor real-time systems. In this work, we reviewed the SMP support in one popular real-time operating system RTEMS and detailed how we develop three state-of-the-art multiprocessor resource synchronization protocols, i.e., MPCP, DPCP, and FMLP, and their variants. With extensive synthetic experiments, the measured results showed that our implementations are comparable to MrsP, which is officially supported in RTEMS. Considering the real system overhead, the performance of resource synchronization protocols might be clarified and decidable by system designers.

Although several dedicated tests are provided to verify the correctness of the implementation, formal model checking is still desirable to prevent the system from potential deadlock, data races, and priority inversions. In the future work, we plan to explore on nested resource synchronization and support the arbitrary processor affinity in RTEMS to improve the generality and the efficiency. An ongoing effort is also provided to support for the latest version of RTEMS.

Acknowledgement

This paper is supported by DFG, as part of the Collaborative Research Center SFB876, subproject A1 and A3 (http://sfb876.tu-dortmund.de/).

References

[1] RTEMs. http://www.rtems.org/.

[2] B. Andersson and A. Easwaran. Provably good multiprocessor scheduling with resource sharing. *Real-Time Systems*, 46(2):153–159, 2010.

[3] B. Andersson and G. Raravi. Real-time scheduling with resource sharing on heterogeneous multiprocessors. *Real-Time Systems*, 50(2):270–314, 2014.

[4] T. P. Baker. Stack-based scheduling of realtime processes. *Real-Time Systems*, 3(1):67–99, 1991.

[5] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *RTCSA*, pages 47–56, 2007.

[6] B. B. Brandenburg and J. H. Anderson. An implementation of the pcp, srp, d-pcp, m-pcp, and FMLP real-time synchronization protocols in litmus$^{\text{rt}}$. In *RTCSA*, pages 185–194, 2008.

[7] B. B. Brandenburg and J. H. Anderson. Optimality results for multiprocessor real-time locking. In *RTSS*, pages 49–60, 2010.

[8] A. Burns and A. J. Wellings. A schedulability compatible multiprocessor resource sharing protocol - MrsP. In *ECRTS*, pages 282–291, 2013.

[9] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson. LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers. In *RTSS*, pages 111–126, 2006.

[10] S. Catellani, L. Bonato, S. Huber, and E. Mezzetti. Challenges in the implementation of mrsp. In *Ada-Europe*, pages 179–195, 2015.

[11] J.-J. Chen, G. von der Brüggen, J. Shi, and N. Ueter. Dependency graph approach for multiprocessor real-time synchronization. In *IEEE Real-Time Systems Symposium, RTSS*, pages 434–446, 2018.

[12] D. Faggioli, G. Lipari, and T. Cucinotta. The multiprocessor bandwidth inheritance protocol. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 90–99, 2010.

[13] S. Gadia, C. Artho, and G. Bloom. Verifying nested lock priority inheritance in RTEMS with java pathfinder. In K. Ogata, M. Lawford, and S. Liu, editors, *ICFEM*, pages 417–432, 2016.

[14] P. Gai, G. Lipari, and M. D. Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *Real-Time Systems Symposium (RTSS)*, pages 73–83, 2001.

[15] A. Gujarati, F. Cerqueira, and B. B. Brandenburg. Multiprocessor real-time scheduling with arbitrary processor affinities: from practice to theory. *Real-Time Systems*, 51(4):440–483, 2015.

[16] G. Han, H. Zeng, M. Natale, X. Liu, and W. Dou. Experimental evaluation and selection of data consistency mechanisms for hard real-time applications on multicore platforms. *IEEE Transactions on Industrial Informatics*, 10(2):903–918, 2014.

[17] W.-H. Huang, M. Yang, and J.-J. Chen. Resource-oriented partitioned scheduling in multiprocessor systems: How to partition and how to share? In *Real-Time Systems Symposium (RTSS)*, pages 111–122, 2016.

[18] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *Proceedings.,10th International Conference on Distributed Computing Systems*, pages 116 – 123, 1990.

[19] R. Rajkumar, L. Sha, and J. P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Proceedings of the 9th IEEE Real-Time Systems Symposium (RTSS '88)*, pages 259–269, 1988.

[20] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Computers*, 39(9):1175–1185, 1990.

[21] J. Shi. DGA-LITMUS-RT. https://github.com/Strange369/Dependency-Graph-Approaches-for-LITMUS-RT, 2018.

[22] J. Shi. HDGA-LITMUS-RT. https://github.com/Strange369/Dependency-Graph-Approach-for-Periodic-Tasks, 2019.

[23] J. Shi, K.-H. Chen, S. Zhao, W.-H. Huang, J.-J. Chen, and A. Wellings. Implementation and evaluation of multiprocessor resource synchronization protocol (mrsp) on litmusrt. In *OSPERT*, 2017.

[24] J. Shi, J. D. T. Pham, K.-H. Chen, M. Münch, J. V. Hafemeister, and J.-J. Chen. Supporting Multiprocessor Resource Synchronization Protocols in RTEMS. https://github.com/Strange369/RTEMS-Resource-Synchronization-Protocols, 2020.

[25] J. Shi, N. Ueter, G. von der Brüggen, and J.-J. Chen. Multiprocessor synchronization of periodic real-time tasks using dependency graphs. In *RTAS*, pages 279–292, 2019.

CABAS: Real-Time for the Masses

Till Smejkal, *TU Dresden*
Jan Bierbaum, *TU Dresden*
Manuel von Oltersdorff-Kalettka, *TU Dresden*
Michael Roitzsch, *Barkhausen Institut*

***Abstract*—Although the real-time community has produced impressive research results, real-time methodology has not gained traction in commodity application development. Only in specialized niches, like avionics and automotive, solid real-time methods are applied because of regulatory requirements and safety concerns. Outside those niches, best-effort-style programming is the norm. We strife to bridge this gap by exposing approachable interfaces to everyday programmers.**

In this paper, we present CABAS, a user-level framework that traces the execution times of jobs online and, based on these data, automatically adapts the parameters of the CBS real-time scheduler inside the Linux kernel. With this approach, we hope to align soft real-time programming with current development methods by freeing the developer from the burden of manually deriving appropriate soft real-time scheduling parameters.

I. INTRODUCTION

The real-time community has accumulated an impressive body of knowledge on task models and schedulability analysis. However, developing a real-time application according to these teachings is a complex undertaking in practice. Therefore, proper real-time methodology is only applied in niches, where regulatory requirements demand a safety certification. From cloud interactive apps to IoT robots and drones, many areas would benefit from solid real-time solutions.

In this work, we present CABAS, a user-level frontend to the constant bandwidth server (CBS) implementation inside the Linux kernel. Offline timing analysis of applications is replaced by automatic online tracing of job runtimes, paired with machine learning to predict execution times. CABAS is inspired by ATLAS [1], which has demonstrated a scheduler design based on tracing of execution times and machine learning to replace worst-case analysis. ATLAS therefore addresses our design goal of simplifying the development of real-time applications, but is limited to single core systems, lacks an approachable programming interface, and requires an in-kernel scheduling component.

After we provide a short overview of CBS and ATLAS, we discuss our framework (Section III). We use synthetic benchmarks and a video player (Section IV) to demonstrate that CABAS is not only able to provide appropriate parameters to the CBS scheduler but also to automatically adapt these parameters when the application's demands change at runtime.

II. BACKGROUND

A. ATLAS *Runtime and Kernel Scheduler*

ATLAS as described by Roitzsch *et al.* [1] consists of two parts. First there are a runtime and programming framework that accept soft real-time jobs from applications and deliver them to the second part, a dedicated soft real-time scheduler. One key aspect of the ATLAS runtime is that programmers do not need to make a complicated execution time analysis of their applications, but only have to specify a relative deadline. The execution time of the application is automatically trained by the runtime using machine learning. This training can be assisted by the programmer by attaching workload metrics to individual jobs which will be considered by the runtime to predict the job's execution time. Thanks to these runtime abstractions, programmers can stay in their respective domains and only have to know which application-specific metrics correlate best with their application's execution time.

However, a significant problem of the ATLAS runtime and scheduler is, that the in-kernel soft real-time scheduler is not part of the mainline Linux kernel. Hence, maintenance of the scheduler is a significant burden, as the Linux scheduling interfaces change regularly. In addition to that, ATLAS cannot simply be used in a plug-and-play fashion, but instead requires a special Linux kernel with the ATLAS soft real-time scheduler built into. Accordingly, although the programming interface and runtime are beneficial for a wide-spread usage of soft real-time, an integration of ATLAS into any system for a normal audience — no kernel developers — is difficult.

B. Real-Time in the Linux Kernel

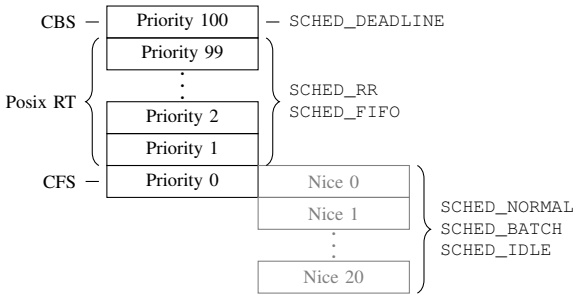


Figure 1: Schedulers of the Linux kernel and their priorities.

The wide variety of use-cases for the Linux kernel and thereby the huge amount of different requirements on the system, resulted in a number of diverse schedulers co-existing within the kernel, as shown in Figure 1. The majority of processes are scheduled using the Completely Fair Scheduler (CFS). CFS handles processes that have no real-time requirements and are thus scheduled with the `SCHED_NORMAL` priority. So-called *nice-levels* can be used to instruct the scheduler to prefer some processes over others, but no guarantees about execution times or when a process is executed are given.

If one strives for more control about a process' execution, the Linux kernel also provides the `SCHED_FIFO` and `SCHED_RR` priorities. Both of them are further separated into 99 sub-priorities and are guaranteed to never be preempted by a process running in a lower priority within `SCHED_FIFO`, `SCHED_RR`, or any process being scheduled by CFS. Although many consider the `SCHED_FIFO` and `SCHED_RR` schedulers already as real-time schedulers, no guarantees among the processes are given by the scheduler regarding, for example, the process' time to finish. Instead, processes are just executed in the order in which they were presented to the scheduler. However, the Linux kernel also supports yet another scheduling priority, namely `SCHED_DEADLINE`. Processes in this priority are handled by a Constant-Bandwidth-Server-like scheduler and are always executed with a priority above any other scheduler in the system.

Constant Bandwidth Server (CBS) [2] is a scheduling algorithm that handles hard and soft real-time tasks simultaneously, guaranteeing deadlines for hard real-time jobs and a constant bandwidth for the soft real-time ones. The CBS implementation in Linux supports hard and soft real-time tasks. In both cases, a task τ_i is defined by its inter-arrival time T_i and its execution time Q_i. For hard real-time tasks, T_i and Q_i refer to the minimum inter-arrival time and the worst case execution time (WCET), whereas for soft real-time tasks the parameters are just considered mean values. CBS will schedule its workload using Earliest Deadline First (EDF). For hard real-time tasks the inter-arrival time T_i is used as the deadline of individual jobs. For soft real-time tasks the Constant Bandwidth Server protocol is used to manage the jobs and calculate their per-job deadlines. Every Constant Bandwidth Server maintains a budget q_i that is expended when a corresponding job executes. Once the budget reaches 0, the deadline of the server is increased by T_i of the soft real-time task that the server manages.

In order to integrate CBS in the Linux kernel, various changes to its behavior were proposed and implemented [3, 4]. Whereas in the original CBS description the budget of a server depleted equally to the executed time of the corresponding job, the implementation in the Linux kernel uses a varying rate based on the state of the server and the overall utilization of the system. Furthermore, Linux distinguishes between active servers — servers that have pending jobs — and inactive ones — servers without pending or running jobs. This state of a server influences the overall utilization of the system and hence the rate at which servers decrease their budget. In addition, the Linux implementation of CBS always leaves some space in its real-time schedule to let other processes that are scheduled with lower priorities to also execute on the CPU.

C. Soft vs. Hard Real-Time

The design of the ATLAS scheduler and framework is tailored towards soft real-time applications. CBS on the other hand supports both hard and soft real-time applications. The nature of ATLAS with its predictor, that might estimate incorrect execution times is an inherent problem for hard real-time applications, which are usually used in mission-critical systems where missing a deadline would lead to a potentially catastrophic failure. For soft real-time applications a deadline miss is usually not critical but just not favorable. Typically, with soft real-time applications the usability of a result decreases the later the job finishes and hence might result in unwanted glitches or visible artifacts. Yet, normally soft real-time applications can still continue even after a deadline miss. Hence, our work CABAS focuses on making the creation of soft real-time applications easier since moving the time-consuming execution time analysis of an application into an online-trained predictor is acceptable. Deadline misses, which might occur especially at the beginning of the predictor training, can be tolerated by the application. However, for hard real-time applications CABAS is probably not suited, although the used Linux CBS scheduler is capable of managing such scenarios as well.

III. IMPLEMENTATION

A. Using Linux CBS

As described in Section II-B the Linux kernel scheduler comprises multiple different schedulers that work together, each handling a different scheduling priority. In order to run real soft or hard real-time applications on a Linux system one has to instruct the kernel to use the `SCHED_DEADLINE` scheduler for this particular application via the `sched_setattr` system call. To define and possibly change the task parameters such as its WCET or its deadline one has to use the additional `sched_setparam` system call.

As the Linux schedulers always schedule individual threads (`struct task_struct` within the kernel), it is even possible to compose an application of normal threads managed by CFS and real-time threads managed by the Linux CBS scheduler. However, this thread-based management within the kernel makes the integration of the task- and job-based model of real-time applications difficult. Especially in a soft real-time scenario, where there are multiple possibly independent jobs, the mapping to separate threads can be burdensome. Our framework CABAS addresses this problem and provides the programmer with an easy-to-use interface that abstracts thread creation, configuration, and management away.

B. The CABAS Framework

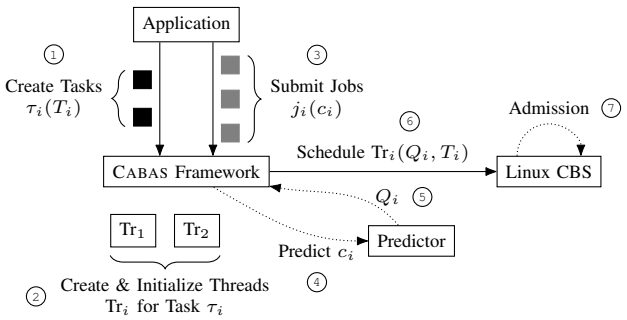


Figure 2: The architecture of the CABAS framework.

The architecture of CABAS is visualized in Figure 2. In general CABAS consists of the framework, which handles the interaction with the kernel's CBS scheduler as well as manages all the necessary state, and the estimator, which is used to predict missing information such as a job's execution time based on workload metrics provided by the application. In order to run a real-time workload using CABAS, the programmer first has to create tasks τ with an expected inter-arrival time T_i ①. For each task CABAS will create a corresponding thread and initialize the thread such that it can be run with the Linux CBS scheduler ②. This initialization includes, for example, the calls to `sched_setattr` and `sched_setparam` with the correct parameters. Later, when the application submits actual jobs to the framework, it can optionally augment them with metrics c_i that correlate positively with the jobs' runtime ③. CABAS uses the workload metrics — if provided — to predict the execution time Q_i of the job based on observations of previous job executions ④ & ⑤. We use the linear least-squares auto-regressive predictor presented by Roitzsch *et al.* [1], which we found to work reasonably well for a set of synthetic benchmarks and a video player; further discussed in Section IV. As shown in the figure, the predictor is not tightly integrated into the CABAS framework and can thus be substituted by the programmer with a specialized implementation. The prediction step can also be omitted when the programmer provides the expected execution time with the job submission. Providing an upper bound from a worst-case execution time analysis allows hard real-time operation, but this is not the focus of our work.

In any case, CABAS will then schedule the thread corresponding to the job's task with the specified parameters on the kernel's CBS scheduler ⑥. In the case that the predictor determined different parameters then for the previous run, CABAS will update the scheduling parameters via the `sched_setparam` system call and the kernel's CBS scheduler will rerun its admission test ⑦ in order to prevent over-utilization of the system. When this admission should fail, the framework will receive an error upon which further steps need to be taken by the programmer. Ignoring the error and running the task with the old parameters might be a valid option, but overload handling can be application-specific and is orthogonal to our solution. If multiple jobs are submitted for one task, the framework will internally queue them and start them either at a specified point in time or as soon as possible.

C. Programming with CABAS

```
1 int create_task(int period, void (*execute)(void *), int
      exec_time);
2 int create_task_pred(int period, void (*execute)(void *),
      struct metrics (*generate)(void *));
3 void add_job_to_task(int task, void *arg);
4 void join_task(int task);
```

Listing 1: The C Interface of the CABAS Framework

CABAS itself is an open-source C/C++ framework[1] and can thus be used by many existing applications. Even integrating it into modern programming languages like Rust is easily doable as foreign function calls to C libraries are usually supported. Listing 1 shows the C-interface of CABAS, which consist of mainly three important functions. There are `create_task_pred` and `create_task` to submit new tasks to the framework. Whereas the former version will instruct the framework to use the estimator, the latter follows the more traditional task model for CBS where a median execution time and inter-arrival time for a task's jobs are specified at creation time. When the predictor is used, CABAS will call the `generate` function before every job execution. That function should return the workload metrics for the given job, which are then used by the framework to predict the job's execution time. Since this function can be specified by the programmer, various techniques are possible for generating the metrics. To run the actual job, the framework will call the `execute` function specified at the task creation and provide it the corresponding arguments for the job. CABAS assumes that a task can be represented by one function. Each job incarnation is a call to this function with varying arguments. Accordingly, the third function that the framework provides (`add_job_to_task`) adds new jobs to a task by specifying the arguments with which the `execute` function should be called. The interface function `join_task` will wait for the completion of all jobs of one task.

D. CABAS *vs. Traditional CBS*

One significant difference between CABAS and traditional CBS is that CABAS dynamically adjusts the scheduling parameters during the execution of the application. In a traditional CBS soft real-time application, a programmer would use offline analysis to determine the mean execution time and the mean inter-arrival time of a task and provide these values to the scheduler. Whereas CBS itself is designed to handle variations within the execution times of the jobs, significant changes in the behavior of jobs, as they can happen with phased applications, can lead to a significant tardiness (see Section IV for an example). CABAS however can alleviate such effects by dynamically adjusting the CBS scheduling parameters at runtime and is thereby able to react to changing application behavior. For example, if an application enters a phase where jobs take longer than usual, CABAS will automatically increase the budget for the corresponding CBS Server, reserving more CPU time for the job.

This dynamic parameter adaptation is also beneficial when an application is ported to new hardware. With traditional CBS one either has to do a new analysis to calculate the mean execution times of the application on the new hardware, or use the previously calculated values and accept a potentially significant job tardiness. With CABAS and its predictor no analysis is necessary as the system will automatically learn the mean execution times and use them accordingly.

[1] https://github.com/TUD-OS/Cabas

IV. Evaluation

We evaluate CABAS with two different applications: a synthetic benchmark and an elementary video player. All experiments were performed on a machine with an Intel Core i7-4790 (4 cores, 2 threads each) and 8 GiB RAM running Ubuntu 21.10 with Linux 5.13. For collecting runtime data, we used the low-overhead *Linux Trace Toolkit: next generation* (LTTng) to set a tracepoint to the end of a job and calculated its tardiness. The time CABAS requires for training and estimation is accounted as part of a job's execution time.

A. Synthetic Benchmark

A synthetic soft real-time workload comprises multiple tasks, each with a target mean inter-arrival time and mean execution time. The values for individual jobs use a standard distribution with the chosen mean value and a standard deviation of 10%. Within this range the values are uniformly distributed.

To acquire representative and reliable results, we generated 1000 workloads for each integer utilisation in the range between 50% and 200% using the *UUniFast* algorithm [5]. Each workload comprises five tasks with at least five jobs each, capping the inter-arrival time at 10 000 µs. The minimum inter-arrival time is set to 200 µs. Workloads run for at least 50 000 µs on either one or two cores. Utilisation above 100% were only run in the two-core scenario as CBS is not expected to handle overload situations. A dedicated thread in `SCHED_FIFO` submits the real-time jobs to the CABAS framework, which then executes them on Linux CBS. The jobs themselves consist of a busy loop consuming the job's appointed runtime.

Figure 3 shows the mean tardiness of a job across the 1000 workloads for a specific system utilisation. CBS is statically configured with the target mean inter-arrival time and mean execution time of a workload. Note, that in a real-world scenario the application developer would need to determine and provide this information. CABAS, on the other hand, is given only the mean inter-arrival time and automatically derives the expected runtime using its predictor (see Section III).

The tardiness of jobs increases up to 100 µs to 250 µs when scheduled with CABAS. The overall behaviour is similar for the dual core scenario. Considering the simplistic nature of the synthetic real-time jobs, the only possible metric is the target runtime itself. Knowing these values, however, would allow to directly use CBS and thus defeat the purpose of CABAS, so we did not use any metrics.

B. Video Player

A typical application that can profit from soft real-time is a media player that wants to provide smooth playback even in the presence of high system load. Unfortunately, adopting real-time scheduling is complex and no major media player makes use of it. To evaluate this interesting scenario anyway, we developed an elementary video player based on *FFmpeg*. The player reads a media file, extracts the video stream, decodes the individual video frames, and renders them using *SDL2*. Any other media streams (audio, subtitles, …) are discarded.

The player maintains three processing tasks: (read and) *decode* frames, *scale* the frame to the expected output size and colour format, and *render* the frame at the right point in time. In contrast to regular video players, ours never drops late frames to highlight the effect of such delays. For the evaluation, we used the animated short film "Big Buck Bunny"[2]. This video is encoded with H.264, has a resolution of 1920×1080 pixels, and a frame rate of 60 Hz. We played the first 3 minutes of the video, i.e. 10 800 frames. Given the target frame rate, all tasks have a period of $\frac{1}{60}$ s. A dedicated management thread, pinned to one core, issues new jobs to the aforementioned tasks and forwards data whenever appropriate. Processing jobs are free to use any of the remaining hardware threads.

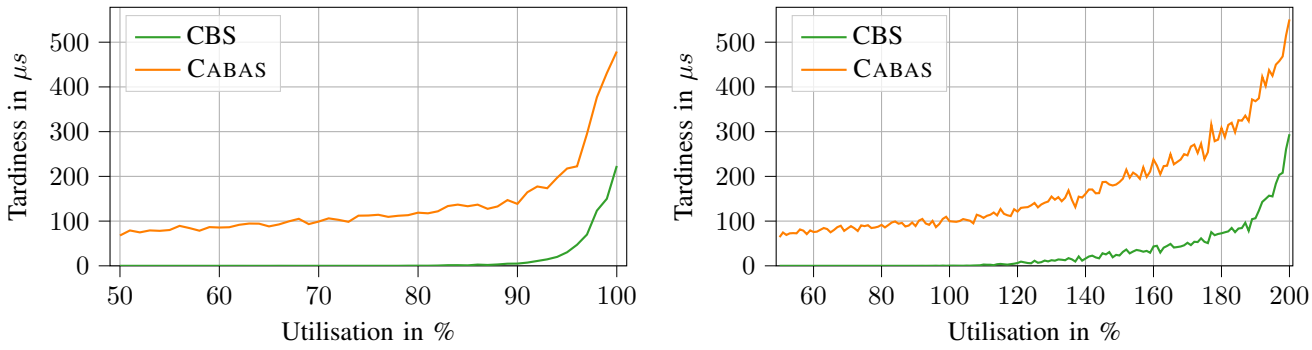
CABAS allows to provide application-specific metrics with a new job, which are then used to improve the runtime prediction (see Section III). Whereas scaling and rendering have a homogeneous workload and take a fixed amount of time, this is different for the decoding task. Due to the nature of modern video codecs, frames might have to be decoded out of order: Decoding a *B frame* (bidirectional coded picture) requires that its successor has been decoded. Thus, the execution time for decoding the chronologically next frame can vary depending on that frame's type and we use this frame type as a metric for CABAS. To give the scheduler more leeway in compensating for deviations, we established a buffering scheme that allows up to 8 scale jobs and 8 render jobs to exists in the system. In contrast to the synthetic benchmark, the mean job runtimes for the video player tasks were not known. We determined these times in an initial run of the player in CFS on an unloaded system. The decoding time for each frame is depicted in Figure 4. To simplify the player's implementation, we also pre-determined the frame type during this initial run.

Users of a media player care about smooth playback, i.e. the timely rendering of frames. Figure 5 shows mean tardiness for each individual frame when using different schedulers. On a lightly loaded system CFS shows no tardiness at all, which also demonstrates that the CPU is capable to decode all frames fast enough to maintain the target frame rate. However, with heavy load, CFS finishes with the video playback almost 9 seconds late. A heavily loaded system for CFS was simulated by running in parallel to the video player as many `stress`[3] benchmarks as the system has CPUs.

CBS, configured with the mean value of all execution times for each specific task, accumulates a delay of 18 seconds over the 3 minute runtime of the video. Further tests showed that long-running decode jobs may leave the scale task without work and thereby depleting the scale task queue eventually. Since the scale task has a very stable runtime, its budget is just enough to complete one job per period. Therefore, the scale task can never catch up with the refilled queue once it was initially delayed. Unfortunately, this creates a downstream effect as the following render task will also eventually deplete its queue and be limited by the throughput of the scale task.

[2] https://peach.blender.org/download/

[3] https://github.com/resurrecting-open-source-projects/stress



(a) Real-time tasks on a single core

(b) Real-time tasks on two cores

Figure 3: Mean tardiness of a job depending on the workload's utilisation.

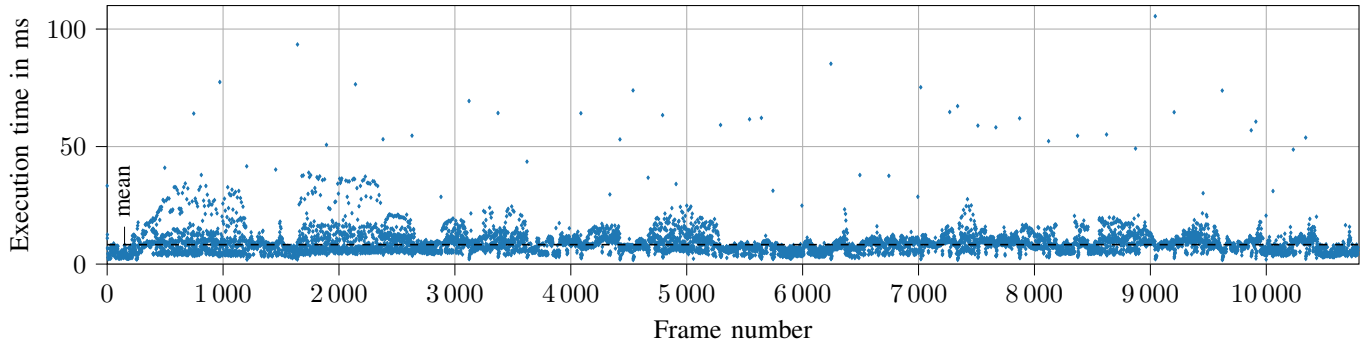


Figure 4: Execution times of the *decode* task in low-load CFS. The black line shows the mean value used for CBS (8.268 ms).

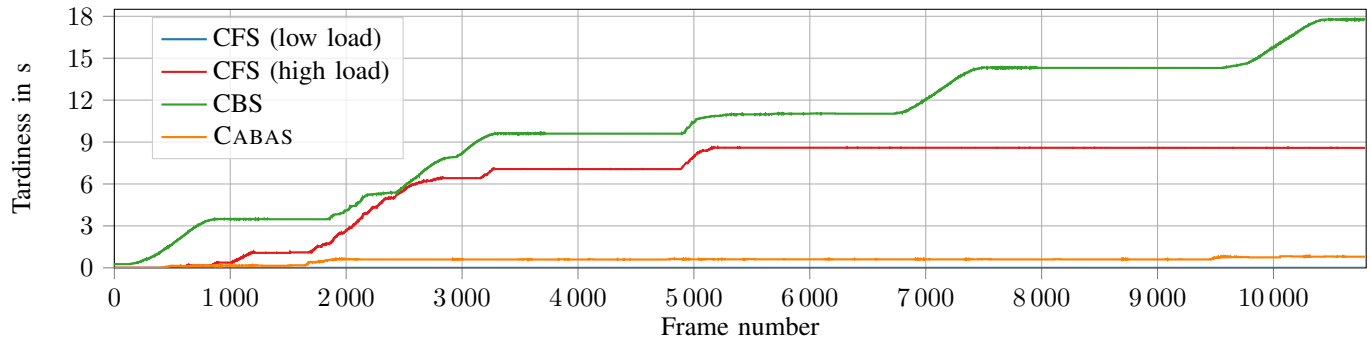


Figure 5: Tardiness of the *render* task for individual frames.

The main problem is that the Linux CBS implementation is not work conserving in contrast to when the player runs in CFS. Further measurements showed that overbudgeting the scale task (e.g. giving it twice the budget) can remedy this situation to some degree, but would not add any additional insight to the comparison of the schedulers. Such a decision is, once again, on the programmer of the application. In addition we found out, that the rendering of the video in the SDL2 player window done by the X-server also introduces some push back on the render task. The main problem is that the X-server itself is not scheduled using real-time priorities. This mismatch of scheduling priorities between the render task and the X-server creates unpredictable additional delay in the render task as the task waits for the acknowledgment of the window manager that the window was updated before it renders another frame. We could verify with additional measurements that not drawing the frame on the X-server window but just in an internal frame buffer reduces the delay of the video playback when run with CBS. When this drawing just to an internal frame buffer is combined with the aforementioned overbudgeting of the scale task, the video player is even able to catch up with the playback eventually after a delay happened and will thus finish with a lower or even no delay in contrast to what is shown in Figure 5 for CBS

When running the player with CABAS tardiness increases, too, but less drastically than with CBS. Adjusting the scheduling parameters by learning and estimating the execution times with CABAS' predictor can better compensate for the long-term variation in execution times. CABAS will temporarily give, for example, the scale tasks more budget when a long-running decode occurs in the decode task, which causes a delay of the scale task. The scale task is thus able to catch up with the refilling queue eventually. When the scale task catches up with the new load and has a stable execution time again, CABAS will reduce the budget accordingly. To sum it up, due to the execution time predictor, CABAS can react to changes in the application behavior which otherwise have to be done manually by the programmer. Additional measurements with CABAS where we disabled the rendering of the frame in the X-server window, similar to the scenario for CBS, resulted in a tardiness of 0 s and show that our framework is able to smoothly play back a video even without any additional scheduling-specific knowledge (e.g. fine-tuning of scheduling parameters) from the programmer.

V. Related Work

CABAS intends to make real-time scheduling more accessible for developers. A considerable expertise barrier exists when application developers implement problems amenable to real-time scheduling, which has also been observed by Brandenburg [6]. Other works share our goal of lowering this barrier, approaching it either using reservation-based methods or the fair-share schedulers present in commodity systems.

a) Reservation Approaches: The original CBS [7] wraps tasks with varying execution times in a server to make scheduling behavior more robust if task parameters are not exact. To improve quality of service for a soft real-time load, later work added dynamic changes to the allocated server bandwidth [8] and slack reclaiming mechanisms [9]. CBS-based schedulers have also been demonstrated on multicore systems [10] and within frameworks for quality of service control [11].

At their core, all reservation-based mechanisms require the developer to report an execution time or bandwidth requirement to the scheduler. Such information is difficult to obtain for highly workload-dependent tasks that end-users run on a variety of hardware platforms with different speeds. The added adaptivity features, however, enable CBS to tolerate over- as well as underspecification. In return, CBS provides timeliness guarantees and is suitable for hard real-time. CABAS relieves the developer from determining execution times entirely, but is not suitable for hard real-time. A common trait of CBS'es adaptive reservations and CABAS is the use of a per-task execution time estimator. However, the estimator presented for CBS [8] extrapolates by using only past execution times. CABAS leverages workload metrics to improve estimations and transparently communicates execution time information to the CBS scheduler without involving the developer.

b) Fair Processor Sharing: Fair-share schedulers allow limited control over scheduling behavior by providing a priority interface like the Unix nice levels. However, a developer cannot determine the correct priority level for an application without a complete overview of all other applications in the system. Borrowed Virtual Time [12] expresses task priorities with a concept called *warp time*, but assigning these parameters still requires global system knowledge, because warp times are not intrinsic to one application. CABAS, CBS, and all other EDF-based schedulers employ deadlines which are parameters from the problem domain of the application. They can be specified without global knowledge.

VI. Conclusion

We present CABAS, an easy-to-use user-level frontend to the CBS real-time scheduler. CABAS mediates between application and scheduler, enabling developers to spawn new work items with individual execution behavior by way of a single function call. Developers need to reflect only on application-local behavior and CABAS only asks for parameters from the application domain: Periodic deadlines express latency requirements, workload metrics describe jobs. CABAS uses these values and execution time information it gathers in the background to derive scheduling parameters for CBS.

We have demonstrated, using synthetic benchmarks and video playback, that CABAS can properly handle common use cases. Not only can it derive suitable CBS parameters, but also automatically adjust them to changing application demands at runtime. Our framework, thus, makes soft real-time scheduling readily available to the average developer.

As a next step we plan to look more closely into why CBS has such a devastating behavior for the video player example and whether we can improve CABAS and Linux' CBS in general to closer match the CFS low load performance. Furthermore, we want to investigate other applications and areas where CABAS can be applied. One interesting aspect would be to use the real-time-specific information such as deadline and estimated execution time for more energy-aware scheduling.

References

[1] M. Roitzsch *et al.*, "ATLAS: Look-ahead scheduling using workload metrics," in *Proceedings of the 19th IEEE real-time and embedded technology and applications symposium*, ser. RTAS, Philadelphia, PA, USA: IEEE, Apr. 2013, pp. 1–10, ISBN: 978-1-4799-0184-5. DOI: http://dx.doi.org/10.1109/RTAS.2013.6531074. [Online]. Available: http://os.inf.tu-dresden.de/papers_ps/rtas2013-mroi-atlas.pdf.

[2] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proceedings 19th IEEE real-time systems symposium (cat. no. 98CB36279)*, IEEE, 1998, pp. 4–13.

[3] L. Abeni *et al.*, "Greedy CPU reclaiming for SCHED_DEADLINE," in *Proceedings of the real-time Linux workshop (RTLWS), dusseldorf, germany*, 2014.

[4] J. Lelli *et al.*, "Deadline scheduling in the Linux kernel," *Software: practice and experience*, vol. 46, no. 6, pp. 821–839, 2016.

[5] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-time systems*, vol. 30, no. 1, pp. 129–154, May 2005. DOI: 10.1007/s11241-005-0507-9.

[6] B. B. Brandenburg, "The case for an opinionated, theory-oriented real-time operating system," in *1st international workshop on next-generation operating systems for cyber-physical systems*, ser. NGOSCPS, Montreal, Canada, Apr. 2019. [Online]. Available: https://www.cse.wustl.edu/~cdgill/ngoscps2019/papers/NGOSCPS2019_Brandenburg.pdf.

[7] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proceedings of the 19th IEEE real-time systems symposium*, ser. RTSS, Madrid, Spain: IEEE, Dec. 1998, pp. 4–13, ISBN: 0-8186-9212-X. DOI: http://dx.doi.org/10.1109/REAL.1998.739726. [Online]. Available: http://retis.sssup.it/~giorgio/paps/1998/rtss98-cbs.pdf.

[8] L. Abeni *et al.*, "QoS management through adaptive reservations," *Real-time systems*, vol. 29, no. 2, pp. 131–155, Mar. 2005, ISSN: 0922-6443. DOI: http://dx.doi.org/10.1007/s11241-005-6882-0. [Online]. Available: http://retis.sssup.it/~lipari/papers/real_time_systems_cucinotta_palopoli_adaptive_reservations.pdf.

[9] L. Palopoli *et al.*, "Weighted feedback reclaiming for multimedia applications," in *Proceedings of the 2008 IEEE/ACM/IFIP workshop on embedded systems for real-time multimedia*, ser. ESTImedia, Atlanta, GA, USA: IEEE, Oct. 2008, pp. 121–126, ISBN: 978-1-4244-2612-6. DOI: http://dx.doi.org/10.1109/ESTMED.2008.4697009. [Online]. Available: http://disi.unitn.it/~palopoli/publications/estimedia08.pdf.

[10] S. Kato *et al.*, "AIRS: Supporting interactive real-time applications on multicore platforms," in *Proceedings of the 22nd euromicro conference on real-time systems*, ser. ECRTS, Brussels, Belgium: IEEE, Jul. 2010, pp. 47–56, ISBN: 978-0-7695-4111-2. DOI: http://dx.doi.org/10.1109/ECRTS.2010.33. [Online]. Available: http://ertl.jp/~shinpei/papers/ecrts10.pdf.

[11] T. Cucinotta *et al.*, "On the integration of application level and resource level QoS control for real-time applications," *IEEE transactions on industrial informatics*, vol. 6, no. 4, pp. 479–491, Nov. 2010, ISSN: 1551-3203. DOI: http://dx.doi.org/10.1109/TII.2010.2072962. [Online]. Available: https://scholar.google.com/scholar?cluster=5484267806271076788.

[12] K. J. Duda and D. R. Cheriton, "Borrowed-Virtual-Time (BVT) scheduling: Supporting latency-sensitive threads in a general-purpose scheduler," in *Proceedings of the 17th ACM symposium on operating systems principles*, ser. SOSP, Charleston, SC, USA: ACM, Dec. 1999, pp. 261–276, ISBN: 1-58113-140-2. DOI: http://doi.acm.org/10.1145/319151.319169. [Online]. Available: http://gregorio.stanford.edu/bvt/bvt.ps.

On the Interplay of Computation and Memory Regulation in Multicore Real-Time Systems

Denis Hoornaert*, Golsana Ghaemi†, Andrea Bastoni*, Renato Mancuso†, Marco Caccamo*, and Giulio Corradi‡
*Technical University of Munich †Boston University ‡Xilinx
*{denis.hoornaert, andrea.bastoni, mcaccamo}@tum.de, †{golsana, rmancuso}@bu.edu, ‡giulioc@xilinx.com

***Abstract*—The ever-increasing demand for high-performance in the time-critical embedded domain has pushed the adoption of powerful yet unpredictable heterogeneous Systems-on-a-Chip. The shared memory subsystem, which is known to be a major source of unpredictability, has been extensively studied, and many mitigation techniques have been proposed. Among them, performance-counter-based regulation techniques have seen widespread adoption. However, the problem of combining performance-based regulation with time-domain isolation has not received enough attention.**

In this article, we discuss our current work-in-progress on SHCReg (Software Hardware Co-design Regulator). First, we assess the limitations and benefits of combined CPU and memory budgeting. Next, we outline a full-stack hardware/software co-design architecture that aims at improving the interplay between CPU and memory isolation for mixed-criticality tasks running on the same core.

***Index Terms*—Criticalities, Real-time, Hypervisor, Budget-based Regulation**

I. Introduction

The real-time community has proposed many successful techniques to mitigate the impact of inter-core memory interference (*e.g.*, [6], [7]). Notably, performance counter (PMC) based techniques such as *Memguard* [7] have received significant attention due to their practicality. In fact, PMC-regulation techniques are used to establish *temporal isolation* by mitigating the problem of non-arbitrated memory bandwidth sharing between cores. In the embedded and real-time domain, these techniques are often implemented within a partitioning hypervisor (*e.g.*, Jailhouse [2]) when the consolidation of multiple RTOSs onto the same multicore system-on-a-chip (MPSoC) is required. At the same time, when consolidating complex applications with mixed-criticality requirements onto MPSoCs with rich OSs like Linux, CPU provisioning still remains a fundamental dimension. Here, server abstractions —*e.g.*, the Constant Bandwidth Server (CBS) [1]— are well known and widely used, with the `SCHED_DEADLINE` [5] policy being the most popular example.

Despite combining CBS-based CPU scheduling and PMC-regulation to achieve isolation in *both* time and memory domains being a logical choice, the effective integration proves to be challenging. The need to enact CBS scheduling at the task level (*i.e.*, in the OS) and PMC-regulation at the CPU level (and hence in the hypervisor) results in a lack of coordination between the two mechanisms. This leaves the system incapable of handling what we refer to as *memory overload conditions*. These correspond to all the cases where high-critical tasks are still eligible for scheduling in the OS, but unable to use the necessary memory bandwidth because they are being throttled via PMC-regulation at the hypervisor level.

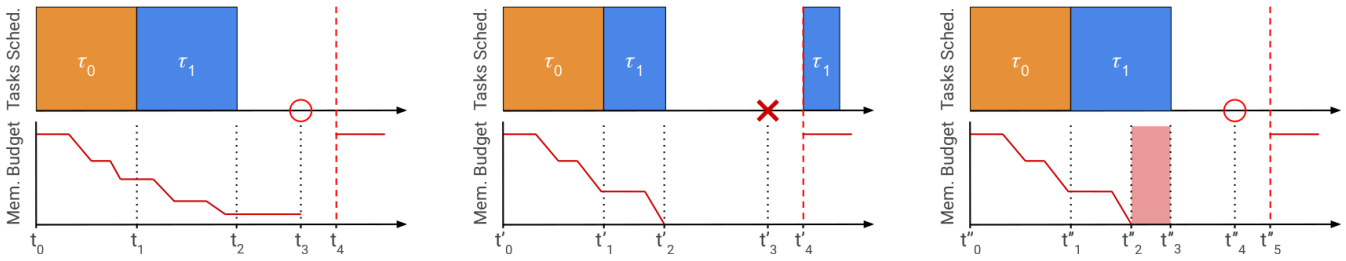
Fig. 1 illustrates a scenario where such an overload occurs. The considered system is composed of one low- and one high-criticality task (respectively τ_0 and τ_1) scheduled using CBS to absorb execution variations. The common PMC-budget assigned is determined beforehand via profiling and the addition of a fixed *safety margin*, which is common practice in industrial applications. While in Fig. 1a, τ_1 is able to complete on time, in Fig. 1b it experiences extra blocking due to the lack of sufficient *memory budget* caused by an increased memory consumption from τ_0. Such an increase can be due to changing computational needs that require additional memory accesses (for example, consider the case of object detection or object tracking in an almost empty street vs. at a crowded intersection). We note that such an increase in memory consumption cannot be determined apriori without resorting to very pessimistic over-estimations.

Our proposed Software Hardware Co-design Regulator architecture (SHCReg) precisely tackles this issue. Under SHCReg (Fig. 1c), when an overload is detected, the critical memory accesses of τ_1 are prioritized *at the hardware level* by switching the policy of the interconnect to the main memory from a fair round-robin to a priority-based one. Consequently, τ_1 can further execute and meet its deadline. Priorities are assigned depending on the criticality of the tasks running on each core. The idea is to facilitate τ_1's completion (possibly at the expanses of other cores) and quickly restore the standard isolation property of the system.

We envision implementing SHCReg on the Xilinx ZCU102 development board leveraging available tools including Linux, Memguard on Jailhouse,[1] and SchIM [4]. The envisioned hardware/software co-design architecture and the strategy employed are outlined in this work-in-progress.

The material presented in this paper is based upon work supported by the National Science Foundation (NSF) under grant number CCF-2008799. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF. Marco Caccamo was supported by an Alexander von Humboldt Professorship endowed by the German Federal Ministry of Education and Research.

[1] `https://github.com/rntmancuso/jailhouse-rt`



(a) Standard PMC-regulation, normal case.

(b) Standard PMC-regulation, memory budget depletion.

(c) SHCReg regulation, with interconnect policy switch.

Fig. 1: Example scenario of a PMC-regulated core, where an increased memory consumption causes τ_1 to miss its deadline.

II. Proposed Regulation Policy

Due to an early depletion of the memory budget A_k on CPU_k (see Fig. 1b), CPU_k could suffer a *memory overload*. Formally, a *memory overload* occurs if a critical task τ_i, running on CPU_k under server- and PMC-regulation, is unexpectedly stalled because its memory budget A_k is depleted, while the associated server S_i is still eligible for execution.

The key insight in the design of our SHCReg mitigation strategy is the following. When a given CPU_k experiences a memory overload, the memory traffic from CPU_k is prioritized over that of other cores, as detailed below.

1) By default, the interconnect is configured to use a fair policy (*i.e.*, akin to round-robin), and each CPU_k memory budget accounting is done following standard PMC-regulation rules.
2) When CPU_k exhausts its memory budget A_k, it is stalled until A_k is replenished unless: i) the criticality of the running task τ_i is high, or ii) a high critical task is released while CPU_k is stalled (*i.e.*, before the next replenishment period). In these cases, the core experiences a ***memory overload***.
3) Upon a *memory overload*, the interconnect policy π is switched to fixed-priority (FP), and each CPU_k's bus priority is set according to the criticality of the executed task. (We assume a finite set of task's criticalities.)
4) A CPU_k leaves the *memory overload* state when the high critical task has completed or when the memory budget A_k is replenished. When all CPUs have left a *memory overload*, the interconnect policy is set to $Fair$.
5) Each CPU_k with leftover memory budget always continues its memory budget accounting until budget depletion is reached. Only critical tasks τ_i can execute even when the memory budget is depleted thanks to the *memory overload* policy.

Interestingly, when considered alone, the individual regulation mechanisms employed by SHCReg are insufficient to achieve the same degree of isolation and flexibility. 1) Perhaps the most straightforward solution would be to over-provision the per-CPU memory bandwidth. However, unfortunately, the safe (conservative) usage of PMC-regulation alone inevitably leads to the under-utilization of the already scarce memory bandwidth. 2) On the other hand, statically prioritizing CPUs when they access main memory (*e.g.*, [4]) might lead to starvation for the low-priority CPUs and prevent them from running non-critical memory-intensive tasks entirely. 3) Dynamically switching the bus priority depending on the criticality level of the running tasks defeats the isolation properties of PMC-regulation and might prevent low-critical tasks from running when the system is not subject to memory overload.

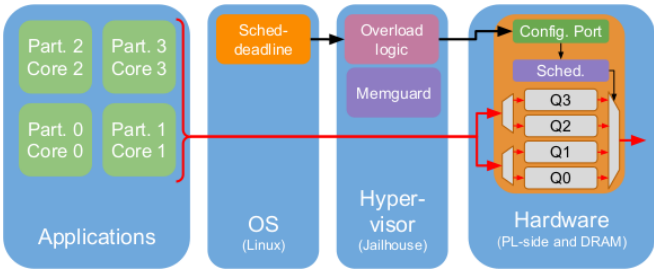


Fig. 2: SHCReg layered architecture

III. Architecture

We target systems that consolidate different RTOSs on top of (lightweight) partitioning hypervisors. SHCReg implements therefore a layered architecture such as the one depicted in Fig. 2. CPU regulation is completely implemented in software at the operating system level, while memory regulation requires a hardware/software co-design, and its implementation is distributed across the hypervisor software level and the hardware-based control of the data link to the main memory (see red arrows in Fig. 2). Furthermore, lightweight communication between layers is required to propagate, for example, information on the criticality of the currently executing tasks (see black arrows in Fig. 2).

The target platform for SHCReg is the Xilinx ZCU102 UltraScale+ development board,[2] a Linux and Hypervisor capable embedded platform associating a tightly integrated programmable fabric (referred to as PL-side) with a traditional Processing System composed of a CPU cluster (referred to as PS-side).

[2]Any PS-PL platform being hypervisor and Linux capable is eligible.

A. CPU Regulation

Real-time tasks execute at the application level on top of an OS with real-time capabilities. The OS supports a server-based scheduling policy (*e.g.*, [3]) that provides isolation among the tasks. We use Linux as OS to prototype our architecture. In Linux, the `SCHED_DEADLINE` scheduling policy [5] realizes a Constant Bandwidth Server. We associate each task to a server and define its maximum utilization. Each server is statically assigned to one of the CPUs.

B. Memory Regulation

The memory regulation is the most complex part of our architecture and consists of two layers, one implemented at the hypervisor level and one implemented at the hardware level, as depicted in Fig. 2.

1) PMC-regulation and Memory Overload Detection: The hypervisor implements a PMC-regulation mechanism limiting the maximum number of memory transactions towards the main memory the cores can issue. Implementing PMC-regulation at the hypervisor level makes the PMC-regulation transparent to the OS level, and it allows using potentially different OSs while ensuring adequate memory bandwidth control. In addition, the proposed architecture allows different OSs to use different types of CPU server regulations. The belief is that separating the PMC-regulation level from the CPU regulation level is a clean and sensible architectural choice.

2) Dynamic FP/Fair Interconnect Policy: The lowest-layer memory regulation technique leveraged by SHCReg is implemented in hardware extending the architecture of the Scheduler in-the-Middle (SchIM) [4].[3]

The SchIM module is implemented on the PL-side and acts as an intermediate step on the data path between cores and DRAM. Similarly to [4], all CPU-originated memory transactions are redirected to the PL-side and to the SchIM. As shown in Fig. 2, each core is associated with a queue storing the memory transactions directed to DRAM. Fig. 2 illustrates that CPU-originated transactions are split into two input links, each being shared by two CPUs. Under heavy traffic, the queuing of the transactions enables the SchIM to schedule them as desired by the system. Scheduling is enacted by deciding which queue's content is forwarded to the target memory and is orchestrated by the hardware transaction schedulers (depicted as *FP & Aging Sched.* and *multiplexer* modules in Fig. 2). The scheduler module defines a set of hardware schedulers (*e.g.*, Fixed-Priority, TDMA) implemented at design time and statically available on the PL at system boot. A scheduler can be selected by operating on a set of registers accessible by the whole system through a memory-mapped configuration port. We extended the original SchIM by enabling the dynamic choice of a specific scheduler at run-time and by adding the $Fair$ scheduling policy.

[3] `https://github.com/denishoornaert/MemorEDF`

C. Design Choices

The synchronization and the communication between the layers constitute a critical performance hurdle of our architecture. It is particularly the case regarding the interplay between the memory budget and the CPU budget (respectively enforced at the hypervisor- level and OS-level).

Considering rules (2) and (4) in Sec. II, the release of a critical task while a CPU is stalled and dynamically switching the priority of the interconnect when a critical task is completed requires careful synchronization between the OS and the hypervisor. For example, while a hypercall can be used by the operating system to signal the completion of a critical task, synchronizing the complex high-resolution timers in Linux with the PMC-regulation logic (at hypervisor level) to detect the release of a high-critical task during a memory-regulation phase is more complex. Either direction introduces foreseeable run-time overheads that might limit the potential benefits of the proposed architecture. Furthermore, the implementation of PMC-regulation in hypervisors such as Jailhouse-RT[4] is realized with interrupt-nesting in hypervisor-context. Therefore, its synchronization with the expiration of Linux's `hrtimers` is particularly challenging and might require considerable hypervisor changes. Nonetheless, while slow, hypercalls completion times can be bounded. On the other hand, not using a hypervisor would automatically prevent the consolidation of independent partitions combining different OSs and RTOSs.

IV. Conclusion and Future Work

We presented our work-in-progress on SHCReg, a hardware/software co-design that aims at solving the *memory overload* problems of real-time workloads with variable memory requirements on architectures that feature both CPU and PMC-based regulations. Our targets are real-world systems that consolidate multiple different RTOSs on a single MPSoC leveraging on hypervisor technologies.

Despite the technical challenges on the design side, we believe that, by exploiting the ability to dynamically change the policy of the interconnect, SHCReg could provide real-time and performance benefits for a wide class of workloads.

References

[1] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279)*, pages 4–13, 1998.

[2] Siemens AG. Jailhouse hypervisor. https://github.com/siemens/. Accessed: 2021-02-08.

[3] Giorgio C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series)*. Springer-Verlag, 2011.

[4] Denis Hoornaert, Shahin Roozkhosh, and Renato Mancuso. A Memory Scheduling Infrastructure for Multi-Core Systems with Re-Programmable Logic. In Björn B. Brandenburg, editor, *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*, volume 196 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 2:1–2:22, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/opus/volltexte/2021/13933.

[4] `https://github.com/rntmancuso/jailhouse-rt`

[5] Juri Lelli, Claudio Scordino, Luca Abeni, and Dario Faggioli. Deadline scheduling in the Linux kernel. *Software: Practice and Experience*, 46(6):821–839, 2016.

[6] 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.

[7] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory Bandwidth Management for Efficient Performance Isolation in Multi-Core Platforms. *IEEE Transactions on Computers*, 65(2):562–576, 2016.

Can we trust our energy measurements?
A study on the Odroid-XU4.

Julius Roeder
University of Amsterdam
Amsterdam, The Netherlands
Email: j.roeder@uva.nl

Sebastian Altmeyer
University of Augsburg
Augsburg, Germany
Email: altmeyer@es-augsburg.de

Clemens Grelck
University of Amsterdam
Amsterdam, The Netherlands
Email: c.grelck@uva.nl

***Abstract*—IoT devices, edge devices and embedded devices, in general, are ubiquitous. The energy consumption of such devices is important both due to the total number of devices deployed and because such devices are often battery-powered. Hence, improving the energy efficiency of such high-performance embedded systems is crucial. The first step to decreasing energy consumption is to accurately measure it, as we base our conclusions and decisions on the measurements. Given the importance of the measurements, it surprised us that most publications dedicate little space and effort to the description of their experimental setup.**

One variable of importance of the measurement system is the sampling frequency, e.g. how often the continuous signal's voltage and current are measured per second. In this paper, we systematically explore the impact of the sampling frequency on the accuracy of the measurement system. We measure the energy consumption of a Hardkernel Odroid-XU4 board executing nine Rodinia benchmarks with a wide range of runtimes and options at 4kHz, which is the standard sampling frequency of our measurement system. We show that one needs to measure at least at 350Hz to achieve equivalent results in comparison to the original power traces. Sampling at 1Hz (e.g. Hardkernel SmartPower2) results in a maximum error of 80%.

I. Introduction

Energy consumption is one of the most important design criteria for battery-powered systems. Thus, it is not surprising that decreasing energy consumption from the software side is an important topic in various research fields such as IoT, edge computing, cyber-physical systems and embedded systems (e.g. [1], [2], [3], [4], [5], [6], [7], [8], [9], [10]; for a survey see [11]).

A crucial part of energy related research is measuring the energy consumption in order to show tangible improvements on real hardware. To measure the energy consumption of a device we need to measure voltage and current, two continuous signals. Continuous signals are measured in discrete intervals at a given sampling rate. From a theoretical point of view we need to measure at twice the highest frequency desired to be measured (Nyquist rate [12]) otherwise the time series might be distorted. However, from a practical point of view it is unclear what the highest desired frequency in this case is.

In general we find that authors and reviewers place little importance on the measurement setup, as papers do not report the setup or lack details on the devices and methods used, e.g. [3], [4], [5], [6], [7], [13]. Publications that do report the measurement system used, do not investigate or consider the impact of the measurement setup on the accuracy of the measurements. For example, [9] naturally used the energy measurement system (SmartPower2[1]) provided by the manufacturer of their target board (Odroid-XU4). According to the publication the SmartPower2 measures at 1Hz. Additionally, we could not find any studies on the measurement error of the SmartPower2. In this paper we raise strong doubts about the reliability of low frequency measurements. As a community, that makes decisions based on energy consumption, we must know that our experimental setups are reliable.

To the best of our knowledge no prior paper has investigated the correlation between sampling frequency and accuracy of energy measurement systems for high-performance embedded systems. Thus, in this paper we systematically investigate the impact of the sampling frequency on the energy measurement accuracy. More specifically we measure the energy consumption of an Odroid-XU4 executing a variety of benchmarks. The measurement system used samples at a high rate; the original power-traces can then be downsampled. We then compare the downsampled traces against the original traces. That way we can alter the sampling frequency of the voltage and current measurements, while keeping all other variables equal.

The paper is organised as follows. In Section II we provide background information and detail our methodology. Section III covers our results and discussion. Then in Section IV we discuss related work. Finally, we present our conclusion in Section V.

II. Background & Methodology

In order to investigate the importance of sampling frequency we need a measurement system, a target system, programs to measure and a way to compare different sampling frequencies. In this section we start with a short discussion about power measurements in general. We then dive into the importance of sampling frequency. Next we introduce our experimental setup and the benchmarks used. Lastly, we detail the statistical tests needed.

A. Power Measurements

Power measurements can be done at the AC source or at the DC source. Discussing and comparing the advantages of

[1]https://www.hardkernel.com/?s=smartpower2&post_type=product&lang=en

either method is beyond the scope of this paper. However, in general the AC-DC converter (i.e. power supply) will have some inefficiencies, and measuring after the converter (i.e. at DC) disregards the loss. Furthermore, the loss can fluctuate with the load, i.e. power supplies are most efficient at a given load and have lower efficiencies at lower/higher loads. An additional reason to measure after the converter is that the energy consumption is most crucial for battery powered systems, which use DC.

For an overview of different DC measurement methods see [14] and [15]. In this paper we consider the shunt resistor method which observes the voltage drop across a resistor as it is widely used. We place the resistor in series with the load. And as we know the resistor value, Ohm's law can be applied to calculate the current of the load. Furthermore, we can place the resistor before the load (high side) or after the load (low side). Low side sensing is cheaper as the amplifier is simpler but has some disadvantages in comparison to high side. More specifically low side sensing is sensitive to ground disturbances and (in this case less importantly) cannot detect fault conditions. Hence, it is mostly used in mass production systems [16]. Therefore, we will focus on high side sensing.

The resistive current sensing method can be deployed directly on a target board, i.e. the board comes with an integrated power measurement function (e.g. Odroid-XU+E[2] used in [13]). Or the method can be deployed on a separate device such as the SmartPower2 or Qoitech Otii[3]. Onboard sensors are polled from the target system itself and can be polled at different frequencies. Additionally, onboard sensors are intrusive as the polling of the sensors impacts the energy consumption of the target.

The voltage drop across the resistor is amplified and then converted using an Analogue-Digital-Converter (ADC). Current sense amplifiers such as the TI INA250[4] can be used in combination with an ADC. The ADC then digitises the information for further analysis.

Once we obtained the voltage and current readings we can calculate the power (Watt). Multiple power readings result in a power trace and as we know the time between different power readings we can calculate the area under the trace, resulting in the energy consumption (Joule).

B. Sampling Frequency

Continuous signals cannot be converted to digital information continuously, instead we have to measure them at discrete intervals. The accuracy of the measurements heavily depends on the sampling frequency. In theory to reproduce an (AC) power signal one needs to measure voltage and current at four times of the highest sinusoidal frequency [17]. However, the DC consumption is not sinusoidal and instead alternates with requirements of the load. In the case of a micro-controller the current requirements change with for example the Dynamic Voltage and Frequency Scaling (DVFS) settings, instructions per clock and the actual instructions being executed [18]. Thus, the required sampling frequency depends on the length of the program being executed and the instruction mix.

[2]https://www.hardkernel.com/shop/odroid-xue/
[3]https://www.qoitech.com/otii/
[4]https://www.ti.com/product/INA250

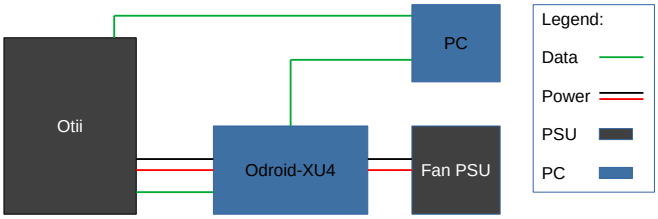


Fig. 1. Measurement setup including: Qoitech Otii, Odroid-XU4, Fan power supply, and PC

C. Setup and Target system

High-performance embedded systems like the Odroid-XU4 and the NVidia Jetson Nano are all relatively similar with respect to the clock frequency and CPU architecture. In this paper we use the Odroid-XU4 board [19] as an example target system. It is an octa-core system with 4 big cores (Cortex-A15), 4 LITTLE cores (Cortex-A7) and a Mali-GPU (T628 MP6). The two separate core clusters and the GPU all form individual voltage islands (i.e. 3 voltage islands). The voltage and the frequency can be set separately for each voltage island. The Odroid-XU4 runs an RT-patched Linux.

The Odroid-XU4 is accompanied by an energy measurement system called the SmartPower2. However, due to the low sampling frequency (1Hz) we decided to not use the system. Instead, we measure the energy consumption of the Odroid-XU4 with the Qoitech Otii on the high side. The Otii has a maximum measurement error of 0.1% + 150µA (i.e. at higher currents the error is approaching 0.1%) and has a sampling frequency up to 4kHz. The main criticism of the shunt resistor method is that a single shunt is only useful in a limited current range [20], [15]. The Otii has multiple shunts to measure very low currents (10 µA with 0.6% error) up to 5A peaks. It measures across all shunts at the same time, thus switching current range (i.e. between shunts) does not result in any lost data points.

Figure 1 shows our setup. The Odroid-XU4 receives its power from the Otii and is at the same time connected via the UART pins to the Otii. This means that the power measurements can be directly linked to messages sent by the Odroid-XU4. The fan of the Odroid-XU4 is powered via a separate circuit, and thus does not affect the power measurements of the Odroid-XU4. Before each set of measurements, we calibrate the Otii. Additionally, we warm up all connected components by executing the *heartwall* benchmark 50 times.

D. Downsampling

The Otii samples at 4kHz. Instead of either forcing a lower sampling rate or using a device with a lower sampling rate we downsample the results. That means if we sample at 4kHz but want a sampling rate of 2kHz we only take into account every

second measurement. Thus, sampling unrelated factors do not play a role (e.g. different measurement error on a different measurement device). In this paper we investigate 22 sampling rates (in Hz: 1, 2, 3, 4, 5, 10, 20, 30, 40, 50, 100, 150, 200, 250, 300, 350, 400, 500, 600, 800, 1000, 2000, 4000).

E. Benchmarks

We use the Rodinia benchmark suite [21] as target programs/tasks. The suite offers a range of targets (C, OpenCL, Cuda), different algorithms & workloads and is widely used. The suite's benchmark selection was inspired by Berkeley's dwarf taxonomy [22]. Each benchmark can further be adjusted via its input parameters. This leads to a large range of runtimes and processor loads.

We use nine benchmarks (backpropagation, BFS, Heartwall, Hotspot, Kmeans, LU-Decomposition, Nearest Neighbour, NW, SRAD) out of the suite as they can be executed on the Odroid-XU4 with minimal adaptations. The other benchmarks would have required significant changes to the code. Besides the input parameters we also vary the target DVFS settings and the target core. We measure all benchmarks on the LITTLE cores, on the big cores and on the GPU (i.e. OpenCL version). However, there are two benchmarks (BFS and SRAD) which were only measured on the big and on the LITTLE cores because the OpenCL versions did not work on the Odroid-XU4. This leads to a total of 842 unique benchmark/target/DVFS combinations. For each combination we collected 50 power traces, thus, in total we collected 42100 power traces.

The resulting dataset is available for download [5][23]. Additionally, the repository containing the analysis scripts is also available [6].

F. Statistical equivalence testing

We measure a non-deterministic system (out-of-order pipeline etc.). Additionally, the measurement system is not perfect and contains some noise. Thus, we repeat measurements for each combination, as there is not a single "correct" value. That also means that downsampling a single time-series and then calculating the error will give an indication of how much worse a lower frequency is. However, this approach does not offer a statistical indication. Therefore, we need to analyse all sets and their downsampled counterparts with statistical tests.

In a regular two-sided t-test we test if two samples are different. The null hypothesis is that there is no difference (μ_D) between two samples (Equation (1)).

$$H0 : \mu_D = 0 \tag{1}$$
$$H1 : \mu_D ! = 0 \tag{2}$$

If the t-test indicates a significant difference (e.g. p-value smaller than 0.05) then we can reject the null hypothesis and accept the alternative hypothesis that the two samples are different (Equation (2)). Thus, a t-test offers evidence in favour of the alternative hypothesis at a given confidence level (e.g. 99%). If the t-test is not significant, this is often counted as support for the null hypothesis, i.e. that there is no difference between the samples or that there is no effect. However, often a non-significant test result is the result of limited statistical power. Thus, it is impossible to know whether a non-significant result indicates equivalence (absence of an effect) or only false equivalence and is lacking statistical power [24].

[5]https://doi.org/10.21942/uva.19665564.v1

[6]https://bitbucket.org/uva-sne/energymeasurementanalysis/

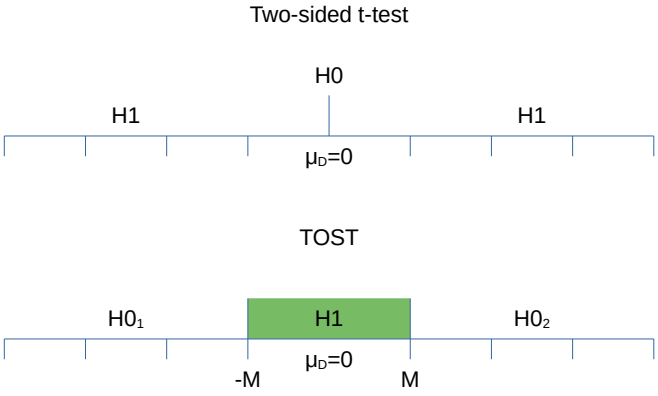


Fig. 2. Comparison of a two-sided t-test and a TOST.

Instead of proving the absence of an effect, we can show that the likelihood of an effect being smaller than a given (low) value to be significant, this is called equivalence testing. To test for equivalence between two samples we use a method called *Two One Sided T-tests* (TOST) [24]. As a TOST consists of two tests, it has two null hypotheses (Equation (3)) and (Equation (4)). The first test is used to determine if the difference between the two samples (μ_D) is smaller than the accepted lower bound ($-M$). The second one tests if the difference is larger than the upper bound M.

$$H0_1 : \mu_D < -M \tag{3}$$
$$H0_2 : \mu_D > M \tag{4}$$

Combining both test results in the alternative hypothesis (Equation (5)) that μ_D falls between $-M$ and M. Thus, if both t-tests are rejected, we have support for the alternative hypothesis that the difference between the two samples is smaller than a chosen M [25]. Figure 2 visualises the difference between a normal t-test and a TOST.

$$H1 : -M < \mu_D < M \tag{5}$$

The majority of our 842 measurement sets are not normally distributed (76.0%) according to both the Shapiro-Wilk test [26] and D'Agostino-Pearson's test [27]. Therefore, we use a non-parametric TOST based on Wilcoxon's Signed Rank test [28]. We do all tests at a 99.9% confidence ($\alpha = 0.1\%$).

One major difference between a standard t-test and an equivalence test is that one needs to determine what (low) difference (M) is acceptable (i.e. considered to be less than a noteworthy effect). We analyse the impact of 8 "acceptable error" levels (20%, 10%, 8%, 6%, 4%, 2%, 1%, 0.5%) and

what sampling level is required to achieve equivalence at that level across all 842 experiment combinations.

III. Results & Discussion

The 42100 power trace time-series can be analysed in multiple different ways. Table I summarises basic statistics of all power traces and shows that our benchmarks/target/DVFS combinations cover a wide range of run times and power. Overall we observe that the downsampled traces mostly resulted in a power consumption underestimation (98.9% of the cases) and in very few cases of overestimation (1.1%).

TABLE I
Summary statistics for all benchmark executions.

	Runtime (s)	Power (W)
Mean	9.87	2.99
Min	0.90	1.82
Max	48.15	8.44

Figures 3 and 4 show one of the power traces. Figure 3 shows the original power trace at the full sampling frequency of 4kHz and Figure 4 shows two downsampled versions. The solid blue line in Figure 4 shows how the power trace looks like if we had sampled at 1Hz. In comparison to the original trace we can see that it misses a majority of the data. Furthermore, it also misses data on the last second completely, as the execution time was 3.98 seconds. It is possible to make up for the last missed measurement by either taking the measurement at second 4 or by using the last known measurement. Either method will still lead to a significant error. The dashed red line shows the same power trace but downsampled to 10Hz. It already has a lot more detail than the 1HZ line but still misses a significant part of the signal.

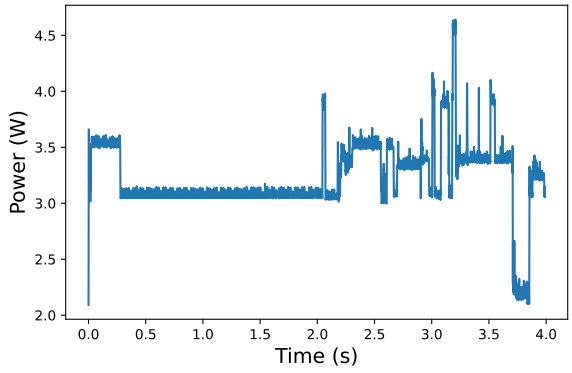


Fig. 3. Original power trace sampled at 4000Hz.

Figure 5 shows the maximum percentage error between the original energy measurement and the downsampled measurement for each frequency. Thus, the maximum error observed across all 842 combinations at 1Hz is 80%. The maximum error only drops below 0.5% at a sampling frequency of 500Hz.

The maximum error only represents a single measurement and does not carry any statistical meaning, which is the reason

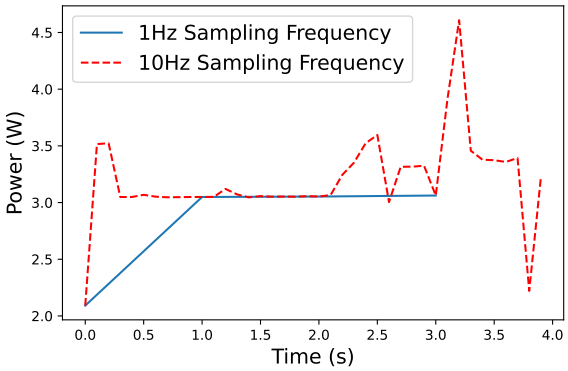


Fig. 4. Downsampled power traces.

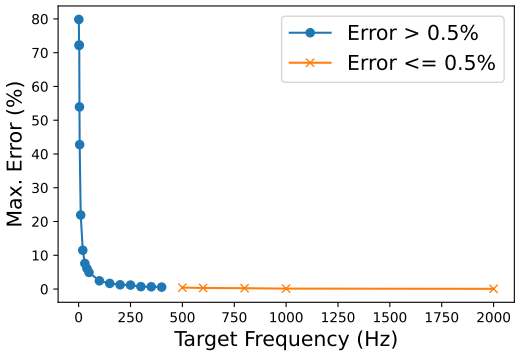


Fig. 5. Maximum error rate at each artificial frequency across all 842 experiment sets.

to employ equivalence testing. Figure 6 shows the minimum frequency required to achieve equivalent results for all 842 combinations in comparison to the full sampling frequency. Thus, if a measurement error of up to 20% is acceptable then a 30Hz sampling rate would lead to an equivalent result for all experimental combinations. At an acceptable error of 0.5%, 600Hz results in an equivalent result. Thus, at a similar level as indicated in Figure 5.

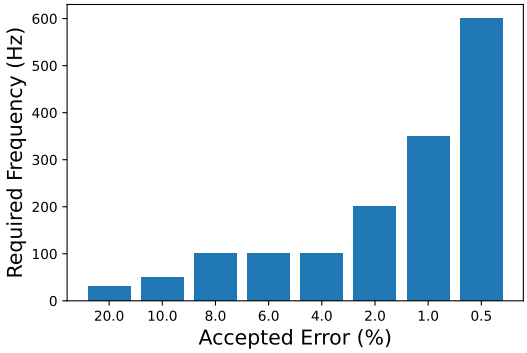


Fig. 6. Frequency required to reach equivalence given an acceptable error.

Lastly, in Figures 7 and 8 we investigate the relation between the error, benchmark run-time and the sampling frequency. Interpreting the 3D graph showing the relation between all three is not straightforward as the resulting graph contains a lot of non-continuous data points (Figure 7). To ease the interpretation, we smooth the data and the relation between the three variables using a polynomial, multi-variable regression based on a Multi-Layer-Perceptron (Scikit-learn: default parameters, hidden layer size = (64, 128, 256, 512)). This also allows us to interpolate the error to other sampling frequencies and run-times. We use 80% of the data for training. The mean absolute error on the test set is 0.0065.

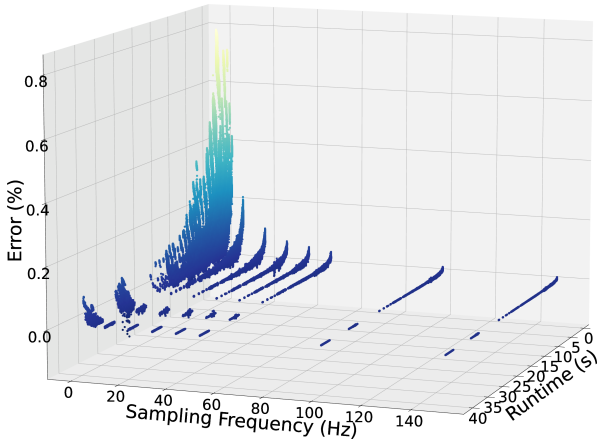


Fig. 7. Relation between the error, benchmark run-time and the sampling frequency for all power traces downsampled to between 1Hz and 140Hz.

We use the regression to predict the error of a measurement given a sampling frequency and run-time. Plotting the regression for the sampling frequency range 1Hz to 140Hz and run-times between 0.5 and 40 seconds results in Figure 8. The figure clearly shows that low sampling frequencies lead to poor results for the selected benchmarks even for longer run-times. That means that the selected long running benchmarks contained a significant amount of faster peaks that were missed at a low sampling rate. The error for short tasks remains higher even with higher sampling frequencies. As such the results obtained with a SmartPower2 are of limited use in an academic setting.

For this set of benchmarks, input parameters, target platform and DVFS settings a sampling frequency between 350Hz and 600Hz is sufficient (given an error of 1% and below). However, much shorter programs might need significantly higher sampling rates or one will have to measure the target task in a different way. For example, measuring a very short task (a few CPU cycles) will be missed even at a sampling frequency of 4kHz, thus, artificially inflating the task could work (e.g. a loop).

IV. Related Work

Cloutier et. al demonstrate that decreasing the sampling frequency from 100Hz to 1Hz results in significant loss of

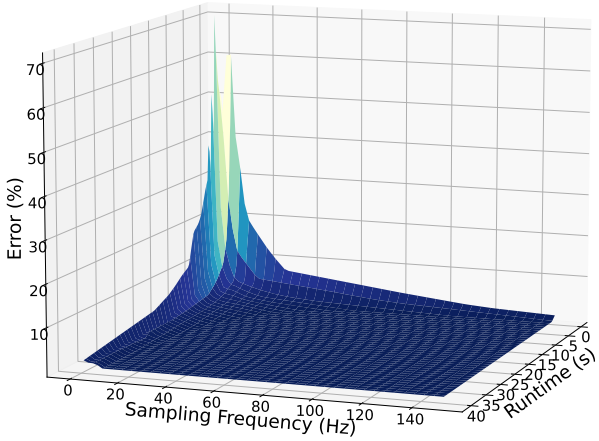


Fig. 8. Regression analysis of the error with respect to the benchmark run-time and the sampling frequency.

the power trace detail [1]. However, they do not further investigate the impact of this decrease on the energy measurement accuracy. Additionally, we can show that the accuracy of measurements at 100Hz is significantly lower than at 4kHz.

Diouri et. al investigate different energy measurement systems for servers [29]. They conclude that higher sampling rates are not necessarily good as they can introduce noise that could mask other trends. However, only because a signal is more noisy doesn't mean that the noise is erroneous and can thus be disregarded for energy measurements. One can always downsample a trace or smooth it to investigate possible hidden trends. Furthermore, server measurements could already be more noisy than high performance embedded systems due to architectural reasons, different target applications and short background tasks. Looking at Figure 3 we cannot confirm that a high sampling rate masks the trends of an application. Lastly, Diouri et. al do not investigate if the downsampled traces lead to equivalent energy measurements.

Djupdal et. al [30] develop a high-performance embedded system oriented energy measurement systems. And in [31] the authors describe two high-sampling frequency power measurement methods (up to 500kHz) for servers and for server components. However, they do not analyse the importance of the sampling frequency and if lower sampling frequencies can achieve similar results.

Buschoff et. al [20] and Jiang et. al [32] developed measurement techniques for low-powered embedded systems. They target devices with long sleep times that only consume energy in a few fast bursts. In contrast we focus on high-performance embedded systems that carry out computationally demanding tasks.

Nakutis et. al [14] and Hergenröder et. al [15] summarise the different power measurement methods and highlight the importance of the sampling frequency. However, neither paper empirically shows the resulting error.

V. Conclusion

Research into reducing energy consumption of embedded systems is popular. Hence, we need to measure the energy consumption of embedded systems. However, researchers and reviewers alike often pay little attention and consideration to how to measure energy consumption. One crucial aspect of energy measurements for high-performance embedded systems is the sampling frequency of the analogue signal.

In this paper we show that for a wide range of Rodinia benchmarks executed on the Odroid-XU4 the minimum sampling rate is 350Hz if a 1% measurement error is acceptable. Measuring at 1Hz results in errors as high as 80%. Thus, showing that systems such as the Hardkernel SmartPower2 (measurement system accompanying the Odroid-XU4) cannot be used to draw conclusions and that measurement methods with low sampling rates are only of limited use in an academic setting. Some papers in the area of reducing energy consumption of high-performance embedded systems should be re-evaluated.

If we want to reliably research and investigate methods for reducing energy consumption we must measure energy consumption accurately. That means that we need to pay more attention to our experimental setup and report our setup accurately. Careless experimental setups lead to two problems: First, we potentially focus too much on the wrong methods (false positive conclusion). Second, we discard methods that do not look promising but are in reality a good option (false negative conclusion).

In the future we would like to establish theoretical minimum requirements for sampling rate. And work on a community based set of guidelines for energy measurements in the high-performance embedded systems area to avoid such problems and confusion henceforth.

Acknowledgements

We would like to thank the reviewers for their time and feedback. This work is supported and partly funded by the HiPEAC project which has received funding by the European Union Horizon-2020 research and innovation programme under grant agreement No. 871174 (HiPEAC6 Network). Additionally, this work is partially supported by the European Union Horizon-2020 research and innovation programmes TeamPlay (grant agreement No. 779882) and ADMORPH (grant agreement No. 871259). Lastly, this work is partially supported by CERCI-RAS COST Action CA19135 funded by COST Association.

References

[1] M. F. Cloutier, C. Paradis, and V. M. Weaver, "A raspberry pi cluster instrumented for fine-grained power measurement," Electronics, vol. 5, no. 4, p. 61, 2016.

[2] J. Roeder, B. Rouxel, S. Altmeyer, and C. Grelck, "Energy-aware scheduling of multi-version tasks on heterogeneous real-time systems," in SAC, 2021, pp. 501–510.

[3] Z. Guo, A. Bhuiyan, D. Liu, A. Khan, A. Saifullah, and N. Guan, "Energy-efficient real-time scheduling of dags on clustered multi-core platforms," in RTAS. IEEE, 2019, pp. 156–168.

[4] D. Liu, J. Spasic, G. Chen, and T. Stefanov, "Energy-efficient mapping of real-time streaming applications on cluster heterogeneous mpsocs," in ESTIMedia. IEEE, 2015, pp. 1–10.

[5] M. Pricopi, T. S. Muthukaruppan, V. Venkataramani, T. Mitra, and S. Vishin, "Power-performance modeling on asymmetric multi-cores," in CASES. IEEE, 2013, pp. 1–10.

[6] A. Balsini, L. Pannocchi, and T. Cucinotta, "Modeling and simulation of power consumption and execution times for real-time tasks on embedded heterogeneous architectures," ACM SIGBED Review, vol. 16, no. 3, pp. 51–56, 2019.

[7] G. Zeng, T. Yokoyama, H. Tomiyama, and H. Takada, "Practical energy-aware scheduling for real-time multiprocessor systems," in RTCSA. IEEE, 2009, pp. 383–392.

[8] U. Odyurt, J. Roeder, A. D. Pimentel, I. G. Alonso, and C. de Laat, "Power passports for fault tolerance: anomaly detection in industrial cps using electrical efb," in ICPS. IEEE, 2021, pp. 152–157.

[9] S. Isuwa, S. Dey, A. K. Singh, and K. McDonald-Maier, "Teem: Online thermal-and energy-efficiency management on cpu-gpu mpsocs," in DATE. IEEE, 2019, pp. 438–443.

[10] N. Brouwers, M. Zuniga, and K. Langendoen, "Neat: A novel energy analysis toolkit for free-roaming smartphones," in SenSys, 2014, pp. 16–30.

[11] A. Z. Sheikh and M. A. Pasha, "Energy-efficient multicore scheduling for hard real-time systems: A survey," TECS, vol. 17, no. 6, pp. 1–26, 2018.

[12] A. V. Oppenheim, J. R. Buck, and R. W. Schafer, Discrete-time signal processing. Vol. 2. Upper Saddle River, NJ: Prentice Hall, 2001.

[13] C. Imes, D. H. Kim, M. Maggio, and H. Hoffmann, "Poet: a portable approach to minimizing energy under soft real-time constraints," in RTAS. IEEE, 2015, pp. 75–86.

[14] Z. Nakutis, "Embedded systems power consumption measurement methods overview," MATAVIMAI, vol. 2, no. 44, pp. 29–35, 2009.

[15] A. Hergenröder and J. Furthmüller, "On energy measurement methods in wireless networks," in ICC. IEEE, 2012, pp. 6268–6272.

[16] S. Arar, "Resistive current sensing: Low-side vs. high-side sensing," Accessed on 21.04.2022. [Online]. Available: https://www.allaboutcircuits.com/technical-articles/resistive-current-sensing-low-side-versus-high-side-sensing/

[17] R. S. Turgel, Sampling Techniques for Electric Power Measurement. US Department of Commerce, National Bureau of Standards, 1975, vol. 870.

[18] E. Vasilakis, I. Sourdis, V. Papaefstathiou, A. Psathakis, and M. Katevenis, "Modeling energy-performance tradeoffs in arm big. little architectures," in PATMOS. IEEE, 2017, pp. 1–8.

[19] Hardkernel Co., Ltd. Odroid-XU4. https://wiki.odroid.com/odroid-xu4/odroid-xu4. Accessed: 2019-09-06.

[20] M. Buschhoff, C. Günter, and O. Spinczyk, "Mimosa, a highly sensitive and accurate power measurement technique for low-power systems," in Real-World Wireless Sensor Networks. Springer, 2014, pp. 139–151.

[21] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in IISWC. Ieee, 2009, pp. 44–54.

[22] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams et al., "The landscape of parallel computing research: A view from berkeley," 2006.

[23] J. Roeder, S. Altmeyer, and C. Grelck, "Energy Measurements of 9 Rodinia Benchmarks executed on the Odroid-XU4." 6 2022. [Online]. Available: https://uvaauas.figshare.com/articles/dataset/Energy_Measurements_of_9_Rodinia_Benchmarks_executed_on_the_Odroid-XU4_/19665564

[24] E. Quertemont, "How to statistically show the absence of an effect," Psychologica Belgica, vol. 51, no. 2, pp. 109–127, 2011.

[25] NCSS, LLC, "NCSS 2022 Statistical Software," 2022, Kaysville, Utah, USA, ncss.com/software/ncss.

[26] S. S. Shapiro and M. B. Wilk, "An analysis of variance test for normality (complete samples)," Biometrika, vol. 52, no. 3/4, pp. 591–611, 1965.

[27] R. D'Agostino and E. S. Pearson, "Tests for departure from normality. empirical results for the distributions of $b^2$and $\sqrt{b^1}$," Biometrika, vol. 60, no. 3, pp. 613–622, 1973.

[28] F. Wilcoxon, "Individual comparisons by ranking methods," in Breakthroughs in statistics. Springer, 1992, pp. 196–202.

[29] M. E. M. Diouri, M. F. Dolz, O. Glück, L. Lefèvre, P. Alonso, S. Catalán, R. Mayo, and E. S. Quintana-Ortí, "Solving some mysteries in power monitoring of servers: Take care of your wattmeters!" in EE-LSDS. Springer, 2013, pp. 3–18.

[30] A. Djupdal, B. Gottschall, F. Ghasemi, and M. Jahre, "Lynsyn and lynsynlite: The sthem power measurement units," in Towards Ubiquitous Low-power Image Processing Platforms. Springer, 2021, pp. 93–114.
[31] T. Ilsche, D. Hackenberg, S. Graul, R. Schöne, and J. Schuchart, "Power measurements for compute nodes: Improving sampling rates, granularity and accuracy," in IGSC. IEEE, 2015, pp. 1–8.
[32] X. Jiang, P. Dutta, D. Culler, and I. Stoica, "Micro power meter for energy monitoring of wireless sensor networks at scale," in IPSN. IEEE, 2007, pp. 186–195.

Revisiting Migration Overheads in Real-Time Systems: One Look at Not-So-Uniform Platforms

Phillip Raffeck, Wolfgang Schröder-Preikschat
Friedrich-Alexander-Universität Erlangen-Nürnberg
raffeck@cs.fau.de, wosch@cs.fau.de

Peter Ulbrich
Technische Universität Dortmund
peter.ulbrich@tu-dortmund.de

***Abstract*—Dynamic migration of tasks between cores is nowadays one of the standard mechanisms of operating systems to exploit multi-core systems. However, migration is practically not used in real-time settings. This is due to the unpredictability of the associated costs and the resulting pessimistic overapproximations. Existing approaches typically rely either on a predictable, static partitioning of tasks or assume uniform costs for all cores; conceptually, migration does not differ from preemption in the latter. However, non-unified memory architecture (NUMA) and NUMA-like embedded hardware platforms are increasingly widespread. Here, intuitively, migration should be more costly than preemption, but the degree is uncertain.**

This paper aims to shed more light on two key influencing factors: (1) the variance of the elementary costs of the hardware and (2) the type and scope of the affected data of a task at the time of migration, the working set. We approach this challenge by deeper investigating application benchmarks, revisiting existing cost experiments, and bringing them to new platforms with not-so-uniform memory architectures. Our results indicate that migration differs from preemption in many relevant cases and thus requires special consideration to incorporate the associated overheads precisely into worst-case analyses.

I. Introduction

A static allocation of workloads to cores can lead to the well-known Dhall effect [1] and is therefore generally considered detrimental to utilization in multi-core settings. Nowadays, we are used to dynamically distributing workloads conveniently between cores in a multi-core system, for example, by migrating processes, containers, or even complete virtual machines. The operating system's primary challenge is to provide transparent and efficient migration mechanisms that foster CPU utilization. However, things are significantly more complicated in the domain of (hard) real-time systems. For example, although scheduling theorists have embraced transparent task migration at the instruction level, it is rarely put into practice in real-world applications. The primary reason for this is that the effects on the temporal behavior are much more complex to predict—a decades-old problem [2]. The *worst-case execution times* (WCET) of application, and operating system functionality are decisive for verifiable scheduling and typically inferred by static analysis techniques. However, the ease and difficulty of such timing analysis are dictated by the given hardware and software's predictability (and their analyzability). Even slight variabilities in execution costs can cause excessive pessimism in WCET estimates and thus jeopardize schedulability and the desired improvement in overall utilization. Conceptually, two main influencing factors can be distinguished: (1) Variable execution costs of the elementary operations caused by memory-access related latencies, i.e., the WCET varies with core allocation. (2) Migration overheads induced by the program structure of the task to be migrated. In particular, these are determined by the working set, i.e., the live part of a program's resident set, which must be transferred between cores.

We first tackled the second factor by static analysis of tasks to determine their resident (RSS) and *working-set sizes* (WSS) in previous works [3], [4]. As a result, we could reduce analysis pessimism by identifying particularly advantageous migration points with low maximum cost. Furthermore, by compiler-supported slicing of the tasks at these points, we obtained smaller jobs that are easier to allocate and schedule, fostering predictable multi-core scheduling.

Challenges and Contribution

Our previous and ongoing work is based on two key assumptions: First, the working-set size varies within the program execution so that there are beneficial migration points to identify. Second, we assume that the hardware exhibits variable access and thus elementary costs on different cores. While we could demonstrate these basic assumptions for a static allocation using core-local memories [3], the generalizability remained an open question.

On the one hand, this issue relates to the evolution of the size of the resident and working set over time in typical applications. Does its size vary within programs and during their execution?

On the other hand, the more challenging question is the variable execution cost on hardware platforms with advanced memory architectures. For example, in their study of preemption and migration delays, Bastoni et al. [5] found these converged with a task sufficiently long preempted. However, they ran their experiments on a *unified memory architecture* (UMA) machine. Because the memory access latencies are equal to all cores, the cache state (i.e., hotness) determines the access latencies. The situation is unclear in systems with less uniform memory characteristics (e.g., *non-unified memory architecture*, NUMA), where certain memory regions may not be accessible from all cores or only with significant overhead. Because of potential data transfers or memory-access overheads for the

migrated task, migration should intuitively be more costly than preemption, but the degree is uncertain.

This paper aims to shed more light on these questions by deeper investigating application benchmarks, revisiting existing cost experiments, and bringing them to new platforms. This will enable us to identify potential scenarios where consideration of migration overhead is beneficial or necessary and clear up misconceptions or uncertainties about the overhead associated with migration.

The paper provides the following contributions: (1) A compile-time analysis of working-set sizes of different benchmarks typical for the domain of real-time systems. (2) The reproduction of previously published results for preemption and migration delays in real-time systems on UMA platforms. (3) Bringing these experiments to a broader spectrum of platforms and memory architectures, namely two x86 platforms with 4 and 8 NUMA domains. (4) A more detailed insight into embedded systems with NUMA-like characteristics by comparable measurements on a typical real-time platform.

II. Approach

We aim to augment the existing data on migration overheads in a two-pronged fashion. First, we investigate the influence of migration on target platforms both with and without NUMA characteristics. Starting with a reproduction of previous results [5], our study provides a more in-depth evaluation of migration delays. Further, we analyze benchmarks at compile time to get a better understanding of how WSSs grow and shrink over the lifetime of a task.

A. Measurements

To obtain a broad overview of the costs and implications of migration, we observed task execution and memory access behavior on a range of platforms. Depending on the platform, we used different observation methods, which we briefly summarize in the following:

On all platforms, we measured memory overheads in a direct approach by recording memory access times to different memory regions and NUMA domains.

On platforms where PREEMPT_RT Linux [6] is readily available, we additionally employed an indirect method to observe migration overheads. Using the ftrace [7] functionality of the Linux kernel, we examined the runtime of a measurement task[1] embedded in specifically crafted task systems consisting of an interference task and multiple blocker tasks. By tuning the period and execution time of the interference task, we were able to trigger preemptions and migrations in the measurement task, which become visible as scheduling events in the ftrace output. The measurement task's execution times derived from these scheduling events subsequentially allowed us to conclude the overhead of preemptions and migrations.

On the embedded platform, we additionally observed the overhead incurred by placing a task's data (e.g., its stack, relevant parts of data sections) in different domains in the memory hierarchy. This setup mimics the loss of core locality due to migration, which enabled us to study two strategies: control-flow-only migration (i.e., data resides in core-local memories) and storing data in shared memory only. We then assessed the potential impacts of such strategies on task execution from the observed overheads.

[1]The measurement task is the subject of the measurement. However, in some experiments, it is also instrumented for measuring execution time.

Name	CPU	Cores	NUMA Domains
M1	Intel i7-2600	4	1
M2	AMD Opteron 6180	48	8
M3	Intel Xenon E7-4830	48	4

Table I: Overview of the x86 machines used for measurements.

B. Analysis Approach

To better understand the variation of the WSS of tasks over their lifetime, we employed a variation of the analysis routine described in [3]. Using a custom LLVM-based [8] compiler [9], [10], we determined the live-data set at each instruction for our benchmarks during compilation. Contrary to our previous work, we were not interested in identifying beneficial points in close vicinity of a target WCET, but instead in the distribution and evolution of the WSS at all points in the execution of the task.

This approach allowed us to evaluate the impact of worst-case WSS estimates on the task execution, as we gained an overview of how often these worst cases actually occur. This, in turn, provided insight into the degree of overapproximation that is introduced by pessimistic worst-case estimates of migration-induced overheads. As our analysis is performed during the compilation stage, the size estimates are based on LLVM data types in the granularity of single bits. Currently, the analysis only considers stack-based dynamic memory allocation, which we deem sufficient for the target domain of (hard) real-time systems. Even though analyzing the representative benchmarks used in our evaluation did not require it, an extension to heap-based memory management is feasible with restrictions on idiomatic C (alias problem) [11]–[13]. In sum, our approach provided valuable information to assess the impact of migration overheads.

III. Observation of NUMA-Effects

We performed experiments on various platforms to cover the characteristics of different memory architectures and their influence on migration overheads. First, as a powerful embedded platform representative, we opted for the Infineon AURIX platform, which is widely used in safety-critical automotive applications (e.g., engine and body control, collision avoidance systems). Specifically, we used the 6-core AURIX TriCore TC397XE [14] for our experiments. The platform features a diverse memory hierarchy, including core-local scratchpads and multiple levels of CPU-local (DLMUx) and domain-local memory (LMUx), as well as global extended memory (EMEM). The shared memory regions are mirrored to two different address ranges to allow cached and non-cached access. The core-local scratchpads are accessible from all cores, albeit at the cost of higher latencies. Cores 0-3 and 4-5, as well

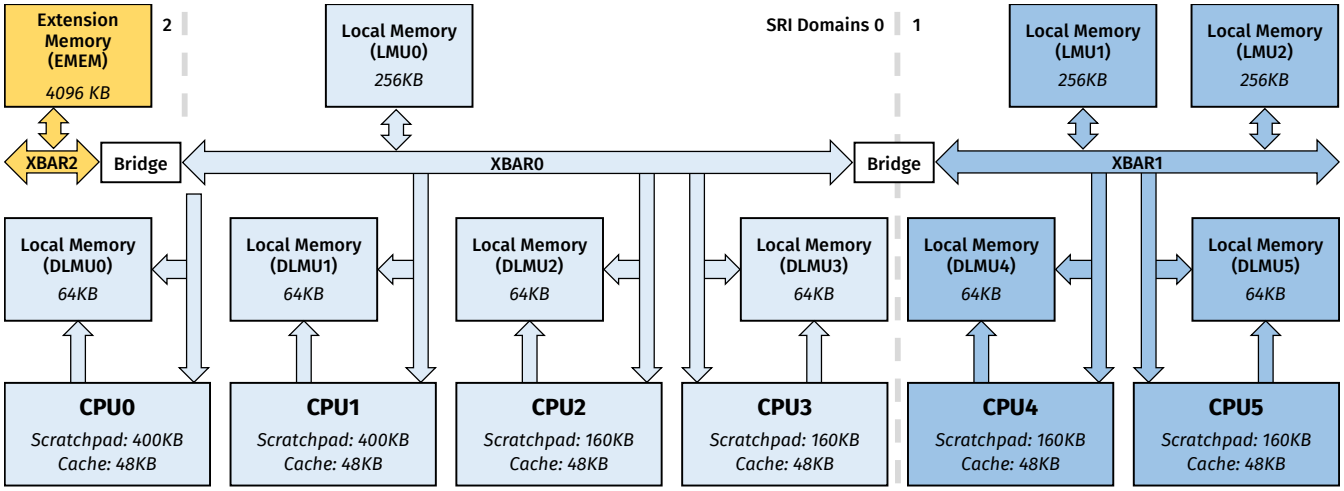


Figure 1: Block diagram of the TC3X microcontroller memory architecture. It features multiple levels with varying access latencies: from core-local scratchpad, over (domain) local memory (DLMU, LMU) up to extended memory (EMEM). Domains are marked by color, linked by the system resource interconnect (SRI, i.e., crossbar), and accessible via bridges.

as the extended memory, are connected to different crossbars (XBARx), called system resource interconnect (SRI), constituting dedicated memory domains. Figure 1 outlines the components of the memory hierarchy relevant to this work.

In the following, we first discuss the results on the AURIX platform before moving on to the second class of systems: three x86 machines with different core counts and NUMA characteristics, ranging from common desktop CPUs to many-core platforms. Table I provides an overview of the respective hardware details of the used CPUs.

A. AURIX embedded platform experimental results

Experiments were performed on an Infineon AURIX TC397XE [14] platform with the help of a Lauterbach PowerDebug [15]. Using this hardware debugger allowed us to record instruction traces in a non-intrusive manner by on-chip tracing. One limitation of this, however, is that the measurements are limited by the 1 MB trace buffer. By recording just the start and end times of the relevant code sections for the experiments, we ensured that the length of the benchmarks did not become an issue of said limited buffer size.

In the first experiment, we examined access times from one core into different parts of the memory hierarchy of the TC397XE platform. All memory regions, including scratchpads of other cores, are accessible from all cores, albeit with different latencies. To quantify these, we captured execution traces for transferring a total of 1 KiB, 8 KiB, 16 KiB and 32 KiB of memory to core 0 from the different memory regions and domains to cover all the various possible access routes through the memory interconnects showcased in Figure 1. The transfer was performed at word granularity, with the next word to transfer chosen randomly to avoid prefetching effects of linear memory accesses. In particular, we evaluated accesses from four memory regions: First, the core-local scratchpad of core 0 as the typical storage for task data. Additionally, one of the shared domain-local memory regions close to core 0, which is (D)LMU0 in Figure 1, in both the cached (LMU0) and non-cached (LMU0 NC) variant. We conducted all measurements with the later, as this research is concerned with the latencies of the memory hierarchy's levels and not the cache coherence (for which there is no HW support in the AURIX). Lastly, we used the scratchpad of core 4 as a remote memory region, constituting the worst-case scenario as it is only accessible for core 0 through the crossbar bridge and subsequently via core 4's interface.

Figure 2 gives a box plot of the determined latencies normalized to access times per word (i.e., 32 bit). Median values are marked by circles, outliers by diamonds. There are three distinct groups visible: (1) core-local and cached access to shared memory, (2) non-cached access to shared memory, and (3) access to the scratchpad of a remote core.

The measurements suggest that the execution time of a task may vary significantly on an embedded platform with NUMA-like characteristics depending on where exactly the required data resides. Especially the near doubling of access times between core-local and remote scratchpad accesses stuck out.

Considering these numbers with migration in mind, they indicate potentially significant overheads, either because the WSS of a task has to be transferred or because the execution time of a task changed due to the higher access times of memory accesses, which became remote accesses after the migration. Depending on the cache-coherency requirements and available (HW) mechanisms, moving data to shared memory is a no viable solution, as indicated by the increased access times for non-cached access to shared memory.

For the second experiment, we evaluated the execution times of tasks derived from the TACLeBench benchmark suite [16] in a simulated migration scenario. We mapped the stack and data regions used by the examined task to different memory regions during linking. On the one hand,

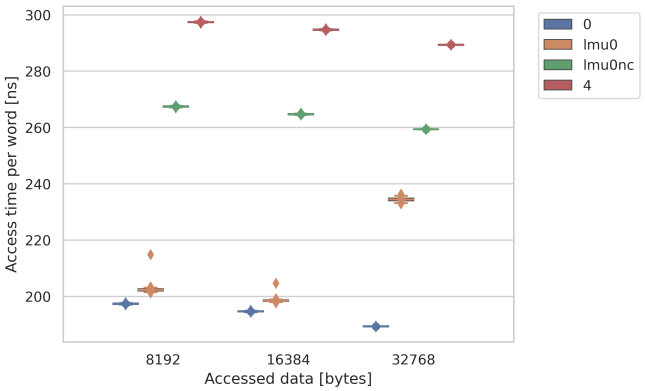


Figure 2: Access times for different parts of the memory map

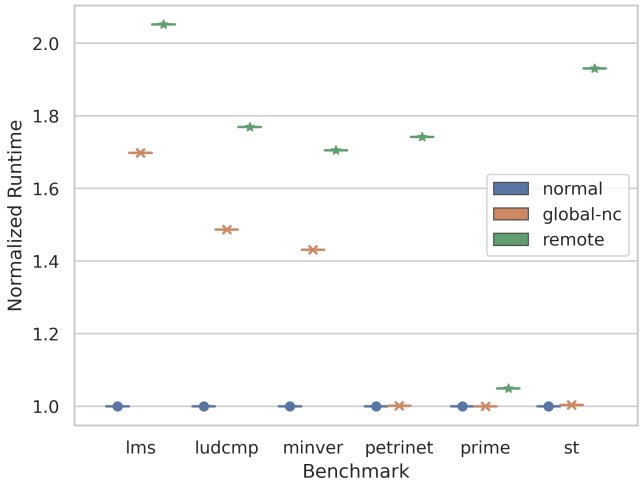


Figure 3: Runtimes for TacleBench benchmarks with their data residing in different parts of the memory map

this simulated a complete task execution after a migration to a different core. On the other hand, this allowed us to compare further the effects of attempts to circumvent problems coming with migration by moving data to global shared memory.

Figure 3 showcases execution times of several benchmarks in three different memory configurations as a boxplot. The data used by the tasks lay either in the scratchpad of core 0 (normal case), in the non-cached shared memory (global-nc), or in the scratchpad of core 4 (remote). The median values of the measurements are marked by a circle (normal), a cross (global-nc), and a star (remote), respectively. For easier comparison, the execution times are normalized to the standard case.

Again, we noticed significant differences in the execution times depending on where the task data resides in memory. These are more or less pronounced depending on the specific benchmark, its memory footprint, and WSS access patterns.

Two conclusions can be drawn from the results, albeit still depending on the concrete nature of the tasks:

(1) Operating entirely in shared memory may come with significant overheads. This signifies the reality of scenarios where migration is beneficial or necessary.

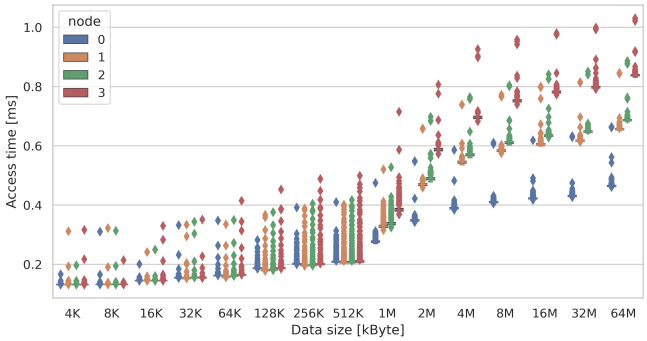


Figure 4: Memory access times from different NUMA domains on `M2`. Benchmark run on domain 0.

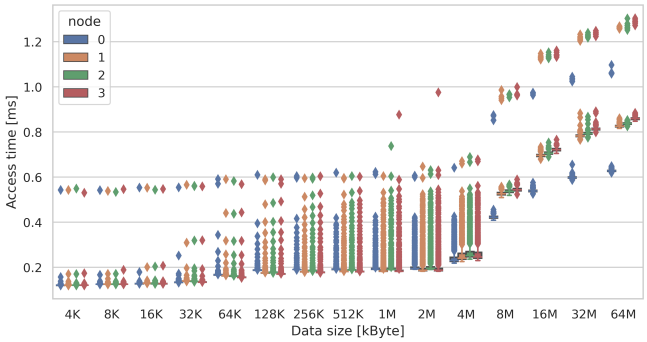


Figure 5: Memory access times from different NUMA domains on `M3`. Benchmark run on domain 0.

(2) Depending on the target core, migration comes with substantial costs. Thus, we mandate predictable migration with precise WSS estimation to manage the overhead.

B. x86 NUMA platforms experimental results

On the three PC platforms, we evaluated memory access times by performing 4096 reads and writes at random indices of an increasingly larger array and controlling the memory allocation of the said array via `numactl`[2]. Figures 4 and 5 display the measured access times as boxplots. For better readability, only 4 of the 8 NUMA domains of `M2` are displayed, as the results are equivalent for the other domains. We can see larger access times across NUMA domains for larger array sizes on both machines.

Additionally, we ran PREEMPT_RT Linux [6] to leverage the Linux tracing functionality [7]. For a finer resolution of the results, we splitted the existing `sched_switch` event in two, `sched_switch_on` and `sched_switch_off`. Evaluations were performed using the `nop` tracer, the `TSC` as the clock source, and with only our scheduling events enabled. Real-time throttling was disabled during the measurements.

The task system under observation comprised one high-priority measurement task and one interference task for each processor core, which all perpetually accessed a WSS the

[2]https://github.com/numactl/numactl

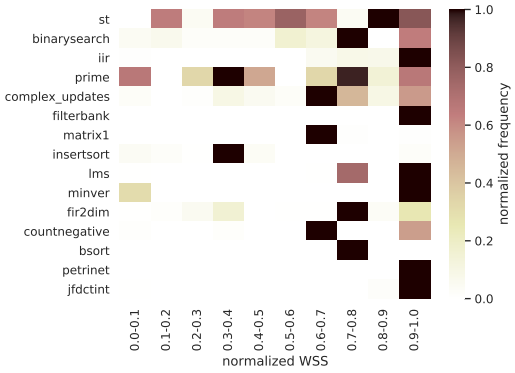


Figure 6: WSS distribution of TACLeBench benchmarks

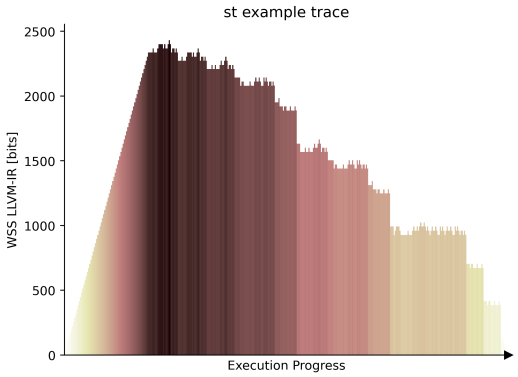


Figure 7: WSS distribution over an exemplary trace of the benchmark `st`

same size. We experimented with two selected microbenchmarks as measurement tasks: one processor-bound (`CPU`) and one memory-bound benchmark (`MEM`). The processor-bound benchmark calculates primes up to two alternating limits. The memory-bound benchmark moves sequentially through a working set of predefined size reading bytes with a step width of 32 B, each read immediately followed by a write to the following byte. We use 2 KiB as a small, 32 KiB as a medium, and 786 KiB as a large WSS. Compared to the results of our analysis of embedded benchmarks (see Section IV), these constitute rather large WSS. To activate worst-case behavior, the measurement task switched CPU every few iterations as indicated by the different colors in Figures 8 to 12.

The traced execution times of the measurement task during these experiments are displayed in Figures 8 to 11. We identified two execution-time clusters for the CPU benchmark on all machines, which fit the code's behavior. `M3` shows a pattern of execution time spikes followed by regions of lower execution times. The effect is better observable in a zoomed-in comparison (see Figure 12): For the MEM benchmark, higher costs are visible at switches to a new processor, followed by faster executions as the caches warm up. At the beginning and the crossing of NUMA domains, distinct spikes are seeable, indicating higher overheads for crossing NUMA domains. The comparison with the execution times of the CPU benchmark shows that memory access latencies are responsible for the observed pattern. Additionally, we evaluated a WSS of 8 MiB for `M3` as the size the access experiments (Figure 5) where NUMA effects become more prevalent. In comparison with Figures 11c and 12b, we can identify higher cost after switching NUMA domains but not after switching cores, indicating that the influence of caches declines at such huge WSSs, while the NUMA influence remains. For `M1`, no clear effect distinguishes preemption and migration, reaffirming the observations of Bastoni et al. [5]. The effect is less prevalent on `M2`, indicating that NUMA architectures do not necessarily come with detrimental overheads.

In summary, these results suggest that dissimilarities between preemption and migration are likely more prevalent in NUMA systems, requiring the explicit consideration of migration for sound and precise WCET estimates.

IV. Distribution of Working-Set Sizes

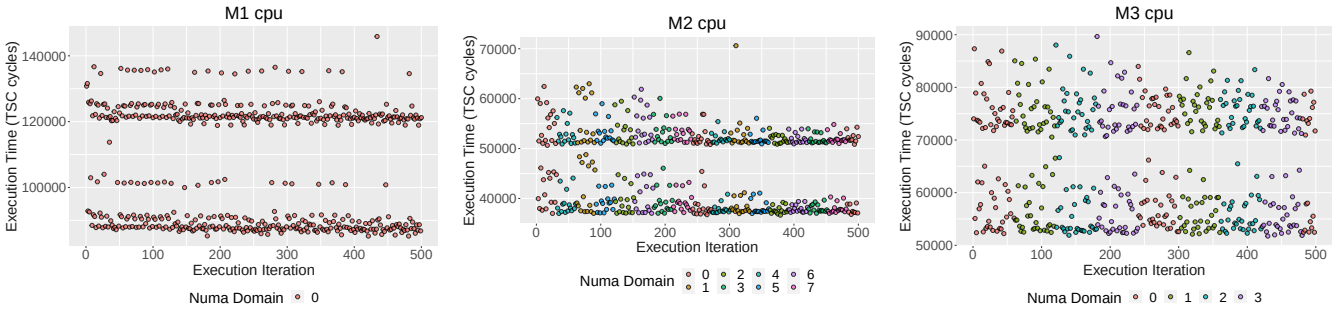
The results of Section III imply that restricting migration to smaller WSSs is beneficial. Thus, we studied the evolution of WSSs over the lifetime of a task and applied our analysis (cf. Section II-B) on TACLeBench benchmarks [16], which serve us as representatives for typical application patterns.

Figure 6 shows a heat map of the resulting WSS distribution. For easier comparison, the results are normalized twice: first, for each benchmark (i.e., each row), the WSS is normalized to the interval of its minimum and maximum size. Second, the occurrence frequency is normalized in the same way. Each column represents a tenth of the WSS interval, while the color of each field indicates how many of the observed WSSs for the benchmark lie within the interval of the column; darker colors denote a higher frequency. By our analysis, we found all working sets in absolute numbers to be smaller than 10 KiB.

The distribution shows no clear trend across all benchmarks but rather a high degree of variation. While some benchmarks (e.g., filterbank, jfdctint, petrinet) exhibit a WSS equivalent to near the worst-case estimate most of the time, others (e.g., prime, st) show a wider distribution and several peaks over different WSS ranges.

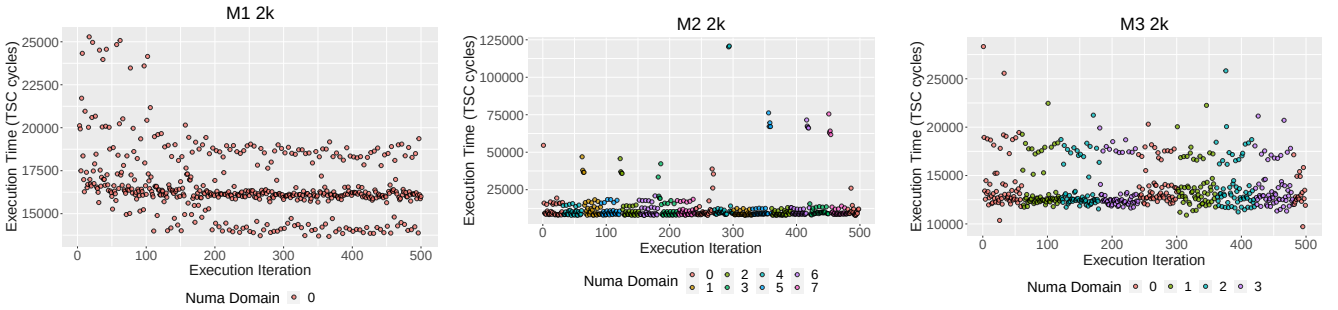
As an example, Figure 7 shows the progression of the WSS over one possible execution trace of the `st` benchmark. Every bar represents the WSS at one instruction. Instructions are ordered from left to right by their time of occurrence in the trace. Note that the selected trace does not necessarily represent the worst-case execution but rather just one potential execution. As the figure shows, there are notable differences in the WSS throughout the execution, indicating that the data to be transferred in the case of migration strongly depends on the exact time the migration takes place.

Two conclusions can be drawn from these analysis: (1) Whether the worst-case WSS is representative of a task execution strongly depends on the nature of the task, as it may as well access only a far smaller WSS most of its time. (2) Predictable migration is essential to improve worst-case analyses by avoiding unnecessary pessimistic WSS estimates.



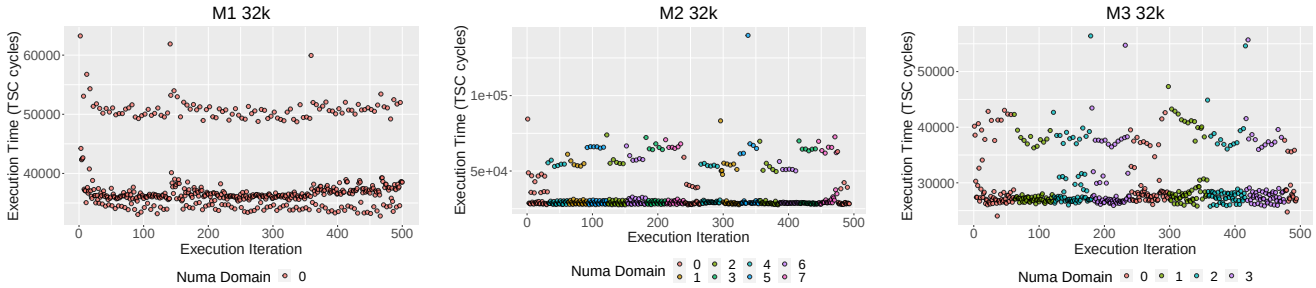
(a) M1 (b) M2 (c) M3

Figure 8: Trace results for CPU.



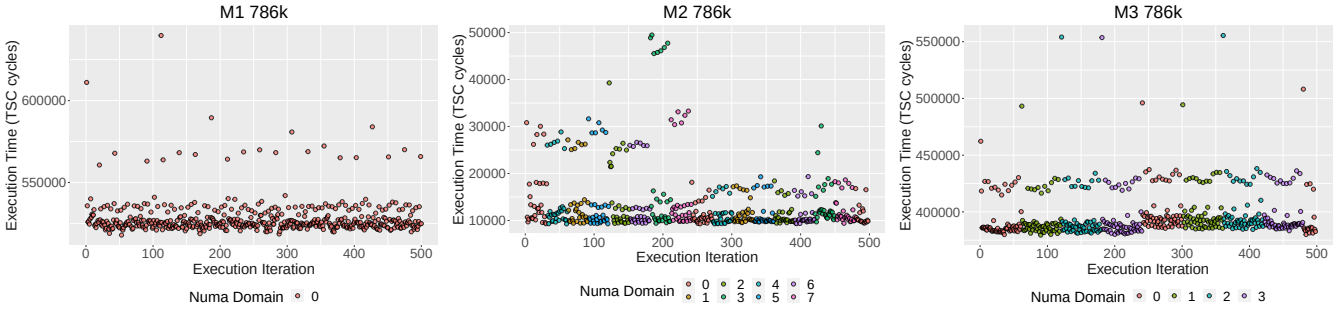
(a) M1 (b) M2 (c) M3

Figure 9: Trace results for MEM with a WSS of 2k.



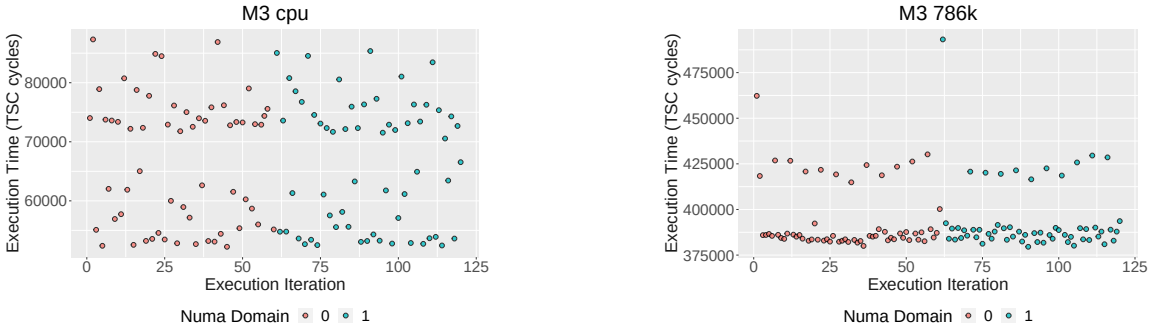
(a) M1 (b) M2 (c) M3

Figure 10: Trace results for MEM with a WSS of 32k.



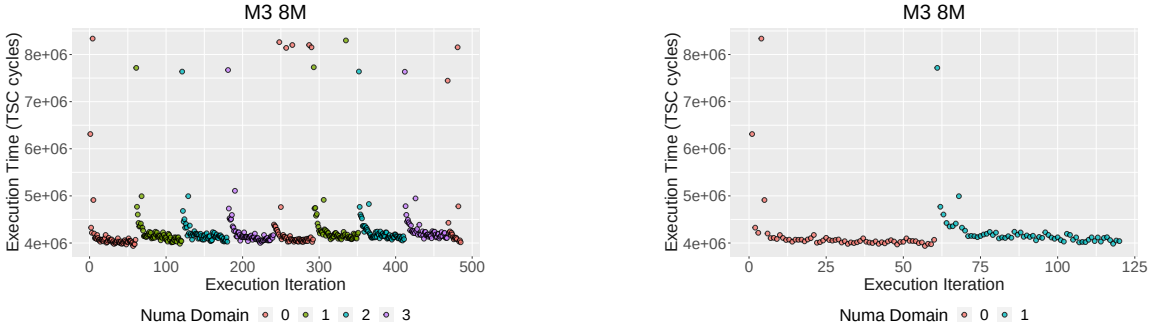
(a) M1 (b) M2 (c) M3

Figure 11: Trace results for MEM with a WSS of 786k.



(a) CPU (b) MEM 786k

Figure 12: Zoomed version of the trace results on `M3`.



(a) Trace results. (b) Zoomed version of the trace results.

Figure 13: Trace results for MEM with a WSS of 8M on `M3`.

Knowledge of the exact point in time a migration happens facilitates precise WSS and, hence, overhead estimation.

V. Related Work

Since the beginning of multitasking, working-set estimation has been a research topic with a large body of work [17]–[19], especially in (virtual) memory management. Brown et al. [20] presented a technique for working-set prediction to make thread migration more efficient. They highlight the need to precisely identify the actual current working set to prevent unnecessary data transfers. In real-time systems, Calandrino et al. [21] and Bastoni et al. [5] identified the WSSs to impact the worst-case timing analysis significantly. Beyond these general investigations, we have studied the specific properties of typical application benchmarks in this work.

Likewise, preemption and migration costs are a vast area of research. For example, Bastoni et al. [5] measured preemption and migration delays on a 24-core UMA machine. They observed no significant differences between preemptions and migrations in systems under load, as with increasing preemption length, cache affinity is lost either way completely. Contrary, in a comparable experiment, Calandrino et al. [21] observed worst-case migration costs to be higher due to forced data invalidation and cache-coherency overheads. Work on cache-related preemption delays (CRPD) [22], [23] aims to determine the effects of preemption more accurately or minimize them systematically. However, we are not aware of any work that considers migration between NUMA domains.

VI. Conclusion & Outlook

Using the tracing capabilities of PREEMPT_RT Linux, we were able to validate previous results for preemption and migration overheads and extend them to the broader range of NUMA and embedded NUMA-like platforms. Our results substantiate the general intuition that migration and preemption deserve nuanced consideration in such scenarios, as the former is associated with more uncertainties and costs.

Further, static analysis of representative application benchmarks indicates that the (worst-case) resident set is an extensive overapproximation of the actual working set for most of the execution. Migration at predictable points in the execution, thus, helps to avoid unnecessary pessimism, as the WSS can be determined more precisely section by section.

Overall, we conclude that, generally, migration cannot be treated like preemption. Especially on NUMA-like systems, migration requires special consideration to precisely determine and incorporate overheads in the system design.

In previous work [4], we outlined possible approaches to enable predictable migration with known overheads by static and dynamic scheduling. The results presented in this paper will help us to refine these approaches and put them into practice.

ACKNOWLEDGMENT

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project numbers 146371743;198891422.

REFERENCES

[1] S. K. Dhall and C. L. Liu, "On a Real-time Scheduling Problem," *Operations Research*, vol. 26, no. 1, pp. 127–140, 1978.

[2] D. S. Milojičić, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou, "Process migration," *ACM Computing Surveys*, vol. 32, no. 3, pp. 241–299, 2000.

[3] T. Klaus, P. Ulbrich, P. Raffeck, B. Frank, L. Wernet, M. R. von Onciul, and W. Schröder-Preikschat, "Boosting Job-Level Migration by Static Analysis (Best Paper Award)," in *Proc. of the 15th Intl. Work. on Operating Systems Platforms for Embedded Real-Time Applications*, 2019, pp. 33–44.

[4] P. Raffeck, P. Ulbrich, and W. Schröder-Preikschat, "Work-in-progress: Migration hints in real-time operating systems," in *Proc. of the 40th IEEE Intl. Real-Time Systems Symp.* IEEE, 2019, pp. 528–531.

[5] A. Bastoni, B. Brandenburg, and J. Anderson, "Cache-related Preemption and Migration Delays: Empirical Approximation and Impact on Schedulability," in *Proc. of the 6th Intl. Work. on Operating Systems Platforms for Embedded Real-Time Applications*, 2010, pp. 17–22.

[6] Real-time linux. [Online]. Available: https://wiki.linuxfoundation.org/realtime/start

[7] Ftrace - function tracer. [Online]. Available: https://www.kernel.org/doc/Documentation/trace/ftrace.txt

[8] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. of the Intl. Symp. on Code Generation and Optimization*, Washington, DC, USA, 2004, pp. 75–86.

[9] F. Scheler and W. Schröder-Preikschat, "The Real-time Systems Compiler: Migrating Event-triggered Systems to Time-triggered Systems," *Software: Practice and Experience*, vol. 41, no. 12, pp. 1491–1515, 2011.

[10] F. Franzmann, T. Klaus, P. Ulbrich, P. Deinhardt, B. Steffes, F. Scheler, and W. Schröder-Preikschat, "From intent to effect: Tool-based generation of time-triggered real-time systems on multi-core processors," in *Proc. of the 19th IEEE Intl. Symp. on OO Real-Time Distributed Computing*. Washington, DC, USA: IEEE, May 2016, pp. 134–141.

[11] M. Stilkerich, J. Schedel, P. Ulbrich, W. Schröder-Preikschat, and D. Lohmann, "Escaping the bonds of the legacy: Step-wise migration to a type-safe language in safety-critical embedded systems," in *Proc. of the 14th IEEE Intl. Symp. on OO Real-Time Distributed Computing*, G. Karsai, A. Polze, D.-H. Kim, and W. Steiner, Eds. IEEE, Mar. 2011, pp. 163–170.

[12] I. Stilkerich, C. Lang, C. Erhardt, and M. Stilkerich, "A practical getaway: Applications of escape analysis in embedded real-time systems," in *Proc. of the 14th ACM SIGPLAN/SIGBED Conf. on Languages, Compilers and Tools for Embedded Systems*, 2015, pp. 1–11.

[13] C. Lang and I. Stilkerich, "Design and implementation of an escape analysis in the context of safety-critical embedded systems," *ACM Trans. on Embedded Computing Systems*, vol. 19, no. 1, 2020.

[14] *Aurix Tc3xx User Manual Part 1*, Infineon, 2020, v1.6.0. [Online]. Available: https://www.infineon.com/dgdl/Infineon-AURIX_TC3xx_Part1-UserManual-v01_00-EN.pdf?fileId=5546d462712ef9b701717d3605221d96

[15] Lauterbach power debug interface usb3. [Online]. Available: https://www.lauterbach.com/powerdebugusb3.html

[16] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sørensen, P. Wägemann, and S. Wegener, "TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research," in *Proc. of the 16th Intl. Work. on Worst-Case Execution Time Analysis*, ser. OpenAccess Series in Informatics (OASIcs), M. Schoeberl, Ed., vol. 55. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, pp. 2:1–2:10.

[17] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 10th ed. John Wiley & Sons, 2012.

[18] P. Denning, "Working sets past and present," *IEEE Trans. on Software Engineering*, vol. SE-6, no. 1, pp. 64–84, 1980.

[19] P. Bryant, "Predicting working set sizes," *IBM Journal of Research and Development*, vol. 19, no. 3, pp. 221–229, 1975.

[20] J. A. Brown, L. Porter, and D. M. Tullsen, "Fast thread migration via cache working set prediction," in *IEEE 17th Intl. Symp. on High Performance Computer Architecture*, 2011, pp. 193–204.

[21] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson, "LITMUSˆRT : A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers," in *Proc. of the 27th IEEE Intl. Real-Time Systems Symp.*, Dec. 2006, pp. 111–126.

[22] L. Ju, S. Chakraborty, and A. Roychoudhury, "Accounting for cache-related preemption delay in dynamic priority schedulability analysis," in *2007 Design, Automation Test in Europe Conf. Exhibition*, 2007, pp. 1–6.

[23] R. Mancuso, H. Yun, and I. Puaut, "Impact of DM-LRU on WCET: a Static Analysis Approach," in *31th Euromicro Conf. on Real-Time Systems*, ser. Leibniz Intl. Proc. in Informatics (LIPIcs), S. Quinton, Ed., vol. 107. Stuttgart, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, conference, pp. 17:1–17:25.

X-RIPE: A Modern, Cross-Platform Runtime Intrusion Prevention Evaluator

Gabriele Serra
Scuola Superiore Sant'Anna
Pisa, Italy
gabriele.serra@santannapisa.it

Sandro Di Leonardi
Scuola Superiore Sant'Anna
Pisa, Italy
sandro.dileonardi@santannapisa.it

Alessandro Biondi
Scuola Superiore Sant'Anna
Pisa, Italy
alessandro.biondi@santannapisa.it

***Abstract*—The complexity of modern software systems, the integration of several software components, and the increasing exposure to public networks are making systems more and more susceptible to cyber-attacks. Operating systems and drivers are typically written in C or C++, which are known to be memory-unsafe languages. As a matter of fact, buffer overflows are still a plague in modern software. Despite the consistent amount of research carried out to counteract cyber-attacks, it is still not straightforward to evaluate the worthiness of a specific countermeasure or to identify the appropriate configuration for existing protection tools. In this paper, we present *X-RIPE*, a modern and cross-platform version of the Wilander and Kamkar's RIPE testbed. The objective of X-RIPE is to evaluate how a given countermeasure behaves against buffer overflow threats. We tested X-RIPE against modern memory-corruption protection techniques supported by GCC and Clang/LLVM compilers such as Stack Protector, ASan, and ARM's Pointer Authentication.**

***Index Terms*—security, evaluator, stack-overflow, ripe**

I. Introduction

Software security is a primary requirement for modern systems. Operating systems, especially those developed for embedded systems, are typically written in C or C++. Undoubtedly, these languages offer flexibility and high performance, and, in many cases, they are often the only language supported by the toolchain provided by hardware manufactures for the target platform. However, C and C++ are known to be memory-unsafe languages. Application and operating systems written using memory-unsafe languages could be the target of memory error exploitations [1]. Historically, the adoption of memory protection support mechanisms (MPU) and memory virtualization support mechanisms (MMU) has allowed operating systems to counter several attacks, especially those targeting code injection through memory corruption. Nonetheless, buffer overflows are still present in modern software. Even considering classic buffer overflows only, this class of memory corruption has kept its position on the podium of the *Common Weakness Enumeration* (CWE) SANS top 25 most dangerous software errors for years. In 2022, *Improper Restriction of Operations within the Bounds of a Memory Buffer* laid again at the first place of the CWE SANS ranking [2]. The latest eminent example dates back to January 2022 and was discovered by Qualys Security Advisory [3]. Briefly, they found a buffer-overflow in the C arguments support used by *Polkit* (formerly PolicyKit) that leads to a local privilege escalation from any user to root. Polkit is a component for controlling system-wide privileges in Unix-like operating systems and is installed by default on all major Linux distributions. Interestingly, this recently-discovered vulnerability is technically a memory corruption exploitable since 2009 and has remained latent until 2022. The preceding example demonstrates that buffer overflows and memory-corruption vulnerabilities are still alive and that the induced problem is far from resolved. As a matter of fact, academic and industrial security researchers are still focused on creating countermeasures that can eventually be implemented at the production level. While most of today's defense methods can be seamlessly evaluated from the point of view of run-time overhead, there is no standard benchmark that allows assessing the robustness of a countermeasure. Hence, researchers often resort to qualitative security analysis for a specific technique. In 2003, Wilander and Kamkar [4] developed an elementary tool used to perform a comparative evaluation on run-time buffer overflows targeting 20 different attack combinations. In 2011, Nikiforakis, together with Wilander, leveraged that original idea to develop *Runtime Intrusion Prevention Evaluator* (RIPE), a testbed suite comprising more than 800 buffer overflow combinations. The embryonic tool from 2003 and the subsequent RIPE were used to demonstrate the effectiveness of several tools and techniques, including [5] [6] [7] [8] [9]. RIPE was released under the MIT license in an attempt to standardize the comparison between different countermeasures; however, it targets only the `i386` processor architecture. Practically, there is no benchmark targeting different architectures that allows evaluating countermeasures with the same yardstick and targeting different architectures or environments. In this paper we present X-RIPE, a modern *cross-platform Runtime Intrusion Prevention Evaluator*. X-RIPE is a revamp of the early RIPE project, and it is designed to target multiple processor architectures. X-RIPE already supports `i386`, `x86-64` and `aarch64`. The main objective of X-RIPE is to provide a quantitative evaluation of the protection coverage offered by a specific mechanism against buffer overflows. The tool is released under the GPL license, with the hope to serve as the foundations for a future comprehensive standard penetration test against memory corruption. To test X-RIPE, we applied it against a few modern memory-corruption protection techniques supported by GCC and Clang/LLVM compilers, such as Stack Protector, Address Sanitizer (ASan),

and ARM's Pointer Authentication.

Contribution. In summary, this work makes the following contributions:

- It presents X-RIPE, a cross-platform testbed to quantitatively and systematically evaluate the robustness of a buffer-overflow prevention technique.
- It reports on an experimental evaluation that was conducted by applying X-RIPE against anti-memory-corruption techniques supported by the widely-used compilers GCC and Clang/LLVM, specifically: Stack Protector, ASan, and ARM's Pointer Authentication.

Paper structure. The remainder of this paper is organized as follows. Section II reviews the related work. Section III presents an overview of the X-RIPE's architecture. Section IV lists the countermeasures we tested against X-RIPE and their working principles. Section V states results of our experimental evaluation and Section VI concludes the paper.

II. Related work

Software systems are growing in size and complexity, hence the number of bugs. Memory corruption vulnerabilities are among the most frequent potential problems. Dangling pointers, heap meta-data overwrites, uninitialized reads, and invalid or double frees are all examples of these problems. As a result, researchers designed several kinds of protection techniques. Consequently, together with databases of vulnerabilities, testbeds were also developed over the years to measure the effectiveness of those defense techniques.

Among others, SARD (Software Assurance Reference Dataset) [10] is a growing database maintained by NIST (The National Institute of Standards and Technology) of approximately 170 000 programs with a set of known security flaws. These test cases are designs, source code, and binaries from all the phases of the software life cycle: they are mainly written in C, C++, Java, PHP, and C# and cover over 150 vulnerabilities. The dataset includes production, synthetic and academic test cases. The dataset intends to encompass various possible vulnerabilities, languages, platforms, and compilers. Users can view test cases and test suites via the SARD online interface or search for test cases by vulnerability kind, name, size, description words, and other parameters. Many cases include comparable good cases to test for false positives, in which flaws are rectified. In SARD, each test case is described by employing metadata, which encompasses most information regarding the specific flaw or defect. Weaknesses are classified using the Common Weakness Enumeration (CWE) ID and name. The SARD database is archival, which means that once a case is added, it cannot be modified or removed. However, if there are issues with a case, it may be tagged as deprecated and a replacement added. Because most defects are stored in metadata, the findings may be reviewed semi-automatically, displaying the kind of bugs that a tool finds and the false positive rate.

Besides test databases, other research teams tried to standardize test benchmarks for emerging platforms. The leading example is *RIPE-ARM*, an implementation of the RIPE benchmark targeting ARM v7 (32 bit) platforms [11] developed in 2020. In their work, Zhou and Chen performed an experiment using their RIPE-ARM against a Raspberry Pi emulated employing QEMU. Unfortunately, their RIPE-ARM was not publicly released; hence, it has been impossible to take advantage of their implementation. In 2022, Calatayud and Meany worked on a comparative analysis of buffer overflow vulnerabilities in high-end IoT devices. Their analysis still targets 32-bit operating systems [12]. The authors modified the original RIPE by replacing the shellcode for code injection attacks and made that shellcode available to the community. Their work, however, is not a full-fledged benchmark platform, but it is tightly coupled with their analysis; thus, it cannot be generalized. Furthermore, it still targets the ARM v7 architecture. Finally, it is worth mentioning RetTag [13], a hardware-assisted hijacking defense method addressing RISC-V platforms. RetTag leverages the ARM's Pointer Authentication design to enforce pointers' integrity. To perform the security analysis of their technique, they wrote a port of RIPE for RISC-V based platforms. Still, the code is not publicly available.

On the opposite side, several works tried to standardize evaluation methods for defenses from a qualitative point of view. A notable example is [14]. In the mentioned work, the author tried to formalize the general requirements that a protection technique shall implement, such as interoperability with legacy software, scalability, and low-performance overhead. Then, the set of derived requirements was applied to widespread protection techniques. Analyzing 24 various buffer overflow protection strategies using the proposed qualitative methodology, the work outputs a report that summarizes the pros and cons of each of these mechanisms.

III. X-RIPE

X-RIPE is structured as two-layer software, the frontend and the backend. Unlike its predecessor, the backend is independent of the underlying architecture. The internal structure is organized using the *facade* design pattern, thus unifying the architecture-dependent components under a standard API. The backend logic lies on top of a hardware abstraction layer in substance. The X-RIPE backend has an attack generator that builds the attack payload and performs the attack on itself. The backend, indeed, contains vulnerable buffers, gadgets and all the logic to calculate offsets. The X-RIPE benchmark is released under the GPL license and available on Github[1]. The original repository was forked to keep the history of the initial benchmark and incorporate the necessary code to realize the hardware abstraction layer.

A. Tool frontend

The tool frontend is written in Python and consists of a script that iterates all different kinds of attacks. In principle, the frontend allows running all the possible attack forms. The script must be invoked by specifying the available overflow

[1]https://github.com/gabriserra/RIPE

technique among direct, indirect or both. Furthermore, the number of times each attack should be launched and which compiler to target (GCC or Clang) can be specified. The last parameter is optional and is used to control the output format.

```
ripe_tester.py
    <direct|indirect|both>
    <num of repetitions>
    <gcc|clang|both>
    [verbose-options]
```

For each test performed, the result log is marked with one possible outcome: `OK` when the attack was executed successfully, `FAILED` when the attack encounters an error before running to completion, `PARTIAL` when attacks did not succeed in each round, or `NOT POSSIBLE`, when the attack is not practically possible (e.g., a direct attack on a stack buffer targeting a global pointer). The frontend is instructed to explore all the attack space available in the backend.

B. Tool backend

The tool backend is written in C and consists of an attack generator. Briefly, it can prepare a malicious payload and perform the attack on itself. The tool is self-contained; namely, it contains both the code to prepare malicious payload and the required vulnerable buffers and gadgets. The attack space has five different dimensions; hence all the possible kinds of attacks generated by the tool are vectors of five components. The exploration of all the attack space combined with the two available techniques (direct and indirect) gives over 2000 various possible attacks.

The dimensions of the attack space are I) the overflow technique, II) the attack code, III) the target code pointer, IV) the memory location, and V) the vulnerable function.

C. Overflow location

The attack location describes the memory section in which the target buffer is located. RIPE supports attacks on the stack, heap, data, and BSS sections.

D. Target function

There are ten vulnerable functions available as attack vectors: `str(n)cpy`, `str(n)cat`, `s(n)printf`, `memcpy`, `homebrew`, `sscanf`, `fscanf`.

The C library string functions allow copying part of a string from a source buffer to a destination buffer without any control on the destination buffer limit. X-RIPE uses them to overflow the destination buffer. The n-version of the C string functions, such as `strncpy`, instead requires to specify of the destination buffer's size. However, it is up to the developer to provide the destination buffer size. Most of the time, that size is calculated dynamically. Therefore, an error in the size computation can frustrate the limit check offered by those functions. The same is true for functions that require the string format. Indeed, an error in providing the format can let the overflow happens. Concluding, in the list of functions, there is also `homebrew`, a loop-based equivalent version of `memcpy`, implemented originally in the previous version of RIPE.

E. Attack code

The attack code represents the kind of shellcode used when the attack occurs. Differently from the original RIPE, our version offers only a shellcode that spawns a shell through `execv` syscall. However, the shellcode is provided in three different flavors.

- Plain shellcode: A shellcode that spawns a shell using `execv` syscall
- Shellcode with NOP sled: The plain shellcode is padded using `NOP` instructions
- Shellcode with polymorphic NOP sled: The basic shellcode is padded using instructions that are equivalent to `NOP` instructions. The polymorphic sled has been generated using the Metasploit framework [15].

Furthermore, X-RIPE offers an additional way of spawning the system shell that does not depend on the provided shellcode avoiding injecting code. The supplementary attack codes started to be carried out after countermeasures such as Data Execution Prevention (DEP) became popular.

- Return-to-libc: Instead of injecting a shellcode, X-RIPE tries to jump to existing libc function, such as `system`
- Return Oriented Programming: X-RIPE uses some instructions already available in the program (gadgets) and combine them to spawn a shell

These last two attack vectors are advanced and more challenging to implement. Therefore, X-RIPE can determine the addresses of libc functions and, additionally, it uses gadgets placed on purpose in the program code.

F. Target code pointer

The target code pointer represents the address exploited by the specific attack. It transfers the control flow to the appropriate offset to trigger the shellcode.

- Previous frame pointer: The address of the frame pointer pushed into the stack, which is used to reference function arguments and local variables.
- Function return address: The return address pointer pushed into the stack is used to jump back to the caller function when the callee terminates
- Longjump buffer: The buffer used to store the current instruction pointer when calling `setjmp`.
- Function pointer: A variable that contains the address of a function callable dynamically.
- Structure with function pointer: A function pointer part of a structure laying adjacent to a buffer.

Clearly, 'Previous frame pointer' and 'Function return address' are stack specific, and this means that they can be used as a target code pointer only when the attack location is the stack. All other targets instead can be allocated in each location.

IV. Tested Defenses

The buffer overflow issue became known and was publicly disclosed in early 1972 by the Computer Security Technology Planning Study [16]. The ability to gain the control of a process by overwriting data, however, received worldwide attention thanks to the Morris worm in 1988 [17]. Since then, defending from buffer overflows attacks has been part of security research and several techniques have been developed. For the reasons stated in Section I, C and C++ are notably the most used languages used to develop operating systems and drivers. One aspect that makes those languages flexible is the lack of a (rich) runtime environment. Accordingly, standard versions of C and C++ do not have any memory-bound checking. This design choice is the basis of the portability of the language but, on the other hand, made buffer overflow attacks possible. Consequently, most defense techniques developed over the years can be categorized under two groups, **(i)** mechanisms that introduce bound checking and **(ii)** mechanisms that prevent the consequences the exploitation. Among others, production-ready systems tend to mostly use only those techniques that, during the years, were seamlessly integrated with popular OSes such as Linux and compilers such as GCC or Clang/LLVM. Therefore, we chose to evaluate X-RIPE against the common techniques supported by GCC and Clang/LLVM, namely: Stack Guard, ASan, and ARM's Pointer Authentication. In this section, we are providing a brief analysis of each technique.

A. Stack Protector

At the beginning of the 2000s, Etoh et al. from IBM [18] suggested a modification to the GCC compiler to protect against stack overflows. The main idea was to place a randomly-generated integer, named `canary`, between any stack-allocated buffers and the return address saved on the stack. The name 'stack canaries' is due to their analogy to coal mine canaries, given that they are used to detect whether it is safe to carry on the program execution. GCC Stack Protector inserts stack canaries on the stack of certain functions and is still used today due to its simplicity and the low overhead introduced at runtime. As illustrated in Figure 1, the canary is placed right after local variables in the current implementation, protecting both the old base pointer and the return addresses from direct overflows. Furthermore, the mechanism arranges the local stack variables to ensure that char buffers are always allocated next to the canary. The latter assumption prevents a direct overflow to corrupt any other local variable.

B. AddressSanitizer (ASan)

The *AddressSanitizer* (also known as ASan) is an open-source memory error detector originally introduced by Serebryany et al. from Google [19]. ASan works as a compiler instrumentation module and is currently implemented in Clang (starting from version 3.1 [20]) and GCC (starting from version 4.8 [21]). ASan targets the most common architectures, including x86 and ARM (both 32- and 64-bit versions of architectures). The tool consists of a compiler pass and the related runtime library. It was designed to find and catch memory errors such as use after free, heap/stack/bss overflows, use after return or scope, etc.

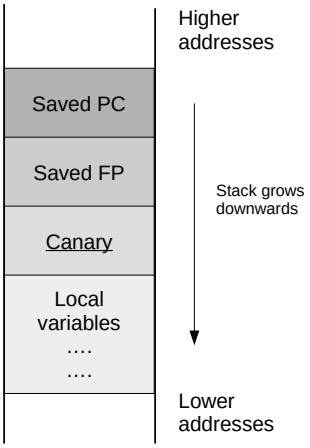


Fig. 1. Conventional layout of a stack frame when stack protector is enabled.

The basic idea of ASan is to divide the virtual address space into two disjoint classes, the main application memory `Mem` and the shadow memory `Shadow`. The regular application code uses the main application memory. On the other hand, the shadow memory consists of a memory area hidden from the application and used to record information about the main memory. The shadow memory contains the shadow values, namely a set of shadow bytes. Shadow bytes, indeed, are mapped to one or more bytes in the main memory. ASan maps 8 bytes of the application memory into 1 byte of the shadow memory. Therefore, the two classes of memory have a correspondence that is built in such a way that computing the shadow memory mapping `mem_to_shadow` is fast. ASan also introduced the idea of poisoned bytes. Poisoned bytes (or *redzones*) are memory areas that cannot be referenced. ASan runtime library can detect accesses to redzones; hence the wider the redzone, the larger the overflows or underflows that will be detected. Poisoning a byte of the memory will result in a particular value written into the corresponding shadow memory.

During the compiler pass, functions such as `malloc` and `free` are replaced with a customized implementation that allocates extra poisoned bytes around the allocated memory region. Furthermore, each memory access that involves a reference to the pointer is transformed. The memory around the area accessed is poisoned too. To make a simple example, suppose that the program accesses a pointer as follows:

```
*address = ...;  // or: ... = *address;
```

If the same program is instrumented by means of ASan, the compiled code would result in:

```
shadow_addr = mem_to_shadow(address);

if (shadow_is_poisoned(shadow_addr))
{
    reportError(address);
}

*address = ...;  // or: ... = *address;
```

That instrumentation causes a runtime error report when the accessed address is not legal.

C. ARM's Pointer Authentication

The relevance of security in modern operating systems pushed chip designers to introduce several security-related hardware facilities in their processors. A relevant example is the feature included by ARM in version 8.3 of their ARMv8 processor architecture named Pointer Authentication (PA). Another relevant example is the ARM's Branch Target Indicator (BTI) feature, introduced since version 8.4 of ARMv8. ARM is not the only manufacturer that has invested in this direction. In recent years, Intel has proposed a similar architectural extension called Control-flow Enforcement (CET). The facilities introduced by ARM and Intel can be used to realize robust control-flow integrity enforcement. Almost all CFI techniques watch over a program execution to ensure that the target of an indirect branch is the intended instruction. Among others, they verify that the control comes back to the calling function at each function. Since control-flow hijacking is essential in many exploits (e.g., including those based on buffer overflows), independently of the exploited vulnerability [22], CFI techniques proved to be effective against several widespread attacks and are considered among the most advanced security countermeasures. In a nutshell, ARM's PA works by cryptographically authenticating the content of a register before using it. Indeed, it is conceived as a protection against modification of code pointers such as return addresses stored in memory. For instance, PA represents a valuable protection mechanism to ensure that functions only return to legal locations as expected by the program according to the CFG, hence preventing stack overflow attacks. The BTI mechanism can secure indirect branches, enforcing that the destination location of the branch contains only instructions of an acceptable list. Combining both instruments allows for a complete forward-backwards control-flow integrity, reducing the possibility of an attacker hijacking the execution flow to execute arbitrary code.

V. Evaluation results

This section presents an evaluation campaign performed on modern compiler-supported techniques. The evaluation campaign was carried out with mainly two purposes. The first objective was to understand if our X-RIPE implementation could provide some working attack on a modern OS. Then, the second objective was to understand how the latest protection techniques available on the market, such as ARM's Pointer Authentication, behave compared to well-established methods. Production-ready systems commonly use protection schemes integrated with popular OSes and compilers such as GCC. Therefore, to test X-RIPE, we applied it against memory-corruption protection techniques supported by GCC and Clang/LLVM compilers, namely: Stack Protector, ASan, and ARM's Pointer Authentication presented in the Section IV. Our evaluation was made using Ubuntu 20.04, the long-term support Ubuntu distribution released in April 2020. The distribution was equipped with version 5.13 of the vanilla kernel, compiled with the support for Pointer Authentication, hence enabling the `CONFIG_ARM64_PTR_AUTH` flag in the configuration. ARM's Pointer Authentication support is available only for processors adopting the ARMv8.3-A architecture version (and above). At the time of writing, there are no COTS development boards available on the market today. Therefore, our evaluation campaign was performed using QEMU v6, enabling the TCG (full-software) emulation of the `FEAT_PAuth` architectural feature. The evaluation campaign does not consider time performance; hence, using a virtual machine does not influence our results. To evaluate the protection degree of each specific technique, X-RIPE is compiled, by default, without stack protector (`-fno-stack-protector`) and with executable stack (`-z execstack`). In the following subsections, the results obtained with each technique are analyzed. The results are summarized in **Table 1**.

A. Stack protector: results

The stack protector mechanism, both in the version offered by GCC and Clang, is focused on protecting the stack. Stack canaries provide detection of a stack buffer overflow before dangerous code is executed. Results show that both the implementation provided by GCC and Clang/LLVM successfully prevent each kind of stack overflow. Thanks to local variable re-ordering, stack protector works in preventing both direct and indirect attacks. On the order hand, the majority of attacks targeting BSS, heap or data segment are not counteracted.

B. ASan: results

ASan is a runtime address sanitizer designed to detect memory errors. Being composed of both compiler instrumentation and a runtime library, ASan is a powerful technique. ASan was intended to prevent out-of-bounds access to the heap, stack, and global objects. Our results show that ASan can detect the most significant part of the overflows, direct and indirect, targeting BSS, data segment, stack and heap. However, some overflows are still not detected, and they are those regarding generic function pointers laying in structures. When a buffer in a structure adjacent to a generic function pointer is overflown, the pointer can be corrupted, replacing its content with, for instance, an address of a different function. Regarding ASan, adjacent buffers in structures are not protected from overflow to avoid backward compatibility issues. It is common, especially in the oldest version of C, to use a structure as an

TABLE I
OVERALL EFFECTIVENESS OF PROTECTION TECHNIQUES FOR BUFFER OVERFLOWS

Setup	**Overall effectivness**	Successful attacks	Failed attacks
Ubuntu 20.04 (GCC, no protection)	62%	1084	1770
Ubuntu 20.04 (Clang, no protection)	61%	1109	1745
Stack protector (GCC)	86%	409	2445
Stack protector (Clang)	84%	470	2384
ASan (GCC)	98%	49	2805
ASan (Clang)	98%	49	2805
Pointer Authentication (GCC)	87%	372	2482
Pointer Authentication (Clang)	83%	477	2377
All protections (GCC)	99%	28	2854
All protections (Clang)	99%	28	2854

array of bytes, independently of declared types of structure members. Furthermore, using ASan to protect a program does not come for free; the typical slowdown introduced by AddressSanitizer is 2x [23].

C. ARM's Pointer Authentication: results

ARM's Pointer Authentication is an architectural feature that comprises a set of instructions and facilities offered by the processor to sign and authenticate pointers. That architectural feature, referred to as `FEAT_PAuth` in Linux, was introduced in 2017 with ARM v8.3-A architecture. ARM's Pointer Authentication support can be used to sign and authenticate any pointer. However, given the relatively recent introduction, the support offered by GCC and Clang/LLVM it is still unripe. So far, Pointer Authentication is used only to authenticate the return address before taking the return branch to the caller. Our results show that the current support is comparable to stack protector in terms of effectiveness; thus, the community's effort must focus on improving the current support available for this outstanding instrument.

D. All protections

When using all protections together, the number of attacks performed with success is nearly null. However, it is still different from zero. In addition, the overhead introduced by that techniques is non-negligible, and, often, not all protection schemes are put in place together. That means all the current protection techniques are still not perfect, and a buffer overflow can still attack a modern OS.

VI. CONCLUSIONS & FUTURE DIRECTIONS

Buffer overflows represent a security threat for applications and operating systems, especially for the embedded systems, where the usage of C and C++ is a common choice. Since the '90s, many countermeasures have been designed and implemented. However, recent exploited vulnerabilities suggest that deliberate memory corruption is still an open problem. In this article, we presented X-RIPE, a cross-platform Runtime Intrusion Prevention Evaluator designed to evaluate, in a quantitative manner, protection coverage offered by a specific mechanism against buffer overflows. We analyzed modern compiler-supported protections against X-RIPE to compare their coverage. X-RIPE attempts to be the basis for a standard way to evaluate a defense technique's robustness; however, it is currently more of a proof-of-concept. In the future, we are willing to extend this work by providing even more, attack vectors to execute several reasonable real-world attacks, which will help quantify the coverage of a specific technique.

REFERENCES

[1] V. van der Veen, N. dutt Sharma, L. Cavallaro, and H. Bos, "Memory Errors: The Past, the Present, and the Future," in *Research in Attacks, Intrusions, and Defenses*, D. Balzarotti, S. J. Stolfo, and M. Cova, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 86–106.

[2] S. Institute, "Cwe/sans top 25 most dangerous software errors," 2022. [Online]. Available: https://www.sans.org/top25-software-errors/

[3] Q. S. Advisory, "pwnkit: Local Privilege Escalation in polkit's pkexec (CVE-2021-4034)," 2022. [Online]. Available: https://seclists.org/oss-sec/2022/q1/80

[4] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen, "RIPE: Runtime intrusion prevention evaluator," in *In Proceedings of the 27th Annual Computer Security Applications Conference, ACSAC*. ACM, 2011.

[5] M. Khandaker, A. Naser, W. Liu, Z. Wang, Y. Zhou, and Y. Cheng, "Adaptive call-site sensitive control flow integrity," in *2019 IEEE European Symposium on Security and Privacy (EuroS P)*, 2019.

[6] R. K. Shrivastava, K. J. Concessao, and C. Hota, "Code tamper-proofing using dynamic canaries," in *2019 25th Asia-Pacific Conference on Communications (APCC)*, 2019, pp. 238–243.

[7] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, ser. OSDI '06. USA: USENIX Association, 2006, p. 147–160.

[8] O. Ruwase and M. Lam, "A practical dynamic buffer overflow detector," in *In Proceedings of The 11th Annual Network and Distributed System Security Symposium*, 2004.

[9] N. Tuck, B. Calder, and G. Varghese, "Hardware and binary modification support for code pointer protection from buffer overflow," in *37th International Symposium on Microarchitecture (MICRO-37'04)*, 2004, pp. 209–220.

[10] P. E. Black *et al.*, "Sard: A software assurance reference dataset," in *Anonymous Cybersecurity Innovation Forum.()*, 2017.

[11] S. Zhou and J. Chen, "Experimental evaluation of the defense capability of arm-based systems against buffer overflow attacks in wireless networks," in *2020 IEEE 10th International Conference on Electronics Information and Emergency Communication (ICEIEC)*, 2020, pp. 375–378.

[12] B. M. Calatayud and L. Meany, "A comparative analysis of buffer overflow vulnerabilities in high-end iot devices," in *2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC)*, 2022, pp. 0694–0701.

[13] Y. Wang, J. Wu, T. Yue, Z. Ning, and F. Zhang, "Rettag: Hardware-assisted return address integrity on risc-v," in *Proceedings of the 15th European Workshop on Systems Security*, ser. EuroSec '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 50–56. [Online]. Available: https://doi.org/10.1145/3517208.3523758

[14] N. R. Kisore, “A qualitative framework for evaluating buffer overflow protection mechanisms,” *Int. J. Inf. Comput. Secur.*, vol. 8, no. 3, p. 272–307, jan 2016. [Online]. Available: https://doi.org/10.1504/IJICS.2016.079187
[15] Rapid7, “Metasploit: Penetration Testing Software.” [Online]. Available: https://www.metasploit.com/
[16] J. P. Anderson, “Computer security technology planning study,” U.S. Air Force Electronic Systems Division Tech., Tech. Rep., 1972.
[17] E. H. Spafford, “The internet worm program: An analysis,” *SIGCOMM Comput. Commun. Rev.*, vol. 19, no. 1, p. 17–57, jan 1989. [Online]. Available: https://doi.org/10.1145/66093.66095
[18] H. Etoh, “GCC extension for protecting applications from stack-smashing attacks,” IBM Research Group, Tech. Rep., 01 2004.
[19] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, “AddressSanitizer: A fast address sanity checker,” in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX Association, Jun. 2012, pp. 309–318. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany
[20] L. Team, “LLVM 3.1 Release Notes,” 2012. [Online]. Available: https://releases.llvm.org/3.1/docs/ReleaseNotes.html
[21] G. Team, “GCC 4.8 Release Changes,” 2014. [Online]. Available: https://gcc.gnu.org/gcc-4.8/changes.html
[22] J. Pincus and B. Baker, “Beyond stack smashing: recent advances in exploiting buffer overruns,” *IEEE Security Privacy*, vol. 2, no. 4, pp. 20–27, 2004.
[23] L. Team, “Clang 15.0.0 documentation: AddressSanitizer.” [Online]. Available: https://clang.llvm.org/docs/AddressSanitizer.html

Work in Progress: Real-Time GRB Localization for the Advanced Particle-astrophysics Telescope

Marion Sudvarg*, Jeremy Buhler†, Roger Chamberlain‡, Chris Gill§
Department of Computer Science & Engineering
Washington University in St. Louis
St. Louis, Missouri
*msudvarg@wustl.edu, †jbuhler@wustl.edu, ‡roger@wustl.edu, §cdgill@wustl.edu

James Buckley
Department of Physics
Washington University in St. Louis
St. Louis, Missouri
buckley@wustl.edu

***Abstract*—The Advanced Particle-astrophysics Telescope is a planned mission to perform real-time gamma-ray burst (GRB) detection and localization using SWaP-constrained embedded hardware aboard an orbiting platform. Due to the dynamic and uncertain nature of GRBs, the parallel localization task is dynamic in both workload and deadline. This implies the need for an adaptable framework that adjusts CPU utilization to accommodate overload. To this end, we propose an elastic framework over the workloads of constituent subtasks that allows both continuous and discrete state spaces. Instead of compressing according to constant weights, it instead uses a nonlinear cost function based on the expected angular error in the localized source direction of observed events.**

I. Introduction

To study the nature of dark matter and to understand the physics of neutron-star mergers, orbiting gamma-ray telescopes observe gamma-ray bursts (GRBs), collecting and transmitting data for later ground-based analysis. Newly emerging areas of astrophysics seek to perform follow-up observations, enabling the study of GRB emissions across several modalities (e.g., X-rays, visible light, radio and microwaves, cosmic rays, and gravitational waves). However, GRBs are transient events; hence, long delays from initial detection of a GRB's light to ground-based computation of its location in the sky (which is nontrivial to infer from the incoming gamma rays but is necessary to physically aim follow-up instruments) cause lost opportunities for observation.

The Advanced Particle-astrophysics Telescope (APT) [1] (Fig. 1) is a planned space-based observatory that will be deployed at the Sun-Earth Lagrange L_2 orbit, affording it a nearly full-sky field of view. It will fly with onboard computational hardware to detect and localize GRBs in real time [2]; this will enable prompt communication and follow-up observations in multiple spectral bands. We characterize APT's localization as a subtask of multiple other tasks: APT can be considered as just one component of a distributed system with multiple cyberphysical follow-up devices that couple *computation* (e.g., a telemetry system to receive the location of a GRB detected by APT) and *actuation* (the repositioning of a telescope). Each such device is associated with a deadline, after which it can no longer collect useful data. Given the worst-case latency of the associated communication, device computation, and actuation, a subdeadline associated with each follow-up device can be assigned to the task of localizing a GRB on APT.

The research presented in this paper was supported in part by NSF grants CSR-1814739 and CNS-17653503 and NASA grant 80NSSC21K1741.

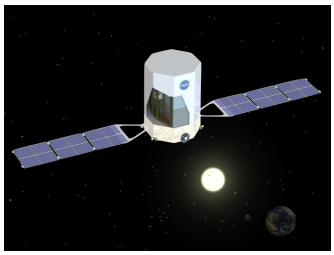


Fig. 1: A Rendering of the APT Instrument

Modeling the GRB detection computation is complicated, since there is no canonical GRB emission spectrum; each GRB is uniquely characterized by how its energy spectrum and brightness evolves over time, which defines the instant after which observing a given band is no longer useful, informing a set of deadlines which are not known a priori. The rate at which data enters APT's onboard computer, as a function of the rate and energies at which photons enter the telescope, is not constant. Further, different physical processes in the detector must be reconstructed by different algorithms [2], [3] in proportions also defined by the spectrum's parameters. Thus, our computational platform must *adapt* to dynamic deadlines and changing workloads to guarantee real-time localization on orbiting hardware with tight SWaP constraints.

To address these problems, we are developing an elastic framework for CPU utilization that aims to estimate workload and deadline constraints based on an initial profile (generated in real time) of a detected GRB. It will then adapt to expected or detected overload first by dedicating CPU resources appropriately to the various interdependent subtasks of *pair reconstruction*, *Compton reconstruction*, and *localization*. If necessary, it will degrade reconstruction accuracy by sampling or dropping a subset of data and reducing refinement iterations (involving elasticity over both continuous and discrete state spaces). Unlike the original elastic scheduling framework, which compresses task utilizations according to proportional weights [4], [5], our framework will need to consider nonlinear weighting over the cost function defined by angular error in source localization. Our framework will target parallel tasks executing on candidate hardware platforms that include both heterogeneous and identical-multiprocessor architectures and will consider compression over each constituent subtask.

II. Background and Related Work

The **Fermi** [6], [7] Gamma-Ray Space Telescope is an existing orbiting observatory with a large field of view (FoV). However, it occupies a low Earth orbit (LEO) and therefore lacks a full-sky FoV. Fermi does not perform onboard GRB localization; while it has produced extensive catalogs of GRBs [8], [9], this limits its ability to contribute to multi-messenger observations of transient astrophysical phenomena. Future planned missions such as **Glowbug** [10] suffer from similar limitations. **APT**, however, will be deployed at the Sun-Earth Lagrange L_2 orbit, where the obscuration of the sky by the earth is minimized and the benefit of the large (nearly 4π-steradian) FoV can be exploited [1]. APT seeks to support efforts in multi-wavelength and multi-messenger astrophysics, allowing follow-up instruments to study detected GRBs across a broad range of emission modalities [11]–[13]. Many such instruments have narrow apertures (often sub-1°) and so must point at the GRB source; APT will perform onboard detection and localization of GRBs in real-time, enabling prompt communication of the source direction.

We have demonstrated that reconstruction of photon trajectories from multiple Compton scattering and subsequent localization of a representative "bright" GRB (i.e., one producing a high volume of data) can be performed in < 200ms on a Raspberry Pi Model 3 B+ (which has a 4-core Cortex-A53) [2]. We built upon the approach in [14] — which reconstructs the path of individual gamma-ray photons by considering all possible orderings of interaction coordinates within a multi-layer detector (Fig. 2 Top) — by instead using a tree search with pruning to provide a deterministic WCET for each photon, then using iterative multilateration over the set of reconstructed photons to estimate a source direction. We extended analysis to heterogeneous platforms [15], with localization implemented in CUDA, achieving estimated < 80ms localization on an NVIDIA Jetson Xavier NX board. In the same work, an FPGA-based approach to infer interaction coordinates consistently completed in 68 cycles (0.23 μs) per event on a Xilinx Alveo U250 accelerator card, which includes an UltraScale+ architecture FPGA.

Our prior work is, however, limited. Evaluation was over a single representative GRB spectrum and did not consider the entire domain of energy spectra and brightness that we desire to detect. All photons were in the Compton regime, and therefore other detection modes (Fig. 2 Bottom) were not considered. Reconstruction was performed over a set of interaction centroids already in a static region of memory when the computation started; realistically, reconstruction will be performed concurrently with data streaming into memory. Finally, the work aimed to minimize execution time but did not consider the various deadlines imposed by the follow-up instruments; therefore, the handling of overload conditions was not considered.

Elastic scheduling [4], [5] provides a framework for dealing with overload by linearly compressing the effective utilizations of individual tasks over a continuous space according to weights assigned to each task. It has been reformulated as a quadratic optimization problem [16], [17] and has been extended to federated scheduling of parallel tasks [18], [19] including those tasks constrained to discrete utilization values [20]. Unlike in prior work, our system will need to compress subtasks individually, assigning weights to each according to a nonlinear cost function, while keeping overhead induced by the framework low such that state transitions do not significantly contribute to system overload. In [21], we demonstrated a quasilinear-time solution and a linear admission control algorithm for elastic scheduling on a uniprocessor, and in [22], we demonstrated pseudopolynomial heuristics to assign processors to integer-valued parallel tasks. Similarly efficient methods will need to be developed for more complex elastic scheduling of parallel tasks.

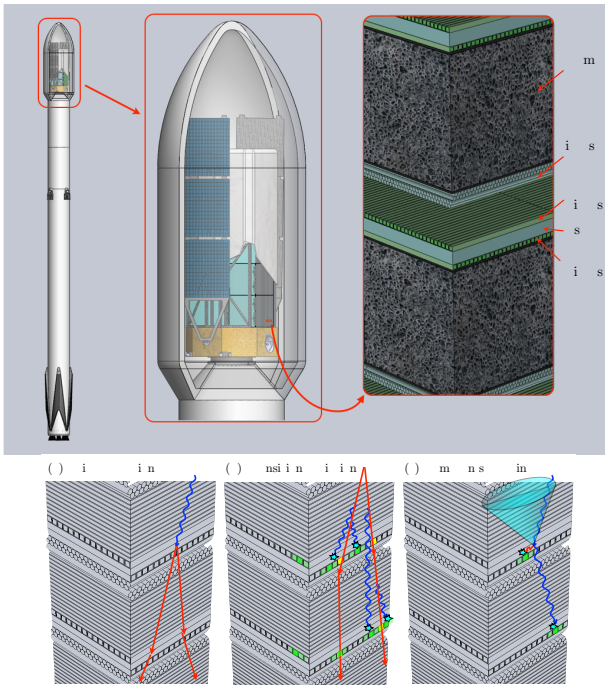


Fig. 2: Top: APT in Falcon-9 faring. Bottom: Detection modes. [1]

III. System Model

Computation Pipeline: We represent the proposed system as a pipeline consisting of several stages diagrammed in Fig. 3. APT's detector has layers of optical fiber arrays; each fiber is read by a photodetector coupled with an analog-pipeline waveform digitizer ASIC [23]. Each layer array is multiplexed by a single FPGA (e.g., a rad-hard Microchip RT PolarFire), which receives a trigger notification when a constituent ASIC detects signal indicative of a GRB event. The FPGA receives, demultiplexes, and time-integrates signal intensities, then performs centroiding (data reduction to infer the coordinates and energies associated with a photon's interactions in the detector) for each detected photon. The FPGA sends data to the CPU's main memory (e.g., over a time-sensitive network handling transmissions from as many as 40 FPGAs).

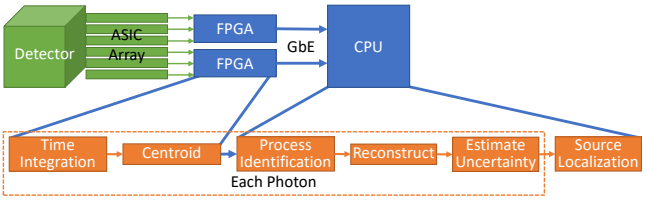


Fig. 3: APT Computation Pipeline

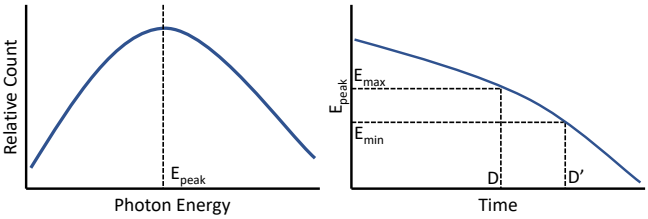


Fig. 4: Left: A GRB spectrum. Right: D, D' from $E_{peak}(t)$, E_{min}, E_{max}

The CPU must combine the received data for each detected photon, then identify the physical process (currently, Compton scattering or pair production) that generated the observed signals. It uses the corresponding algorithm to reconstruct the photon and estimate the uncertainty of the associated result. These are propagated to the source localization stage, which combines data from multiple incident photons. Localization must be completed in time to guarantee the end-to-end deadline requirements for pointing secondary instruments. For now, we assume a sufficient memory buffer such that throughput for processing data transmitted from the FPGA is not a concern; consideration of memory constraints is deferred to future work.

End-to-End Deadline: We define a collection of follow-up instruments $\mathbf{I} = \{I_i\}$, each sensitive to a spectral range $[E_i^{min}, E_i^{max}]$. At a given instant t, a GRB emits a spectrum characterized, among other parameters, by its peak energy $E_{peak}(t)$; the spectrum evolves over time, and the function is unique to each GRB. We define $t = 0$ as the time at which photons emitted by the GRB first enter the detector, t_i^{min} as the time where $E_{peak} = E_i^{max}$ and similarly for t_i^{max}. For now, we assume that E_{peak} is monotonically decreasing in the region $[E_i^{min}, E_i^{max}]$, as in Fig. 4; study of GRB catalogs is ongoing to verify monotonicity and to identify other properties of E_{peak} (e.g., concavity). Peak times imply a soft deadline $D_i = t_i^{max}$, after which the GRB begins to emit fewer photons in the observable spectrum of I_i, and a firm deadline $D_i' = t_i^{min}$, after which I_i cannot make useful observations.

Each instrument is associated with a latency δ_i that includes delivery of the burst alert (speed-of-light from L_2 orbit to Earth is ≈ 5s, though some instruments may also be in an L_2 orbit or even on board APT itself) and the time to repoint the instrument. We additionally assume a worst-case latency δ_{CPU} between a gamma-ray photon's arrival in the detector and the associated data's arrival in memory. This allows us to define a subdeadline D for event reconstruction and source localization on the CPU as $\min\{D_i - \delta_i\} - \delta_{\text{CPU}}$. We expect D to be subsecond for fast GRBs, though it may be on the order of several seconds for slow events.

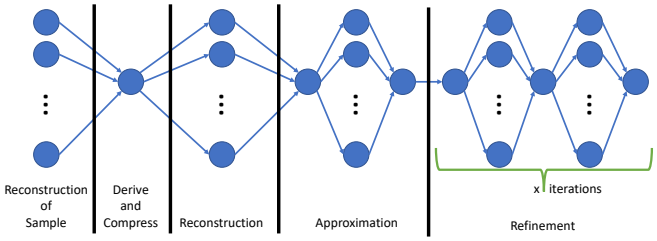


Fig. 5: DAG representation of the parallel CPU execution task.

Overload: Individual photon processing (which includes process identification, reconstruction, and propagation of uncertainty) depends on its associated physical process; we define WCETs C_c for Compton-scattering and C_p for pair-production. The expected fraction $r(t)$ of Compton-scattering photons (a function of the emission spectrum) allows us to define an average WCET $C = r(t)(C_c - C_p) + C_p$ per photon. The rate of photon arrival, R, is a function of the brightness of the GRB.

The function $E_{peak}(t)$ is not known a priori. However, given an initial sample of n_{fit} photons, $E_{peak}(0)$ can be fit to a Band function [24] and matched against known GRB data (e.g., from the Fermi catalogs in [8], [9]) to estimate $E_{peak}(t)$ and derive the deadline and WCET for reconstruction and propagation of uncertainty. For simplicity, we use a constant WCET C_r derived from the worst-case estimated $r(t)$. The derivation subtask has WCET C_{fit} and is released at time $\delta_{\text{CPU}} + n_{fit}/R$.

Given a sufficiently large D, a streaming execution model can be used, where all data is reconstructed as it is collected; then, once data stops arriving (or the arrival rate slows significantly), the source localization stage runs. However, if D cannot be met, the system is considered overloaded. In this case, we model computation on the CPU as the parallel DAG task illustrated in Fig. 5 and elastically compress its constituent subtasks by solving the following optimization problem, which seeks to minimize the expected error in source localization while meeting the deadline constraint:

$$\text{min:} \quad \textbf{error}(n, n_a, x) \tag{1}$$

$$\text{s.t.:} \quad C_r \cdot n/m + C_{fit} + C_l(n, n_a, x) \leq D \tag{2}$$

$$n \leq R \cdot (D - C_l(n, n_a, x)) \tag{3}$$

$$n_a \leq n \tag{4}$$

$$x \in \mathbb{N} \tag{5}$$

Expected error **(1)** is a nonlinear, monotonically increasing function of three variables: n, the number of photons selected for trajectory reconstruction; n_a, the number of reconstructed photons sampled for an initial approximation of the GRB's location; and x, the number of subsequent refinement iterations to improve the location estimate (the localization algorithm is detailed in [2]). Similarly to E_{peak}, **error** is not known a priori. Using simulations of known GRBs from the catalogs, we can estimate error functions offline. This allows the online compression framework to select an **error** according to the Band function fit from the initial sample of photons.

Because of the highly parallelizable nature of several stages of the pipeline, latency can be characterized according to the

expression on the left side of **(2)**. This constraint guarantees that the time between photon arrival in the instrument and associated data arrival in main memory, plus total reconstruction time (parallelized over the CPU's m cores), fitting, and localization WCET C_l does not exceed the deadline. C_l is polynomial in n, n_a, x and is characterized by the following equation (described in [2]), where each a_i is constant:

$$x(a_0 \cdot n^2 + a_1 \cdot n) + a_2 \cdot n + a_3 \cdot n_a + a_4 \tag{6}$$

In **(3)**, the photons selected for reconstruction are constrained by the number that have become available before localization must begin. Equation **(4)** constrains the photons sampled for initial approximation to those that have been reconstructed. The values n and n_a are expected to be large enough to approximate a continuous space, but **(5)** restricts x to the natural numbers.

Solving this optimization problem is the topic of ongoing work. While the functions **error** and C_l have not yet been fully characterized, we suspect the nonlinearity will make it too computationally intensive to solve online. Generating an offline solution for each representation GRB spectrum from the catalog would reduce the execution time for online compression. However, neither R nor $\mathbf{I}$ are known a priori: the collection of available instruments changes as ground-based telescopes may be out-of-view due to Earth's rotation, and instruments may be occupied or taken offline. However, as this is not a hard real-time problem, approximate solutions should be sufficient. An offline solution might be given as a function of R (or for a set of discrete values of R). Further, we might define a few sets of available instruments depending on the time of day, which would allow a deadline D to be assigned to each representative GRB from the catalog, similarly to E_{peak} and **error**. We are also considering fast methods to search for an approximate solution online, e.g., by using a genetic algorithm [25].

IV. CPU and OS Requirements

Our task pipeline will run atop SWaP-constrained embedded hardware onboard an orbiting platform. We have tested several of its algorithms on a Raspberry Pi Model 3B+ ([2], [15]). A suborbital demonstration mission, in which a smaller version of the APT instrument will fly on a high-altitude balloon, is currently being designed with an Intel Atom-based single-board computer. APT, however, will fly at the L_2 Lagrange point, which presents the additional challenge of radiation hardening.

The current APT architecture requires an FPGA per detector layer; at 40 layers, sufficient networking capabilities, including possible support for a TSN protocol, will be required. The CPU's board will need a high-bandwidth (e.g., gigabit) network adapter with DMA capabilities, requiring OS and driver support. It will additionally need to communicate burst alerts, requiring additional support for telemetry equipment (which will likely be accessed over a serial bus).

Execution of the CPU stages of our pipeline (process identification, photon trajectory reconstruction, estimation of uncertainty, and localization) can execute as a single binary which can also encode the logic for both determining and implementing task compression. As such, a targeted unikernel compile of Linux [26] that integrates all necessary drivers and the GRB source localization program might be ideal for our purposes. However, other processes may need to execute concurrently, including real-time mission-critical instrument control tasks. For such task sets, the target operating system may need to provide both priority-based scheduling and strong temporal isolation. For example, CPU reservations (such as those provided by cgroups and real-time group scheduling in Linux) can be used to enforce the target utilization of the localization task and prevent overruns from affecting other tasks on the system. Furthermore, mechanisms such as scCaps in seL4 [27] can, in addition to providing bandwidth constraints, enable a system to switch criticality modes in the event of overruns.

V. Conclusion

We have presented an elastic model for compressing task utilization by reducing individual subtask workloads according to a nonlinear cost function. Characterization of representative GRB spectra, their evolution in time, and the corresponding parameters of the optimization problem presented in Sec. III are ongoing through simulations, measurements, and study of GRB catalogs. However, we suspect the problem will be computationally intensive to solve online as part of the onboard localization pipeline. But because this is a soft real-time problem, we intend to instead find an approximate solution (either by precomputing a set of compression modes from which the closest one can be selected, or with online approximation using a fast search technique such as a genetic algorithm). Overrun might result in missed opportunities for follow-up observations but will not cause system failure. Time remaining before the deadline can be used to reconstruct additional photons, then perform additional refinement over the larger set of data.

Our model has room for further refinement. As other tasks may run concurrently, we need to consider how this affects schedulability of the parallel localization task. Under federated scheduling, our pipeline would be assigned dedicated cores, but with only 4 cores on the considered hardware platforms, this may result in unnecessary resource waste. Alternative analytical frameworks, such as semi-federated scheduling, could reduce resource waste but would further complicate the proposed elastic scheduling model. Additionally, memory constraints must be considered: to avoid dropping data transmitted from the FPGA (which must additionally be saved to secondary storage), a buffer must be allocated according to the maximum expected data volume and rate and the reconstruction throughput, which itself is elastic. A suitable OS, such as a real-time microkernel (e.g. seL4 [27]) or a targeted unikernel compile of Linux [26], is still being sought. We welcome suggestions and feedback from the community.

REFERENCES

[1] J. Buckley, S. Alnussirat, C. Altomare *et al.*, "The Advanced Particle-astrophysics Telescope (APT) Project Status," in *Proc. of 37th International Cosmic Ray Conference — PoS(ICRC2021)*, vol. 395, Jul. 2021, pp. 655:1–655:9.

[2] M. Sudvarg, J. Buhler, J. H. Buckley, W. Chen *et al.*, "A Fast GRB Source Localization Pipeline for the Advanced Particle-astrophysics Telescope," in *Proc. of 37th International Cosmic Ray Conference — PoS(ICRC2021)*, vol. 395, Jul. 2021, pp. 588:1–588:9.

[3] W. Chen, J. Buckley, S. Alnussirat *et al.*, "The Advanced Particle-astrophysics Telescope: Simulation of the Instrument Performance for Gamma-Ray Detection," in *Proc. of 37th Int'l Cosmic Ray Conference — PoS(ICRC2021)*, vol. 395, 2021, pp. 590:1–590:9.

[4] G. C. Buttazzo, G. Lipari, and L. Abeni, "Elastic task model for adaptive rate control," in *IEEE Real-Time Systems Symposium*, 1998.

[5] G. C. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni, "Elastic scheduling for flexible workload management," *IEEE Transactions on Computers*, vol. 51, no. 3, pp. 289–302, Mar. 2002. [Online]. Available: http://dx.doi.org/10.1109/12.990127

[6] C. Meegan, G. Lichti, P. N. Bhat *et al.*, "The Fermi gamma-ray burst monitor," *The Astrophysical Journal*, vol. 702, no. 1, pp. 791–804, aug 2009. [Online]. Available: https://doi.org/10.1088%2F0004-637x%2F702%2F1%2F791

[7] W. B. Atwood, A. A. Abdo, M. Ackermann *et al.*, "The large area telescope on the Fermi gamma-ray space telescope mission," *The Astrophysical Journal*, vol. 697, no. 2, pp. 1071–1102, may 2009. [Online]. Available: https://doi.org/10.1088%2F0004-637x%2F697%2F2%2F1071

[8] M. Ackermann, M. Ajello, K. Asano *et al.*, "The first Fermi LAT gamma-ray burst catalog," *The Astrophysical Journal Supplement Series*, vol. 209, no. 1, p. 11, oct 2013. [Online]. Available: https://doi.org/10.1088%2F0067-0049%2F209%2F1%2F11

[9] M. Ajello, M. Arimoto, M. Axelsson *et al.*, "A decade of gamma-ray bursts observed by Fermi-LAT: The second GRB catalog," *The Astrophysical Journal*, vol. 878, no. 1, p. 52, jun 2019. [Online]. Available: https://doi.org/10.3847/1538-4357/ab1d4e

[10] J. E. Grove, C. C. Cheung, M. Kerr *et al.*, "Glowbug, a low-cost, high-sensitivity gamma-ray burst telescope," 2020. [Online]. Available: https://arxiv.org/abs/2009.11959

[11] I. Bartos and M. Kowalski, *Multimessenger Astronomy*, ser. 2399-2891. IOP Publishing, 2017. [Online]. Available: https://dx.doi.org/10.1088/978-0-7503-1369-8

[12] A. Neronov, "Introduction to multi-messenger astronomy," in *Journal of Physics: Conference Series*, vol. 1263, no. 1. IOP Publishing, 2019, p. 012001.

[13] P. Mészáros, D. B. Fox, C. Hanna, and K. Murase, "Multi-messenger astrophysics," *Nature Reviews Physics*, vol. 1, no. 10, pp. 585–599, 2019.

[14] S. Boggs and P. Jean, "Event reconstruction in high resolution Compton telescopes," *Astronomy and Astrophys. Supp. Series*, vol. 145, no. 2, pp. 311–321, 2000.

[15] J. Wheelock, W. Kanu, M. Sudvarg *et al.*, "Supporting multi-messenger astrophysics with fast gamma-ray burst localization," in *Proc. of IEEE/ACM HPC for Urgent Decision Making Workshop (UrgentHPC)*, Nov. 2021.

[16] T. Chantem, X. S. Hu, and M. D. Lemmon, "Generalized elastic scheduling," in *IEEE International Real-Time Systems Symposium*, 2006.

[17] ——, "Generalized elastic scheduling for real-time tasks," *IEEE Transactions on Computers*, vol. 58, no. 4, pp. 480–495, April 2009.

[18] J. Orr, C. Gill, K. Agrawal, S. Baruah *et al.*, "Elasticity of workloads and periods of parallel real-time tasks," in *Proceedings of the 26th International Conference on Real-Time Networks and Systems*, ser. RTNS '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 61–71. [Online]. Available: https://doi.org/10.1145/3273905.3273915

[19] J. Orr and S. Baruah, "Multiprocessor scheduling of elastic tasks," in *Proceedings of the 27th International Conference on Real-Time Networks and Systems*, ser. RTNS '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 133–142. [Online]. Available: https://doi.org/10.1145/3356401.3356403

[20] J. Orr, J. C. Uribe, C. Gill, S. Baruah *et al.*, "Elastic scheduling of parallel real-time tasks with discrete utilizations," in *Proceedings of the 28th International Conference on Real-Time Networks and Systems*, ser. RTNS 2020. New York, NY, USA: Association for Computing Machinery, 2020, pp. 117–127. [Online]. Available: https://doi.org/10.1145/3394810.3394824

[21] M. Sudvarg, C. Gill, and S. Baruah, "Linear-time admission control for elastic scheduling," *Real-Time Systems*, vol. 57, no. 4, pp. 485–490, Oct 2021. [Online]. Available: https://doi.org/10.1007/s11241-021-09373-4

[22] M. Sudvarg and C. Gill, "Analysis of federated scheduling for integer-valued workloads," in *Proceedings of the 30th International Conference on Real-Time Networks and Systems*, ser. RTNS 2022. New York, NY, USA: Association for Computing Machinery, 2022, pp. 12–23. [Online]. Available: https://doi.org/10.1145/3534879.3534892

[23] K. Bechtol, S. Funk, A. Okumura, L. Ruckman, A. Simons, H. Tajima, J. Vandenbroucke, and G. Varner, "TARGET: A multi-channel digitizer chip for very-high-energy gamma-ray telescopes," *Astroparticle Physics*, vol. 36, no. 1, pp. 156–165, 2012.

[24] D. Band, J. Matteson, L. Ford, B. Schaefer, D. Palmer, B. Teegarden, T. Cline, M. Briggs, W. Paciesas, G. Pendleton, G. Fishman, C. Kouveliotou, C. Meegan, R. Wilson, and P. Lestrade, "BATSE observations of gamma-ray burst spectra. I. spectral diversity," *Astrophys. J.*, vol. 413, p. 281, Aug. 1993.

[25] K. A. De Jong, "An analysis of the behavior of a class of genetic adaptive systems." Ph.D. dissertation, University of Michigan, USA, 1975, aAI7609381.

[26] A. Raza, P. Sohal, J. Cadden, J. Appavoo, U. Drepper, R. Jones, O. Krieger, R. Mancuso, and L. Woodman, "Unikernels: The next stage of linux's dominance," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 7–13. [Online]. Available: https://doi.org/10.1145/3317550.3321445

[27] "The sel4 microkernel," https://docs.sel4.systems/projects/sel4/, seL4 Foundation, accessed: January 23, 2022.

Notes

OSPERT 2022 Program

Tuesday, July 5 2022

8:00 – 9:30	Registration
8:30 – 10:00	Welcome Session 1: Keynote Mixed Criticality on RISC-V: Experiences from Porting a Partitioning Hypervisor *Konrad Schwarz, Siemens Coprorate Technology*
10:00 – 10:30	Coffee Break
10:30 – 12:00	Session 2: Broadening RTOS Understanding RTOS-Independent Interaction Analysis in ARA *G. Entrup, J. Neugebauer, D. Lohmann* Supporting Multiprocessor Resource Synchronization Protocols in RTEMS *J. Shi, J. Pham, M. Münch, J. Hafemeister, J. Chen, K. Chen* Cabas: Real-Time for the Masses *T. Smejkal, J. Bierbaum, M. von Oltersdorff-Kalettka, M. Roitzsch* On the Interplay of Computation and Memory Regulation in Multicore Real-Time Systems *D. Hoornaert, G. Ghaemi, A. Bastoni, R. Mancuso, M. Caccamo, G. Corradi*
12:00 – 13:30	Lunch
13:30 – 15:00	Session 3: Use Your Data, Trust Your System Can we trust our energy measurements? A study on the Odroid-XU4i *J. Roeder, S. Altmeyer, C. Grelck* Revisiting Migration Overheads in Real-Time Systems: One Look at Not-So-Uniform Platforms *P. Raffeck, W. Schröder-Preikschat, P. Ulbrich* X-RIPE: A Modern, Cross-Platform Runtime Intrusion Prevention Evaluator *G Serra, S. Di Leonardi, A. Biondi* Work in Progress: Real-Time GRB Localization for the Advanced Particle-astrophysics Telescope *M. Sudvarg, J. Buhler, R. Chamberlain, C. Gill, J. Buckley* Closing
15:00 – 16:00	After-Workshop Coffee Break

Wednesday, July 6th – Friday, July 8th 2022

ECRTS main conference.