ARA: Static Initialization of Dynamically-Created System Objects

Björn Fiedler, Gerion Entrup, Christian Dietrich, Daniel Lohmann
Leibniz Universität Hannover, Germany
{fiedler, entrup, dietrich, lohmann}@sra.uni-hannover.de

***Abstract*—After power-on, crash or reboot, the system-setup point is the first deadline that a safety-critical system has to reach. Up to this point, the application not only initializes its own state but it also creates all necessary system objects (e.g., threads, mutexes, alarms, ...) in the real-time operating system. And, while the strict requirements for real-time analyses often result in a rather static set of created system objects, the commonly-provided *real-time operating system (RTOS)* interfaces force developers to execute these creations at run time, resulting in an unnecessarily prolonged boot process.**

With ARA, we present a static whole-system transformation that discovers pseudo-dynamic *system-object creations (SOCs)* which yield the same object on every boot. By modifying the application and by RTOS specialization, we transform these SOCs to semantically equivalent static SOCs, which moves their instantiation from the run time to the compile time. Thereby, we maintain the well-known RTOS interfaces for dynamic SOCs but let developers enjoy the benefits that static initialization provides. In our case studies with FreeRTOS applications, we could reduce the boot time by up to 43 percent at a moderate increase of flash usage.

I. Introduction

The system setup is the first task that a real-time system executes after power-on, reboot, or crash. After we have initialized the hardware, configured the RTOS, and kicked off the application execution, we reach the *system-setup point (SSP)* and can begin the normal operation. The SSP is not only the first important deadline for a safety-critical real-time system, but it also impacts the resilience of fail-safe systems that recover from transient errors by rebooting [2], [3], [4]. For some safety-critical systems, the time to the SSP is even subject to official regulation: For example, the FMVSS111 [26] states that a rear-view camera in an US vehicle must provide an image within 2 seconds after vehicle start and reverse-gear selection. For automotive systems recovering from an error, the functional safety standard ISO26262 defines the fault tolerant time interval as the maximum amount of time allowed from fault occurrence until a safe state is reached again. If those systems perform a reboot as error recovery mechanism, the SSP has an impact on that interval.

An important aspect of the system setup is the configuration of the RTOS and the creation of system objects, like threads, mutexes, or message queues. And since real-time analyses require a decent amount of static knowledge ahead-of-time, the RTOS configuration, as well as the set of system objects, does not vary much from boot to boot. While some RTOSes, like OSEK [27] or μITRON [21], reflect this static nature and drive a system generator with a configuration file to instantiate system objects as preconfigured static objects at compile time, many recent RTOSes, like FreeRTOS [16] or RIOT [6] only provide dynamic system-object creation at the system-call interface.

However, there is a trade-off between dynamic and static SOCs: On the one hand, the dynamic variant requires no external tooling, allows for run-time configurability, and provides familiar OS-usage patterns for developers that come from general-purpose computing (e.g., POSIX [1]). On the other hand, if we use dynamic SOCs for objects whose parameters are statically known, we end up creating the same objects with the same parameters, over and over again, on every boot. These *pseudo-dynamic* SOCs delay the SSP without providing actual flexibility. Furthermore, the initialization routines inflate the code segment, decrease the resilience against transient hardware faults [19], and entail the presence of other complex components, like a heap manager.

So, while a dynamic system setup is easier to manage on the development side, it comes at the cost of a longer, more expensive, and less robust initialization phase that could be avoided in cases where the system configuration is known in advance. Therefore, some RTOSes, like Zephyr [37], already provide distinct APIs for static and dynamic SOCs. However, if developers have no in-depth understanding of the costs of different system-setup strategies, they will often default to dynamic SOCs as it is the more familiar and the easier to maintain usage pattern.

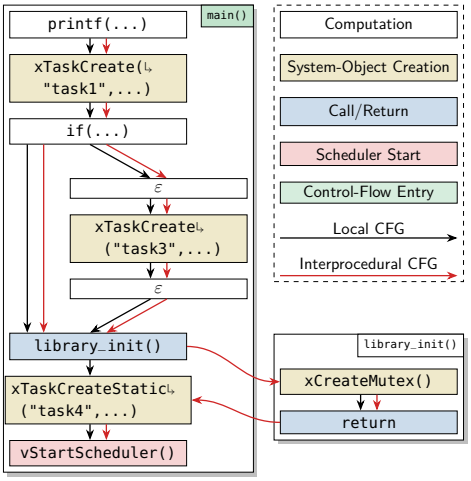
About this paper In order to bridge the semantic gap between dynamic and static system-setup strategies, we present ARA, which is an RTOS-aware whole-system compiler that transforms pseudo-dynamic SOCs to static ones. Based on a precompiled real-time application, which uses dynamic object creation, ARA performs the *static instance analysis (SIA)* to find pseudo-dynamic SOCs and extracts their creation parameters. Equipped with this knowledge, ARA performs a *partial specialization* of the real-time application and the RTOS and transfers as many SOCs as possible from the run time to the compile time to decrease the SSP delay. In particular, we claim the following contributions:

1) We present the *static instance analysis (SIA)*, which performs RTOS-aware detection of pseudo-dynamic system-object creations.
2) We provide specialization methods that exploit this knowledge to speed up system setup and to reduce the RTOS' code segment.

```
1  StackType_t stack[1024];
2  StaticTask_t tcb;
3
4  int main() {
5    printf("Starting system\n");
6    xTaskCreate("task1", t1_entry, 512, 2);
7    if (unknown_condition()) {
8      xTaskCreate("task3", t3_entry, 128, 1);
9    }
10   library_init();
11   xTaskCreateStatic("task4", t4_entry,
12                     256, 2, stack, &tcb);
13   vStartScheduler();
14 }
15 SemaphoreHandle_t mutex1;
16 void library_init() {
17     mutex1 = xCreateMutex();
18     return;
19 }
```

```
20 void t1_entry() {
21   xTaskCreate("task2", t2_entry, 256, dynPrio);
22   for(;;) { /* ... */ }
23 }
24
25 void t2_entry() { for (;;) { /* ... */  } }
26
27 void t3_entry() { for (;;) { /* ... */  } }
28
29 void t4_entry() {
30     xTaskCreate("task5", t5_entry, 512, 2);
31     for (;;) { /* ... */  }
32 }
33
34 MessageBufferHandle_t buffer1;
35 void t5_entry() {
36     buffer1 = xMessageBufferCreate(12);
37     for (;;) { /* ... */  }
38 }
```

(a) Application Source Code



(b) LCFGs and ICFG for `main()`

Fig. 1: Example FreeRTOS Application (simplified). Six independent control flows (green) visit SOC sites where they create and initialize (yellow) system objects. The control-flow entries of the five threads only execute after the scheduling is started explicitly at the end of the `main()` entry point (red).

3) We demonstrate the applicability of our approach with two real-world FreeRTOS applications and reduce their SSP delay by up to 43 percent. For micro benchmarks, the speedup is up to 67 percent.

The rest of the paper is structured as following: We explain the system model that ARA assumes in Section II and describe our analysis and system-specialization approach in Section III. We demonstrate the potential and the applicability of ARA in Section IV, discuss the results in Section V and the related work in Section VI, before we conclude the paper in Section VII.

II. System Model

For this paper, we consider real-time systems where application and RTOS are statically combined into a single system image that gets deployed onto the target platform's flash memory. We demand that the whole application is available as source code, or at least as an *intermediate representation (IR)* (e.g., LLVM IR [22]) that provides us with information about (function-local) control-flow graphs and function-call sites. With considerable effort and extensive machine-code analyses [32], this requirement could even be weakened to the availability of the application binary. Upon system start, the application loads its data segment from flash, zeroes out the BSS segment, and dynamically creates system objects (i.e., threads) at distinguished locations (SOC sites). After the scheduling is started explicitly, threads are allowed to dynamically create further system objects. We assume that every thread gets scheduled eventually, but make no further assumptions about scheduling strategy or multi-processor capability of the RTOS.

Without loss of generality, we use FreeRTOS [16] as a representative example throughout this paper. FreeRTOS, whose development Amazon stewards to use it together with their cloud instances, comes in the form of a library operating system that is configurable by C-preprocessor macros. For example, system calls may be (de)activated and it ships with five different heap manager implementations. FreeRTOS applications create and initialize *threads*, *semaphores*, *message queues*, and other instances of system abstractions dynamically at run time. And although newer versions of FreeRTOS (since 9.0.0) have the possibility to use statically-allocated memory for system objects instead of dynamic-heap allocations, system-object initialization is always done at run time.

Figure 1 shows an example system that uses the FreeRTOS API to create system objects on boot. Within the `main()` function, the application unconditionally creates the threads `task1` and `task4`, while `task3` is only created under an unknown condition. Besides thread creation, the call to `library_init()` invokes a SOC for a mutex object. While the user provides the memory for `task4`, FreeRTOS uses its internal heap manager to allocate memory for all other system objects in the example. After the scheduler starts, `task1` creates `task2` with a dynamically calculated priority but a static stack size of 256 bytes. Furthermore, the execution of `task4` results in a SOC for `task5`, which by itself creates a message buffer of size 12; Besides `task2` (dynamic argument) and `task3` (conditional creation), the creation of all other system objects is pseudo dynamic and could be performed at compile time instead of executing the SOCs over and over again on every system start.

III. The ARA Approach

ARA is a whole-system compiler (see Figure 2), which takes the application code as well as the RTOS configuration as inputs. Thereby, ARA has the whole system at hand and performs the following analyses and modifications as link-time optimizations. In the analysis phase (❶-❸), we read in the application code as LLVM IR [22], perform necessary *control-flow graph (CFG)* preprocessing steps, and execute the *static instance analysis (SIA)*, which calculates the list of possibly-created system-object *instances*. In the synthesis phase (❹-❼), we decide on the maximal and desired *level of specialization*, transform the real-time application to statically allocate and initialize objects, and generate a specialized RTOS. Furthermore, we execute an additional *post-processing step* on the binary (❽) to further speed up the system setup.

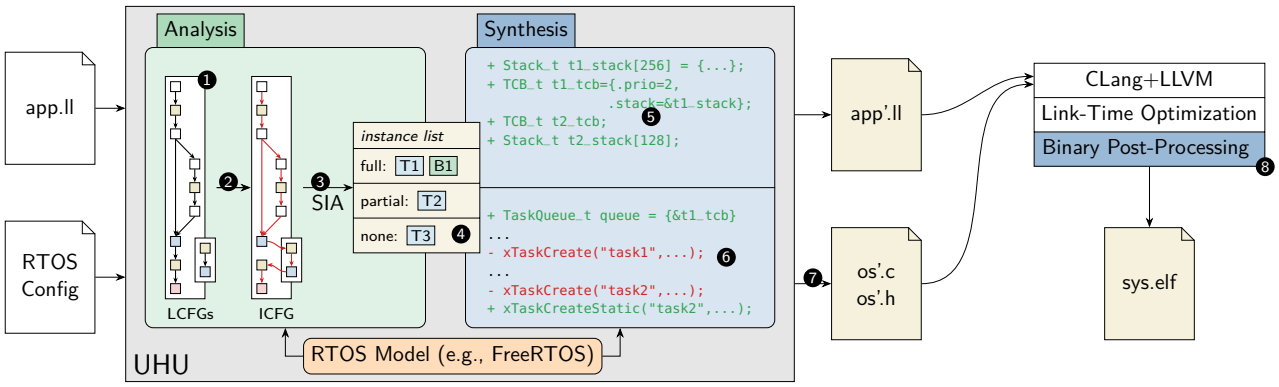


Fig. 2: ARA Overview: We analyze the real-time application in combination with the RTOS configuration. With the instance list, the synthesize phase modifies the application and specializes the RTOS that we link together, further optimize and output a system image.

To provide RTOS independence, ARA divides all algorithms into a generic part and an RTOS specific part, the RTOS model. It provides information about SOCs, especially its names, signatures and parameter interpretation. To capture the parameter values of instances, it provides instance specific objects. For later specialization, it also provides system calls that only perform partial initialization and object registration.

A. Static Instance Analysis

Goal of the *static instance analysis (SIA)* is the static detection of instances that the application creates in its setup phase. Thereby, we search not only for instances that are created unconditionally with constant parameters, but we also detect instances where we can deduce partial knowledge about parameters or instantiation count. The SIA is a static analysis, which - roughly speaking - traverses the application's CFG in a flow-sensitive and RTOS-specific manner while scanning for SOCs. For each identified SOC, we extract its creation parameters with help of the *static value-flow analysis (SVF)* [30] and evaluate the SOC according to an RTOS model, which is supplied by ARA, to build up a list of (possibly) created instances. Before the actual SIA starts (❸), ARA first has to preprocess the *local control-flow graphs (LCFGs)* (❶) to combine them into multiple *interprocedural control-flow graphs (ICFGs)* (❷).

❶ **Extract the *local control-flow graph (LCFG)***: After we have extracted the application code from the intermediate representation, the analysis preprocesses the application's CFG such that system calls and function invocations reside in their own basic block:

1) Split every (maximal) basic block that contains a function or system call directly before and after the call site, possibly creating empty basic blocks.
2) Categorize every basic block according to its contents into system-call blocks, function-call blocks, or computation blocks.

By following these rules, we get one function-local CFG (the LCFG) for each function. In their entirety, their basic blocks cover the whole application code. With the categorization, we put a focus on the application logic that is visible from the operating system's perspective and subsume all irrelevant computations into computation blocks, while interaction with the kernel can only take place in system-call blocks. This concept is strongly inspired by the *atomic basic block* concept of Scheler and Schröder-Preikschat [28], which also distills RTOS interaction into distinct blocks but performs further block-merge operations in order to form larger single-entry–single-exit computation regions within the CFG. Figure 1b shows the LCFGs (black edges) for the main entry point of Figure 1a.

❷ **Derive the *interprocedural control-flow graph (ICFG)***: For the `main()` function, and for each subsequently discovered thread entry, we combine the reachable LCFGs into one ICFG by connecting call sites with their possible callees. While our LCFG construction already isolated call-sites into their own basic block, we still have to determine the possible call targets for each call site. For this, we calculate an over-approximation of possible call targets, which we further filter down with help of the RTOS model.

For direct calls, the callee is explicitly mentioned in the LLVM IR and trivially to extract. For indirect calls via function pointers, the actual call target only becomes available at run time and we perform a static (pointer) value analysis to determine a set of possible callees. For this, we first try to resolve the function-pointer value with help of the SVF, which internally executes the Andersen pointer analysis [5]. If this fails, we filter the list of all possible functions with the required function signature at the call site. For arguments of non-pointer types the filter function simply checks for type equivalence. For arguments of pointer types, we must consider C++ inheritance. Pointers to the derived class are compatible to base-class pointers. Unluckily, LLVM implements C++ inheritance by embedding the base class into the derived class. Therefore, we accept arguments of pointer types as equivalent if the callers pointee type is embedded into the callee's pointee type. This results in a slight but safe over-approximation. Techniques of the field of control-flow integrity [25] could lead to an even tighter over-approximation, but require metadata of the compiler frontend, which we explicitly forbid. Furthermore, we also allow for manual annotations to restrict the targets of a specific call site.

As a second step, we filter these call-target set to only contain functions that could invoke a SOC. With help of the RTOS model, we mark functions that contain a SOC site as *creation relevant*. Recursively, every function whose call-target sets contain a creation-relevant function becomes creation relevant itself. For each call-site, we intersect the creation-relevant functions with the set of possible call-targets. If the set becomes empty, the function-call block is re-categorized as a computation block.

With the reduced callee set, we construct the ICFG: each function-call block is connected to its callee's entry blocks, while the callee's return blocks are connected to the successor blocks of the call sites. Please note that each call site has, due to our LCFG construction, a unique (although possibly empty) successor block. Figure 1b shows the ICFG (red) that we derive from Figure 1a for the main entry point. We see that the ICFG does only contain an edge to `library_init()` but not to `printf()`, as the latter function is not creation relevant.

❸ **Execute the *static instance analysis (SIA)***: The SIA statically analyses the ICFGs to compute the list of instances that the application can create. In a nutshell, we start the SIA for the main entry point and perform a flow-sensitive traversal of the ICFG to iterate over all pairs of the form (call path, SOC site). For these pairs, we extract the SOC parameters, interpret them according to the RTOS model, and collect the created objects in the analysis-time *instances list*. If we discover a new thread, we spawn another SIA for the thread's entry function. The detailed SIA algorithm is described in Listing 1 as simplified pseudo-code.

The SIA execution starts by calling `findSOC()` with the control-flow entry of the application (`main()`) and an initial empty call path, which we will extend with function-call blocks. `findSOC()` recursively follows the ICFG by iterating over ICFG successors (line 5, line 29f). When we visit a call site, we extend the call path (line 7) with the current function-call block in order to make precise call-path–dependent returns at the end of a function (line 13f). We end the ICFG traversal (line 9ff) if the last function on the call path returns or if we encounter the scheduler-start system call as the initial control flow terminates there. Furthermore (not shown for clarity), we visit loops only once and do not follow recursive function calls. While we descent into the call hierarchy, we also track (`in_thread`) whether we are currently executing within a thread context or within the initial control flow that executes before the scheduler starts.

For every basic block that `findSOC()` visits, we use the RTOS model to identify SOC sites and derive information about the created instances from it (line 16-27). First, we invoke the `evalSOC()` function with the current call path and the discovered SOC site to create an analysis-time representation (*instance*) of the system object that contains the RTOS-specific creation parameters. For the returned instance, we also store whether it will be created before or after the scheduler has started. Furthermore, we inspect the call path to determine whether this SOC site will yield a unique and unconditionally (`exactly_once`) created system object or if the creation cardinality could differ from one. The SOC is visited exactly once, if neither the current basic block or any of the function-call blocks on the call path are located in a conditional branch or within a loop in their respective function context. Both properties (`inLoop` and `inCond`) are calculated on the LCFG with dominator information and a reachability analysis. When `evalSOC()` discovers a new thread, we spawn another `findSOC()` recursion (line 24) with the thread's entry function as starting point and remember that we are now executing within a thread context.

```
1  type   CallPath  = Stack[BasicBlock]
2  global instances : List[SystemObjects]
3
4  def findSOC(callpath: CallPath, bb: BasicBlock, in_thread: bool):
5      next_bbs = bb.icfg_successors
6
7      if bb.isCallSite:
8         callpath.push(bb)
9      else if bb.isReturn and callpath.empty() \
10            or os_model.isSchedulerStart(bb):
11        next_bbs = []
12     else if bb.isReturn:
13        caller_bb = callpath.pop()
14        next_bbs  = caller_bb.lcfg_successors or []
15
16     if os_model.isSOC(bb):
17        obj = evalSOC(callpath, bb.syscall)
18        obj.before_sched = not in_thread
19
20        inLoop = any([caller.inLoop for caller in callpath + [bb]])
21        inCond = any([caller.inCond for caller in callpath + [bb]])
22        obj.exactly_once = (not inLoop) and (not inCond)
23
24        if type(obj) is Thread:
25           spawn findSOC([], obj.entry_block, true)
26
27        instances.append(obj)
28
29     for next_bb in next_bbs:
30        findSOC(callpath, next_bb, in_thread)
31
32 def evalSOC(callpath: CallPath, syscall: SOC)-> SystemObject:
33     vfg = SVF.getVFG(filter_by=callpath)
34     args = []
35     for param, ptype in os_model.params(syscall):
36        value_node = vfg.get(param)
37        while value_node.hasUniquePredecessor():
38           value_node = value_node.predecessor
39
40        if value_node.isConstant and ptype == CONSTANT:
41           args.append(value_node)
42        else if value_node.isSymbol and ptype == SYMBOL:
43           args.append(value_node)
44        else:
45           args.append(null)
46
47     obj = os_model.create(syscall, args)
48     vfg.get(syscall.return_value).setValue(obj)
49     return obj
```

List. 1: The *static instance analysis (SIA)* (Pseudo-Code). `findSOC()` iterates the call-paths of SOC relevant functions and invokes `evalSOC()` for every discovered SOC. `evalSOC()` extracts the creation parameters and returns an analysis-time *instance*. If we discover a new thread, we spawn another `findSOC()` recursion for the thread-entry function.

As explained, we use `evalSOC()` to extract call-path–dependent information for the SOC site. For this, `evalSOC()` extracts the SOC arguments with a value analysis and creates an instance of the system abstraction. As base for our value analysis, we use the SVF [30], which performs a flow-sensitive and inter-procedural value tracking and returns the *value-flow*

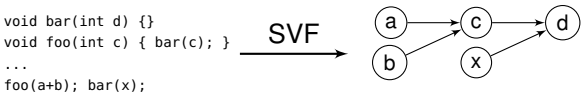


Fig. 3: Example Value-Flow Graph as constructed by the SVF

graph (VFG), which covers value flows within functions and across function boundaries. The VFG is a directed graph of value nodes that represent a symbolic or constant value. If a value depends on other values (i.e., by copy or calculation), the corresponding value node is a successor of the respective source-value nodes. In Figure 3, we see how the SVF tracks the arguments of `foo()` and `bar()` back to the variables `a`, `b`, and `x`.

In order to extract the SOC arguments, `evalSOC()` filters down the VFG (line 33) to contain only value nodes that are valid on the call path that leads up to the given SOC site. Thereby, we narrow down the VFG to the relevant value flows and get a more unambiguous view on the SOC arguments. In Figure 3, filtering with the call path `[foo(a+b), bar(c)]` would exclude the value node for `x`.

For the argument extraction, we ask the RTOS model for the signature of the SOC site. Thereby, we not only get the parameter list but also the information whether the RTOS model requires a compile-time constant or a globally-defined symbol. For example, for the thread entry, we require a function symbol to uniquely identify the thread entry.

We start the argument extraction (line 36ff) at the respective value node and search the filtered VFG backward until we find a node that has not exactly one predecessor. Thereby, we follow value–copy chains until we reach a constant (no predecessor), a global symbol (no predecessor), or a dynamic computation (more than one predecessor). We inspect the found value node (line 40ff) and extract, depending on the required parameter type, a constant or a symbol; otherwise we mark the SOC argument as unknown. With the argument list (line 47f), we create an RTOS-specific system-object instance and return it to `findSOC()`. By pinning the instance to the return-value node in the VFG, we are able to deduce previously discovered instances as SOC arguments.

During its execution, the *static instance analysis (SIA)* collects a global list of discovered and RTOS-specific instances (line 27). Due to the call-path-sensitivity of `findSOC()`, it is certain that SOC sites that are invoked on different call paths will yield multiple instances. For SOC sites in loops or in recursion, we only return one instance with `exactly_once=false` that represents the possibility of multiple (or none) SOCs at run time. Furthermore, as the set of call targets is an over-approximation, we are certain that the SIA visits all reachable SOCs. In Figure 4, we show the discovered instance list for the running example (Figure 1). Please note that `task3` is not created exactly once, as its SOC site is located within the conditional branch in `main()`.

B. Synthesis

Equipped with the instance list, we start the synthesis phase, where we specialize the application and RTOS to replace dynamic SOCs with static ones. For this, we first filter (❹) the instance list for SOCs that are suitable for specialization and manipulate the application (❺) to statically allocate and initialize them. Furthermore, we modify the application's SOC site (❻) to use the statically created objects instead of calling the RTOS API. To finalize the process, we generate (❼) and link a matching RTOS implementation and post process the system image (❽).

System Objects	task1	task2	task3	task4	task5	mutex	buffer
Created before scheduler	✓	×	✓	✓	×	✓	×
Created exactly once	✓	✓	×	✓	✓	✓	✓
RTOS allocates memory	✓	✓	✓	×	✓	✓	✓
All parameters known	✓	×	✓	✓	✓	✓	✓
❹ Specialization Depth	F	P	N	F	F	F	F
❺ Object Registration	S	D	–	S	D	S	D

Creation Parameters:
task1 = {type=Thread, stacksize=512, prio=2, ...}
task2 = {type=Thread, stacksize=256, prio=UNKNOWN, ...}
...
mutex1 = {type=Mutex}
buffer1 = {type=MessageBuffer, size=12}

Fig. 4: Detected instances and their creation parameters for Figure 1. The maximal specialization depth can either be **F**ull (static allocation+initialization), **P**artial (only static allocation), or **N**one (dynamic SOC). The created object is either registered **S**tatically or **D**ynamically at the RTOS.

❹ **Decision on Specialization Depth**: First, we categorize the discovered instances into different *specialization depths* based upon the static knowledge that we could derive in the SIA. We differentiate between three depths: full, partial, and none. We define these depths according to the amount of feasible specialization and the resulting potential for improvements. While we can pre-allocate the memory for both full and partial specialized system objects, we only initialize instances at the full specialization depth at compile time. For a depth of none, we perform no specialization and leave the dynamic SOCs unchanged.

As we perform compile-time specialization, we can only create objects statically, if we know that the object will surely exist. Hence, all SOCs which are not executed *exactly once* are categorized as none. Furthermore, as we do not perform call-path–specific code modification, we also categorize all after-scheduler SOCs that share their SOC site with another SOC as none. Thereby, we know for every specialized SOC site within a thread that at most one object will be created. For the yet unclassified SOCs, we categorize the ones where all creation parameters are known with a depth of full, while all others are categorized as partial. Thereby, we fall back to static memory allocation if a creation parameter depends on run-time knowledge.

For the running example, we show the specialization depth for the seven detected SOCs in Figure 4. We categorize `task3` as none, since `main()` creates it only under a condition. Furthermore, for `task2`, we cannot know its priority at compile time and therefore choose a depth of partial. For all other objects we perform a full specialization.

❺ **Static Object Instantiation**: For the deeper specialization depths, we statically instantiate system objects and register them in the RTOS data structures (e.g., insert a thread into the ready queue). Technically, we perform static memory allocation for partial and full specializations, but generate static initializers only at the full specialization depth.

For the static allocation, we chose to generate C code with global variables instead of directly encoding objects in LLVM IR. This is not only simpler to implement, it also fosters the portability of ARA: As the generated code fragment uses all necessary RTOS headers and we process it with the same compiler as used for the application and RTOS, we remain largely independent of the CPU architecture and the RTOS version. Even if the LLVM IR states to be platform independent, there are calculations, such as the computation of a data types size, which the frontend (clang) performs based on the selected architecture. By generating C code, we make sure that that the original application and the newly generated RTOS parts remain compatible in this respect. We statically allocate memory for the partial and full specialization depth and, thereby, reduce the necessity for dynamic memory allocation.

For the full specialization depth, we also initialize the object state: We derive the required C-initialization statements with the help of the RTOS model, which interprets the creation parameters as the RTOS would do it at run time. For example, we derive the initial top-of-stack address from the supplied stack size. Thereby, ARA initializes all fields that are direct properties of the system object.

However, FreeRTOS uses parts of the object state for its internal housekeeping. While we leave these parts untouched for SOCs that execute after the scheduler starts, we modify them on before-scheduler SOCs: By statically preparing the RTOS state, we emulate the effects of the SOC at compile time and thereby *register* statically-initialized objects at the RTOS. For example, FreeRTOS embeds pointers within its *task control block (TCB)* to enqueue the thread into the double-linked ready queue. While FreeRTOS normally boots with an empty ready queue, we are now pre-populating the queue with all threads that are surely created before the scheduling starts.

In Listing 2, we show a (simplified and reduced) version of the generated C source that statically creates and enqueues `task1` in FreeRTOS' ready queue. We statically allocate memory for the `task1`'s stack and its TCB, as well as for a multi-level ready queue with five priority levels. Furthermore, we set up the pointers of `xStateListItem` and the ready queue such that `task1` is already enqueued at priority 2.

```
#include "FreeRTOS.h"

StackType_t _task1_stack[512] = { ..., &t1_entry};.
TCB_t _task1_tcb = {
  .uxPriority = 2,
  .pxTopOfStack = &_task1_stack + 495,
  .pxStack =      &_task1_stack,
  .xStateListItem = {
    .pxNext =     &pxReadyTasksLists[2].xListEnd,
    .pxPrevious = &pxReadyTasksLists[2].xListEnd,
  }
};

List_t pxReadyTasksLists[5] = {
  ..., /* priority2 = */ {
    .xListEnd = {
        .pxNext =     (ListItem_t *) &_task1_tcb.xStateListItem,
        .pxPrevious = (ListItem_t *) &_task1_tcb.xStateListItem
    },
    .uxNumberOfItems = 1,
    .pxIndex = (ListItem_t*) &pxReadyTasksLists[2].xListEnd
  }, ...
};
```

List. 2: Generated Stack, TCB and Ready-Queue for `task1`

❻ **SOC Site Transformation**: After we have allocated and initialized as many SOCs as possible, we modify the application to use these objects instead of dynamically creating system objects. For reasons of readability, the following examples are denoted using C code, even though ARA performs all this modifications by manipulating the application's LLVM IR code at the SOC sites. Along the process, ARA takes the specialization depth into account.

For the partial specialization depth, ARA replaces the call target at the SOC site to a version of the system call that avoids the dynamic memory allocation but uses a user-supplied memory area. FreeRTOS already provides such system calls with the `Static` suffix, which eliminates the need for the RTOS model to provide them. In our example, `task2` has a partial specialization depth, so we replace the `xTaskCreate()` with `xTaskCreateStatic()`.

For the full specialization depth, the required IR modification depends on whether the SOC is created before or after the start of the scheduler. As we already registered all before-scheduler SOCs (❺), we have to replace the SOC site with IR code that only produces the expected return value. For example, ARA replaces `xMutexCreate()` in the running example with "`mutex = &_mutex1`".

If we know that a SOC is executed within a thread, we replace the creation call with a system call that only performs the object registration (e.g., thread enqueue). If the RTOS already provides such an API, we use it, otherwise ARA's RTOS model provides one. In the example, we replace the `xTaskCreate()` call for `task2` with a `vTaskResume()` call, which only enqueues the thread without initializing it.

With these modifications in place, the transformation of dynamic SOCs into static SOCs is complete and ARA writes the application's IR to a file to be further processed by the compiler toolchain.

❼ **RTOS Specialization**: Besides application modifications, we also specialize the RTOS to the now changed application requirements and incorporate analyses results for further optimizations. For this, we change the RTOS configuration and activate previously unused system calls while deactivating now unused ones. Furthermore, we adapt the startup routines to be compatible with our pre-initialized RTOS state and remove redundant initialization steps. For example, FreeRTOS normally initializes the ready queue during the startup process; a task that ARA now performs statically.

Furthermore, we also reduce the configured RTOS heap size: Since partial and full specialized system objects are now allocated as global variables, they do not draw memory from the RTOS-internal heap. If the SIA detects that all system objects are at least partial specializable, we even disable the RTOS's heap manager altogether.

```
@syscall(categories={SyscallCategory.create},
         signature=(SigType.symbol, SigType.value, SigType.value,
                    SigType.symbol, SigType.value, SigType.symbol))
def xTaskCreate(cfg, abb, state):
    ...
```

List. 3: Excerpt of the RTOS model describing the signature of the `xTaskCreate` SOC

C. Link-Time Optimization

After application and RTOS are modified, ARA performs the system-image assembly: We compile the generated source files with Clang to LLVM IR and link it with the modified application's IR and the specialized RTOS. Furthermore, we process the resulting system image with the link-time optimization passes of LLVM such that all system components can be optimized as a unit. As ARA initializes field values of statically-initialized system objects only with constant expressions, the optimizer is able to pre-compute the actual values and transfers these objects to the *data segment*, which is copied from flash to RAM at the boot time. Please note, that these pre-computable values also include pointer values with constant offsets (e.g., `&_task1_stack + 495` in Listing 2), although they are expanded at link time by the help of relocations.

For most data structures, it is profitable to use a pre-initialized image of the object in the data segment, as this avoids the execution of pseudo-dynamic run-time code during the system setup. However, this is not universally true for all (system) objects: Our early evaluations have shown that it is sometimes cheaper to dynamically write a few values to a sparsely-populated BSS-initialized object instead of putting the whole object into the data segment. The reason for this trade-off is the fact that initializing memory with zero is cheaper than moving data around. To avoid this setup-time degradation, we introduce a binary post-processing step.

❽ **Binary Post-Processing:** The described issue arises for data structures that mostly contain repeating values and are only sparsely populated with differing values. The prime example for this issue are pre-initialized stacks: In FreeRTOS, a freshly initialized thread stack only contains a small initial call frame to kick off the thread execution. Furthermore, we also found similar patterns of sparsely-populated objects in one of our case studies. Such sparsely-populated objects suffer from being statically initialized as copying their values from flash into RAM is more expensive than copying only the non-zero values.

To solve this problem, ARA provides a link-time optimization that performs *run-length encoding (RLE)* of sparsely-populated memory objects. During the system setup, while we prepare the BSS and data segment, we also expand the compressed image of the objects. This allows us to minimize the flash reads for repeating values to a minimum, resulting in reduced setup times.

IV. Evaluation

We implemented the above concepts within ARA in a combination of Python and C++. The latter is used for the LLVM-specific code in form of several Python extension modules. The model and high-level analyses are written in Python for higher flexibility. Especially the model is expressed as a Python class with domain-specific decorators to maintain a concise and readable interface.

For example, Listing 3 shows an excerpt of the model describing the signature of the xTaskCreate system call. In particular, the method is used in three cases: To classify a basic block as system call (`os_model.isSOC`, line 16, Listing 1), to provide the necessary knowledge for `os_model.create` (line 17, Listing 1), and to specify the arguments within the value analysis (`os_model.params`, line 35, Listing 1).

To show the improvements of our approach, we apply ARA to three micro benchmarks and to two real-world case studies. With the micro benchmarks, we quantify the maximal reachable enhancement that our approach can achieve without interference from other application initalizations. With the case studies, we furthermore demonstrate the general applicability of ARA on existing projects and also quantify the boot-time improvements and its moderate impact on flash usage.

As case studies, we chose the LibrePilot CopterControl[1] firmware as a safety-critical real-time application for the flight controller of a quadcopter and the GPSLogger[2], an embedded application for logging GPS positional data on an handheld device. Both projects use FreeRTOS as operating system and create system objects before and after the scheduling has begun. All our evaluations run on a STM32F103 MCU (ARM© Cortex©-M3 @72MHz, 128KB Flash, 20KB SRAM) on a STM32 Nucleo-F103RB evaluation board.

Until the SSP is reached, the boot process consists of four initialization steps: (1) copy the data segment from flash to RAM, (2) decompress the RLE segment, (3) initialize the BSS with zeroes, and (4) run the application setup by calling `main()`. For our time measurements, we record the timestamps for the SSP and each preceding initialization step with the CPU's *Data Watchpoint and Trace* (DWT) Unit. We repeated all time measurements 100 times and the standard deviation was always below 30 cycles, which is not surprising as the system setup for our benchmarks is not dependent on external inputs. Thus, the figures error bars are vanishingly small. Additionally, we collect the sizes of the code, data, RLE, and BSS segments.

For our evaluation, we define three benchmark scenarios where we limit the maximal specialization depths of the discovered SOCs:

Baseline As baseline for run-time evaluations and size comparisons, the build system bypasses the ARA framework and, therefore, we introduced no changes.

Partial For this scenario we ran the SIA but limited the synthesis phase to a maximal specialization depth of *partial*, even for objects were a higher specialization level would have been feasible. Thereby, ARA only replaces dynamic memory management with static object allocation.

Full For this scenario, we executed the full ARA approach and applied the maximal possible specialization depth for all discovered SOCs wherever they are feasible.

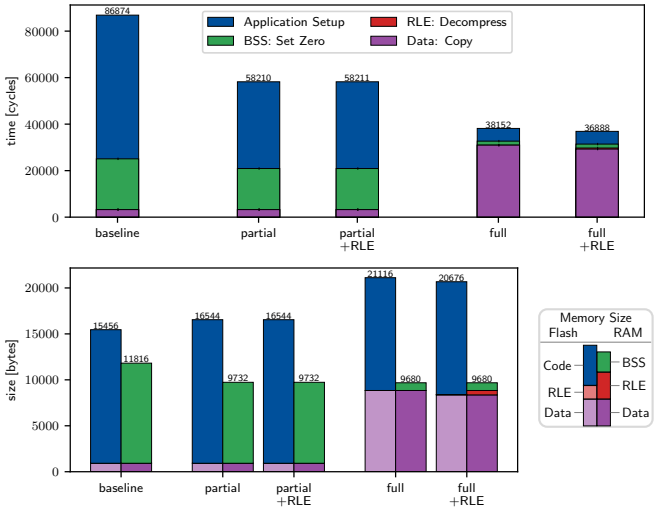


Fig. 5: Result for Queue Instantiation

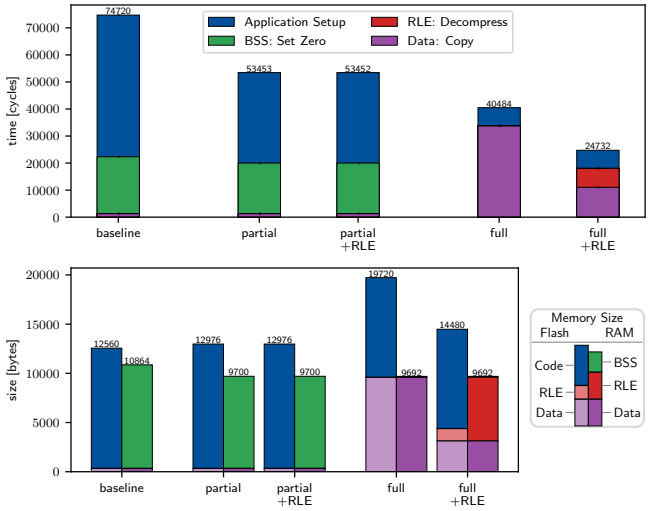


Fig. 6: Results for Thread Instantiation, Pre-Scheduler

A. Micro-Benchmarks

To evaluate ARA, we constructed three micro benchmarks that instantiate system objects such as threads and queues. Thereby, we can observe the impact of ARA without disturbance caused by other application logic.

a) Queue Instantiation: For this benchmark, we create 100 RTOS-managed queues via pseudo-dynamic `xQueueCreate()` calls. Each SOC is placed in its own statement (not using a loop) and we consider the scheduler start as the SSP of this system. In FreeRTOS, queues are only initialized but not registered with the RTOS. Only after a thread is waiting on the queue, it becomes known to the RTOS.

Figure 5 shows results for the SSP delay and memory requirements. For the partial scenario, the specialized system reaches the SSP 28 663 cycles (32.99 %) earlier than the baseline. However, since system calls that use statically-allocated memory have more parameters, each SOC site requires more instructions. In total, we end up with a flash-size increase of 11 bytes per queue creation.

For the full scenario, ARA even saves 49 986 cycles (57.54 %) compared to the baseline. Looking at the memory consumption, we see that objects move from the BSS to the data segment. However, as the data segment is loaded from flash, we require 52 additional bytes of flash memory per queue. In this scenario, the RLE compression only has a small influence as initialized queues are not sparsely populated with values.

b) Thread Instantiation, Pre-Scheduler: For this benchmark, we create 30 threads with `xTaskCreate()` in a row (without a loop) before starting the scheduler, which is the SSP for this benchmark. All threads come with a stack size of 200 bytes and have a maximal specialization depth of full.

Figure 6 shows that specializing to a partial depth already saves 21 267 cycles (28.46 %), while the full specialization even achieves a boot-time speedup of 34 236 cycles (45.82 %). Similar to the Queue instantiation benchmark, the per-object specialization comes at the cost of increased flash usage (full: +239 bytes per thread). However, with the additional usage of RLE, we can compress the sparsely-populated thread stacks and, thereby, limit the flash increase to 64 bytes per thread. When looking at the run-time measurements, we see that enabling RLE compression further speeds up the boot process of the full scenario by another 16 798 cycles. In total, this results in a decrease of the SSP delay by 49 988 cycles (66.9%).

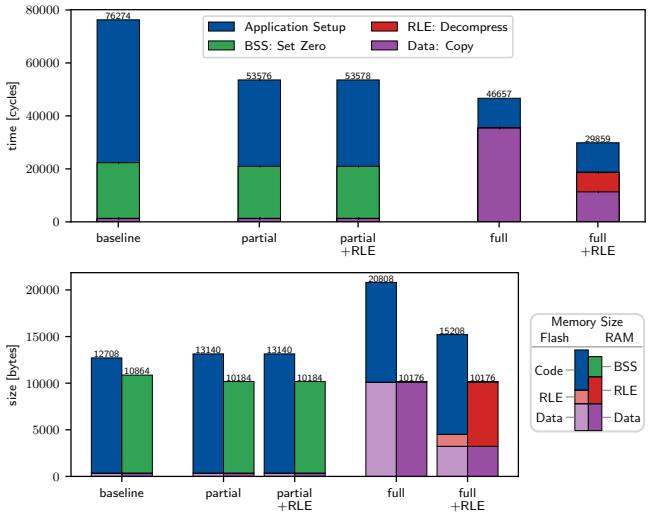


Fig. 7: Results from Thread instantiation, Post-Scheduler

c) Thread Instantiation, Post-Scheduler: As a second thread-creating micro benchmark, we build a system that creates only one initialization thread before the scheduling start. This thread subsequently creates 30 more worker threads with a priority that is less than the initialization thread. Each thread has a stack of 200 bytes and is fully specializable. The SSP is reached when the initialization thread finishes.

Since the 30 worker threads are only created after scheduling has started, ARA cannot avoid the run-time cost of thread enqueuing. Still, Figure 7 shows a decrease of the SSP delay by 46 415 cycles (60.85%) (full+RLE). For this benchmark, the flash cost per thread (81 bytes) is a little higher as the SOC sites still invoke the RTOS, but we see a similar benefit of enabling RLE for the data-segment size and the SSP delay.

[1]https://www.librepilot.org/, Version 16.0.9
[2]https://github.com/grafalex82/GPSLogger, Git commit: `8808b922`

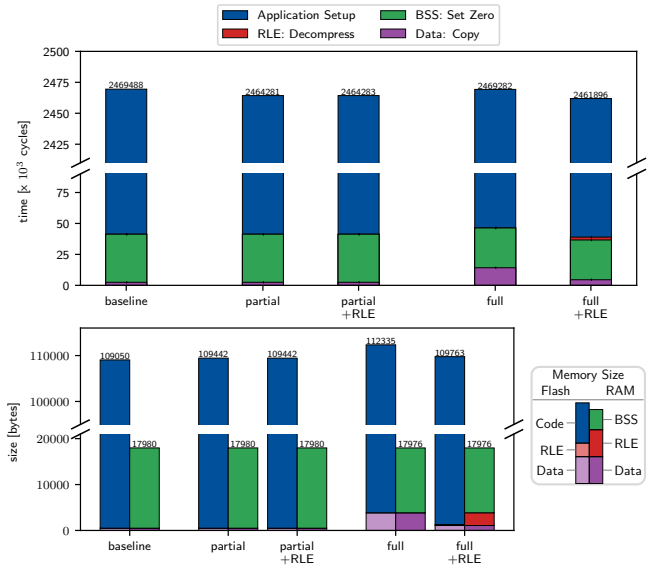


Fig. 8: Results from LibrePilot

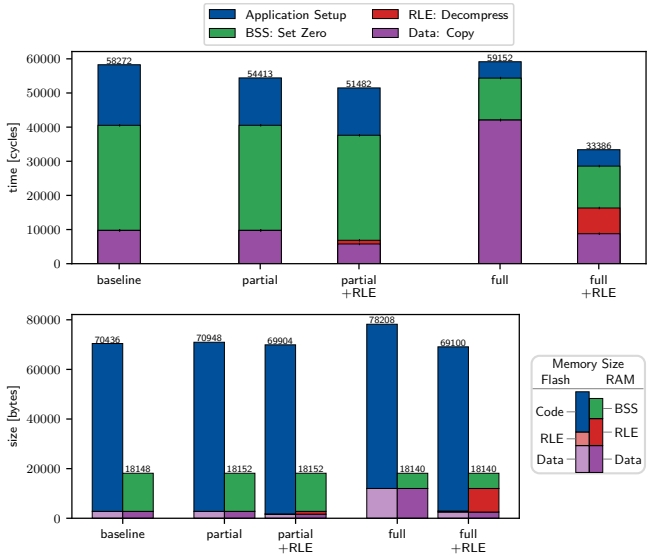


Fig. 9: Results from GPSLogger

B. Case Study: LibrePilot CopterControl

The LibrePilot CopterControl firmware is a configurable and flexible flight-controller firmware for quadcopters that executes on an ARM Cortex M3 microprocessor, which is connected to external peripherals and to an external flash storage. For this evaluation, we mocked the communication with the peripherals and used the board's internal flash memory. Thereby, we reduced the influence of external factors on the boot process. The system reaches its SSP, when the application reports a successful boot.

The application consists of 2673 functions with 17 265 basic blocks, which originate from 78 787 lines of code. It has 2882 normal function-call sites and 223 system-call sites of which 33 are SOC sites. The maximal call-path length to a SOC is 8. For its initialization, LibrePilot uses a custom linker script to collect its initialization functions in a function-pointer section, which is then iterated at boot time. To incorporate this link-time knowledge, we manually annotated the three call sites that use this section to use a smaller call-target set.

With these annotations in place, ARA detects 17 thread SOCs and 24 queue SOCs. Please note that the number of SOC sites is lower than the number of detected SOCs as some SOCs are visited on different paths. From the SOCs, ARA can specialize 5 thread SOCs fully and 2 thread SOCs partially; none of the queues was specializable as ARA marked them as being created conditionally. Reason for this low number of specializable SOCs are `assert()` statements, which are effectively irregular control flows that prematurely halt the boot process. As a failing assert bypasses coming SOCs, ARA marks all following SOCs as being conditional. In future work, we want to extend ARA to specially handle failures during the boot process.

With full specialization and RLE, we achieve a reduction of the system-setup time by 7592 cycles (see Figure 8). However, since the start-up phase of the LibrePilot makes liberal usage of dynamic initialization for library and application objects, ARA's SSP reduction boils down to a vanishing 0.31 percent speedup.

We manually investigated on LibrePilot's long boot process and found that wide parts of its system-setup code dynamically allocates user-level objects and initializes them with data that does not vary between reboots. Even more, these objects are never deleted and LibrePilot's FreeRTOS configuration uses a heap manager without a functional implementation for `free()`. Hence, LibrePilot shows the dynamic–object–creation problem not only on the level of system objects but also on the level of user-level objects.

C. Case Study: GPSLogger

The GPSLogger is a freely available application that collects GPS information. It consists of the MCU connected to a graphical display (I^2C), a GPS receiver (UART), an SD card (SPI), and two buttons (GPIO), all of which are only used after the SSP. The application code consists of 1311 functions and 11 165 basic blocks, which originate from 79 573 lines of source code. In total, there are 2303 function-call sites and 36 system-call sites of which 10 are SOCs. The maximal call-path length within the creation-relevant functions is three. As all system objects are created before the scheduler starts, we use the `vSchedulerStart()` as the SSP.

ARA detects 6 threads, 3 queues and one mutex and all discovered instances are suitable for the full specialization depth. In Figure 9, we see a startup-time reduction of 6790 cycles (11.65%) for the partial specialization with RLE. However, due to the delay to copy the data segment, a full specialization of all ten SOCs leads to an increased SSP delay of 880 cycles (+1.51%). This increase is mainly driven by the copy operation of the thread stacks from the flash memory. Therefore, enabling the RLE limits this effect and ARA achieves an overall SSP-delay reduction of 24 886 cycles (42.71%).

When looking at the flash requirements (full+RLE), two effects nearly cancel out each other: On the one hand, ARA enlarges the data segment by 124 bytes due to pre-initialized system objects. On the other hand, we are able to eliminate all dynamic memory allocations from the RTOS and, therefore,

ARA automatically disables the RTOS-internal heap manager, which reduces the code size by 1460 bytes. Thereby, the total flash requirements decrease by 1336 bytes.

Besides the targeted system objects, our RLE compression of sparsely-populated objects also had an impact on the GPSLogger application itself: The SD-card library statically allocates a large object (1136 bytes) that contains a buffer for the SD-card communication and some constantly-initialized state variables. Due to the static initialization of a few values, the compiler puts the whole object, including the large uninitialized buffer area, to the data segment. The RLE compression of this buffer accounts for 2931 cycles of our observed boot-time savings.

V. Discussion

In the evaluation, we have seen that ARA consistently reduces the SSP delay, although the observed improvements varied widely between 0.31 percent and 66.9 percent. We pay this reduction with an increased usage of flash memory as the binary's data segment now includes the pre-initialized system objects. As the system setup is only a rarely occurring event, the question arises if our achieved improvements are worth the effort?

For systems that recover from errors by rebooting [2], [3], the setup phase occurs at a critical moment and its worst-case delay determines whether the system is able to meet its deadlines even in the presence of errors. In essence, if we look at the whole real-time system, the setup phase must be considered as a non-preemptable sporadic task, executed with the highest priority, that has a (hopefully) very long inter-arrival time. Therefore, we believe that boot-time reductions will directly lead to better guarantees for such systems.

Compared to this boot time improvement, the increased usage of flash memory seems reasonable. In our case-studies, we have seen that overall impact on the flash usage varies (LibrePilot: +0.65%, GPSLogger: -1.9%) but is not significant compared to the rest of the application. Furthermore, for the class of embedded systems that ARA targets, the amount of available flash is often not a limiting factor. However, these systems often run at low clock speeds, which makes execution time a precious resource.

Furthermore, applying ARA is fast and automatic: The combination of ARA's analysis and synthesis phases ranges from about 2 seconds to about 70 seconds (LibrePilot) and happens without user intervention. In comparison to the rest of the build process, which takes about the same amount of time, ARA's compile-time overheads are reasonable and, as functional properties remain untouched, can be limited to important system-image builds. Furthermore, ARA is currently a Python prototype that we have not optimized for analysis time yet and whose conceptually most complex step (the SVF) completes in at most two seconds for our benchmarks. By using ARA in a continuous-integration and deployment pipeline, it will yield boot-time reductions in production without slowing down the normal development process.

Also, in contrast to the severe effects of switching to an RTOS that provides only static SOCs, ARA limits the RTOS-API usage only in a single aspect: In our current implementation, it forbids the user to delete static system objects as their memory was not allocated from the heap manager. However, system objects that are created exactly once with fixed creation parameters are unlikely to get destroyed and neither the GPSLogger nor LibrePilot perform such a system-object reclamation, which we enforced with an additional check in ARA. Although FreeRTOS provides heap manager implementations that are capable of handling memory reclamation, all examined applications use the simple bumping-pointer allocator which does not even provide a method to reclaim memory. Nevertheless, if memory reclamation is desired, ARA could extend the statically allocated objects with pre-initialized heap-manager information such that even those objects can get freed.

Another important aspect is the generalizability of our results to other applications. Although the Embedded Market Survey [15] suggests that 18 percent of the embedded developers use FreeRTOS, we had a hard time find available real-world benchmark scenarios that use it. However, with LibrePilot, we have successfully processed a representative application that makes heavy use of dynamic SOCs, which makes us confident that our approach will also yield boot-time reductions for other projects.

Although our implementation only supports FreeRTOS, the ARA approach is generalizable to other RTOSes that adhere to our system model: Since the presented algorithms are operating-system agnostic, we were able to encapsulate all RTOS specifics in a replaceable RTOS model. This RTOS model provides information about SOC signatures, the interpretation of the creation parameters, and the necessary methods to encode static system-object initializers as C code. For the synthesis, ARA also requires the RTOS model to provide system calls that only perform a partial initialization and object registration. At the moment, we are working on Zephyr [37] support for ARA.

The main factor to achieve large boot-time reductions is the precision of the SIA, which is mainly driven by the ability to restrict the call-target set for function-call blocks (see Section III-A, ❷). Since ARA is conservative and only specializes exactly-once SOCs, recursive call paths limit the specialization depth to none. So, if the call-target set for an indirect call site cannot be restricted enough, our over-approximation of the ICFG becomes very loose, and many SOCs that are actually static are left untouched by ARA. However, since indirect function calls are also problematic for other real-time analyses (e.g., WCET calculation), their usage is often restricted [24] or manual developer annotations are already available in the source code. Therefore, ARA is able to make use of such annotations and informs the user about large call-target sets.

Apart from the boot-time reductions, ARA's analyses results also allows for some basic sanity checks of the creation parameters against the given RTOS configuration. For example, we check that the given thread priority is in the range of allowed priorities and are able to emit a warning, where FreeRTOS silently caps the priority to the maximal available priority. Furthermore, after we have shrunk the size of the statically-allocated FreeRTOS heap (❼), we check that the remaining heap is large enough to hold at least one instance of each unspecialized SOC and emit a warning otherwise.

Besides all the technical improvements that ARA provides for the boot process, there is another benefit of ARA's hybridization between dynamic and static SOCs. Developers can use dynamic SOCs and enjoy their flexibility, while still getting the benefits of static SOC whenever possible. Thereby, developers can stick to their known OS-usage patterns as we retain the "POSIX-like haptic" of the RTOS. This is especially useful for the broader domain of embedded systems (e.g., IoT) where hard real-time constraints are less important and the availability of developers is a main driver of management decisions. In these domains, a flexible API is often preferred over the strict corset of a static system design, although the latter one is easier to analyze and yields better run-time characteristics.

Additionally, based on ARA's analysis results, we also provide a visual representation of the application's creation-relevant call graph and captured system objects. This can help developers to understand the project's initialization process as it highlights the important source-code parts where the system objects originate from. Thereby, ARA assists the developers to further optimize the boot process.

In future work, we plan to extend the ARA approach to support static initialization of user-level objects. For LibrePilot, we have seen such dynamic initializations can slow down the boot time of dynamic systems considerably. Therefore, we want to extend ARA to also recognize `malloc()` as a SOC-like function call and collect as much constant values as possible from the initialization site.

Furthermore, we want to extend the process of automatic staticalization to the used data structures: Currently, ARA remains compatible with the dynamic API of FreeRTOS and therefore has to generate objects as if they were allocated and initialized at run time. Thereby, we allocate and initialize data structures that are easy to manipulate (e.g., linked lists) for a system that might never modify these data structures. Hence, we want to extend ARA to exchange the RTOS-internal dynamic data structures by fixed-size structures and arrays to further improve the RTOS' run-time behavior and memory consumption.

VI. Related Work

Statically deducing computation results is the essence of an optimizing compiler and it is an incarnation of the seminal concept of partial evaluation [17]. For example, the LLVM framework [22] evaluates trivial constructors of global objects at compile time and places the initialized object image in the data segment. Even more, *compile-time function evaluation (CTFE)* is a topic of ongoing research [31] and several programming languages put a focus on better support for explicit CTFE: Since C++-11 [12], the `constexpr` keyword enabled developers to evaluate expressions at compile time and the keyword became less restrictive with every following language update. This development was fanned by the D language [14], whose compilers include an interpreter, and it also inspired the meta-programming capabilities of the Circle language [7]. In contrast to ARA, these implicit and explicit CTFE methods focus on single compilation units instead of the whole system and they perform no interpretation of RTOS-specific SOCs.

Wimmer et al. [35] extended the GraalVM to allow for application startup: At build-time, they execute parts of the application and produce a pre-populated JVM image that contains user-level objects as well as a pre-heated JIT cache. Thereby, the application startup boils down to a few `mmap()` operations. However, this approach only works for memory-safe languages as a precise points-to analysis is required and it remains limited to the application alone. On a broader scope, *link time optimization (LTO)* also performs a whole system optimization. For example, GCC's WHOPR framework merges global constructor invocations with the same priority and it reorders statically-allocated objects to increase code and data locality [18], [9]. In contrast to ARA, all these techniques focus on the application but leave the operating system aside.

Cockx [11] performs a *manual* staticalization of SOCs and thereby shows that moving pseudo-dynamic SOCs to the compile time is beneficial and he argues for an RTOS-aware compiler that performs this manual procedure automatically. Titzer [33] sacrifices compatibility with legacy applications by providing a domain-specific language for embedded systems that only allows compile-time initialization. However, the proposed language does not provide basic operating-system concepts like threads or mutexes. In contrast to this, ARA keeps compatible with legacy applications and with the existing RTOS interface and semantic.

Yang et al. [36] reduce Android startup times by up to 90 percent by loading whole-system images that were previously created with the suspended-to-disk mechanism of Linux. Singh et al. [29] performed a manual analysis of the Android startup and decrease the SSP delay by identifying parallelization bottlenecks, dropping unnecessary execution, or moving less-important initialization tasks beyond the SSP. In contrast to ARA, these approaches focus on a specific operating system and involve a lot of manual intervention by the system integrator.

Similar to ARA, Chanet et al. [10] analyze a concrete combination of application and RTOS to remove unreferenced system calls and unused system call parameters. Additionally, they deduce system-wide–constant system-call arguments and propagate them into the kernel. Lee et al. [23] also perform a whole system analysis for embedded Linux systems and eliminate dead code from the kernel. Bertran et al. [8] generalized this concept of system-wide dead-code elimination by construction of a global CFG that covers applications, libraries, and the operating system. Dietrich et al. [13] performs abstract interpretation of the interaction between RTOS and application to specialize system-call handlers. Unlike ARA, these automated whole-system analyses focus on the kernel behavior after the SSP and do not consider the static execution of SOCs.

Another direction of whole-system analysis in the real-time domain is dynamic tracing: Tools like Grasp [20] and Tracealyzer [34] record the system's operation to measure its timing characteristics, whereby they can report on observed SOCs. However, their reports are intended for developer feedback and automated testing procedures, and they are not used for automated system optimization.

VII. Conclusions

With ARA, we presented an RTOS-aware analysis and specialization method to reduce the *system-setup point (SSP)* delay. In a call-path sensitive and RTOS-aware analysis of the real-time application, we detect all *system-object creations (SOCs)* before and after the scheduling starts and infer if a specific SOC happens exactly once. For these SOCs, we statically allocate memory and, in case of compile-time–constant creation parameters, we also perform static initialization and registration with the RTOS. Thereby, ARA automatically transforms pseudo-dynamic SOCs, which yield the same object on every boot, to static SOCs, which come at much lower overhead during the boot process. We demonstrated the applicability of our approach with two real-world FreeRTOS applications and reduced their SSP delay by up to 43 percent at a moderate increase in flash usage. For micro benchmarks, the speed up was even up to 67 percent.

VIII. Acknowledgments

We like to thank the anonymous reviewers for their feedback and fruitful comments. This work has been supported by the German Research Foundation (DFG) under the grant no. LO 1719/4-1.

The source code and evaluation artifacts are available at:
`https://www.sra.uni-hannover.de/p/ara-rtas21`

References

[1] Portable operating system interfaces (POSIX®) – part 1: System application program interface (api) – amendment 1: Realtime extension, 1998.

[2] F. Abdi, R. Mancuso, S. Bak, O. Dantsker, and M. Caccamo. Reset-based recovery for real-time cyber-physical systems with temporal safety constraints. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8, 2016.

[3] F. Abdi, R. Mancuso, R. Tabish, and M. Caccamo. Restart-based fault-tolerance: System design and schedulability analysis. In *2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–10, 2017.

[4] Benny Akesson, Mitra Nasri, Geoffrey Nelissen, Sebastian Altmeyer, and Robert I. Davis. An empirical survey-based study into industry practice in real-time systems. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 3–11, 2020.

[5] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[6] Emmanuel Baccelli, Cenk Gündoğan, Oliver Hahm, Peter Kietzmann, Martine S Lenders, Hauke Petersen, Kaspar Schleiser, Thomas C Schmidt, and Matthias Wählisch. Riot: An open source operating system for low-end embedded devices in the iot. *IEEE Internet of Things Journal*, 5(6):4428–4440, 2018.

[7] Sean Baxter. Circle feature preview homepage. https://www.circle-lang.org/.

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

[9] Preston Briggs, Doug Evans, Brian Grant, Robert Hundt, William Maddox, Diego Novillo, Seongbae Park, David Sehr, Ian Taylor, and Ollie Wild. Whopr - fast and scalable whole program optimizations in gcc, initial draft. 2007.

[10] Dominique Chanet, Bjorn De Sutter, Bruno De Bus, Ludo Van Put, and Koen De Bosschere. System-wide compaction and specialization of the linux kernel. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES '05)*, pages 95–104, New York, NY, USA, 2005. ACM Press.

[11] A. Johan Cockx. Whole program compilation for embedded software: The adsl experiment. In *Proceedings of the Ninth International Symposium on Hardware/Software Codesign*, CODES '01, page 214–218, New York, NY, USA, 2001. Association for Computing Machinery.

[12] C++ Standards Committee et al. *IEC 14882: 2011 Information technology — Programming languages — C++*. 2011.

[13] Christian Dietrich, Martin Hoffmann, and Daniel Lohmann. Global optimization of fixed-priority real-time systems by rtos-aware control-flow analysis. *ACM Transactions on Embedded Computing Systems*, 16(2):35:1–35:25, 2017.

[14] Compile time function evaluation (ctfe) - dlang tour. https://tour.dlang.org/tour/en/gems/compile-time-function-evaluation-ctfe.

[15] EETimes, Aspencore. 2019 embedded markets study, March 2019.

[16] FreeRTOS homepage. http://freertos.org/.

[17] Yoshihiko Futamura. Partial evaluation of computation process–an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, page 381–391, 1999, Originally published in 1971.

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

[19] Martin Hoffmann, Christian Dietrich, and Daniel Lohmann. Failure by Design: Influence of the RTOS Interface on Memory Fault Resilience. In *Proceedings of the 2nd International Workshop on Software-Based Methods for Robust Embedded Systems (SOBRES '13)*, Lecture Notes in Computer Science. Gesellschaft für Informatik, 2013.

[20] Mike Holenderski, Reinder J. Bril, and Johan J. Lukkien. Grasp: Tracing, visualizing and measuring the behavior of real-time systems. In *in Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, pages 37–42, 2010.

[21] TRON Association ITRON Commitee. *µItron 4.0 Specification, Ver. 4.00.00*. 1999.

[22] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Washington, DC, USA, March 2004. IEEE Computer Society Press.

[23] C.T. Lee, J.M. Lin, Z.W. Hong, and W.T. Lee. An application-oriented linux kernel customization for embedded systems. *Journal of information science and engineering*, 20(6):1093–1108, 2004.

[24] *Guidelines for the Use of the C Language in Critical Systems (MISRA-C:2004)*. October 2004.

[25] Paul Muntean, Matthias Neumayer, Zhiqiang Lin, Gang Tan, Jens Grossklags, and Claudia Eckert. Analyzing control flow integrity with llvm-cfi. *Proceedings of the 35th Annual Computer Security Applications Conference*, Dec 2019.

[26] U.S. Department of Transportation National Highway Traffic Safety Administration. Laboratory test procedure for FMVSS111 – rear visibility (other than schoolbusses), 2018.

[27] OSEK/VDX Group. OSEK implementation language specification 2.5. Technical report, OSEK/VDX Group, 2004. http://portal.osek-vdx.org/files/pdf/specs/oil25.pdf, visited 2014-09-29.

[28] Fabian Scheler and Wolfgang Schröder-Preikschat. The RTSC: Leveraging the migration from event-triggered to time-triggered systems. In *Proceedings of the 13th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '10)*, pages 34–41, Washington, DC, USA, May 2010. IEEE Computer Society Press.

[29] Gaurav Singh, Kumar Bipin, and Rohit Dhawan. Optimizing the boot time of android on embedded system. *Proceedings of the International Symposium on Consumer Electronics, ISCE*, pages 503–508, 06 2011.

[30] Yulei Sui and Jingling Xue. Svf: Interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, page 265–266, New York, NY, USA, 2016. Association for Computing Machinery.

[31] Lucian Radu Teodorescu, Vlad Dumitrel, and Rodica Potolea. Moving

computations from run-time to compile-time: hyper-metaprogramming in practice. ACM, 2014.

[32] Henrik Theiling. Extracting safe and precise control flow from binaries. In *Real-Time Computing Systems and Applications, 2000. Proceedings. Seventh International Conference on*, pages 23–30. IEEE, 2000.

[33] Ben L. Titzer. Virgil: objects on the head of a pin. In *Proceedings of the 21st ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '06)*, pages 191–208, New York, NY, USA, 2006. ACM Press.

[34] Percepio Tracealyzr - Homepage. https://percepio.com/tracealyzer/.

[35] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss, and Thomas Würthinger. Initialize once, start fast: application initialization at build time. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):184:1–184:29, 2019.

[36] X. Yang, N. Sang, and J. Alves-Foss. Shortening the boot time of android os. *Computer*, 47(07):53–58, jul 2014.

[37] Zephyr Project homepage. https://www.zephyrproject.org/.