Leibniz Universität Hannover
Institut für Systems Engineering
Fachgebiet System- und Rechnerarchitektur

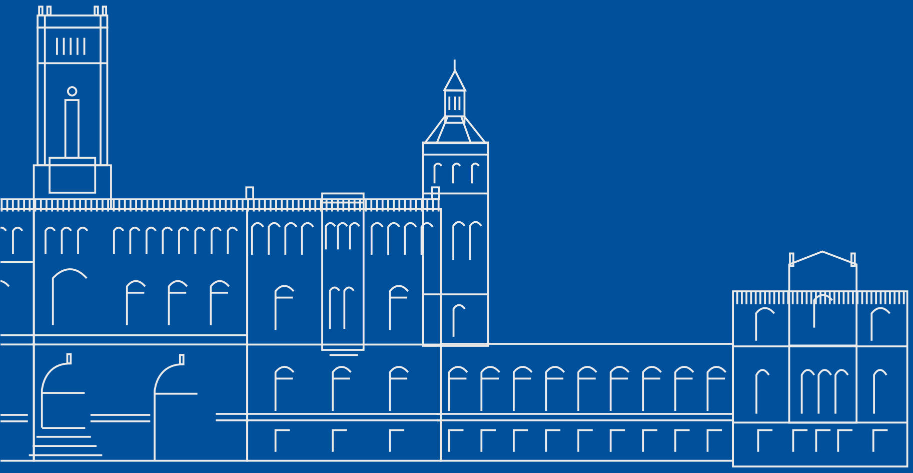
Leibniz
Universität
Hannover

Andreas Kässens

Synthesis of Optimized AUTOSAR Embedded Systems: Automated System-Call Specialization and Lock Elision on Multicore Applications as a Whole-System Approach

Masterarbeit im Fach Technische Informatik

14. Juni 2023



Leibniz Universität Hannover
Institut für Systems Engineering
Fachgebiet System und Rechnerarchitektur
Appelstr. 4 · 30167 Hannover · Germany

Synthesis of Optimized AUTOSAR Embedded Systems: Automated System-Call Specialization and Lock Elision on Multicore Applications as a Whole-System Approach

Masterarbeit im Fach Technische Informatik

vorgelegt von

Andreas Kässens

geb. am 18. Oktober 1995
in Papenburg

angefertigt am

Institut für Systems Engineering
Fachgebiet System- und Rechnerarchitektur

Fakultät für Elektrotechnik und Informatik
Leibniz Universität Hannover

Erstprüfer:	**Prof. Dr.-Ing. habil. Daniel Lohmann**
Zweitprüfer:	**Prof. Dr.-Ing. Bernardo Wagner**
Betreuer:	**Björn Fiedler, M.Sc.**
	Gerion Entrup, M.Sc.

Beginn der Arbeit:	**14. Dezember 2022**
Abgabe der Arbeit:	**14. Juni 2023**

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties. I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Andreas Kässens)
Hannover, 14. Juni 2023

ABSTRACT

Embedded real-time systems in safety-critical areas, such as in the automotive and aerospace industries, have particularly high software requirements regarding safety and reliability. In the automotive sector, the operating system standard AUTomotive Open System ARchitecture (AUTOSAR) has been established as a common basis among manufacturers and suppliers to meet these criteria. With priority-based real-time scheduling and static configuration, AUTOSAR-compliant operating systems are well suited for deployment in real-time systems. However, there is an aim to further optimize these software systems to improve their non-functional characteristics, including dependability, memory consumption, and runtime delays.

In this thesis, I present the development of the **M**ulticore **A**UTOSAR **C**ompatible **A**pplication-specific **W**hole-system-optimizer (MACAW) system generator that supports the key interfaces of the AUTOSAR operating system specification. Using the Automated Real-time system Analyzer (ARA) framework, previous work developed the MultiSSE, a static code analysis to obtain a graph enumerating and connecting all possible system states. By automatically detecting certain interaction patterns between operating system objects in this graph, the performance of operating system calls can be optimized. Costly actions such as Inter-Processor Interrupts (IPIs) or spinlock operations can be eliminated if they do not affect subsequent system states. Spinlock operations that are never executed by two processor cores simultaneously can be omitted without affecting functionality, and IPIs that do not affect scheduling on the target core can be skipped.

MACAW is implemented as the synthesis step within the ARA framework for real-time applications and currently supports the POSIX platform. At compile time, the detected optimizations will be automatically applied to a specialized system call variant for each call site. Evaluation of available test applications shows that about 28 % of the affected cross-core system calls can be optimized. In particular, the avoidance of IPIs leads to a measurable reduction in delays and jitter.

KURZFASSUNG

Eingebettete Echtzeitsysteme in sicherheitskritischen Bereichen, wie beispielsweise in der Automobil- und Luftfahrtindustrie, stellen besonders hohe Softwareanforderungen bezüglich Betriebssicherheit und Zuverlässigkeit. Im Automobilsektor wurde der Betriebssystemstandard AUTOSAR als gemeinsame Basis zwischen Herstellern und Zulieferern etabliert, um diese Kriterien zu erfüllen. Mit prioritätsbasiertem Echtzeit-Scheduling und statischer Konfiguration sind AUTOSAR-konforme Betriebssysteme für die Anwendung in Echtzeitsystemen gut geeignet. Weiterhin gibt es das Bestreben, diese Softwaresysteme zu optimieren, um ihre nicht-funktionalen Eigenschaften einschließlich Zuverlässigkeit, Speicherverbrauch und Laufzeitverzögerungen zu verbessern.

In dieser Arbeit stelle ich die Entwicklung des MACAW-Systemgenerators vor, der die wichtigsten Schnittstellen der AUTOSAR-Betriebssystemspezifikation unterstützt. Mithilfe des ARA-Frameworks wurde in bisherigen Arbeiten mit der MultiSSE eine statische Codeanalyse entwickelt, um einen Graphen, der alle möglichen Systemzustände aufzählt und verknüpft, zu erhalten. Durch die automatische Erkennung bestimmter Interaktionsmuster zwischen Betriebssystemobjekten in diesem Graphen kann die Ausführungsgeschwindigkeit von Betriebssystemaufrufen optimiert werden. Teure Aktionen wie IPIs oder Spinlock-Operationen können eliminiert werden, falls sie keine Auswirkungen auf nachfolgende Systemzustände haben. So können Spinlock-Operationen, die nie von zwei Prozessorkernen gleichzeitig ausgeführt werden, ohne Beeinträchtigung der Funktionalität ausgelassen werden, und IPIs, die sich nicht auf das Scheduling auf dem Zielkern auswirken, können übersprungen werden.

MACAW ist als ein Syntheseschritt innerhalb des ARA-Frameworks für Echtzeitanwendungen implementiert und unterstützt derzeit die POSIX-Plattform. Bei der Kompilierung werden die ermittelten Optimierungen automatisch auf spezialisierte Systemaufruf-Varianten für jede Aufrufstelle angewandt. Die Auswertung anhand von verfügbaren Testanwendungen zeigt, dass etwa 28 % der betroffenen kernübergreifenden Systemaufrufe optimiert werden können. Insbesondere die Vermeidung von IPIs führt zu einer messbaren Reduzierung von Verzögerungen und Laufzeitschwankungen.

CONTENTS

INTRODUCTION 1

In recent years, the complexity of embedded software systems, such as automotive control systems, has increased rapidly. Next to traditional control units and infotainment systems, advanced driver assistance systems and autonomous driving capabilities have become crucial technologies in modern vehicles. Such real-time control applications require complex system design and software architecture, and the automotive industry developed the AUTomotive Open System ARchitecture (AUTOSAR) standards as a common framework for application reusability and communication between control units. The (classic) AUTOSAR specification is based on the predecessor Open Systems and their Interfaces for the Electronics in Motor Vehicles (*german*: Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen, OSEK), targeted at event-triggered real-time control systems with safety and deterministic execution as primary targets [AUT]. An OSEK/AUTOSAR system is generated from the application code and a system description, producing a single application-specific binary image [OSE04b].

Real-time control systems have very strict requirements regarding timing, such as task deadlines and maximum delays [Coo17]. Therefore, Real-Time Operating Systems (RTOSs) like AUTOSAR implement suitable scheduling policies, as well as mechanisms for resource allocation and event-handling functionality. Extending OSEK, the AUTOSAR standard also supports multicore systems and advanced protection features [AUT22].

With the high complexity of software systems and limited computational resources in embedded devices, fulfilling the timing requirements is increasingly challenging for application developers. Deploying more powerful processors for typical embedded applications is not desirable due to additional costs, power consumption, or space requirements. To ensure optimal performance and meet the safety goals, the automatic optimization of such complex software systems is an important area for research. In addition to performance and response timing improvements, RTOS optimizations can involve various other non-functional aspects, such as energy consumption, memory footprint, or fault tolerance and dependability. One possible way of enabling optimizations for RTOS systems is static code analysis of the complete software system, including the application code and the operating system. Using the static system configuration of OSEK/AUTOSAR systems, knowledge about the runtime execution can be extracted for system optimization already at system generation time.

By consideration of the AUTOSAR system call interface semantics, the MultiSSE whole-system analysis in the Automated Real-time system Analyzer (ARA) project statically analyzes interactions between system objects and collects them in a system state graph [EFL23]. This state graph includes all possible states of the complete AUTOSAR system, including all cores, their local states like current execution points, and the global system objects. In this thesis, the implementation of an AUTOSAR system generator that can leverage this static knowledge to optimize cross-core system calls in multicore applications is presented. Building upon the Dependability-Oriented Static Embedded Kernel (*d*OSEK) RTOS generator for OSEK systems as a foundation, one key objective of this work is to optimize timing parameters like delay and jitter to improve the system performance, reliability, and predictability. A generic implementation of a system call may contain unnecessary overhead for certain call sites, depending on the current system state. In such cases, specialized system call variants can minimize this overhead in the kernel path to improve the timing behavior of the system call. Based on the system analysis in ARA, this static application-specific specialization of AUTOSAR systems will be applied automatically during the synthesis of the AUTOSAR system.

As a first step, I will extend *d*OSEK with multicore features like Inter-Processor Interrupts (IPIs), locking mechanisms, and synchronization of cores. The AUTOSAR system generator must be integrated into the ARA framework to access the analysis results. Next, the optimization of system calls will be added and automatically applied during the AUTOSAR system generation. To restrict the scope of this thesis, the implementation is limited to the POSIX platform, but a modular software design allows porting to other architectures without code duplication. Finally, I will evaluate the resulting RTOS generator regarding the implemented AUTOSAR features, and measure the performance improvements of the optimizations.

FUNDAMENTALS 2

2.1 Operating System Design

An operating system provides the interface for the user to interact with the computer hardware. From the users' perspective, one key task of operating systems is providing the possibility to execute programs conveniently and efficiently. For applications, the operating system offers abstractions that simplify the development process. In a system-level view, the operating system is a software layer between the hardware and the applications. This layer manages the hardware resources and multiplexes access to the hardware by virtualization. To ensure safety and security, operating systems can isolate the applications and the system itself. Figure 2.1 provides an overview of operating system components [Loh20]. The applications can access the abstracted hardware and the operating system functionality using the system call interface [SGG18].

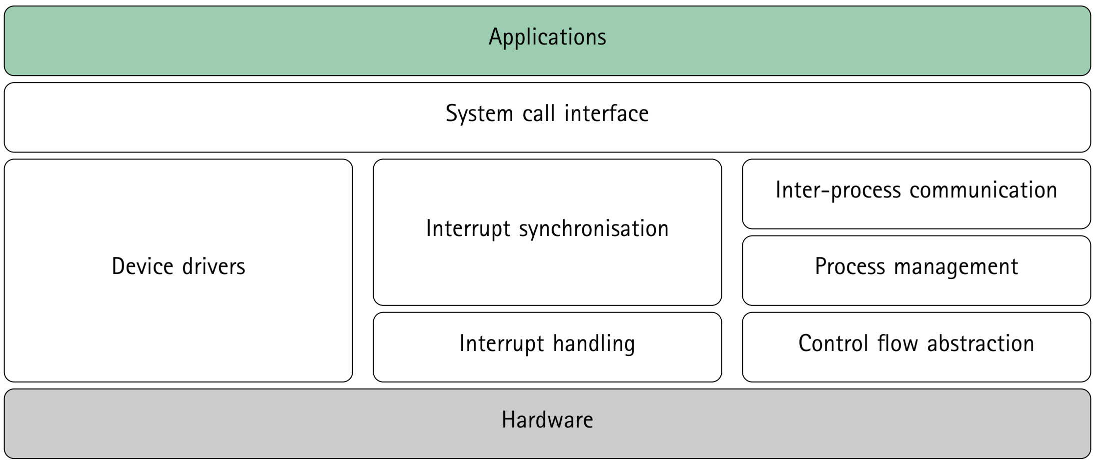


Figure 2.1 – Operating system components between applications and hardware [Loh20, adapted]

Due to hardware-specific dependencies, some components of operating systems require platform-specific adaptations. An important design objective for operating systems is to ensure portability to simplify the adaption to new hardware platforms. Operating systems with modular designs abstract the hardware-specific details, allowing common functionality to be implemented on top of these abstractions [SGG18]. General-purpose operating systems typically feature a monolithic kernel, which provides all functionality like scheduling, file systems, memory management and device drivers inside this kernel. Next to other existing operating system architectures like microkernels, a library operating system is another design concept that enables specialized, efficiency-focused environments for single-application use cases. Similar to the beginning of operating system design, the system image is compiled by linking the application and system libraries together into a single binary, the system image [Sta18].

Multicore design and interrupt handling impose additional challenges to operating system development. If concurrent tasks or interrupt handlers call the same function and access the same shared memory, this can lead to unexpected and potentially hazardous behavior. Such reentrancy issues can be addressed by careful design or with synchronization mechanisms to ensure the safe execution of shared code [Coo17].

Operating system development involves many conflicting design goals, resulting in significant variations in the architecture, features, or design principles for specific use cases. The following section focuses on the requirements of multicore embedded operating systems regarding scheduling and timing.

2.2 Embedded Operating Systems

Embedded operating systems are highly specialized software systems that run on embedded devices, such as microcontrollers, Internet of Things (IoT) devices, and Electronic Control Units (ECUs) in vehicles. Such devices are often tailored to their specific use case and have limited hardware resources, such as main memory, computing power, or storage capacities. Consequently, these constraints must be considered when developing an operating system for embedded systems.

Since embedded systems typically have a single use case, all necessary operating system interfaces are known at build time. In addition to functional requirements like interrupt-driven events, mutual exclusion and synchronization between cores, RTOSs have many more non-functional requirements like timing and response constraints. Furthermore, because of the limited hardware resources, the usage of storage and main memory must be optimized to not exceed the system's capabilities. In contrast to general-purpose operating systems, unused code can be automatically removed to reduce the image size, and the performance can be improved through application-specific system optimizations. For the implementation of an AUTOSAR kernel later in this work, the following subsections provide detailed elaboration on priority-based scheduling for real-time use cases and multicore design concepts.

2.2.1 Priority-Based Scheduling

A key feature of event-driven embedded operating systems is their ability to support real-time applications with specific timing requirements, such as fuel injection control units or robotic systems. An RTOS shall be able to guarantee fixed deadlines and deterministic execution sequences through scheduling decisions.

In a multitasking RTOS, the scheduler executes the tasks according to their priority. While non-preemptive schedulers require cooperative multitasking, preemptive schedulers can suspend the execution of one task in favor of a different task with higher priority [Aud+95]. In the OSEK/AUTOSAR context, larger numbers are higher priorities [OSE05].

With resource sharing between tasks, the problem of priority inversion can occur. When a low-priority task holds a resource that a critical high-priority task requires, the high-priority task must wait until the low-priority task frees the resource. If another task with a higher priority than the resource-holding task becomes active, the critical task must wait for this other task to terminate before the low-priority task can free the resource. This issue is particularly problematic when multiple other tasks run before the highest-priority task can access the resource because the low-priority task cannot free the resource yet, leading to a chained priority inversion.

The priority inversion problem can be mitigated by using priority inheritance or the Priority Ceiling Protocol (PCP). The more commonly used PCP assigns a ceiling priority to system resources, which is equal to the maximum priority of all tasks that contend for this resource. When one task locks this resource, the PCP elevates the task's dynamic priority to the ceiling priority of that resource. After unlocking, the priority returns to the predefined task priority. With this mechanism, tasks cannot be preempted by other tasks with lower or equal priority compared to the resource, preventing the priority inversion problem and deadlocks [Coo17].

2.2.2 Multicore Systems

In a multicore system, one chip houses two or more computing cores. In a generic Symmetric Multiprocessing (SMP) system, these cores are identical and can execute all tasks in the same manner. However, the increase in processing power with each additional CPU core does not scale linearly because of multiple factors. A shared bus system connects all CPU cores, and access to the memory decreases the overall throughput. Figure 2.2 depicts a typical multicore system with shared level 2 cache and local level 1 cache [SGG18]. If one core updates a word in its own cache, the cache coherence mechanism notifies the other cores to prevent access to an invalidated copy of the word [Sta18]. Interrupts in multicore systems are local to one core, and if a core needs to communicate with another one, an IPI is required to trigger the rescheduling on the receiving side. Along with the costs associated with interrupt handling, such as register saving or cache invalidations, IPI transmission over the shared bus system causes additional overhead, making IPIs a costly signaling mechanism.

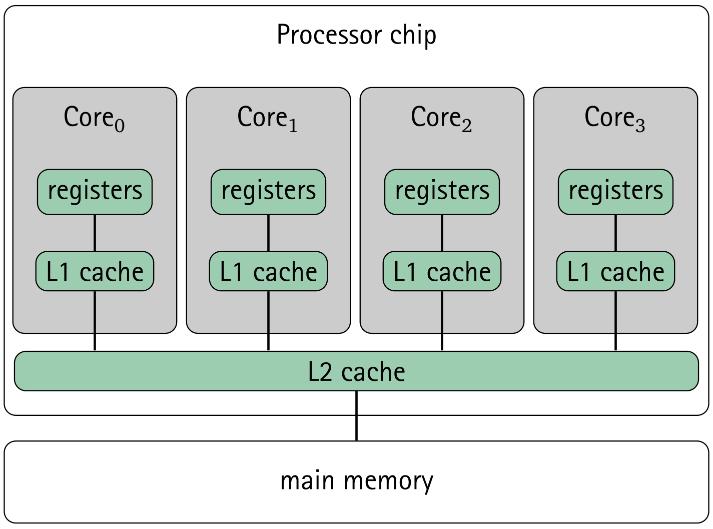


Figure 2.2 – Multicore system with caching [SGG18, adapted]

Additionally, efficient application design for multicore systems is more complex and access to shared resources must be serialized to prevent race conditions, leading to reduced throughput. This serialization is commonly achieved through mutual exclusion (mutex) and the operating system provides locking primitives like spinlocks or semaphores for the applications. Because spinlocks are actively waiting, they are typically only used for short critical sections where a rescheduling operation including context switch would take up too much time. Alternatively to using spinlocks as a mutex mechanism, passively waiting mutex concepts like binary semaphores lead to a suspension of the thread until it can enter the critical section. All implementations of multicore locking mechanisms require atomic operations, which are operations that cannot be interrupted by other cores simultaneously because the shared bus is locked [SGG18]. Instead of using higher-level locking mechanisms for variables, atomic operations can also be used directly, for example, to synchronize a multithreaded counter [Sta18]. The programming language or compiler intrinsics can support such operations and the compiler selects the correct atomic instructions. On an x86 machine, atomic machine instructions have a `lock` prefix.

2.3 OSEK and AUTOSAR Standards

The OSEK standard is a set of specifications developed by a joint project of the automotive industry called OSEK/VDX with the objective to standardize the system architecture for ECUs in vehicles. In 2005, the project published the latest version of the real-time capable operating system specification [OSE05].

The successor to the OSEK system specification in the automotive industry is driven by the AUTOSAR Group, a global partnership of automobile manufacturers that aims to standardize more system functions and functional interfaces. The AUTOSAR consortium has established two platforms: the Classic and the Adaptive Platform, each for distinct target applications. While the Adaptive Platform focuses on runtime services for high-performance control units used in automated vehicles, the Classic Platform is designed to replace OSEK for embedded systems with high requirements regarding predictability, safety, and responsiveness [AUT; AUT22].

This work is based on the Classic Platform, therefore following references to AUTOSAR always relate to that specification. The first version of the AUTOSAR Operating System specification was published in 2005 and has since been updated and extended continuously. AUTOSAR adopts most of the OSEK specification and interfaces but adds additional functionality, including support for multicore systems [AUT22].

2.3.1 OSEK System Specification

OSEK is a statically configured operating system using a priority-based scheduling policy. In contrast to other embedded operating systems, OSEK does not allow dynamic memory management, hence it does not require a memory allocation mechanism or a heap. This design decision simplifies the operating system architecture and mitigates software bugs at the expense of reduced flexibility. By eliminating the handling of dynamic instances, a static system analysis can rely solely on the static system configuration for optimization. The system generation requires the static configuration to create static OS instances and their properties [OSE05].

The system is described using the OSEK Implementation Language (OIL), a text-based system configuration that is generated into application-specific system code. Figure 2.3 describes the development process of an OSEK system. The user-defined application code, the library operating system kernel, and the generated system code are compiled together to form the complete system image, as described in section 2.1 [OSE04b].

The Listing 2.1 and 2.2 provide an example of a small OSEK-conform system. The OIL excerpt describes a `Task` with its scheduling properties and the related OS resources [OSE04b]. In the

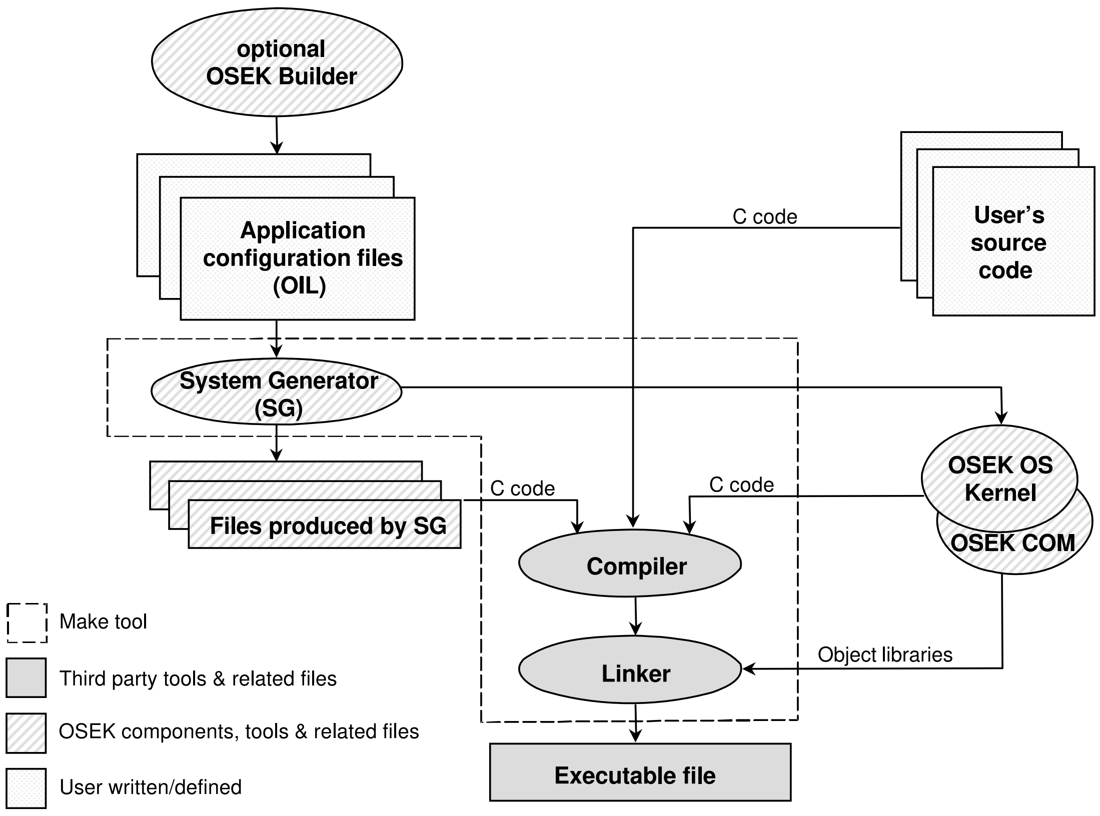


Figure 2.3 – Application development example in OSEK [OSE04b]

application code, OSEK system calls like `ActivateTask` can be used to interact with the configured OS objects.

```
TASK T0 {
    PRIORITY = 1;
    /* preemptable task */
    SCHEDULE = FULL;
    ACTIVATION = 1;
    AUTOSTART = FALSE;
    RESOURCE = R1;
};

ISR I0 {
    CATEGORY = 2;
};
```

Listing 2.1 – OIL configuration

```
int main() {
    StartOS(0);
}
Task(T0) {
    GetResource(R1);
    /* critical section */
    ReleaseResource(R2);
    TerminateTask();
}
ISR2(I0) {
    ActivateTask(T0);
}
```

Listing 2.2 – OSEK application

Automotive use cases have real-time system requirements for OSEK implementations. For this reason, the OSEK specification provides the functionality required to support event-driven systems [OSE05]. However, the standardized interfaces for operating system primitives, scheduling policies, and priority mechanisms in OSEK do not cover all aspects of an operating system. OSEK only provides framework conditions for the implementation of hardware abstraction and interaction with devices.

The key system elements for control software in OSEK are `Tasks`. The scheduler manages and switches the `Tasks` based on their activation state and priority. For synchronization between `Tasks`, the OSEK specification defines `Resource` and `Event` mechanisms. `Resources` provide access control for logic resources and devices and should be implemented using the PCP. With the `Event` system services, different `Tasks` in the system can be synchronized. To support the aforementioned event-driven systems, Interrupt Service Routines (ISRs) and `Alarms` based on `Counters` or `Timers` are available. Actions triggered by an `Alarm` can be the activation of a new `Task` or the setting of an `Event`. Calls to the OSEK system services return a `StatusType` variable, which can either be `E_OK` or a coded error. The system configuration supports user-defined `Hooks` that are called on certain events. In case of a system call error, a user-defined `ErrorHook` is called to handle the error. Other user-defined hooks include a `Pre/PostTaskHook` for time tracing and debugging and the `Startup/ShutdownHook` for system initialization and deinitialization. The OSEK specification defines system services as Application Programming Interface (API) calls to C functions or preprocessor macros, although other languages can be used for the system implementation as long as they work with the C interface [OSE05]. In order to reduce the memory footprint for small embedded devices, `Basic Tasks` can use a shared stack if they do not need a waiting state for `Events`. Such `Basic Tasks` always run to completion or are preemptable by higher-priority tasks but do not have a waiting state for `Events`, making stack sharing possible [DL18]. On the other hand, `Extended Tasks` can use the `Event` services but require a separate stack, resulting in a larger system image.

An additional OSEK standard defines sender/receiver-based communication for data transfer between tasks and interrupt service routines in ECUs [OSE04a].

2.3.2 AUTOSAR Classic Platform

In general, AUTOSAR adapts most of the OSEK system design, system service interfaces, and scheduling mechanisms. Although the API is backward compatible with OSEK, AUTOSAR discontinued the system configuration with OIL and proposed AUTOSAR XML (ARXML). However, to support both OSEK and AUTOSAR, most vendor implementations provide an import and conversion tool for OIL. AUTOSAR adapts and extends some OSEK system services to support multicore systems and adds new services. `Tasks` are bound to specific cores by static configuration and cannot be reassigned dynamically to other cores. The advantage of this approach is that scheduling on each core can be performed independently, without the need for shared waiting queues and locking. Some OSEK system services such as `Interrupt` enabling or disabling, as well as `Resources` and `Timers`, are explicitly limited to core-local usage, while other services are extended for multicore support. In multicore systems, `Resources` implemented using the PCP are insufficient to enforce mutually exclusive access to critical sections between concurrent tasks on different cores. For this purpose, AUTOSAR introduces `Spinlocks`, which are busy waiting locks to enable sequential access to these sections. AUTOSAR specifies checks to prevent deadlock situations, nesting, and interference with `Resource` handling. New API calls are defined for managing multicore hardware, and a synchronized startup and shutdown concept across all cores is required.

To support cross-core interactions, AUTOSAR extends services like `ActivateTask` or `SetEvent` with an IPI mechanism. Such cross-core interactions are synchronous, the calling task will only return after the remote core completes the interaction. Additional `StatusTypes` are defined in AUTOSAR, for example for `Spinlock` errors or issues during the startup sequence. The Table 2.1 provides an overview of all AUTOSAR system calls relevant to this work and their description [AUT22].

System Call	Description	OSEK comparison
`ActivateTask`	activate a new task and reschedule	cross-core
`TerminateTask`	terminate the current task and reschedule	cross-core
`ChainTask`	combination of `ActivateTask` and `TerminateTask`	cross-core
`GetSpinlock`	acquire a lock	new
`TryToGetSpinlock`	acquire a lock, return if locked already	new
`ReleaseSpinlock`	free the acquired lock	new
`GetResource`	enter critical section	only core-local
`ReleaseResource`	leave the critical section	only core-local
`WaitEvent`	wait until the event is set and reschedule	cross-core
`SetEvent`	set the event for a certain task and reschedule	cross-core
`GetAlarm`	read relative alarm value in ticks	cross-core
`SetRelAlarm`	set relative alarm value and enable alarm	cross-core
`StartCore`	start a new core, before StartOS	new
`StartOS`	start system and reschedule	only core-local
`GetCoreID`	retrieve the core identifier	new
`ShutdownOS`	stop scheduling of tasks	only core-local
`ShutdownAllCores`	stop all activated cores synchronously	new
`EnableAllInterrupts`	enable interrupts	only core-local
`DisableAllInterrupts`	disable interrupts	only core-local

Table 2.1 – AUTOSAR system call interface subset [AUT22; OSE05]

The AUTOSAR standard defines more advanced features, some of which require hardware support. The implementation in this work does not need the following protection mechanisms and features, therefore they are only mentioned briefly. In AUTOSAR, `Tasks` and their related system resources are collected in `OS-Applications` to provide access protection among multiple `OS-Applications`. Furthermore, memory protection isolates a `Task` and allows access only to certain memory areas. This feature requires hardware support, particularly a Memory Protection Unit (MPU). AUTOSAR also defines mechanisms to prevent timing faults like missed deadlines or inter-arrival times for `Task` activation against spurious interrupts or erroneous sources. For backward compatibility, OSEK applications behave like trusted `OS-Applications` in AUTOSAR, where these protection mechanisms can be disabled. The Inter-OS-Application Communicator (IOC) component is responsible for communication between applications, particularly across memory protection boundaries. The Software Free Running Timer (SWFRT) module enables measurements using timers and adds new API calls like `GetElapsedValue` and `GetCounterValue`. A series of tasks or events with certain timing can be activated using `ScheduleTables` [AUT22].

2.4 POSIX on Linux

The implementation of the AUTOSAR kernel in this thesis is using the Portable Operating System Interface (POSIX) standard as a virtual platform. This section provides the fundamentals of the system interface defined by POSIX and highlights features that are important for this work.

The Austin Group, a joint working group of the IEEE, The Open Group, and ISO/IEC is responsible for the development and maintenance of the POSIX standards. Among other things, they define standard concepts and interfaces between system services in operating systems and applications.

The system interface is defined as C functions and intended to allow the implementation of efficient and portable applications across UNIX-like systems. It is important to note that POSIX does not define a complete operating system, including the administration of system objects and users. Instead, it focuses on a minimal interface definition and allows optional extensions.

Next to some basic concepts like file access permissions, file names, or byte orders, POSIX defines memory synchronization and thread handling in the `pthread` interface. These functions provide mutex, spinlock, signal and thread abstractions, which are scheduled according to the defined scheduling policy. As an example, the system interface `pthread_create` shown in Listing 2.3 is used to create a new thread with given attributes. POSIX supports priority-based real-time scheduling policies and the thread attributes include the real-time priority. Next to the threads, spinlocks are defined as a busy waiting mechanism for thread synchronization and mutual exclusion. For inter-thread signaling, POSIX supports standard signals and real-time signals, with the difference that real-time signals can be queued and always arrive in a guaranteed order. Although the `pthread_kill` function name suggests differently, any defined signal identifier can be sent to the receiving thread [IEE17].

```c
int pthread_create(pthread_t *restrict thread,
        const pthread_attr_t *restrict attr,
        void *(*start_routine)(void*), void *restrict arg);

int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_trylock(pthread_spinlock_t *lock);
int pthread_spin_unlock(pthread_spinlock_t *lock);

int pthread_kill(pthread_t thread, int sig);
```

Listing 2.3 – POSIX thread, spinlock and signal interfaces [IEE17]

POSIX does not mandate a particular implementation of the defined system interfaces. In GNU/Linux systems, the GNU C Library (glibc) project including the Native POSIX Thread Library (NPTL) and the librt library implements the POSIX thread interface, including real-time extensions, along with some specific additions [Mic; DM03]. To utilize the POSIX interface, user applications link against the glibc libraries. The glibc libraries, in turn, use the Linux system call interface as shown in Figure 2.4.

A remarkable difference between the glibc libraries and the POSIX standard is the addition of Linux-specific extensions. Next to additional scheduling options, the `timer_create` interface in glibc extends the POSIX interface with an additional option to specify the target thread that shall receive the timer interrupt. Not only is glibc extending existing POSIX interfaces,

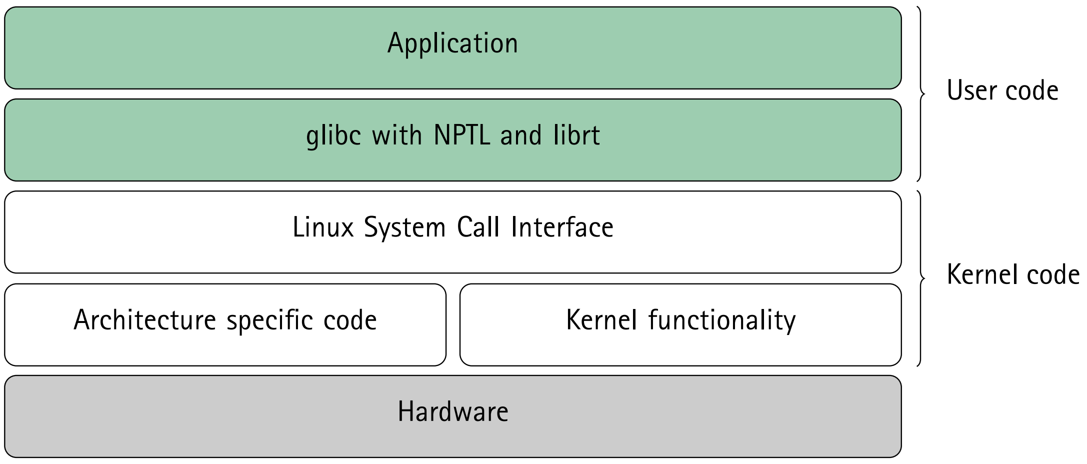

Application
glibc with NPTL and librt
Linux System Call Interface
Architecture specific code
Kernel functionality
Hardware
User code
Kernel code

Figure 2.4 – POSIX system call interface in GNU/Linux

it additionally provides useful, albeit non-portable interfaces that are not part of POSIX, like `pthread_setaffinity_np` to set a thread's affinity to certain cores [Mic].

2.5 Compiler and Code Toolchain

In order to execute programs on a CPU, either compilation or runtime interpretation is required to translate the high-level language code into machine code that can be executed by the underlying hardware microarchitecture. In compiler design, the classical approach for this translation consists of three phases: frontend, optimization, and backend. The frontend involves parsing the source code and checking it for errors, resulting in a language-specific Abstract Syntax Tree (AST) that contains all relevant tokens for the program's functionality. In the next phase, the AST is converted to an intermediate format that is suitable for the optimization steps. The third step is the backend, which transforms the intermediate code into the target instruction set. One advantage of this design is that frontends for different programming languages or backends for different instruction sets can be exchanged, while common optimization steps do not need to be reimplemented. This modular architecture reduces the effort of supporting new programming languages or porting the compiler to a new target platform because it minimizes code duplication.

LLVM is a collection of compiler tools that follows this approach and strictly separates the three phases. The project has designed the LLVM Intermediate Representation (IR) language, which is used for the optimization and backend steps. Compared to GCC, where the intermediate language is not designed to be used externally, the LLVM IR can be modified or optimized using the LLVM API [Fre; Lat11]. In Figure 2.5 the phases of the LLVM compiler toolchain are depicted for C code on an x86 architecture.

For the generator implementation in this thesis, the next subsection will describe the LLVM IR code in more detail.

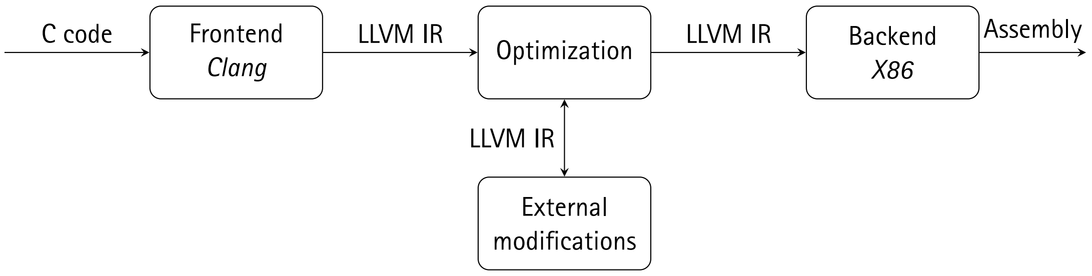


Figure 2.5 – LLVM toolchain for C code

2.5.1 LLVM Intermediate Representation

The LLVM IR is designed to be a flexible and language-agnostic representation of high-level languages. It serves as a common language during compilation and optimization steps in the LLVM toolchain and can be available in both human-readable and bytecode formats. In LLVM IR, functions are decomposed into basic blocks, which are linked together to form a Control Flow Graph (CFG). The CFG represents the branching of possible control flows between the basic blocks. Each basic block has a defined set of predecessors and successors, which is the base for optimizations at the function level. The first basic block of a function has no predecessor, and it is executed automatically on function entry. The LLVM IR language shares similarities with Reduced Instruction Set Computer (RISC) opcodes, but it has a strong and simple type system [Lat11]. This type system offers various advantages, such as increased readability and more possible optimizations at the IR level [LLVa].

Listing 2.4 displays a comparison of an LLVM function with one basic block with the corresponding C and assembly code. The increment operator on variables declared `_Atomic` in the C language translates to an `atomicrmw add` instruction in LLVM. For x86 assembly, this will insert a `lock` instruction prefix to prevent simultaneous access to the shared data bus, as described in subsection 2.2.2.

```c
void atomic_fetch_and_inc(_Atomic(int) *i) {
    (*i)++;
    return;
}
```

```
atomic_fetch_and_inc:
    mov     0x4(%esp),%eax
    lock addl $0x1,(%eax)
    ret
```

```
define void @atomic_fetch_and_inc(i32* %0) {
entry:
    %2 = atomicrmw add i32* %0, i32 1 seq_cst, align 4
    ret void
}
```

Listing 2.4 – Atomic instruction in C (left), x86 assembly (right) and LLVM IR (bottom)

As mentioned before, the IR code is very well suited for optimization and external modification using the LLVM API. The upcoming subsection will provide further details on such modifications.

2.5.2 Automatic Source Code Modification

Compiler technologies have evolved over time to provide many optimizations, which are possible at every abstraction level. These optimizations aim to improve non-functional requirements such as execution time, memory footprint, and power consumption. The functional requirements of the source code remain unchanged, and optimizations must not alter the program's visible state during execution. Optimizations performed at the high-level source code level in the AST are universal and machine independent. However, code modifications at this level may not fully exhaust all possibilities and advantages of the hardware architecture. Alternatively, optimization at the assembly instruction level is only applicable to a specific instruction set or architecture and can improve code performance by replacing or reordering instructions. Finally, optimization at the intermediate code level is both machine-independent and similar to assembly opcodes, allowing for many hardware-related optimizations using heuristics or suitable metrics.

To provide an interface for operations on the LLVM IR code, the LLVM C++ API can be used. The API can be utilized to replace, insert, or modify instructions, basic blocks, or functions [LLVb]. LLVM is designed to be highly modular, every optimization pass is independent of the others or may declare dependencies to other passes where necessary. This results in a pipeline, where each pass directly works on the IR code. This modularity allows for extensions and granular unit-testing [Lat11].

Listing 2.5 shows an example usage of the LLVM API.

```cpp
Function &Func = module.getFunction("main");
cout << "Function " << Func.getName() << endl;
for (BasicBlock &BB : Func)
    cout << "Basic block " << BB.getName() << endl;
```

Listing 2.5 – Accessing basic blocks with the LLVM C++ API [LLVb]

A related domain to code compilation and optimization is source code generation from a static system description. The generator is a translation unit that creates lower-level code from a higher-level description, thus it has the same function as a compiler. Statically configured embedded operating systems like OSEK follow this approach, where the system description in OIL format must be transformed into C code [OSE04b]. The categorization of optimizations specifically targeted to statically configured RTOS is described in the following subsection.

2.5.3 RTOS Specialization

Automatic code optimization for static RTOSs can be much deeper than in normal applications because all possible system inputs can be calculated a priori. At compile-time, the system can be specialized to the use case of the specific application. The key objective of system specialization is to improve non-functional requirements, while the functionality of the system and application is not affected. This specialization usually comes at the cost of flexibility, which is not required in static RTOS as the application scenario does not change for the product's lifetime. To provide a classification for various specialization levels, the following taxonomy has been introduced [Fie+18]:

Specialization of Abstractions

Removing certain abstractions from the system is a simple specialization that can be used even in general-purpose operating systems. If a real-time application does only require a single core, an operating system that can support multicore applications can be specialized in various aspects. Semantics of cores, locking mechanisms, and cross-core communication interfaces can be removed, without affecting the application. Another example is reducing the system call interface by removing alarm or event handling if the application does not require these mechanisms.

Specialization of Instances

A more specialized system can create concrete instances of abstractions, which requires more static knowledge about the application. This can involve static initialization of tasks, including their scheduling parameters, activation state, or name. RTOS with statically defined instances, such as OSEK/AUTOSAR, enable this specialization by design. The OIL system description contains the static information that is required for this type of specialization.

Specialization of Interactions

The most specialized optimization can be applied at the interaction level between instances. With knowledge about interactions between system objects, kernel paths can be accelerated by reducing the overhead that is not needed for specific interactions. This concept can be utilized for whole-system optimization with static code analysis that examines all possible interaction sequences.

This specialization classification is used in the following sections to differentiate the depth of specializations in related work.

2.6 *d*OSEK System Analysis and Synthesis

As this work is using the *d*OSEK project as a foundation, it will now be introduced in detail.

*d*OSEK is a research project focused on improving dependability and tolerance against transient hardware faults in embedded operating systems. By static system design with *Specialization of Interactions*, transient hardware faults are mitigated, as fewer indirections improve the tolerance. Additionally, system objects are arithmetically encoded to harden the kernel and allow error detection to prevent silent data corruption. Apart from the dependability features, *d*OSEK is designed as a statically configured OSEK/AUTOSAR-compliant RTOS [Hof+15]. Because of the focus on dependability, *d*OSEK does not implement all OSEK features, like multiple tasks per priority or multiple activations per task. While *d*OSEK supports x86 and ARM platforms, as well as POSIX as a virtual platform, multicore functionality is not part of the implementation. To generate a *d*OSEK system image, a whole-system compilation approach involving two steps is employed: static analysis of the application and tailoring the system to the application's needs. Because inter-task interactions have a predictable behavior with the compile-time knowledge from the system analysis, system calls can be optimized to specialize interactions and to improve the non-functional properties [Hof+15].

One application of this whole-system approach is the System State Enumeration (SSE), which creates a global CFG that incorporates both the user application and kernel logic. In most modern compilers, CFGs are created to allow optimization on function or program level (as discussed in subsection 2.5.1). However, such optimizations cannot cover the transition from the application to the kernel code as it is required for system calls, because detailed system behavior is not predictable in non-real-time and non-deterministic operating systems. The SSE incorporates semantics of an RTOS and the static configuration knowledge into the control-flow analysis to provide the global CFG. The elements in the global CFG are grouped abstract system states, which include all relevant status information of the system, such as task states, occupied resources, and their call stack at a specific point in the control flow. Unlike the local CFG in LLVM, the global CFG does not operate on basic blocks. Instead, the concept of Atomic Basic Blocks (ABBs) is used, where all basic blocks that do not interact with the operating system are merged into one ABB, while system calls remain visible as a single ABB. This abstraction hides the complexity of the application logic and focuses on the interaction with the system objects during analysis. During the system analysis, all possible abstract system states are enumerated and connected in a state transition graph, which is then used to create the global CFG by grouping the states. This graph structure provides a view of all possible control flows across the system call interface and enables the specialization of system calls for each call site. While most OSEK implementations use a generic system call interface, fine-grained control over the implementation of system services per call site can reduce overhead by providing only the minimum necessary functionality. If the global CFG contains only one successor state to an `ActivateTask` system call, the scheduling decision is known ahead of time, and the system call can be specialized to directly dispatch to the following task. When the exact scheduling decision cannot be determined by the analysis, at least a subset of possible tasks can be considered in a partially specialized system call. System call specialization using cross-kernel control flow knowledge increases the runtime performance in the kernel path while increasing the code memory usage [DHL15]. The SSE enables optimization by *Specialization of Interactions*, therefore it provides very extensive specialization possibilities [Fie+18].

Similar to LLVM passes, the analysis steps are orchestrated in a pipeline with interdependencies, and the results are stored in a common graph. The system generator uses the results from the OIL reading pass and several other analysis steps to create the *d*OSEK code. To reduce indirections, each system call in the application is replaced with a specialized variant that statically contains the parameters. This replacement requires modifications to the application, which are performed in the IR code. The generated code is then linked together with the modified application and the system libraries to create the system image.

2.7 MultiSSE: Whole-System Analysis and Optimization with ARA

ARA is a framework and compiler toolchain for static system analysis and optimization of embedded applications and supports multiple RTOS. It was created as a research project to analyze instance creation and interactions in OSEK and FreeRTOS [ESD19]. Based on the instance analysis, ARA was extended to also transform systems with dynamic instantiation like FreeRTOS by statically initializing objects that are dynamically allocated on each boot. With this approach called Static Instance Analysis (SIA), ARA utilizes whole-system analysis to modify application code and specialize the system to reduce boot time [Fie+21]. Building upon previous results, the whole-system analysis was applied to an abstract OS model to enable analysis steps to work across multiple different operating systems. The system model for each RTOS is used to statically analyze system behavior, state transitions, and system object instances. While this is applicable for various analysis tasks, OS-specific semantics, such as scheduling decisions, must be considered as well [ENL22]. Like in *d*OSEK, steps are separated and can store global knowledge about the analyzed application into a common graph.

For synthesis, the ARA generator was adapted from *d*OSEK for FreeRTOS on an ARM platform. Figure 2.6 provides an overview of the structure of ARA optimization steps including the abstract OS model.

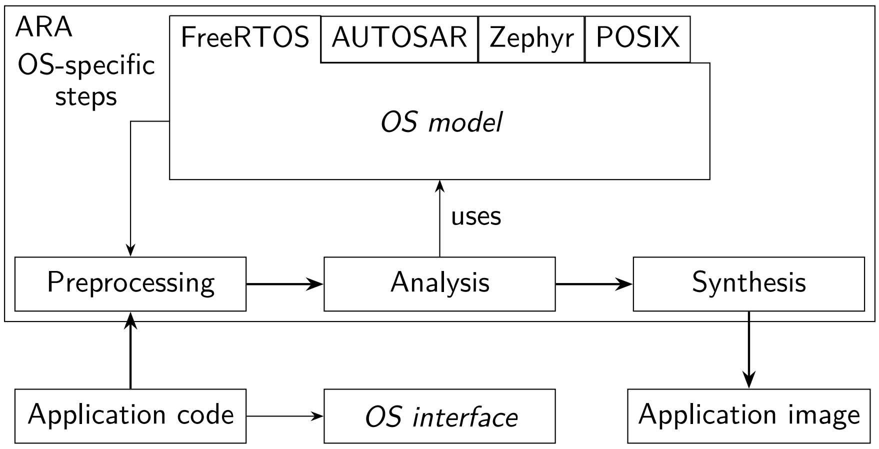


Figure 2.6 – Whole-system optimization with ARA [ENL22]

Next to the SIA and the SSE, which was ported from *d*OSEK, ARA includes the MultiSSE analysis. While the SSE supports only single-core applications, the MultiSSE extends the cross-kernel control-flow analysis concept to multicore systems.

The key idea of MultiSSE approach is to create a Multicore State Transition Graph (MSTG) of all possible abstract system states for the multicore system. One multicore system state includes the local state of multiple cores and global OS objects at a given time in the system. To keep the number of states as low as possible, the MultiSSE first creates core-local system states like the SSE and synchronizes these states only when required, instead of combining all possible system states of all cores. Such Synchronization Points (SPs) are created for the affected cores of cross-core system calls and depend on the RTOS semantics. For example, a `GetSpinlock` system call in AUTOSAR synchronizes all cores that contend for the same spinlock. As a result, the MSTG contains two different types of nodes, either core-local states or SPs between two or more cores. Every possible execution order of the parallel cores must be considered when analyzing multicore applications, so multiple local states can originate from the same ABB. Additionally, pre-calculated timing information per ABB can be leveraged to reduce the number of states in the MSTG by considering worst-case and best-case execution times when pairing the local states at SPs. This reduction of the MSTG leads to a reduced analysis time and potentially more optimizations.

An example synchronization point, including four tasks on three cores, is shown in Figure 2.7. At the system start, only task `T11` is running on core 1, while the other cores are idling and their tasks are suspended [EFL23].

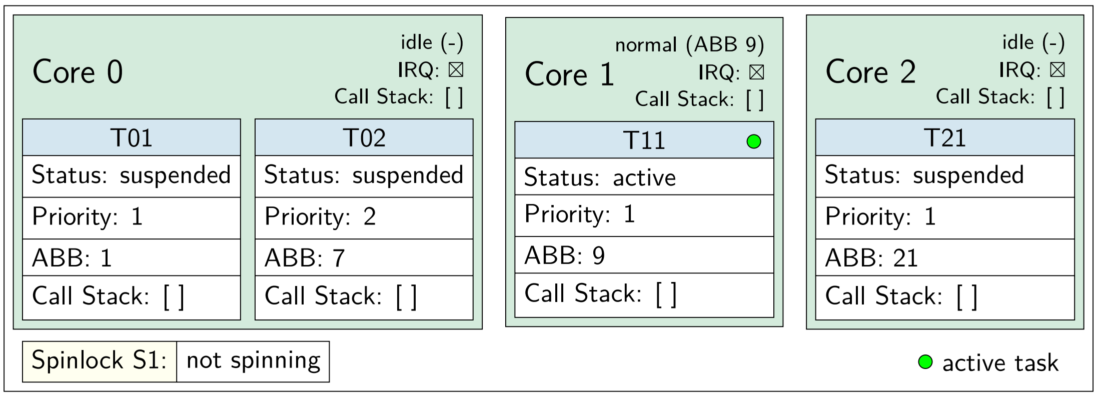


Figure 2.7 – MultiSSE example initial system state (synchronization point) [EFL23]

Detailed information about interactions between the application and the RTOS can be extracted from the MSTG. Just like the SSE enables system call specializations for single-core applications, the MultiSSE can be used for the following multicore optimizations in AUTOSAR [EFL23].

Lock Elision

In a multicore AUTOSAR system, `GetSpinlock` is used for mutual exclusion of critical sections. If no state in the MSTG is actually waiting for the global spinlock object, the complete lock, including all related system calls, could be removed from the generated system. If the spinlock only spins for certain call sites, these system calls can be specialized, while the others are unaffected. This optimization can mitigate architecture-specific costs of spinlocks like shared bus locking and cache coherence overhead. Additionally, locking structures that are never required can be removed to reduce the system image size.

Deadlock Detection

Deadlocks happen if one core executes `GetSpinlock` on a lock that is already held by the same core. Such deadlocks must be prevented in AUTOSAR by bookkeeping and returning an error code to the system caller. The same issue can arise if nested spinlocks are not locked and unlocked in the correct order. According to the AUTOSAR specification, such deadlocks must be prevented already during the generation phase by the definition of a fixed spinlock ordering [AUT22]. As the MSTG already contains the necessary information to allow checks for deadlock prevention independent of their ordering, this is not required in ARA.

IPI Avoidance

For cross-core `ActivateTask` system calls, an IPI must be issued to trigger rescheduling on the target core. This rescheduling is not needed if the currently running task has a higher priority than the task marked ready, and the IPI can be omitted. The same logic applies to other cross-core system calls like `SetEvent`. If the receiving core does not have an active task that is waiting for this `Event`, or a task with higher priority than the waiting task is running, the IPI is unnecessary. With this optimization, the runtime performance of the system can be improved because fewer interrupts are triggered, which mitigates additional IPI-related timing overhead.

In summary, the specialization of cross-core system calls can increase the system performance by reducing delay and jitter with the elision of costly cross-core operations [EFL23].

2.8 Related Work

Numerous operating systems have been developed for embedded systems and vary in terms of their features, supported platforms, and level of specialization. This section focuses on open-source RTOS projects with whole-system optimization and specialization in mind. Some commonly used RTOS in the industry are Zephyr, FreeRTOS, and Erika OS.

Zephyr is a versatile and relatively new RTOS with lots of modern features. It supports multiple platforms and offers a virtual POSIX platform for testing and development. Like a Linux kernel, Zephyr can be configured using Kconfig to disable unnecessary functions. Furthermore, Zephyr allows resource definition at compile time, which can be used for *Specialization of Instances* [Zep]. However, Zephyr does not use static application analysis to specialize system calls on a per-usage level.

FreeRTOS is a popular open-source RTOS kernel in the market that supports many platforms and provides a port for a Linux/POSIX system for development [Ama]. Specializing FreeRTOS systems is challenging because it provides dynamic system object creation. While *Specialization of Instances* is possible by preallocation of pseudo-dynamic objects, the application of static analysis for further optimization is limited, and interactions between dynamic system objects cannot be specialized [ESD19; Fie+21].

Erika Enterprise v3 is another well-established embedded operating system kernel that conforms to the OSEK/AUTOSAR standards. It was designed with multicore support and utilizes the RT-Druid tool for generating static RTOS code from an OIL file [Evi]. Although the RT-Druid generator supports some analysis plugins, it is unable to take application-specific analysis information into account, and its ability to optimize the RTOS beyond the instances defined in the system description is limited.

Apart from the industry-grade systems, scientific research has created other relevant RTOSs. Trampoline RTOS is an embedded operating system that implements the OSEK/AUTOSAR specifications and also supports multiple hardware platforms. It supports all OSEK features, memory protection features, and isolation [Bec+06]. As an OSEK conforming OS, *Specialization of Instances* by statically defining all OS resources is possible. Nevertheless, Trampoline does not support multicore applications, except for the PowerPC platform. To specialize the RTOS for the application, a Petri net OS model is used for analyzing reachability and pruning dead code from the system [Tig+17]. However, this OS model analysis was introduced to replace preprocessor macros in the code to automate the *Specialization of Abstractions*. In addition to the dead code elimination, the operating system model is used for formal verification, including support for multicore applications [HBR21].

In contrast to the previously mentioned systems, *d*OSEK provides the deepest application-specific optimization by using knowledge from static code analysis for *Specialization of Interactions*. As explained in section 2.6, *d*OSEK specializes system call sites to improve dependability and performance. Further publications like Semi-Extended Tasks for optimization of memory usage or OSEK-V hardware specialization based on interaction analysis are implemented in the *d*OSEK framework [DL18; DL17]. Yet, due to the absence of support for multicore applications, the *d*OSEK generator in its current form cannot be used for cross-core optimizations such as Lock Elision and IPI Avoidance.

ARCHITECTURE 3

This chapter presents the architecture and implementation of the **M**ulticore **A**UTOSAR **C**ompatible **A**pplication-specific **W**hole-system-optimizer (MACAW) RTOS generator. While MACAW is largely based upon the *d*OSEK system generator, this thesis shifts the focus from dependability to cross-core runtime optimizations by interaction specialization as described in section 2.7.

3.1 Architecture and Framework

The architecture of the operating system generator follows the generic OSEK/AUTOSAR image generation approach (see Figure 2.3), with the additional specialization of cross-core interactions. The desired level of specialization for multicore applications requires static code analysis with knowledge about the RTOS semantics and enumeration of all possible system states. With the AUTOSAR OS model and the MultiSSE analysis in ARA, this knowledge is available for AUTOSAR applications. In order to leverage this knowledge, MACAW RTOS is implemented into the ARA whole-system-compiler. Specifically, the system generator component serves as the synthesis module of the ARA toolchain (illustrated in Figure 2.6).

For implementing the core functionality of an AUTOSAR kernel and the generator, *d*OSEK is used as a base because it already decouples the kernel and the system call implementation for interaction-specific optimization. As *d*OSEK only supports single-core applications, additional system features need to be implemented to support multicore AUTOSAR use cases. As the focus of this work is cross-core runtime optimization, MACAW currently only supports the virtual POSIX platform on Linux. This restriction allows for faster development because the system generator can be tested without additional hardware and deployment, and hardware-specific details can be ignored.

The diagram in Figure 3.1 displays a high-level overview of the AUTOSAR image generation process in ARA.

To perform the static system analysis, ARA reads the system description and the application's IR code to construct the required data structures, such as the instance interaction graph, CFG, and MSTG. Optimizations for cross-core interactions, such as Lock Elision, leverage the MultiSSE output to identify patterns or system calls that can be replaced. The system generator then employs the analysis results to synthesize specialized code that associates the application-specific system calls with the application-independent AUTOSAR libraries. Additionally, the generator modifies the IR code to enable system call specializations and inserts startup code to ensure the correct initialization of the system and tasks.

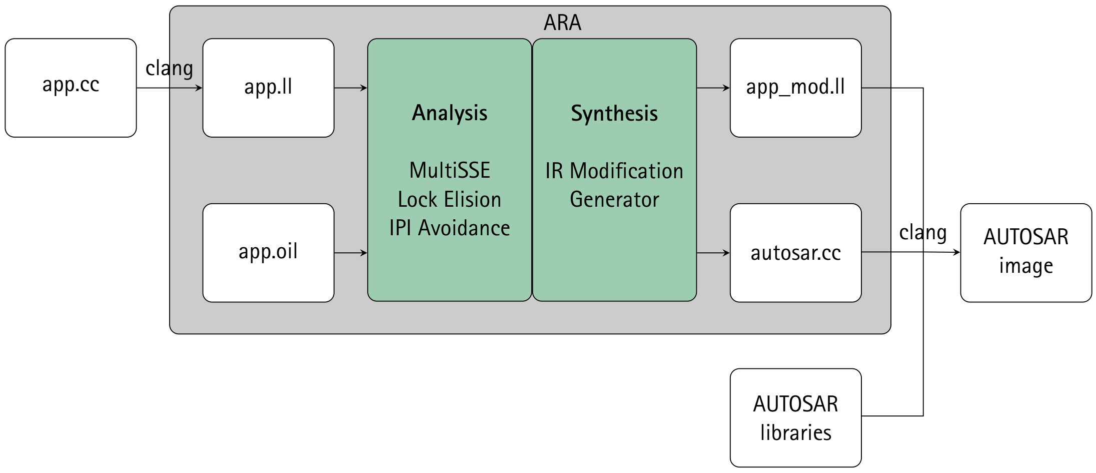


Figure 3.1 – AUTOSAR image generation with ARA

The following sections provide the implementation details of the application-independent AUTOSAR kernel libraries and the code generator.

3.2 AUTOSAR Kernel Libraries Implementation

ARA is configured using the Meson build system, which enables the choice of architecture, test cases, and the selection of compilers and tools. To add the MACAW system generation to ARA, I have added initial support for the POSIX target platform, and inappropriate test cases for this architecture must be disabled.

The implementation of the kernel functionality is split into multiple separate libraries. While the `libautosar_os` library provides fundamental interfaces like the scheduler, hook declarations, and all OSEK/AUTOSAR types, the `libautosar_posix` and `libautosar_generic` libraries provide the architecture-dependent functionality like the spinlock implementation, Task Control Blocks (TCBs), and interrupt handling. Another library named `libautosar_test` is required for testing functionality only, not for the actual AUTOSAR implementation.

With *d*OSEK as a base, these libraries need extensions and adaptations to support the required multicore functionality. The following paragraphs describe the components that are implemented into the kernel libraries of every MACAW system, independent of the application.

Logging

For development and debugging, *d*OSEK uses a `DEBUG` preprocessor symbol to enable logging output to `STDERR`. The implementation of the logging uses the C++ stream operator and the `write` system call. With multicore debugging output, per-core buffering is required to prevent character interleaving which renders the log useless. Additional locking is not needed because the `write` system call itself is atomic per POSIX specification, if the buffer length does not exceed 512 bytes [IEE17]. In the MACAW implementation, the buffer is flushed after 80 characters or line breaks.

Spinlocks

The AUTOSAR specification mandates the `GetSpinlock/TryToGetSpinlock/ReleaseSpinlock` system calls. Implementations for spinlocks are architecture-specific and require atomic operations like Compare and Swap (CAS) (see subsection 2.2.2) for synchronization between multiple cores [AUT22]. The spinlock functionality of the POSIX interface, as described in section 2.4, is used to implement the AUTOSAR-conformant spinlock functions. The Linux manual states that spinlocks should be used in combination with a real-time scheduling policy only, and thread placement must be considered to prevent deadlocks [Mic]. The following paragraph (Core Handling) describes how MACAW handles thread placement and provides real-time behavior.

To ensure the reliability of the AUTOSAR system, it is required to handle certain cases to prevent deadlocks. Firstly, when invoking the `GetSpinlock` system call with the same lock on the same core, the system should return `E_OS_SPINLOCK`. Secondly, if a task calls `ChainTask/TerminateTask`, the system must check for unreleased spinlocks and return the same error if the task holds any locks. Similar to the event implementation in *d*OSEK, each task in MACAW contains a bitmask to store whether any spinlocks are currently locked. Furthermore, spinlocks are allowed to be nested only in a predefined order to prevent deadlocks, which is a requirement for OS generation. The MultiSSE already enables Deadlock Detection, thereby fulfilling this requirement without any need for nesting order definitions.

Unlike in OSEK, the `ChainTask/TerminateTask` system calls can fail because of unreleased spinlocks. When reaching the end of the task function without a successful rescheduling, another `ErrorHook` is triggered with the status code `E_OS_MISSINGEND` [AUT22]. If the user-defined hook does not reschedule, MACAW forces a shutdown.

Core Handling

Multicore functionality requires architecture-specific code for starting and stopping cores. In the POSIX platform in *d*OSEK, a core corresponds to a thread that is scheduled by the host kernel. For TCB switching, the POSIX virtual platform makes use of x86-specific assembly code.

In contrast, the FreeRTOS POSIX simulator for Linux creates a new POSIX thread for each task. While this approach has the benefit of using platform-independent `pthread` condition signals instead of assembly code for TCB switching, it also results in many signals being delivered to different threads in the process [Ama]. As a consequence, task dispatching becomes slow in comparison to using assembly instructions. Also, instead of per-core timer interrupts implemented using POSIX timers, multiple threads on one core would require additional interrupt dispatching. To add multicore functionality to the *d*OSEK approach, new cores are created with `pthread_create`. This call takes a start routine argument that specifies the entry point of the created thread (as shown in Listing 2.3). For IPIs, the core startup function stores the `pthread_t` identifier in an array. Additionally, MACAW keeps track of the number of cores in a global `cores` counter variable for the synchronization function (in Listing 3.2). Finally, the new core executes the `main` function defined in the application. Nevertheless, the cores cannot be shut down safely with `pthread_exit`, because the task-switching mechanism manipulates the stack, and the stack unwinding will fail. To circumvent this, the shutdown sequence will disable all interrupts and wait in an endless loop, as defined in AUTOSAR [AUT22]. The last core then executes `ShutdownMachine` which corresponds to `exit` in POSIX, thus all threads will be terminated, and the process will be cleaned up (shown in Figure 3.2) [IEE17].

To run all threads actually in parallel on a Linux host, the number of threads is limited to the number of available cores on the host and a maximum of 16 cores by static definitions. To achieve real-time behavior, MACAW sets the CPU affinity for each thread to a different host core, and the whole process must be scheduled with a real-time policy. Although a common Linux kernel as a general-purpose operating system does not support real-time applications by default, the `PREEMPT_RT` patch set provides real-time priority scheduling. To enable this scheduling policy for the MACAW system, the process can be started with `chrt(1)` [Mic].

Synchronization, Startup and Shutdown Sequence

Differences in hardware platforms and synchronization problems require a predefined multicore startup and shutdown sequence. During the startup phase, the first core, which is identified by `OS_CORE_ID_MASTER`, can start other cores using the `StartCore` system call as shown in Listing 3.1. Each core can activate new cores itself until `StartOS` is called.

```
int main(void) {
    StatusType rv;
    switch(GetCoreID()) {
    case OS_CORE_ID_MASTER:
        StartCore(OS_CORE_ID_1, &rv);
    default:
        StartOS(0);
    }
}
```

Listing 3.1 – Startup sequence for two cores

To execute the `StartupHook` and the scheduling of the first task on all cores simultaneously, AUTOSAR defines two synchronization points in `StartOS`. For the shutdown sequence, AUTOSAR specifies two different mechanisms for backward compatibility with OSEK. In an OSEK application, the calling core stops scheduling after calling `ShutdownOS`, immediately followed by the `ShutdownHook`. The synchronized shutdown of multiple cores in AUTOSAR is triggered using `ShutdownAllCores`, where a synchronization point is introduced before the simultaneous execution of the `ShutdownHook` on all cores [AUT22]. MACAW adds a fourth synchronization point in the shutdown sequence and waits until all cores have finished the `ShutdownHook` before the machine is shut down completely. Figure 3.2 displays these synchronization points from system start until shutdown for two AUTOSAR cores.

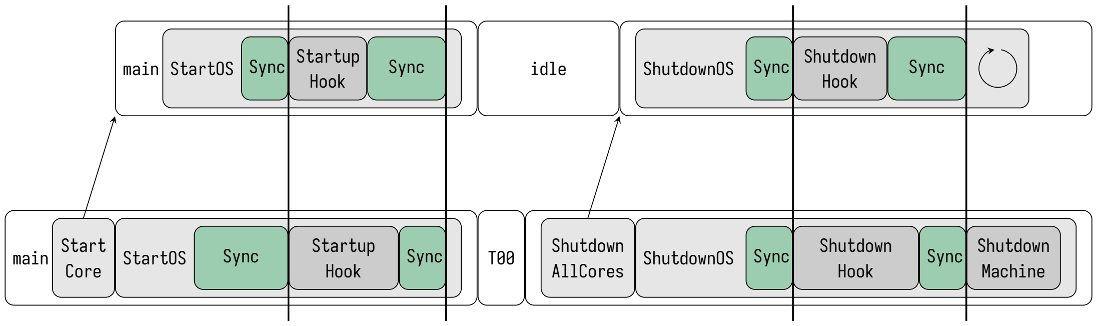


Figure 3.2 – Multicore synchronization points [AUT22, adapted]

The implementation of the synchronization points requires locking or atomic operations because all threads concurrently modify a global counter. POSIX provides exactly this synchronization mechanism with `pthread_barrier` [IEE17], which is internally implemented using spinlocks in glibc. However, the `pthread_barrier` requires a fixed number of threads to be synchronized at initialization, which is not given at the first synchronization point during the startup phase. Therefore, MACAW directly uses atomic operations for the synchronization of cores.

Listing 3.2 shows the implemented synchronization function that must be executed by all active `cores` to achieve synchronicity. The number of `syncing_cores` is counted until all threads are synchronized to the last core that enters the while-loop. Similar to the Trampoline implementation, multiple `sync_points` ensure correct synchronization throughout the lifetime of the system. Unlike the `pthread_barrier`, the implemented synchronization mechanism is capable of handling the simultaneous increment of active `cores` during the startup phase.

```
_Atomic(int) cores = 1;
_Atomic(int) syncing_cores[arch::SYNC_MAX] = {0};
void sync_all_hardware_threads(cpu_sync_point_t sync_point) {
    syncing_cores[sync_point]++;
    // wait until all cores have synced
    while (syncing_cores[sync_point] < cores);
}
```

Listing 3.2 – Synchronization of multiple cores

Scheduling

Compared to Trampoline, where each system call is locked using a kernel spinlock, the system calls and the scheduling implementation in MACAW are lock-free. Each core schedules independently and in the case of cross-core interactions like `ActivateTask`, the data structures of the scheduler on the target core are modified atomically to prevent data races.

The scheduling implementation is created by template expansion as part of the code generator described in subsection 3.3.2.

Interrupts and Timers

Interrupts can be modeled using signals on the POSIX platform. In *d*OSEK, sending a signal to the whole process is sufficient because only one thread in the process exists that will handle this signal. Interrupts in multicore systems, and particularly IPIs on POSIX, require that the receiving thread can be specified.

Cross-core interactions like `SetEvent` in AUTOSAR need a synchronized IPI that returns only after the system call completes on the remote core. The interrupt trigger implementation in MACAW takes the signal identifier, target core, and a `sync` flag as arguments, as displayed in Listing 3.3. After the IPI is triggered, the calling core waits until the target core has processed all interrupt handlers and reschedules.

In the AUTOSAR specification, a `Counter` can be defined either as a manually incremented software counter or as a hardware counter triggered by a timer interrupt. Each core must provide its own hardware counter, and the counter value of other cores must not be modified [AUT22].

```
_Atomic(bool) ipi_cleared[MAX_CPUS];
void IRQ::trigger_interrupt(int irq, int cpuid, bool sync) {
    if (!cpu_online(cpuid)) {
        return;
    }
    if (!sync) {
        pthread_kill(get_thread_id(cpuid), irq);
        return;
    }
    /* synchronized interrupt: wait until target core does reschedule */
    ipi_cleared[cpuid] = false;
    pthread_kill(get_thread_id(cpuid), irq);
    while(ipi_cleared[cpuid] == false);
}
```

Listing 3.3 – Interrupt trigger mechanism

While the POSIX standard does provide `timer_create` and `timer_settime` system calls, the thread identifier where the interrupt will be triggered cannot be specified [IEE17]. As described in section 2.4, the implementation of `timer_create` in Linux extends the POSIX interface with an additional option to specify the target thread that shall receive the timer interrupt [Mic]. MACAW uses this extension to provide timer functionality for each thread. An alternative approach would be to create one timer per core with different signals and unblock only one of those timers per thread. However, this would also impact the implementation of other interrupt-related system calls, such as `EnableAllInterrupts`.

For user-defined interrupt handlers for AUTOSAR hardware interrupts, MACAW also uses POSIX signals. Because POSIX standard signals can be queued only once according to the standard, such interrupts should be defined within the range of real-time signals, specifically between `SIGRTMIN` and `SIGRTMAX`, to prevent lost interrupts. The `guardian` function is registered as the handler for all POSIX signals in the process, dispatches to the appropriate ISR for the received signal, and reschedules if the ISR requests it. Table 3.1 provides an overview of the signals used in the system.

Signal	Function
`SIGUSR1`	Trigger `Reschedule`
`SIGUSR2`	Trigger synchronized shutdown
`SIGALRM`	Timer interrupt with 1 kHz
`SIGRTMIN..SIGRTMAX`	user-defined ISRs

Table 3.1 – POSIX signals employed in MACAW

Test Code

The evaluation of the implemented system in chapter 4 requires a testing interface within the system image. When modified for thread safety using atomic test state variables, the test functionality provided by *d*OSEK is acceptable for multicore testing as well. Tests can be implemented as an execution order test using a `test_trace` function and comparison to an expected trace or as assertions of single expected values. The initialization of the test framework is done by calling the `TEST_MAKE_OS_MAIN` preprocessor macro with the desired startup sequence. Listing 3.4 shows the resulting output of a successful test execution of an example application (Appendix A).

```
expect: 0123E
traced: 0123E
    +
SUCCESS 1 0:0
Tests finished: ALL OK
```

Listing 3.4 – Test execution result

The Meson unit test system is utilized to automate the execution of available tests by reading the output of the test results. Available tests are described in chapter 4.

After the details about the implemented system functionality, the following section describes the MACAW code generator and the application-specific optimization of system calls.

3.3 Code Generator Implementation

As already shown in Figure 3.1, the MACAW RTOS synthesis consists of two steps: IR modification and code generation. The IR modification makes use of the LLVM API to insert startup code and error handling calls into the application code. Details about these modifications are explained in subsection 3.3.1.

As introduced in section 2.7, the code generator step of ARA implemented in Python is based on the *d*OSEK code generator and consists of three components. Firstly, the operating system generic generator rules include the instantiation and initialization of operating system objects. In the case of AUTOSAR, this includes the architecture-independent setup of statically defined instances like tasks, alarms, and spinlocks. Secondly, the architecture-specific part is required for hardware-dependent code blocks like interrupt handling, TCBs, stack allocation, and linker scripts. The interface to this part is defined in a common generic architecture. The last part is the system call implementation, where either generic or specialized system calls are generated for the application. These components, described in detail from subsection 3.3.2 to 3.3.5, generate the code required for linking the application with the library kernel.

3.3.1 IR Code Modification

The C++ AUTOSAR generator step uses the LLVM API to iterate through the functions and basic blocks defined in the application (see Listing 2.5). At the beginning of the `main` function, a call to `arch_startup` is inserted to initialize the hardware before executing the first system call. Instead of this code insertion, the system initialization could also be done at the first system call, but that would require additional logic inside multiple system calls that could potentially be the first one. Furthermore, the ARA generator step modifies each `Task` function at function entry and exit. For each `Task`, a call to `kickoff` is prepended to enable interrupts at the start of each function. Compared to a stack modification to jump to this `kickoff` function, the insertion approach has multiple advantages. The call insertion at the IR level is independent of the calling convention, does not require hardware-specific assembly code, and allows for the insertion of specialized kickoff variants as well. At the end of AUTOSAR `Task` functions, the specification requires a call to `Terminate/ChainTask`. If a function returns without such calls, the `E_OS_MISSINGEND` error is raised to prevent returning to an invalid location.

The most important IR modification is call-site-specific system call replacement. By appending a call-site-specific number to the call instruction of supported system calls, each call site can be implemented by a specialized function. Because the creation of ABBs in a previous analysis step first splits the LLVM basic blocks directly before and after a system call, one system call site is represented exactly by one basic block and one ABB. Although the multicore analysis works mostly on ABBs, in the domain of IR code, the basic block is the preferred identifier because its usage is straightforward and it is decoupled from the graph representation. For example, the test application in Appendix A contains a call to `ActivateTask`, which is specialized to `AUTOSAR_ActivateTask_BB18`. Currently, all system calls regarding the OS objects can be specialized, including `SetEvent` for IPI Avoidance and `GetSpinlock` for Lock Elision.

Instead of iterating over the functions and their basic blocks in the IR code, the links between ABB and IR code in the ARA control flow graph could be used for system call replacement and code insertion, too. However, it is easier to directly use the LLVM API for code insertion and modification compared to obtaining the required information from the ARA graph, so the MACAW generator modifies the IR code without depending on the ARA graph data.

3.3.2 AUTOSAR OS Generation

After the modification of the application's IR code, the application-specific AUTOSAR code is generated into the `autosar.cc` file (see Figure 3.1).

In the architecture-independent part, the system objects from the system description file are created. This includes the initialization of `Tasks` with their scheduling options such as priority, the core they are running on, and a task identifier. If the system description contains `Alarms`, the required objects with the related `Counter` and initial alarm configuration are allocated. `Events` and `Spinlocks` are generated as constant integer identifiers that are used by the scheduling functions. To enable bitmasking, these identifiers are enumerated by one-hot coded integers, so they are limited to 32 instances on the implemented architecture. Additionally, `Spinlock` identifiers are backed by objects allowing the actual busy-waiting functionality.

Next to the data objects, the system hooks like `StartupHook` and `ShutdownHook` are generated, which call the corresponding user hooks if they are defined.

A large part of the generated OS code is created from the scheduler and tasklist templates introduced in *d*OSEK. These templates contain the framework logic for priority-based scheduling for a fixed number of tasks and provide an interface for system calls to interact with the scheduler. For example, the interface for activating a task consists of two operations. First, the activated task is marked as ready in the static tasklist of the scheduler, then the dispatcher switches the context to the task with the highest priority.

3.3.3 Architecture Specific Code

In addition to the generic AUTOSAR instances, architecture-specific code must be generated. To reduce the memory footprint, MACAW supports and extends stack sharing for `Basic Tasks`, as introduced by *d*OSEK. Since tasks are statically assigned to specific cores, each core must have its own shared stack for `Basic Tasks`. Only if the task requires `Events`, the generator creates an `Extended Task` with a separate stack. The TCB for each `Task` is instantiated with its dedicated stack, task function pointer, and task identifier corresponding to the generic task instance. Unlike real architectures, the POSIX virtual architecture does not require an additional linker script.

Besides the TCBs, each architecture handles ISRs differently. For the POSIX architecture, interrupts do not need to be acknowledged manually by the ISR, and new interrupts do not need to be blocked during interrupt handling because the POSIX signals already provide that functionality. Therefore, the generated ISR wrappers for POSIX only call the corresponding user-defined ISR that contains the actual routine. Two artificial ISRs for IPI handling are added to the list of interrupt handlers to trigger actions for `SIGUSR1` and `SIGUSR2` (previously described in Table 3.1). Each generated ISR is initialized during the `StartOS` system call and registered with the `guardian` for dispatching.

3.3.4 Generic System Calls

The system call generator component is responsible for the call-site-specific implementation of system calls. Although the system calls are specific to call sites already because *d*OSEK does not handle the system call parameters at runtime, the generic system call generator does not specialize interactions based on analysis results. These generic system calls provide a baseline for the evaluation in chapter 4, where the impact of call-site-specific optimizations is evaluated.

The generation of system calls follows a fixed pattern. When system calls are called from tasks, interrupts need to be disabled to prevent preemption. Then, the system call implementation is generated, surrounded by a `SystemEnterHook/SystemLeaveHook` pair. Although these hooks are not part of the OSEK standard, they were introduced in *d*OSEK to implement common functionality and therefore simplify the generation of system calls. If interrupts were disabled, they are enabled again, and a `StatusType` is returned. The Listing 3.5 displays the generated system call for an `ActivateTask` system call (Appendix A). In this case, the call site is a task and not an ISR, so interrupts are blocked. Because the activated task must run on a different core, an IPI is triggered instead of rescheduling on the current core.

As the system call arguments for a certain call site are known at code generation time, the MACAW RTOS generator inserts required parameters like the task to activate and the target core statically. This static system call generation is the foundation for interaction specialization described in the following subsection.

3.3.5 Specialized System Calls

System call specializations can be enabled or disabled by configuring the ARA step. If there are no specializations for a given system call possible, the generator defaults to the generic system call.

As described in section 2.7, the MSTG contains local states and SPs that synchronize multiple cores. For each ABB and therefore for each system call, one or more local states may be available in the MSTG. Cross-core system calls and locking operations lead to a SP following the respective system call.

```
extern "C" StatusType AUTOSAR_ActivateTask_BB18(TaskType arg0) {
    StatusType result = E_OK;
    // Callsite: T00
    Machine::disable_interrupts();
    // Hook: SystemEnterHook
    { debug_core << __FUNCTION__ << endl; }
    {
        scheduler_[1].SetReady_impl(OS_T10_task);
        Machine::trigger_interrupt(10, 1, true);
    }
    // Hook: SystemLeaveHook
    {
        if (result != E_OK) {
            CALL_HOOK(ErrorHook, result);
        }
    }
    __OS_enable_irq_after_kernel();
    return result;
}
```

Listing 3.5 – Generated system call

Specialization of Cross-Core System Calls

The `GetEvent`, `ActivateTask`, and `ChainTask` system calls can be specialized using the results from IPI Avoidance. When an MSTG SP that follows a cross-core system call does not lead to rescheduling on the affected core, then the IPI is unnecessary. If multiple local states are available for the given system call, the IPI can be avoided only if it is unnecessary in all possible states. The system call can then be specialized by skipping the `trigger_interrupt` call as displayed in Listing 3.5. This optimization is applied where possible to the system call implementations automatically when the generator flag `ipi_avoidance` is set.

Specialization of Locking Operations

For `GetSpinlock/TryToGetSpinlock/ReleaseSpinlock` specialization, the Lock Elision analysis step shall provide the information if a locking operation can be elided. The previous implementation only checks whether a `GetSpinlock` leads to any spinning state on the same core. Because another core could also call `GetSpinlock` before the first one has released the lock, the first locking operation cannot be elided in this case, although it is initially not leading to a spinning state. To correctly elide a locking operation, the MSTG states must be traversed. For each state, it must be checked whether other cores are interacting with the same lock between the first `GetSpinlock/TryToGetSpinlock` and the matching `ReleaseSpinlock`.

To visualize the lock elision algorithm, Listing 3.6 and Figure 3.3 show a test application with two cores contending for the same lock and the corresponding (reduced) MSTG. While the nodes in the MSTG represent SPs between both cores, red edges display system calls and subsume local states implicitly, and the green edge stands for a spinning state.

```
 1 void main() {
 2     StatusType rv;
 3     switch (GetCoreID()) {
 4     case OS_CORE_ID_MASTER:
 5         StartCore(OS_CORE_ID_1, &rv);
 6     default:
 7         StartOS(0);
 8     }
 9 }
10
11 /* core 0 */
12 TASK(T00) {
13     GetSpinlock(S1);
14     ActivateTask(T10);
15     /* other task may try to lock */
16     ReleaseSpinlock(S1);
17     ShutdownOS(E_OK);
18 }
19
20 /* core 1 */
21 TASK(T10) {
22     GetSpinlock(S1);
23     ReleaseSpinlock(S1);
24     ShutdownOS(E_OK);
25 }
```

Listing 3.6 – Test application

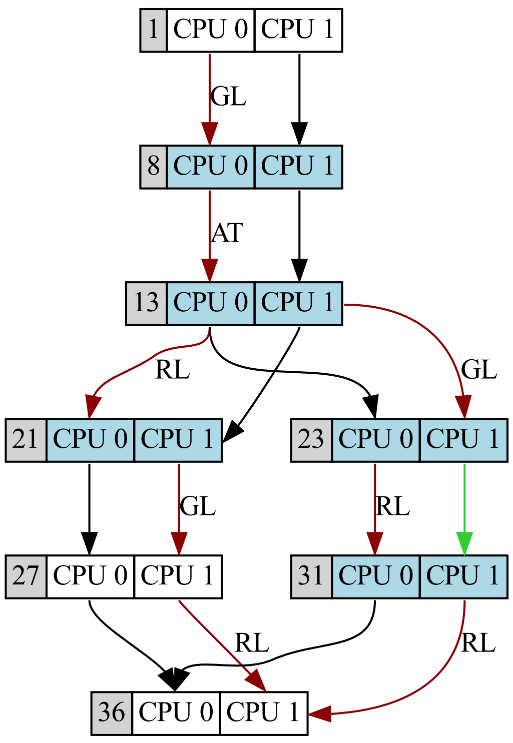


Figure 3.3 – MSTG of test application

After the task `T00` gets the lock (line 13, GL, 1 → 8), it activates task `T10` on the second core (line 14, AT, 8 → 13). While task `T00` releases the lock (line 16, RL, 13 → 21 or 23 → 31), the second task is accessing the same spinlock simultaneously (line 22, GL), which may already be unlocked (21 → 27) or still locked (13 → 23).

The following algorithm can determine whether Lock Elision is possible. If the spinlock is already held by any application while the given state is attempting to lock, the locking operation can never be elided. After this initial requirement, the MSTG must be partially searched to analyze the interactions related to the given spinlock operation. Starting from the initial SP after the given state of a `GetSpinlock`/`TryToGetSpinlock` call (SP 8), all following SPs before the release (blue nodes) must be visited. For each visited SP, the predecessor states must be examined. If a predecessor state is not a system call related to the given spinlock, it is ignored. When another core interacts with the spinlock before it is released, the locking operation cannot be elided in the given state (as in SP 31). However, if a `ReleaseSpinlock` is executed on the same core for the given spinlock (RL, 13 → 21), the subtree must not be searched further, and the ABB is stored for the specialization of `ReleaseSpinlock`. Finally, when the spinlock has not been released in the current path, all following SPs are added to the queue.

The MSTG is built using the `graph-tool` library, which provides various search algorithms like Breadth First Search (BFS). These search algorithms do not allow stopping the exploration of a subtree, instead, they only support stopping the complete search [PP]. Therefore, an adapted BFS on the MSTG is required for this algorithm. The (simplified) implementation of this MSTG search is depicted in Listing 3.7.

```python
def can_elide(state, cpu_id)
    visited = []
    elide = True
    queue = list(get_next_states(state))
    spinlock = get_syscall_args_spinlock(state)
    release_abb = None

    for initial_sp in queue: # check if lock is already held
        if lock_held(initial_sp, spinlock):
            elide=False

    while queue:
        curr_sp = queue.pop(0)  # FIFO -> BFS
        visited.append(curr_sp)
        stop_search_subtree = False

        for p in get_predecessor_states(curr_sp):
            if not get_syscall_args_spinlock(p) == spinlock:
                continue # irrelevant predecessor

            # Find ReleaseSpinlock on same cpu id with same spinlock
            if cpu_id == get_cpuid(p) and syscall_name(p) == "ReleaseSpinlock":
                stop_search_subtree = True
                release_abb = get_abb(p) # store where the lock is released

            if cpu_id != get_cpuid(p): # interaction on any other cpu
                elide = False

        if not stop_search_subtree:
            for next_sp in get_successor_sps(curr_sp):
                if next_sp not in visited and next_sp not in queue:
                    queue.append(next_sp)

    return elide, release_abb
```

Listing 3.7 – MSTG search for Lock Elision

Just like the cross-core system call specialization, all possible states regarding a certain call site for spinlock operations must be analyzed. Only if the lock can be elided in every possible state and the generator flag `lock_elision` is set, the call site will be specialized. For a specialized call site, the actual locking operation in `GetSpinlock`/`TryToGetSpinlock` and the unlocking operation in the corresponding `ReleaseSpinlock` will be omitted. Still, in order to correctly handle unreleased locks and prevent deadlocks, the bitmask of occupied spinlocks needs to be updated, and the system call cannot be removed completely.

3.4 Conclusion

Figure 3.4 gives an overview of all components of the MACAW RTOS generator that were explained in detail in the previous sections. These components are compiled together into the final system image that can be executed on an x86 Linux host.

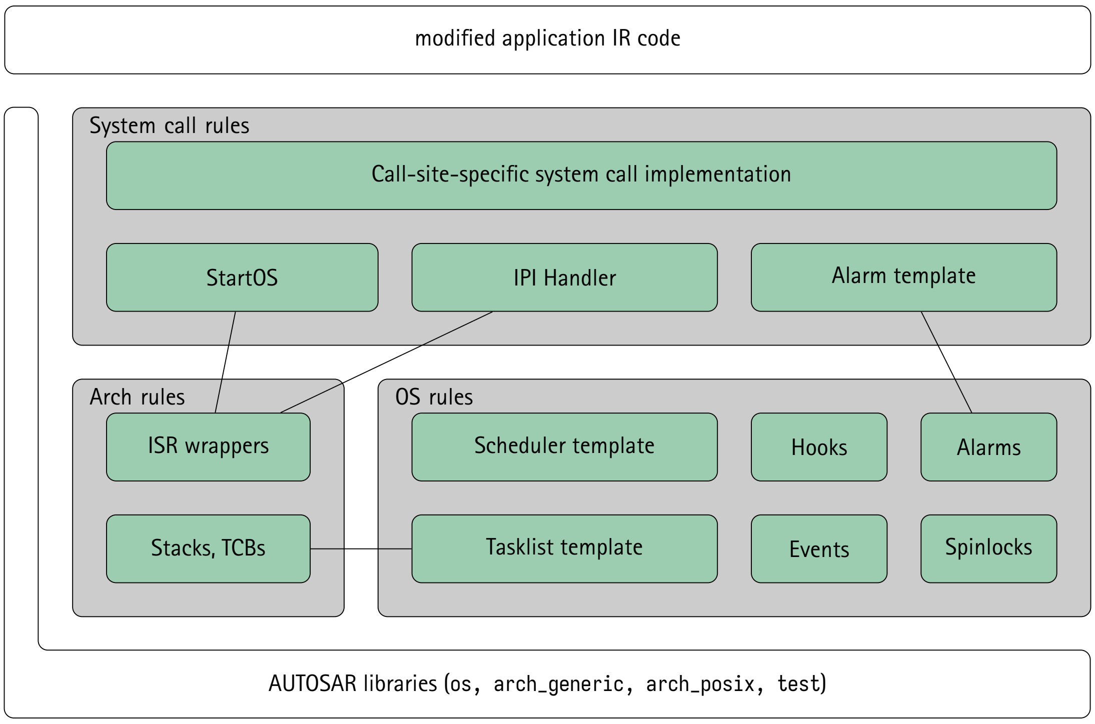


Figure 3.4 – MACAW RTOS generator overview

In summary, MACAW modifies the application IR code by inserting startup and error handling calls and renaming each system call site. Fundamental AUTOSAR functionality is implemented in the application-independent kernel libraries. MACAW automatically generates application-specific code, such as system call implementations and static instances, based on the system description. The system call generator can leverage static knowledge from previous analysis steps for the *Specialization of Interactions* in the system calls. Also, in the development process of the system synthesis, multiple fixes and extensions to improve other ARA steps and the AUTOSAR OS model were added.

ANALYSIS 4

In this chapter, the implemented components of the MACAW RTOS generator are discussed and evaluated. The kernel libraries and system generation are compared to those of other systems like *d*OSEK and Trampoline RTOS, using their respective test cases.

With MACAW, the specialization of cross-core system calls and elision of locking operations can be achieved. While these specializations improve the timing parameters such as delay and jitter of real-time applications, they come at the cost of increased memory footprint. The AUTOSAR kernel libraries integrated into ARA are based on *d*OSEK but do not support all features of it. Porting the dependability extensions like arithmetic encoding of OS objects in *d*OSEK is not in the scope of this work and only the POSIX platform is supported. Although the decision to use POSIX as a virtual platform simplified multiple aspects of the development, it limits the applicability of the implemented system. Exact measurements of performance and timing improvements of the complete application on this platform, especially when executed on a non-real-time scheduling x86 architecture, are affected by jitter, thus requiring microbenchmarks and suitable metrics for the system evaluation.

Next to the optimization goals, conformance to the OSEK/AUTOSAR standards is required for the interoperability and reuse of the AUTOSAR test applications in ARA. In the following section, the extent to which the MACAW RTOS conforms to the OSEK/AUTOSAR standards is discussed.

4.1 AUTOSAR Test Applications

MACAW implements the crucial AUTOSAR services required for scheduling and multicore handling and supports most OSEK and AUTOSAR applications. Specifically, all *d*OSEK features, including `Events`, `Alarms`, `ISRs`, and `Timers/Counters`, are adapted to a multicore operating system. The multicore extensions to the AUTOSAR API, such as the `StartCore` service, synchronized startup and shutdown, as well as `Spinlocks`, are implemented in MACAW. For actual AUTOSAR conformance, additional primitives like `ScheduleTables` and protection checks need to be implemented into the ARA OS model and the system generator, hence MACAW is "AUTOSAR Compatible."

For functional testing, the following paragraphs list multiple sets of test applications for the system generator. An overview of the complete Meson test suite, including over 100 working test cases, is given in Appendix B.

*d*OSEK Tests

The test cases from *d*OSEK are used as a baseline for the correct code generation of single-core applications, as they cover the OSEK primitives like `Events` and `Alarms`, as well as the implemented PCP with `Resources`. All functional test cases that succeed on *d*OSEK are also generated and executed correctly on MACAW.

Trampoline RTOS Tests

The multicore tests from Trampoline RTOS check for a variety of features, including correct error handling, cross-core system calls, and startup sequences. The system description of these tests was ported to ARA, and the Trampoline test functionality, including assertions, is redirected to the already implemented test framework described previously (paragraph Test Code). After these adaptations, many multicore Trampoline RTOS tests can be built and executed successfully. The tests listed in Table 4.1 are not part of the test suite because the ARA OS model and the MACAW RTOS synthesis are missing certain functionality. Remarkably, test applications with many interrupts, like `mc_alarms_s1`, are complex to analyze without further modifications or timing restrictions, leading to a combinatorial explosion during the state enumeration.

Trampoline test case	Limitation / missing function
`mc_appTermination_s1,2`	`TerminateApplication`
`mc_reschedule_s1`	`Schedule`
`mc_taskTermination_s1,2`	Timing protection
`mc_autostart_s3, sched_tables_s1`	`ScheduleTable`
`mc_alarms_s1`	Too complex (number of interrupts in MultiSSE)
`mc_spinlock_s1`	Too complex, timing protection, spinlock ordering (1)

(1) A modified `mc_spinlock_s1` test without timing protection and unused ISR is working. The test evaluates 20 separate assertions, of which two are failing due to unsupported spinlock ordering.

Table 4.1 – Unsupported Trampoline test cases

MACAW Tests

Next to the *d*OSEK and Trampoline RTOS tests, I have added further multicore tests specifically for IPIs, core handling, and spinlocks. These tests partially overlap with features tested by Trampoline but also exploit other edge cases like hooks or IPI synchronicity and are suitable for testing specialized system calls. Compared to the Trampoline tests, where the test suite is limited to only two cores, this test suite can support up to 16 cores.

MultiSSE Tests

Although the test cases for MultiSSE can be used to validate some aspects of the generator, the tests require multiple modifications, including an updated startup sequence, insertion of the test framework functions, and additional evaluation logic. But those tests do not check for any functionality beyond the test coverage of the other tests, so they are not part of the Meson test suite. Nevertheless, when the startup sequence is fixed, these applications compile and execute correctly. A subset of these tests is used to examine the specialized system calls in subsection 4.2.2.

In summary, all implemented AUTOSAR features are working as expected, and the application support of the implemented MACAW RTOS generator is comparable to other AUTOSAR implementations like Trampoline, with only a few limitations setting it apart. The objective to extend the *d*OSEK system with multicore functionality to enable cross-core system call specialization leveraging the static knowledge from the MultiSSE has been accomplished. In the next section, the performance advantages of the implemented system call specializations are evaluated and discussed.

4.2 Evaluation of the System Call Specialization

In this section, the specialization of the system calls is evaluated regarding their performance advantages compared to the generic system calls. Due to the global locking in every system call, the Trampoline AUTOSAR implementation is unsuitable for such a comparison. However, along with more granular locking, that system could also benefit from Lock Elision on the system call interface.

4.2.1 Microbenchmarks

In general, the performance gains of Lock Elision and IPI Avoidance are highly dependent on the implementation and the platform. Especially when cheap-when-successful instructions are used for locking operations, the performance gain is expected to be low. For the POSIX virtual platform, the microbenchmarks in Appendix C are used to measure the delays of an IPI and spinlock operations. Table 4.2 displays the results of these measurements.

The tests were compiled using clang-14 for i386, like MACAW, and executed on an Intel Xeon W3670 at 3.2 GHz running Ubuntu 22.04. Because the measurement of timing is influenced by jitter, scheduling, and system load, each benchmark is executed ten million times, and the median is calculated. Using this method, test runs where scheduling on the host interrupts the test execution are excluded, and the measured delays are nearly constant across multiple runs.

Benchmark	Delay median
None	66 ns
IPI	1776 ns
IPI (synchronous)	2504 ns
Spinlock	91 ns
`getpid`	312 ns

Table 4.2 – Results for POSIX IPI and spinlock microbenchmarks in Appendix C

As a baseline, the delay between two directly consecutive system calls to `clock_gettime` is 66 ns. IPIs implemented using POSIX signals take about 1.7 μs, and about 2.5 μs if an additional synchronization flag is used to signal IPI completion. With this synchronization flag, the interrupt overhead on the receiving core is considered on top of the delay of the signaling itself. Remarkably, the calls to `pthread_spin_lock` and `pthread_spin_unlock` together only take 91 ns. The spinlock operations in glibc use a cheap atomic decrement operation and do not require system calls, so they cause a very low delay. For reference, the Linux `getpid` system call is added to the comparison, as it always requires a kernelspace transition since the result is not cached in glibc [Mic].

After many execution cycles, these benchmarks provide a comparable and consistent timing result. In real applications, the delay of a single operation may have much higher delays because of cache misses or microarchitectural side effects. These microbenchmarks therefore only provide an estimation of the optimization potential on the POSIX platform. IPI Avoidance can lead to a measurable reduction in delay and jitter, whereas Lock Elision does not improve the performance noticeably on this platform. On other platforms, the delays can be very different, thus architecture-independent metrics to evaluate the system call specialization are discussed in the next subsection.

Next to the microbenchmarks, I have measured the runtime delay between `StartupHook` and `ShutdownHook` of some test applications. Although each measurement is repeated 1000 times, the jitter is still in the order of hundreds of nanoseconds, so the runtime measurements in Table 4.3 are rounded to microseconds. As expected, only tests where at least one IPI is avoided show a measurable performance improvement over the generic implementation. The first IPI between two cores is by far the slowest, delays of consecutive IPIs are close to the values given by the microbenchmarks. Because this is probably a consequence of the lazy allocation of control structures in the kernel, the performance improvement for `ipi_g` does not reflect the benefits resulting from IPI Avoidance. The timing improvements of spinlock elision cannot be measured exactly because the jitter is too high. For the `ipi_d,f` test applications, the avoidance of one IPI reduces the average execution time by roughly 3 μs as expected from the microbenchmarks.

Application	Generic	Specialized	Description
`ipi2_g`	22 μs	7 μs	1 / 1 IPIs avoided
`ipi2_f`	22 μs	19 μs	1 / 2 IPIs avoided
`ipi2_d`	26 μs	23 μs	1 / 3 IPIs avoided
`spinlock_a`	29 μs	29 μs	0 / 1 IPIs avoided, 4 / 8 locking operations elided

Table 4.3 – Comparison of test application runtime with specialized and generic system calls

4.2.2 Performance Impact Metrics

To analyze the possible impact of the system call specialization platform independently, the actual number of performed specializations must be examined. For this purpose, it would be possible to evaluate the results of the ARA analysis steps IPI Avoidance and Lock Elision. Using this approach, the outcome of the MultiSSE is analyzed by counting states where IPIs can be avoided for `ActivateTask` and `GetSpinlock` calls that do not lead to a spinning state [EFL23]. Another option is to execute relevant test applications and use tracing tools like `ltrace` to count the number of calls to the `pthread` library or `printf`-tracing.

As both options do not directly count the call site specializations, the preferred approach for this work is to count the specializations directly during the system generation phase. All call sites for cross-core `ActivateTask`, `ChainTask`, `SetEvent` system calls, as well as the spinlock operations `GetSpinlock`, `TryToGetSpinlock`, and `ReleaseSpinlock`, are counted. Additionally, the number of specialized variants for these system calls is noted. As a result, in the multicore test applications including MACAW, Trampoline, and MultiSSE (`locks/2cores`, `locks/3cores`, `paper`) test cases, 76 out of 274 system calls ($\approx$ 28%) are replaced with a specialized variant. In Figure 4.1, the number of specialized system calls compared to generic system calls are grouped by test sets. Specialization is not possible for the Trampoline tests, as they are testing only very specific features with two tasks on two cores, making interaction specializations not applicable. On the other hand, the MultiSSE tests for spinlocks on two or three cores allow for many spinlock specializations, while the `paper` tests include both specialization types. Further, the MACAW RTOS tests (`ipi`, `spinlock`) show both possible specializations.

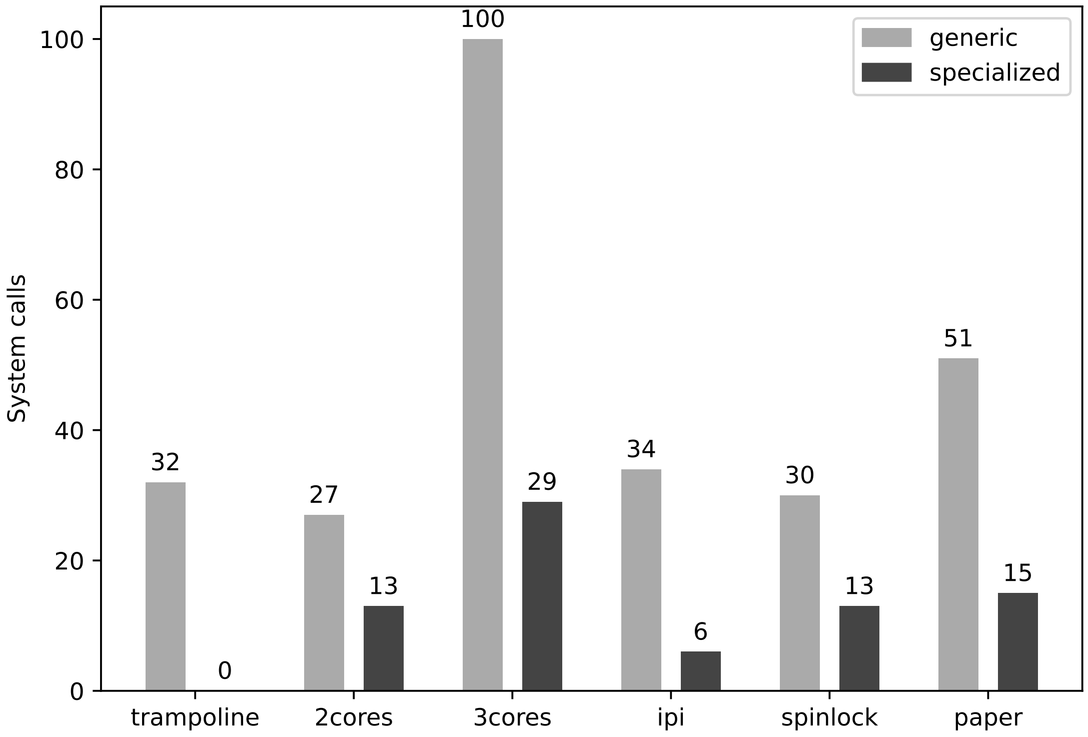


Figure 4.1 – Specialized system calls for test applications

The results show that a noticeable number of spinlock operations in the test cases are specialized, while IPIs are specialized less frequently. Although this is a direct consequence of what the test applications are targeting, this is an expected result in general. With cross-core system calls that require an IPI, a direct interaction with the receiving core is intended by the caller and can only be avoided in rare cases. On the other hand, spinlock interactions are less direct, and the operations are required only when other cores are simultaneously accessing the same spinlock, making Lock Elision possible on many occasions.

4.3 Result Discussion

The previous section covers performance improvements by eliding costly operations, but other nonfunctional properties like memory usage are affected by specialization as well. As the implemented system calls are generated and statically specialized per call site, they increase the memory footprint of the system linearly with each system call [Hof+15]. With this memory overhead in mind, the removal of data structures for locks that are never used to free up memory does not improve the overall system memory usage. To reduce this memory usage, not-specialized system calls could use a common, generic, and non-static system call implementation, like Trampoline RTOS. However, this is currently not supported without additional changes to the *d*OSEK task template, as the system call generation requires the insertion of static knowledge for each system call. Additionally, the specialization of system calls is limited to skipping only certain operations, not removing the entire system call, because it may contain other side effects. Only if functional requirements are relaxed to exclude incorrect application behavior like double-locking or missing lock releases, an aggressive specialization can remove the complete system call and reduce memory usage. The system analysis could employ additional steps to check for such conditions and allow for the elision of the whole system call. If memory protection is in place, this kind of specialization would also remove the overhead of kernelspace transitions, further improving the effectiveness of the optimization.

Another important factor is the mutual influence of the MultiSSE pre-calculated timings with the system call specialization. If the MultiSSE leverages static timing knowledge, the optimized system calls may reduce the best-case execution times for single ABBs, which must be considered in the timing calculation.

Regarding performance improvements or jitter reduction, the defined metric shows significant potential for system call specialization for the test applications. As real-world multicore AUTOSAR applications are rarely available, improvements by specialization remain to be analyzed in such cases. In general, the applicability of cross-core system call specialization is not limited to the use case of AUTOSAR timing improvements, the concept can be applied to other statically defined RTOSs as well.

CONCLUSION 5

In this thesis, I have created an AUTOSAR-compatible RTOS generator, supporting interaction-based specialization of multicore applications. By utilizing the results from the MultiSSE analysis, MACAW can specialize system calls per call site to eliminate costly operations and improve the timing parameters of the system. The MultiSSE analysis builds an MSTG that is evaluated to determine whether certain system calls require IPIs or locking operations.

MACAW is based on the *d*OSEK project, which supports only single-core OSEK applications [Hof+15]. I have implemented the multicore features like spinlocks, core startup, and synchronization defined by the AUTOSAR standard for the POSIX virtual platform. Cores are modeled as POSIX threads, signals represent interrupts, and the resulting AUTOSAR image can be executed on x86 GNU/Linux systems. With thread placement on different host CPUs and real-time scheduling, the generated system can achieve actual real-time behavior and parallel computation on physical cores. To mitigate locking on the system call interface, each core schedules independently, and cross-core operations like thread activation on a different core use atomic operations. While the kernel libraries are independent of the whole-system approach, the generator with system call specializations depends on the ARA analysis steps like the MultiSSE. The multicore states in the MSTG enable optimization through *Specialization of Interactions* if the resulting system call semantics remain unchanged. With Lock Elision and IPI Avoidance, two possible types of multicore specializations are available and can be applied to every AUTOSAR application by the MACAW generator automatically. As one part of the application-specific system generator, the LLVM IR code of the application is modified to call the statically specialized variant for each supported system call.

As a result, it is possible to improve the timing behavior of statically defined real-time applications with a reduction in delay and jitter. The test results suggest that spinlocks can be elided in many applications, but the performance impact is low, at least on the POSIX platform. On the other hand, avoiding an unnecessary IPI can reduce the delay in the order of microseconds, which is clearly measurable even in a jitter-influenced environment. In a subset of available and relevant multicore test applications, 28 % of cross-core system calls can be specialized. Although a few features are missing, MACAW supports the key functionality of the AUTOSAR specification and can handle most test applications present in the ARA repository. The currently implemented system only supports the POSIX platform, future work could extend the platform support to an ARM-based microcontroller as this is a more realistic scenario for automotive control systems. Also, more AUTOSAR features like memory protection could be added to MACAW, and the benefit of removing the complete system call can be analyzed.

LIST OF ACRONYMS

ABB	Atomic Basic Block
API	Application Programming Interface
ARA	Automated Real-time system Analyzer
AST	Abstract Syntax Tree
AUTOSAR	AUTomotive Open System ARchitecture
BFS	Breadth First Search
CAS	Compare and Swap
CFG	Control Flow Graph
***d*OSEK**	Dependability-Oriented Static Embedded Kernel
ECU	Electronic Control Unit
glibc	GNU C Library
IOC	Inter-OS-Application Communicator
IoT	Internet of Things
IPI	Inter-Processor Interrupt
IR	Intermediate Representation
ISR	Interrupt Service Routine
MACAW	**M**ulticore **A**UTOSAR **C**ompatible **A**pplication-specific **W**hole-system-optimizer
MPU	Memory Protection Unit
MSTG	Multicore State Transition Graph
NPTL	Native POSIX Thread Library
OIL	OSEK Implementation Language
OSEK	Open Systems and their Interfaces for the Electronics in Motor Vehicles (Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen)
PCP	Priority Ceiling Protocol
POSIX	Portable Operating System Interface
RISC	Reduced Instruction Set Computer
RTOS	Real-Time Operating System
SIA	Static Instance Analysis
SMP	Symmetric Multiprocessing
SP	Synchronization Point
SSE	System State Enumeration
SWFRT	Software Free Running Timer
TCB	Task Control Block

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

REFERENCES

[Ama] Amazon Web Services. *FreeRTOS*. URL: freertos.org (visited on 03/24/2023).

[Aud+95] Neil C. Audsley et al. “Real-Time System Scheduling.” In: *Predictably Dependable Computing Systems*. Springer Berlin Heidelberg, 1995, pp. 41–52. DOI: 10.1007/978-3-642-79789-7_3.

[AUT] AUTOSAR GbR. *AUTOSAR*. URL: https://www.autosar.org (visited on 02/20/2023).

[AUT22] AUTOSAR. *Specification of Operating System*. R22-11. Nov. 2022. URL: https://www.autosar.org/fileadmin/standards/R22-11/CP/AUTOSAR_SWS_OS.pdf (visited on 06/12/2023).

[Bec+06] Jean-Luc Bechennec et al. “Trampoline An Open Source Implementation of the OSEK/VDX RTOS Specification.” In: *2006 IEEE Conference on Emerging Technologies and Factory Automation*. 2006, pp. 62–69. DOI: 10.1109/ETFA.2006.355432.

[Coo17] Jim Cooling. *Real-time Operating Systems. Book 1 - The Theory*. The Engineering of Real-Time Embedded Systems Series. 2017.

[DHL15] Christian Dietrich, Martin Hoffmann, and Daniel Lohmann. “Cross-Kernel Control-Flow-Graph Analysis for Event-Driven Real-Time Systems.” In: *Proceedings of the 16th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*. Ed. by ACM. Portland, Oregon, USA, 2015, pp. 1–10. DOI: 10.1145/2670529.2754963.

[DL17] Christian Dietrich and Daniel Lohmann. “OSEK-V: Application-Specific RTOS Instantiation in Hardware.” In: *Proceedings of the 2017 ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES ’17)* (Barcelona, Spain). New York, NY, USA: ACM Press, 2017. DOI: 10.1145/3078633.3078637.

[DL18] Christian Dietrich and Daniel Lohmann. “Semi-Extended Tasks: Efficient Stack Sharing Among Blocking Threads.” In: *Proceedings of the 39th IEEE Real-Time Systems Symposium 2018*. Ed. by Sebastian Altmeyer. Nashville, Tennessee, USA: IEEE Computer Society Press, 2018. DOI: 10.1109/RTSS.2018.00049.

[DM03] Ulrich Drepper and Ingo Molnar. *The Native POSIX Thread Library for Linux*. Red Hat. Feb. 2003. URL: https://static.redhat.com/legacy/whitepapers/developer/POSIX_Linux_Threading.pdf (visited on 02/20/2023).

[EFL23] Gerion Entrup, Björn Fiedler, and Daniel Lohmann. “MultiSSE: Static Syscall Elision and Specialization for Event-Triggered Multi-Core RTOS.” In: *Proceedings of the 29th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS’23)*. May 2023.

[ENL22] Gerion Entrup, Jan Neugebauer, and Daniel Lohmann. "RTOS-Independent Interaction Analysis in ARA." In: *Proceedings of the 16th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT '22)* (Modena, Italy). July 2022.

[ESD19] Gerion Entrup, Benedikt Steinmeier, and Christian Dietrich. "ARA: Automatic Instance-Level Analysis in Real-Time Systems." In: *Proceedings of the 15th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT '19)* (Stuttgart, Germany). July 2019.

[Evi] Evidence Srl. *Erika Enterprise RTOS v3*. URL: `https://www.erika-enterprise.com` (visited on 03/25/2023).

[Fie+18] Björn Fiedler et al. "Levels of Specialization in Real-Time Operating Systems." In: *Proceedings of the 14th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT '18)* (Barcelona, Spain). 2018.

[Fie+21] Björn Fiedler et al. "ARA: Static Initialization of Dynamically-Created System Objects." In: *Proceedings of the 27th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'21)* (Virtual Event). May 2021, pp. 400–412. DOI: `10.1109/RTAS52030.2021.00039`.

[Fre] Free Software Foundation, Inc. *RTL (GNU Compiler Collection (GCC) Internals)*. 12.2. URL: `https://gcc.gnu.org/onlinedocs/gcc-12.2.0/gccint/RTL.html` (visited on 02/25/2023).

[HBR21] Imane Haur, Jean-Luc Béchennec, and Olivier Henri Roux. "Formal schedulability analysis based on multi-core RTOS model." In: *RTNS '2021. The 29th International Conference on Real-Time Networks and Systems*. Nantes, France, Apr. 2021. DOI: `10.1145/3453417.3453437`.

[Hof+15] Martin Hoffmann et al. "dOSEK: The Design and Implementation of a Dependability-Oriented Static Embedded Kernel." In: *Proceedings of the 21st IEEE International Symposium on Real-Time and Embedded Technology and Applications (RTAS '15)*. Washington, DC, USA: IEEE Computer Society Press, 2015, pp. 259 –270. DOI: `10.1109/RTAS.2015.7108449`.

[IEE17] IEEE. *Portable Operating System Interface (POSIX(TM)) Base Specifications, Issue 7*. IEEE Std 1003.1-2017. 2017. DOI: `10.1109/IEEESTD.2018.8277153`. URL: `https://pubs.opengroup.org/onlinepubs/9699919799/`.

[Lat11] Chris Lattner. *The Architecture of Open Source Applications: LLVM*. 2011. URL: `https://www.aosabook.org/en/llvm.html` (visited on 02/25/2023).

[LLVa] LLVM. *LLVM Language Reference Manual*. 14.0. URL: `https://releases.llvm.org/14.0.0/docs/LangRef.html` (visited on 02/25/2023).

[LLVb] LLVM. *LLVM Programmers Manual*. 14.0. URL: `https://releases.llvm.org/14.0.0/docs/ProgrammersManual.html` (visited on 02/25/2023).

[Loh20] Daniel Lohmann. *Betriebssystembau*. 2020.

[Mic] Michael Kerrisk. *Linux manual page*. URL: `https://man7.org/linux/man-pages` (visited on 03/27/2023).

[OSE04a] OSEK/VDX. *Communication*. 3.0.3. July 2004. URL: `https://www.osek-vdx.org/mirror/OSEKCOM303.pdf` (visited on 03/25/2023).

[OSE04b] OSEK/VDX. *System Generation. OIL: OSEK Implementation Language*. 2.5. July 2004. URL: `https://www.osek-vdx.org/mirror/oil25.pdf` (visited on 02/20/2023).

[OSE05] OSEK/VDX. *Operating System Specification*. 2.2.3. Feb. 2005. URL: `https://www.osek-vdx.org/mirror/os223.pdf` (visited on 02/20/2023).

[PP] Tiago de Paula Peixoto. *graph_tool.search - Search algorithms*. 2.5.3. URL: `https://graph-tool.skewed.de/static/doc/search_module.html` (visited on 04/16/2023).

[SGG18] A. Silberschatz, G. Gagne, and P.B. Galvin. *Operating System Concepts*. 10th ed. Wiley, 2018.

[Sta18] William Stallings. *Operating Systems. Internals and Design Principles*. 9th ed. Pearson Education Limited, 2018.

[Tig+17] Kabland Toussaint Gautier Tigori et al. “Formal Model-Based Synthesis of Application-Specific Static RTOS.” In: *ACM Trans. Embed. Comput. Syst.* 16.4 (2017). ISSN: 1539-9087. DOI: `10.1145/3015777`.

[Zep] Zephyr Project. *Zephyr Project Documentation*. URL: `https://docs.zephyrproject.org` (visited on 03/24/2023).

IPI TEST APPLICATION A

```c
#include "autosar/os.h"
#include "test/test.h"

DeclareTask(T00);
DeclareTask(T10);
DeclareTask(T20);
DeclareTask(T30);

DeclareEvent(E1, 1);

void start()
{
    StatusType rv;
    switch(GetCoreID()) {
    case OS_CORE_ID_MASTER:
        StartCore(OS_CORE_ID_1, &rv);
        StartCore(OS_CORE_ID_2, &rv);
        StartCore(OS_CORE_ID_3, &rv);
    default:
        StartOS(0);
    }
}

TEST_MAKE_OS_MAIN(start())

TASK(T00) {
    test_trace('0');
    ActivateTask(T10);
    WaitEvent(E1);
    test_trace('E');
    ShutdownAllCores(E_OK);
}

TASK(T10) {
    test_trace('1');
    ChainTask(T20);

}
```

```

TASK(T20) {
    test_trace('2');
    ChainTask(T30);
}

TASK(T30) {
    test_trace('3');
    SetEvent(T00, E1);
    TerminateTask();
}

void ShutdownHook(StatusType status) {
    (void) status;
    /* The testcase has finished, check the output */
    if (GetCoreID() == OS_CORE_ID_0) {
        test_trace_assert("0123E");
        test_finish();
    }
}
```

Listing A.1 – Test application for IPIs

B MACAW GENERATOR TEST SUITE

```
autosar_generator_autosar_singlecore_bcc1_alarm2_a                  OK
autosar_generator_autosar_singlecore_bcc1_alarm2_c                  OK
autosar_generator_autosar_singlecore_bcc1_alarm2_b                  OK
autosar_generator_autosar_singlecore_bcc1_alarm1_c                  OK
autosar_generator_autosar_singlecore_bcc1_alarm1_a                  OK
autosar_generator_autosar_singlecore_bcc1_alarm1_b                  OK
autosar_generator_autosar_singlecore_bcc1_alarm1_e                  OK
autosar_generator_autosar_singlecore_bcc1_alarm1_d                  OK
autosar_generator_autosar_singlecore_bcc1_complex1_a                OK
autosar_generator_autosar_singlecore_bcc1_counter1_a                OK
autosar_generator_autosar_singlecore_bcc1_complex2_a                OK
autosar_generator_autosar_singlecore_bcc1_isr2_a                    OK
autosar_generator_autosar_singlecore_bcc1_isr2_b                    OK
autosar_generator_autosar_singlecore_bcc1_isr2_c                    OK
autosar_generator_autosar_singlecore_bcc1_isr2_d                    OK
autosar_generator_autosar_singlecore_bcc1_isr2_e                    OK
autosar_generator_autosar_singlecore_bcc1_resource1_a               OK
autosar_generator_autosar_singlecore_bcc1_resource1_b               OK
autosar_generator_autosar_singlecore_bcc1_resource1_c               OK
autosar_generator_autosar_singlecore_bcc1_resource1_d               OK
autosar_generator_autosar_singlecore_bcc1_complex1_b                OK
autosar_generator_autosar_singlecore_bcc1_complex1_c                OK
autosar_generator_autosar_singlecore_bcc1_complex1_d                OK
autosar_generator_autosar_singlecore_bcc1_resource1_e               OK
autosar_generator_autosar_singlecore_bcc1_resource1_f               OK
autosar_generator_autosar_singlecore_bcc1_resource1_g               OK
autosar_generator_autosar_singlecore_bcc1_resource1_j               OK
autosar_generator_autosar_singlecore_bcc1_resource1_h               OK
autosar_generator_autosar_singlecore_bcc1_resource1_k               OK
autosar_generator_autosar_singlecore_bcc1_resource1_l               OK
autosar_generator_autosar_singlecore_bcc1_resource2_a               OK
autosar_generator_autosar_singlecore_bcc1_sse1_a                    OK
autosar_generator_autosar_singlecore_bcc1_sse1_c                    OK
autosar_generator_autosar_singlecore_bcc1_sse1_b                    OK
autosar_generator_autosar_singlecore_bcc1_resource2_b               OK
autosar_generator_autosar_singlecore_bcc1_task1_a                   OK
autosar_generator_autosar_singlecore_bcc1_alarm1_f                  OK
autosar_generator_autosar_singlecore_bcc1_alarm3_a                  OK
```

```
autosar_generator_autosar_singlecore_bcc1_alarm3_e                  OK
autosar_generator_autosar_singlecore_bcc1_task1_b                   OK
autosar_generator_autosar_singlecore_bcc1_task1_d                   OK
autosar_generator_autosar_singlecore_bcc1_task1_c                   OK
autosar_generator_autosar_singlecore_bcc1_task1_f                   OK
autosar_generator_autosar_singlecore_bcc1_task1_e                   OK
autosar_generator_autosar_singlecore_bcc1_task1_g                   OK
autosar_generator_autosar_singlecore_bcc1_task2_a                   OK
autosar_generator_autosar_singlecore_bcc1_task2_b                   OK
autosar_generator_autosar_singlecore_ecc1_bt1_a                     OK
autosar_generator_autosar_singlecore_bcc1_task2_c                   OK
autosar_generator_autosar_singlecore_ecc1_bt1_b                     OK
autosar_generator_autosar_singlecore_ecc1_bt1_e                     OK
autosar_generator_autosar_singlecore_ecc1_bt1_c                     OK
autosar_generator_autosar_singlecore_ecc1_bt1_f                     OK
autosar_generator_autosar_singlecore_ecc1_bt1_d                     OK
autosar_generator_autosar_singlecore_ecc1_bt1_g                     OK
autosar_generator_autosar_singlecore_ecc1_bt1_h                     OK
autosar_generator_autosar_singlecore_ecc1_event1_a                  OK
autosar_generator_autosar_singlecore_ecc1_event1_d                  OK
autosar_generator_autosar_singlecore_ecc1_event1_c                  OK
autosar_generator_autosar_singlecore_ecc1_event1_b                  OK
autosar_generator_autosar_singlecore_ecc1_event1_f                  OK
autosar_generator_autosar_singlecore_ecc1_event1_e                  OK
autosar_generator_autosar_singlecore_ecc1_event1_g                  OK
autosar_generator_autosar_singlecore_ecc1_eventisr1_c               OK
autosar_generator_autosar_singlecore_ecc1_eventisr1_d               OK
autosar_generator_autosar_singlecore_ecc1_eventisr1_e               OK
autosar_generator_autosar_singlecore_ecc1_eventisr1_f               OK
autosar_generator_autosar_singlecore_sched_a                        OK
autosar_generator_autosar_singlecore_ecc1_eventisr1_a               OK
autosar_generator_autosar_singlecore_ecc1_eventisr1_b               OK
autosar_generator_autosar_singlecore_bcc1_alarm3_b                  OK
autosar_generator_autosar_singlecore_bcc1_alarm3_d                  OK
autosar_generator_autosar_singlecore_bcc1_alarm3_c                  OK
autosar_generator_autosar_multicore_trampoline_mc_autostart_s1      OK
autosar_generator_autosar_multicore_trampoline_mc_autostart_s2      OK
autosar_generator_autosar_multicore_trampoline_mc_coreid_s1         OK
autosar_generator_autosar_multicore_trampoline_mc_eventSetting_s1   OK
autosar_generator_autosar_multicore_trampoline_mc_events_s1         OK
autosar_generator_autosar_multicore_trampoline_mc_scheduling_s1     OK
autosar_generator_autosar_multicore_trampoline_mc_startOs_s1        OK
autosar_generator_autosar_multicore_trampoline_mc_startup_s1        OK
autosar_generator_autosar_multicore_trampoline_mc_taskActivation_s1 OK
autosar_generator_autosar_multicore_trampoline_mc_taskChaining_s1   OK
autosar_generator_autosar_multicore_andreas_ipi_a                   OK
autosar_generator_autosar_multicore_andreas_ipi_b                   OK
autosar_generator_autosar_multicore_andreas_ipi_c                   OK
autosar_generator_autosar_multicore_andreas_ipi_d                   OK
autosar_generator_autosar_multicore_andreas_ipi_e                   OK
autosar_generator_autosar_multicore_andreas_ipi_f                   OK
autosar_generator_autosar_multicore_andreas_ipi_g                   OK
```

```
autosar_generator_autosar_multicore_andreas_ipi2_a                  OK
autosar_generator_autosar_multicore_andreas_ipi2_b                  OK
autosar_generator_autosar_multicore_andreas_ipi2_c                  OK
autosar_generator_autosar_multicore_andreas_ipi2_d                  OK
autosar_generator_autosar_multicore_andreas_ipi2_e                  OK
autosar_generator_autosar_multicore_andreas_ipi2_f                  OK
autosar_generator_autosar_multicore_andreas_ipi2_g                  OK
autosar_generator_autosar_multicore_andreas_spinlock_a              OK
autosar_generator_autosar_multicore_andreas_spinlock_b              OK
autosar_generator_autosar_multicore_andreas_spinlock_c              OK
autosar_generator_autosar_multicore_andreas_spinlock_e              OK
autosar_generator_autosar_multicore_andreas_spinlock_f              OK
autosar_generator_autosar_multicore_andreas_cores_4                 OK
autosar_generator_autosar_multicore_andreas_cores_12                OK
autosar_generator_autosar_multicore_andreas_cores_16                OK

Ok:                 105
Expected Fail:      0
Fail:               0
Unexpected Pass:    0
Skipped:            0
Timeout:            0
```

Listing B.1 – Test Suite results

POSIX MICROBENCHMARKS C

```c
typedef enum {
    NONE = 0,
    IPI = 1,
    IPI_SYNC = 2,
    SPINLOCK = 3,
    GETPID = 4,
} MODE_t;

_Atomic(int) ctr = 0;
_Atomic(bool) finished = false;
_Atomic(bool) exit_flag = false;
pthread_t t0, t1;
pthread_spinlock_t lock;
struct timespec start, end;
MODE_t MODE = 0;
#define REPEAT 10000000
unsigned int diff[REPEAT];

void test_start(int i) {
    clock_gettime(CLOCK_REALTIME, &start); // start test
}

void test_end(int i) {
    clock_gettime(CLOCK_REALTIME, &end); // end test
    if (end.tv_nsec < start.tv_nsec)
        end.tv_nsec += 1e9; // handle seconds
    diff[i] = end.tv_nsec - start.tv_nsec;
}

int compare (const void * a, const void * b) {
    return *(int*)a - *(int*)b;
}

void test_print() {
    qsort (diff, REPEAT, sizeof (unsigned int), compare);
    printf("median:  %u ns\n", diff[REPEAT/2]);
}

```

```
void thread_sync() { while (++ctr < 2); }

void handler(int signum) { finished = true; }

void thread_0() {
    thread_sync(); // sync both threads
    while (!exit_flag);
}

void thread_1() {
    thread_sync(); // sync both threads
    for (int i = 0; i < REPEAT; i++) {
        switch (MODE) {
        case NONE:
            test_start(i);
            test_end(i);
            break;
        case IPI:
            test_start(i);
            pthread_kill(t0, SIGUSR1);
            test_end(i);
            break;
        case IPI_SYNC:
            finished = false;
            test_start(i);
            pthread_kill(t0, SIGUSR1);
            while (!finished);
            test_end(i);
            break;
        case SPINLOCK:
            test_start(i);
            pthread_spin_lock(&lock);
            pthread_spin_unlock(&lock);
            test_end(i);
            break;
        case GETPID:
            test_start(i);
            getpid();
            test_end(i);
            break;
        }
    }
    exit_flag = true;
}

int main(int argc, char* argv[]) {
    /* init signal handler, spinlock, threads */
    test_print();
}
```

Listing C.1 – Microbenchmark application