Protection and Relocation Extension for RISC-V

Maja Malenko*
malenko@student.tugraz.at
Graz University of Technology
Graz, Austria

Leandro Batista Ribeiro*
lbatistaribeiro@tugraz.at
Graz University of Technology
Graz, Austria

Marcel Baunach
baunach@tugraz.at
Graz University of Technology
Graz, Austria

Abstract

The current trend of transforming static embedded systems into open platforms (in which several software providers are able to directly load their software) drives the need to design and implement modular embedded software. Additionally, a myriad of embedded devices are expected to operate and provide services for years, or even decades, while remaining correct and secure at all times. Therefore, one of the emerging challenges for highly adaptive embedded computing platforms is to offer dynamic software composition at runtime and internal device housekeeping, which in conjunction improve device maintainability. With our hardware/software co-designed concept, the software can be dynamically updated at module granularity for application and middleware layers, while maintaining dependencies through loose coupling. At the hardware layer, a RISC-V extension enables the loose coupling by implementing effective runtime relocation and protection. This allows modules to freely move in memory without invalidating references.

Keywords: hardware/software co-design, modular software design, risc-v, partial software updates

1 Introduction

The advent of the Internet of Things (IoT) and the smartization of Cyber Physical Systems (CPS) has led to a drastic increase in the number of connected devices [1, 2]. A myriad of these connected embedded devices are expected to operate for years, or even decades (e.g., automotive, environmental monitoring, aerospace, industrial IoT). This long-term operation requires the capacity to add new software features, fix security vulnerabilities and bugs, adapt to changing legislation, etc. In order to provide these capabilities, reliable and convenient software updates are necessary.

While today's embedded systems are commonly updated by full image replacement, where the monolithic firmware image contains the entire software, for open systems, where the software composition develops over the year, this becomes especially complex. Partial updates must be supported while guaranteeing continued system correctness after each modification. With the trend of creating open embedded systems [7, 8] in which multiple parties are allowed to directly load their software into the device [9], support for modular (partial) updates becomes crucial. Parts of the system can be independently modified, while the rest of the system continues with its normal operation.

The support of modular updates brings an inherent overhead in comparison with the classical full image replacement, mainly due to dependency resolution. While monolithic systems solve all dependency issues at linking time, modular systems must do it at runtime, during the update process. Figure 1 shows three common strategies to handle dependencies updates. Strategy (a) must modify the references in all dependant modules, which can be very expensive. Strategy (b) uses an old version of the dependency to redirect references to the new version. Strategy (c) uses a dedicated indirection table that must be adapted upon updates.

Systems which are frequently updated and operate long-term exhibit memory fragmentation problems. Our approach eases the the relocation of memory segments without any additional effort for resolving dependencies. Two necessary mechanism that we provide are logical addresses and runtime relocation. Another paramount aspect of modular embedded systems is their maintainability. After a sequence of modular updates, the device memory potentially becomes fragmented, which leads to the waste of anyhow limited resources. Therefore, memory defragmentation is highly desirable. However, embedded systems are predominantly relocated with physical addresses, and moving modules to defragment the memory would require a further relocation of all moved modules. Additionally, the dependants of such modules would have to be relinked, in order to adapt their references to the moved (dependency) modules.

We propose a concept which uses the indirection approach from Figure 1 (c), enforced by hardware support for runtime relocation, in order to provide modular systems that can be dynamically updated and furthermore defragmented without the aforementioned overhead.

2 Background

2.1 SmartOS

SmartOS [4] is a small, modular, real-time OS designed for embedded devices. It features a small-sized microkernel that provides a minimal number of system services in privileged mode, including preemptive multitasking, priority-driven scheduling, task synchronization, dynamic resource management as well as centralized interrupt and system call handling. In SmartOS, a task is an independent and preemptive execution unit with its own code, data (including stack), and priority. The memory space is linear and shared between the

*Both authors contributed equally to this research.

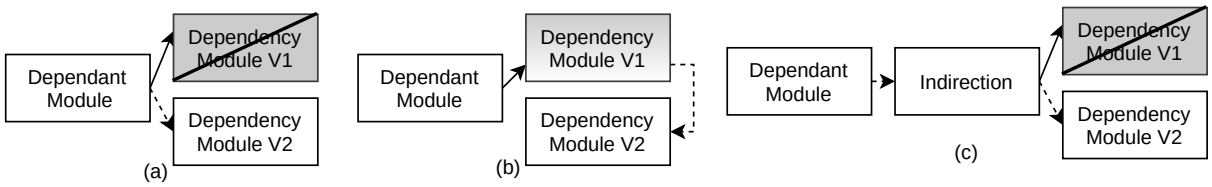

Figure 1. Strategies to keep references to dependencies consistent after an update.

OS and the tasks, including middleware code (e.g., drivers, libraries) executed in task context.

2.1.1 Modules. We define a module as software that is independently developed and utilizes the system's interfaces. While application modules contain one or more tasks, middleware modules implement library or driver routines and contain no tasks. The memory layout and the structure of modules is shown in Figure 2.

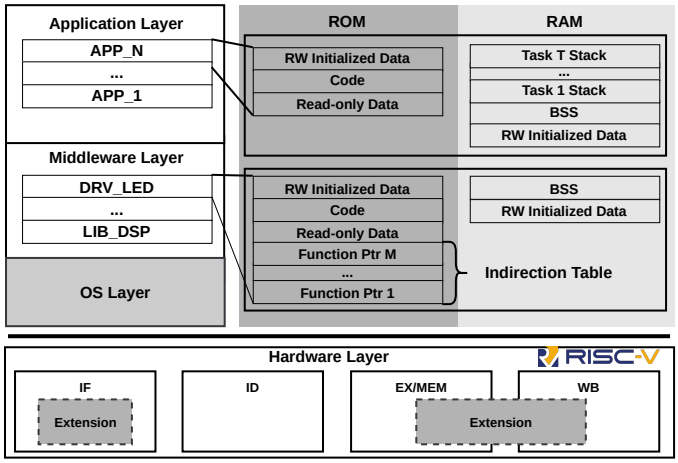

Figure 2. Memory Layout in SmartOS.

2.1.2 Partial Updates. Like most embedded operating systems, the original version of SmartOS is modular at compile-time but not at runtime and supports updates only through a full image replacement. An image is a monolithic software statically built by integrating the OS and the modules.

Initial support for partial runtime updates on module granularity was added to SmartOS in [3]. The client-server update protocol distributes the update operations between the embedded device and the high-performance update servers which act as software repositories. Compiling and linking the module is performed on the server. The device is only responsible for installing and loading the module, subsequently creating the necessary data structures required for its execution. Figure 3 illustrates a simplified version of the update process. Eventhough secure binary transfer is a crucial step in over-the-air updates, and it comprises of cryptographically signing and verifying modules, it is out of our scope.

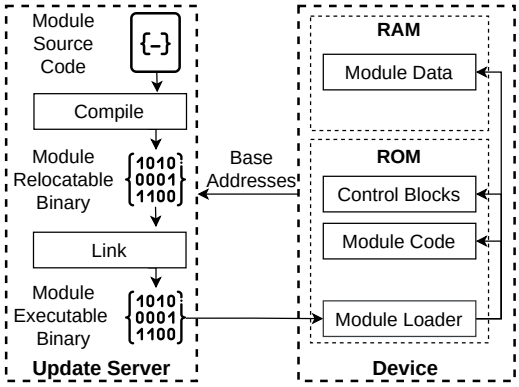

Figure 3. Simplified update process from [3].

2.1.3 The Effect of Using Physical Addresses. To link the module, the update server needs detailed information about the current memory layout of the device. The device must, therefore, provide the base addresses where the module will be stored and loaded and the addresses of the module's dependencies (e.g., middleware module functions). However, in [3] the exchanged addresses are physical, and modules are built with references to Physical Addresses (PAs). In systems with partial runtime updates, the downside of using PAs is obvious. Once a module has been loaded into the device's memory, it cannot be moved (e.g., during memory defragmentation), and if any of its dependencies change, its references become invalid. As a result, updating a module may also require adapting other modules (without any functional modifications).

2.2 RISC-V

We use an MCU based on RI5CY, now known as CV32E40P, a 32-bit single-issue, four stage pipeline implementation of the RISC-V ISA. It is directly connected to data and instruction memories with a single-cycle access and contains several memory-mapped on-chip peripherals. As per RISC-V privileged specification [6], RI5CY supports machine (M) and user (U) modes as well as a Physical Memory Protection (PMP) unit, necessary for developing secure embedded systems. The PMP grants permissions and protects up to 16 memory regions. This standard PMP specification does not meet our system's requirements and as described in Section 3.2, our approach implements a modified and extended version

which facilitates not only protection, but runtime relocation as well.

3 Design

Our system facilitates dynamic partial updates by providing Runtime Relocation (RTR) of modules. At the same time it isolates modules while allowing for limited and well de ned interactions.

3.1 Software Design

We divide the system in three logical layers: application, middleware, and OS, as illustrated in Figure 2. The OS layer is monolithic and self contained, while the other two layers are modular. Modules can be added, removed, or updated at runtime. Applications can use middleware modules and the OS, but not other applications. Middleware modules can use the OS and other middleware modules. Moreover, the accesses between modules must happen through well de ned interfaces.

3.1.1 Logical Addresses. To support modular updates, we leverage the concept from [8] as a base. As mentioned before, using this concept a dependant module (M) is linked against the Physical Addresses (PAs) of its dependency module (EM) (we will call this module external). If the dependencies in EM are updated and receive new PAs, the references in M become invalid. Moreover, if M or EM are relocated (e.g., during memory defragmentation), internal and external references (e.g., local variables, external function calls, etc.) become invalid again.

To overcome these challenges and ease RTR, we build modules using Logical Addresses (LAs). Since application modules are not directly used by any other module, all application modules are compiled to the same LA. However, middleware modules can be accessed by other modules and are executed in the context of the running task. Therefore, we assign them di erent and unique LAs. As a fortunate by-product, this strategy facilitates memory defragmention (on module granularity), since the RTR allows modules to freely move along the memory space.

3.1.2 Inline Tagging. The assignment of LAs to di erent types of modules is done using inline address tagging. The logical address space is partitioned as shown in Table 1. We embed metadata in the upper unused bits of the LA. In our current con guration by using 5 bits (i.e., Module Tag [29:25]) of the 32 bit LA, we support up to 31 tagged middleware modules and one Module Tag is reserved for all application modules. To be able to relocate module's code and data sections separately, we compile them with di erent Memory bit (bit 30). We set the Privilege bit (bit 31) when compiling modules. The remaining 25 bits represent the address o set and they limit the maximum size of a module to 32 MB.

Table 1. Address patterns of a 32-bit logical address space.

Privilege [31]	Memory [30]	Module Tag [29:25]	O set [24:0]	Access Type
0	x	xxxxx	xx...x	Kernel
1	x	xxxxx	xx...x	User
x	0	xxxxx	xx...x	ROM
x	1	xxxxx	xx...x	RAM
1	x	00000	xx...x	APP Mod
1	x	00001	xx...x	MW Mod 1
...				
1	x	11111	xx...x	MW Mod 31

3.1.3 Indirection Tables. By using LAs in conjunction with RTR, a module M can be easily relocated and its internal data and code references will stay consistent. However, this does not solve all the issues when M is accessing an external module EM. If M is tightly coupled to its dependencies (e.g., functions in EM) and those dependencies within EM change, M's references become invalid. This can happen in case the functions in EM grow or shrink and change their o sets (i.e., LAs within the module).

By decoupling a module from its dependencies a module is not required to be changed in case any of its dependencies are updated. Therefore, we provide loose coupling by using the concept of indirection. We build an indirection table into every middleware module, which consists of function pointers to the implemented functions in that module, as shown in Listing 1. At build time, we ensure that the indirection table is always located at o set zero, i.e., at the beginning of the module. The indirection table is the only valid entry point into the middleware module. More than a simple interface, it allows modules and their functions to grow and shrink at will between versions, since it always stores references to their appropriate positions within the module. The only requirement is to keep the indirection table consistent throughout versions. For example, in Listing 1 _LEDConfigure is in the rst slot in the indirection table of the LED middleware module. For backward compatibility, future versions must also have _LEDConfigure in the rst slot. Adding new functions to a module is allowed. However, incompatibilities arise if the position of the addresses of existing functions within the indirection table changes. We see version control orthogonal to our concept, and therefore do not address in this work.

```
1
2 // led.c
3 static Res_t _LEDconfig(led_t led){ ... }
4 static Res_t _LEDtoggle(led_t led){ ... }
5 static Res_t _LEDsetState(led_t led, ledSt_t st)
      {... }
6 const void* _SECTION_(".IT") IT_DRV_LED[]=
7 {&_LEDconfigure, &_LEDtoggle, &_LEDsetState};
```

```
// led.h
#define LEDconfig((Res_t(*)(led_t ))(    IT_DRV_LED[0])
#define LEDtoggle ((Res_t(*)(led_t))(    IT_DRV_LED[1])
#define LEDsetState((Res_t(*)(led_t led, ledSt_t
    st)( IT_DRV_LED[2])
```

Listing 1. Indirection table (IT) for a LED middleware module. IT_DRV_LED is the interface, and it contains pointers to all offered functions.

Figure 4. Relocation and Protection Logic.

3.2 Hardware Design

The software design decisions described in Section 3.1 can only function correctly with a tailored hardware support. Therefore, we design and implement a hardware extension which is responsible for two main mechanisms, memory protection and Runtime Relocation (RTR) of modules. It also facilitates the loose coupling concept of middleware modules.

The hardware extension is designed as a separate hardware component and attached to the RI5CY pipeline as shown in Figure 2. The data for configuring the protection and relocation registers is transferred via the CSR module. It implements the relocation logic and generates exceptions when the module's boundaries are not respected. Upon exception, the current execution is aborted and the execution flow is redirected to the kernel's exception handling routine.

3.2.1 Memory Protection. Complementary to the core's vertical isolation between kernel and user mode, our system leverages the hardware extension for horizontal isolation between modules. To enforce the memory boundaries of a module the hardware uses a set of start and end Protection Registers (PRs), a concept used in Memory Protection Units (MPUs) to define the address range of the module's memory regions. Each memory access initiated by the module is intercepted and checked against the allowed ranges in these registers (i.e., if no address range is defined, it can not be accessed). The kernel is accessed only through re-entrant system call wrappers which are always open for access. These checks are only performed when the system is running in user mode, inherently allowing full memory access to the kernel.

3.2.2 Relocation Logic. To allow modules to freely move along the physical memory space, we add a complementary RTR logic to the range protection checks using Relocation Registers (RRs). The CPU fetches instructions and accesses data using the module's Logical Address (LA). The hardware subsequently, after checking the range permissions, calculates the correct PA by adding the LA to the corresponding RR, as shown in Figure 4.

3.2.3 Number of registers. PRs and RRs of middleware modules are only (re-)configured by the kernel every time when modules are updated, deleted, or installed for the first time. In contrast, PRs and RRs of application modules are populated with the information stored in the Module Control Block (MCB) upon context switch.

As modules have fixed LAs, we eliminate the need for the start PR, saving hardware resources. According to Figure 2, the code and data sections of modules are continuous. Therefore, our architecture requires only two memory regions per module, which are protected (2 PRs) and relocated (2 RRs). While all application modules share the same PRs and RRs, the middleware modules have dedicated ones. Thus, the number of registers is defined as:

PRB = RRB = 2 · 1 , (D??>AC43 "833;4F0A4 ">3D;4B

If middleware modules also had shared registers, we would need to reconfigure them every time an application or middleware module interacts with different middleware modules. Since PRs and RRs must be configured in kernel mode (for security reasons), that would require a context switch upon every middleware interaction, which would add considerable execution overhead. Hence, we pay for faster software execution with extra hardware registers.

3.2.4 Manipulation of Tag Bits. In Section 3.1, we have explained how we partition the logical address space for relocatable modules. Figure 4 illustrates how the hardware extension uses this information to perform the LA to PA conversion. On each memory access, the hardware extracts the inline address tag bits from the LA and controls the memory access permissions. As shown in Figure 4, based on the Module Tag bits the hardware distinguishes whether an application or any of the middleware modules is currently running, and accesses the correct set of registers to protect (Range Checker) and relocate (Relocator) accordingly. Furthermore, depending on the Memory (bit M), the data or code protection and relocation are chosen. The most significant bit in the LA (bit P) identifies its address space (kernel or user). While protection is active only in user mode, the relocation happens continuously on every memory access. Having in mind that our OS is monolithic and not relocatable, the content of the RRs has no effect in kernel mode.

The hardware extension also ensures that middleware modules are accessed only through the indirection table, as a single valid entry point. Namely, the indirection table of a middleware module is accessed only through load data access instructions (i.e., lw) which retrieve the function pointer

from the respective slot. Any load data access outside the indirection table of a called middleware module will generate an exception. The nature of data accesses is identified by comparing the Module Tag bits between the caller and the callee. If the data LA and the current PC have same tags, the access is *intra-module* and is always allowed. If they are different, the access is *inter-module*, and it is allowed only within the indirection table of the called module.

4 Evaluation

We analyzed the memory overhead, hardware logic overhead, and execution time overhead in our modular version and compared it with the monolithic version with the same software configuration.

The system prototype is implemented on a Basys3 Artix-7 evaluation board, running at 50 MHz clock speed. The prototype consists of a RI5CY-based (RV32IM) MCU [5] with several on-chip peripherals (e.g., UART, GPIO) and SmartOS running on top.

Simulation. To evaluate the Runtime Relocation (RTR) logic, we check if the application module M shown in Listing 2 works as expected, even after updating and moving its dependency, the LED driver. The LAs of both modules before the update are summarized in Table 2.

The simulation waveform in Figure 5 shows all relevant signals to evaluate the RTR. The value of PC_PA is the sum of PC_LA and APP_ROM or MW_RT. The first data access (Data_LA = 0xBA000008) reads _LEDsetState' LA (0xBA00000C) from the LED driver's indirection table. Upon the function call (PC_LA = 0x80000038), the PC jumps to _LEDsetState, whose PA is 0x00002F44. On the second function call (PC_LA = 0x800000a8), the PC jumps to the updated _LEDsetState, whose new LA (0xBA000008) and its corresponding PA (0x00003068), were affected by updateLEDDriver. Finally, after moveLEDDriverCode, _LEDsetState is again called with the same LA (0xBA000008), but with a different PA (0x00003110), since moving a module affects its PA, but not the LA.

```
1 OS_TASKENTRY(M){
2         LEDsetState(LED_01, LED_ON);
3         updateLEDDriver();
4         LEDsetState(LED_02, LED_ON);
5         moveLEDDriverCode()
6         LEDsetState(LED_03, LED_ON);
7 }
```

Listing 2. Triggering problematic actions.

Table 3 shows the *memory overhead* of the system. The modular version is larger because it requires extra functionalities, such as management of modules and memory. These functionalities keep track of the installed modules and the available memory.

Table 4 shows the *execution time overhead* of both versions. The focus is on the comparison of local (within the module, no indirection) and external (in another module, with indirections) function calls, as shown in Listing 3. In the modular version, local function calls have virtually the same performance as the monolithic one. On the other hand, the indirection on function calls introduces a considerable overhead. However, the more cycles the external function requires, the smaller the relative overhead, since the indirection always needs a fix amount of cycles. Our test issues a result very close to the worst case.

Module		Start LA	End LA
taskModuleM	Code	0x80000000	0x80000110
LED Driver	IIT	0xBA000000	0xBA000008
	Code	0xBA00000C	0xBA0000A0

Table 2. Logical Addresses (LAs) of application and LED middleware modules analyzed in Figure 5

```
1 OS_TASKENTRY(taskModuleM){
2     // Local  function calls
3     for (int i=0; i < REPEAT; i++) localFunc();
4     // External function calls
5     for (int i=0; i < REPEAT; i++) externalFunc();
6 }
```

Listing 3. Benchmarking the overhead of local and external function calls.

Table 3. Comparison of memory overhead (in bytes).

	.text	.data	.bss	Total
Monolithic	10784	200	76	11060
Modular	12159	216	84	12459
	+13%	+8%	+11%	+16%

Table 4. Execution Time [µs] of the monolithic and modular software from Listing 3, with REPEAT = 100

Operation	Monolithic	Modular
Local Function Call	59.36	59.47 (+ 0.19 %)
External Function Call	52.64	64.62 (+ 22.76 %)

We do not analyze the processing time required for transferring, installing, and loading of modules, nor we measure the amount of exchanged data. These evaluation are presented in [3].

To evaluate the additional *hardware logic*, we measured the utilization of logic cells on FPGA as reported after synthesis and shown in Table 5. In the evaluation are included 2 on-chip peripherals, BRAM-based data and instruction memories, and the Baseline RI5CY implementation, which