Bringing Segmented Stacks to Embedded Systems

Zhiyao Ma and Lin Zhong
Yale University
{zhiyao.ma,lin.zhong}@yale.edu

Abstract

Microcontrollers are the heart of embedded systems. Due to cost and power constraints, they do not have memory management units (MMUs) or even memory protection units (MPUs). As a result, embedded software faces two related challenges both concerned with the stack. First, in a multi-tasking environment, physical memory used by the stack is usually statically allocated per task. Second, a stack overflow is difficult to detect for lower-end microcontrollers without an MPU.

In this work, we argue that segmented stacks, a notion investigated and subsequently dismissed for systems with virtual memory, can solve both challenges for embedded software. We show that many problems with segmented stacks vanish on embedded systems and present novel solutions to the rest. Importantly, we show that segmented stacks, combined with Rust, can guarantee memory safety without MMU or MPU. Moreover, segmented stacks allow memory to be dynamically allocated to per-task stacks and can improve memory efficiency when combined with proper scheduling.

1 INTRODUCTION

Most widely used software uses the *contiguous stack*, which occupies a contiguous region of memory. In multi-tasking systems, every task has a contiguous stack with a statically determined maximum size. For example, in 64-bit Linux, a thread has a default stack of 8 MB of virtual memory. In embedded systems without virtual memory, a task can have a stack of 100s or 1000s of bytes of physical memory.

Contiguous stacks raise two challenges for embedded software. First, without virtual memory, they are inefficient because physical memory is statically allocated to them. In §2, we report a case study with the firmware of a commercial miniature drone. We show that most of the allocated stack memory can remain unused for most of the time. This can be problematic because modern embedded systems can have many tasks, multiplying this inefficiency.

Second, without virtual memory, stack overflows become tricky to detect. For microcontrollers with an MPU running multiple tasks, stack overflows can be detected by either allocating a contiguous memory region for each task and reconfiguring the MPU at context switch, as in Tock [15], which sacrifices memory efficiency, or adding stack probing instructions, which can incur higher overhead for large stack frames and is not correctly implemented by major embedded compilers, e.g., ARM GNU [25] and LLVM. Moreover, many low-end microcontrollers do not have an MPU, e.g., those running the wireless network stack in Ambiq Apollo Blue series [1] and Nordic nRF52 series [22], which makes stack overflows difficult to detect. For embedded software on such microcontrollers, memory safety remains an elusive goal, even when written in a safe language like Rust.

In this paper, we consider a rather old idea, called *segmented stack* [14], toward solving the above two challenges for embedded software. Unlike the contiguous stack, a segmented stack consists of stacklets that are dynamically allocated from the heap (See §3). It relies on compiler-inserted function prologue to detect impending stack overflows and grows on-demand, and it also frees the memory when the offending function returns. The segmented stack is known to suffer in performance and memory efficiency [12, 13, 24]. Not surprisingly, while it is supported by both GCC and Clang, it is rarely used in mainstream systems. New languages such as Go and Rust have dropped their support for the segmented stack [12, 24].

In §4, we present an in-depth investigation into why the segmented stack suffers in performance and memory efficiency, confirming a known problem, i.e., hot-split, and revealing many new ones. In §5, we show that many of the issues with the segmented stack vanish on microcontroller-based systems while achieving memory safety and creating new opportunities for memory efficiency. In §6, we report a preliminary implementation of the segmented stack for Rust and C, with a novel solution to the hot-split problem and other optimizations. In §7, we report preliminary experimental results that show the performance of our implementation of segmented stack is within 94% of that of contiguous stack. Our current implementation, however, doubles the maximum stack usage for a task. We show this overhead can be substantially reduced with some ABI change. More importantly, when tasks are scheduled properly such that they do not reach maximum stack usage at the same time, the actual memory usage by all stacks can be still much lower than that by statically allocated stacks. This introduces a new dimension into the task scheduling problem.

In summary, this work makes the following contributions:

- A small case study to reveal the problem with the contiguous stack for embedded software.
- An in-depth analysis of problems with the segmented stack and their (ir)relevance to embedded software
- A preliminary implementation of the segmented stack for embedded C and Rust with important optimizations to evaluate its feasibility on microcontroller.

2 A CASE STUDY OF EMBEDDED STACK

We use the firmware of Crazyflie [11], an open-source, commercially available miniature drone, as an example to shed insight into the memory usage by stacks in embedded software. The firmware is based on FreeRTOS [8], a widely used open-source embedded operating system that supports statically allocated per-task contiguous

HotMobile '23, February 22–23, 2023, Newport Beach, CA, USA

ACM ISBN 979-8-4007-0017-0/23/02.
https://doi.org/10.1145/3572864.3580344

Task name	Period	Min stack usage	Max stack usage	Sleep time	Allocated stack
`Stabilizer`	1 ms	8 B	384 B	95.9%	3600 B
`Sensors`	1 ms	88 B	420 B	46.9%	2400 B
`CRTP RX`	3.5 ms	36 B	168 B	99.2%	2400 B
`CRTP TX`	∞ ms	36 B	104 B	100.0%	1200 B

Table 1: Stack Usage by Crazyflie Core Tasks

stacks. The Crazyflie firmware implements its core algorithm and sensor drivers in 21 tasks.

We analyze four core tasks as shown in Table 1. We derive stack usage by statically analyzing the disassembly code. The maximum stack usage is the greatest total size of stack frames in all feasible code paths, while the minimum stack usage is the size of context that must be carried over every task sleep. The allocated stack size of each task is determined by the configuration macros. We record the temporal stack behavior by gathering runtime traces while the drone hovers. When a task sleeps on a periodic timer, hypothetically it should keep its stack at the minimum usage, but in reality it takes an extra 200+ bytes, due to the stack frame of system call library functions and the exception frame and register context pushed by FreeRTOS onto the task's stack. Despite this, we believe with code refactoring the stack usage can be reduced close to the hypothetical minimum if we store the context outside of the user stack and streamline the library functions.

We have the following observations from our case study.

- *Stacks use substantial memory.* The stacks of all 21 tasks occupy 45600 bytes, which is 32% of the total system memory usage.
- *The maximum stack usage is significantly smaller than the allocated stack size* for all four tasks. This is likely because determining the maximum stack usage is difficult and the developers chose to leave a large margin to avoid stack overflows.
- *Tasks spend most time blocked with minimum stack usage, which is significantly smaller than the maximum.* The `sensors` task is an exception, which blocks on the DMA system call when the stack usage is high. The `CRTP TX` task is driven by outgoing packets, which happen rarely.
- *Large arrays and structures are commonly declared as static variables.* Some of them do not necessarily require a static lifetime, e.g., the `sensorData` struct in the `stabilizer` task. However, changing them to stack variables does not make any difference in memory usage when the stack is statically allocated.

The above observations highlight the memory waste of statically allocated, contiguous stacks: while the task stack usage is dynamic and low on average (over time), the system has to dedicate a chunk of memory to the stack based on the maximum usage, no matter how briefly the task may actually use that much stack, and usually with a large margin. This motivated us to investigate an alternative stack design: segmented stack.

3 WHAT IS SEGMENTED STACK?

The segmented stack is a type of call stack, which was initially invented for efficient first-class continuation implementation [14]. It detects an upcoming overflow in software and avoids it by growing by a contiguous chunk of memory, called a stacklet. A runtime library allocates the stacklet, frees it when the offending call returns, and tracks allocated stacklets as a linked list.

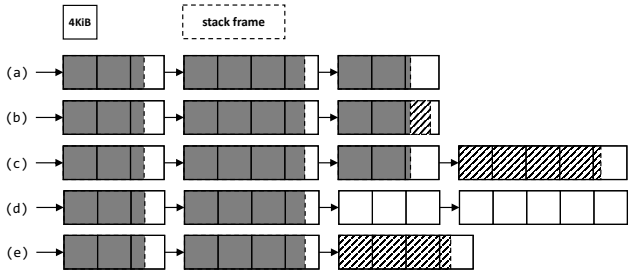

Figure 1: Different states of a segmented stack

GCC and Clang implement the segmented stack by prefixing each compiled function body a prologue that at runtime examines the remaining free space in the stack. If there is enough, the prologue proceeds to the function body. Otherwise, it invokes the runtime library to obtain a new stacklet first. Both GCC and Clang depend on the runtime library provided within libgcc [9, 18]. To compile code with the segmented stack, one only needs to set the `-fsplit-stack` option flag.

Below we explain the implementation of segmented stack in libgcc. Figure 1 shows how the stack, represented by the linked list of stacklets, changes.

Figure 1(a) shows that the stack starts with three stacklets with the current stacklet at the end of the linked list. When a function is called, the function prologue calculates the new stack frame boundary by subtracting the required stack frame size, which is known at compile time by the compiler, from the current stack pointer. It then decides whether to request a new stacklet or not by comparing the new stack frame boundary with the current stacklet boundary stored in the thread local storage (TLS).

If the new stack frame boundary is within the current stacklet boundary, the prologue follows the *Fast Path*: it continues to the function body. The function's stack frame lives in the current stacklet, as shown in Figure 1(b)'s striped frame. When the function returns, it cleans its stack frame by simply moving the stack pointer: the stack transits from Figure 1(b) to (a).

Otherwise, the prologue follows the *Slow Path*: it invokes the runtime library, which allocates a new chunk of memory. The allocation must be of at least a minimum size and comply to a granularity, both of which are the memory page size (4 KiB) on x86_64. The library then copies the arguments from the previous stacklet to the new one, sets the stack pointer to the new stacklet, and updates the stacklet boundary record in the TLS. The library finally *calls* into the user function body. Figure 1(c) illustrates the resulting linked list of stacklets, with the striped frame representing the function's stack frame. When the function returns, it will *return*

to the runtime library, which will free the current stacklet and adjust everything back to the previous stacklet. The stack would transit back to Figure 1(a).

The runtime library caches freed stacklets to serve subsequent requests, as shown in Figure 1(d). If the request size exceeds the size of the next cached stacklet, all cached ones are returned back to the heap before the runtime library makes a new allocation, transitioning to Figure 1(e) where the striped frame triggers the new allocation.

4 WHY NOT SEGMENTED STACK

In this section, we present a set of previously unknown problems with the segmented stack that may have contributed to its poor memory efficiency and performance reported in the past. We also confirm the negative impact of *hot-split*, a well-known problem for the segmented stack [12, 13, 24]. Later we will show that these problems either largely vanish or can be solved on microcontroller-based embedded systems.

Internal Fragmentation due to ABI Compatibility. The segmented stack intends to improve memory efficiency by allocating memory for the stack dynamically. On Linux, however, each stacklet must reserve considerable space to conform to System V ABI and to cooperate with Linux dynamic linker and signal handlers.

System V ABI for x86_64 [26] allows a program to use the immediate 128 bytes beyond the stack top, called the red-zone, without adjusting the stack pointer, saving two instructions in leaf functions. To conform to this ABI, every stacklet must reserve 128 bytes for the red-zone.

Dynamic linking requires every stacklet to reserve even more free space. A function call may invoke the dynamic linker for runtime symbol resolution, but the dynamic linker is not aware of the segmented stack so is unable to request a new stacklet. Thus, every stacklet's reserved space must also be large enough to accommodate the need of the dynamic linker. To the knowledge of the authors, the stack space needed by the dynamic linker is not documented.

Signal handling in Linux similarly requires free space beyond the stack top. The kernel pushes the thread execution context onto the user stack by default before returning control to the userspace signal handler. The context can take up to 3174 bytes with the presence of AVX512 SIMD registers.

Considering all above, libgcc reserves 3584 bytes in each stacklet on x86_64 Linux [17]. The fragmentation ratio can be as high as 87.5% when 4 KiB stacklet is used.

Poor Compatibility with Legacy Code. If code compiled with the segmented stack must be linked with conventional code with the contiguous stack, the compiler tool-chain must deal with the fact that the conventional code cannot allocate a new stacklet. Before the control is transferred from the code with segmented stack to the conventional code, the linker must ensure that the stacklet in use is sufficiently large. Specifically, when the linker detects that a function compiled with the segmented stack calls into conventional code, it rewrites the function prologue to ensure large free space in the stacklet, leveraging the knowledge that the prologue follows a specific instruction pattern. We note that the linker cannot add a prologue to the conventional code because doing so may break PC-relative instructions.

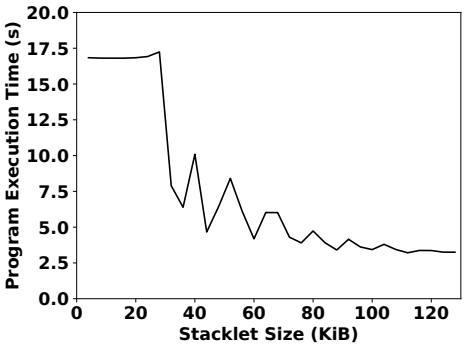

Figure 2: The hot-split problem. The program execution time has a general trend down as the stacklet size increases, but rebounds when hot-split is triggered.

This approach, however, faces four problems. (*i*) First, the default GCC linker ld.bfd does not rewrite the prologue although newer ld.gold and ld.lld do. With it, overflows are likely because stacklets are usually small. (*ii*) Second, the approach is not foolproof because a conventional function using significant stack space can still overflow the stacklet. Even worse, a stacklet overflow is more dangerous than a contiguous stack overflow, because no guard page sits at the stacklet boundary. It can happen silently, forming a stack clash [23]. (*iii*) Moreover, the approach reduces memory efficiency because the stacklet before the control transfer to the conventional code must have abundant free space, e.g., larger than 64 KiB in GCC [17], whose effect is further amplified by stacklet caching. (*iv*) Finally, it may break legacy code that requires an executable stack [10]. libgcc marks allocated stacklets as always non-executable due to safety concerns.

Sub-optimal libgcc *Implementation.* We discovered that the current implementation of the segmented stack in libgcc for x86 and x86_64 intentionally introduces a missed return address prediction [16] that may significantly degrade runtime performance. The code path is triggered when code compiled with the segmented stack is linked with conventional code with the contiguous stack. We improved the implementation for x86_64, eliminating the missed return address prediction by leveraging `%r10` and `%r11` as scratch registers. According to our measurement of a JSON parser program with frequent calls to C++ standard libraries without the segmented stack, this optimization reduces the performance overhead of the segmented stack from 113% to 24%.

Hot-Split. Also known as stack-thrashing, it occurs when a leaf function call in a tight loop follows the slow path. We reproduce hot-split with an artificial workload written in C++ where the program randomly allocates some stack frames and then runs a tight loop calling a leaf function. We vary the stacklet size from 4 KiB to 128 KiB by compiling the code with patched GCC, enforcing the size in `libgcc/config/i386/morestack.S`. Figure 2 shows that the program execution time fluctuates with a general trend down as the stacklet size increases. When hot-split is triggered, as when the

stacklet size is 40 KiB, the program runs about 60% slower than that with 36 KiB.

5 WHY SEGMENTED STACK ON MCUS

We next show that many of the problems with the segmented stack vanishes on embedded systems. More importantly, the segmented stack would bring important benefits to such systems in addition to memory efficiency.

5.1 Disadvantages vanish

Less fragmentation in stacklet. It is usually unnecessary to reserve large free space in stacklets for microcontroller-based systems, unlike on Linux. ABIs designed for MCUs, e.g., ARM EABI [3], do not employ red-zone. Executable images for microcontrollers are usually fully linked at compile time, obviating dynamic linking. User-provided callback functions can run with a separate stacklet because the kernel is aware of the use of the segmented stack, unlike Linux. Thus, there is no need for reserving large free space in a stacklet on microcontrollers.

However, each stacklet still must include an unavoidable overhead to save the previous stack pointer and the pointer to its predecessor stacklet. Extra reserved space might be necessary depending on the specific hardware architecture and segmented stack implementation. On ARM Cortex-M3/4, register `%r4` and `%r5` are used to pass the requested stack frame size and stack argument size to the runtime library and thus must be pushed into the stacklet reserved space. `%lr` must also be saved to preserve the return address. That is in total 20 bytes.

Additional reserved space might be required to handle interrupts. ARM Cortex-M pushes the interrupt frame to the user stack before invoking an interrupt handler, thus extra 32 bytes are needed. On the other hand, x86_64 pushes the interrupt frame to the kernel stack, while ARM Cortex-A and Cortex-R depend on banked registers to preserve the interrupt site, all obviating additional reserved space in stacklet.

Tighter control of code base. Microcontroller developers often have full source-level control of the code base, where the executable image is compiled completely from source code and not linked against pre-compiled binary libraries. They can enable the segmented stack for every function compiled, thus eliminate the need to cooperate with code compiled with the contiguous stack. In the case where a peripheral vendor supplies binary libraries, they can provide a version compiled with the segmented stack.

Speed loss more acceptable. Microcontroller-based systems are typically I/O bound and care more about predictability rather than absolute speed. Additionally, the energy cost due to speed loss from the segmented stack can be potentially compensated by energy saving from using a smaller memory. We do note that the segmented stack can complicate worse-case execution time (WCET) estimation because it involves dynamic memory allocation. Accurate WCET estimation is important for real-time scheduling so that an event can be handled within a deadline. We believe that this problem can be relieved with real-time dynamic allocation designs such as TLSF [19].

5.2 Benefits emerge

Memory Safety by Compiler. Most microcontrollers are not equipped with memory management unit (MMU); instead, tasks share the physical address space rather than private virtual address spaces. Stack overflows in one task can easily cause silent data corruption in another task or the kernel.

A programmer can use the memory protection unit (MPU) to solve the problem by specifying access permissions to memory segments for a task. The MPU will trap a stack overflow if it leads the task to access a disallowed memory segment. This approach, however, either suffers from poor memory efficiency or incurs higher runtime overhead for functions with a large stack frame. One implementation, as adopted by Tock OS [15], is to place the down-growing stack at the lowest address of all allowed memory segments of a task. It requires a contiguous chunk of memory to be allocated for a task upon creation, which is less efficient than dynamic allocation supported by our approach.

Another implementation, as adopted by some compilers, is to add stack probing instructions at compile time, which can incur higher runtime overhead than the segmented stack prologue. These instructions probe the stack frame, i.e., writing locations in the frame separated by a constant interval, before the function body runs. Because the smallest segment is 32 bytes on ARM Cortex [2], the probing is required when the stack frame is larger than 32 bytes. More importantly, there will be one probing for each every 32 bytes of the stack frame. That is, if the stack frame has N bytes, there will be $\lfloor N/32 \rfloor$ probings. On ARM Cortex-M3/4, 2 to 3 instructions are needed for each probing. In contrast, the segmented stack prologue is per-stack frame. In the fast path, it takes 7 instructions, of which 1 to 3 can be optimized out for common cases. As such, the segmented stack has better performance in guarding overflows for functions with large stack frames (>96 bytes). During our investigation, we discover that ARM GNU (arm-none-eabi-gcc) erroneously configures the probing interval as 4 KiB, rendering the probing almost useless for embedded software. This is also noted by others [25]. We also find that LLVM silently ignores the stack probing option `-fstack-check` for ARM embedded. As a result, the popularity of compiler-based stack probing is questionable in embedded software where stack overflows remain a hard, practical problem.

Even worse, many low-end microcontrollers are not even equipped with an MPU. For them, compiler enforced memory safety provides a promising alternative approach. For this, the segmented stack complements language-level memory safety, like safe Rust, by preventing overflows by the compiler generated prologue.

Hardware and Energy Efficiency. The segmented stack helps to reduce the size of SRAM required on chip for a given set of tasks because SRAM can be shared by stacks temporally. Smaller SRAM is beneficial in two ways. First, hardware can be made cheaper. The savings can be nontrivial because embedded systems are usually produced in large numbers. Second, power consumption is reduced. Since SRAM must be powered to retain data, the smaller its size, the less power it consumes. As we show in §2, when the contiguous stack is used, the system wastes energy via statically allocated but unused stack space. The segmented stack would substantially reduce such waste.

Figure 3: System SRAM address space layout. Heap includes segmented stacks and dynamic memory. Unlike Linux where each thread has its own kernel stack, embedded kernels usually employ a single kernel stack for all tasks.

6 IMPLEMENTATION

We next describe the first known implementation of the segmented stack for microcontroller and a novel technique to alleviate hot-split. Our implementation aims at ARM Cortex-M4 and supports C and Rust by patching LLVM ARM backend (18 lines). We implement the segmented stack runtime library as supervisor call (`SVC`) handlers in C (500 lines). We also patch the Rust frontend (rustc) to include a new function attribute `no_split_stack` to suppress the generation of segmented stack prologue for special functions (10 lines), whereas the C frontend (clang) already has this attribute.

Below we highlight some implementation details that render the segmented stack practical on microcontrollers.

Guard against stack overflows completely. We prevent stack overflows for both user code running with the segmented stack and kernel code running with the contiguous stack. The kernel code that allocates and switches stacklets cannot run with the segmented stack. To prevent the kernel code from overflowing the contiguous stack, we use a single kernel stack and place it at the lower border of the SRAM region, below which the address is mapped to Flash thus not writable, so that we can trap kernel stack overflows with a hardfault, inspired by Drone OS [27] and Tock OS [15]. User tasks run with segmented stack supported by the system heap, inherently immune to stack overflows. The system address space layout is shown in Figure 3.

Retrieve stacklet boundary swiftly. We employ the read-write position independence (RWPI) relocation model to support fast retrieval of the stacklet boundary, since every function prologue reads and compares to this value. RWPI reserves register `%r9` to point to the start of the `.data` section. We place the stacklet boundary with other metadata at a fixed negative offset from `%r9` so that reading it takes only a single `ldr` instruction.

Alleviate hot-split memory-efficiently. The runtime library starts with each function stack frame having its own stacklet of exactly right size, and then merges stacklets to the extent just enough to prevent hot-split, adapting to the program's runtime behavior as follows. The key insight is that the detection of a hot-split event can prevent future ones for periodical tasks. If the runtime library detects that a function `F` directly or indirectly calls `G` and `G` experiences hot-split because the stacklet used by `F` cannot accommodate `G`, it calculates the difference between the remaining free space in the stacklet for `F` and `G`'s requested stack frame size, denoted as δ. Later when `F` is invoked another time, the runtime library allocates its stacklet with additional δ bytes to prevent `G` from hot-splitting again.

7 EVALUATION

We evaluate our implementation on the STM32F407VG microcontroller, which features an ARM Cortex-M4 core running at 168 MHz, 1 MiB Flash, and 128 KiB SRAM. We demonstrate that the segmented stack can achieve decent performance along other benefits it brings.

We port CoreMark [4], an industry-standard benchmark that measures the performance of embedded CPU, to our microcontroller. The benchmark contains three types of workload: matrix multiplication, linked list manipulation, and state machine transition. Matrix multiplication primarily works with loops while function calls are rare. On the other hand, function calls are abundant in the other two types. Each benchmark iteration goes through the three types. The performance is reported in the number of iterations per second.

Performance. We show CoreMark performance under 5 configurations in Table 2. Contiguous stack with static relocation is the conventional setup and delivers the best runtime performance. RWPI relocation introduces an extra layer of indirection when addressing global or static data, but its overhead is negligible. Segmented stack always works with RWPI relocation. The naive version simply allocates a stacklet for every function's stack frame. Prior work also called it "linked frames" [7]. It performs the worst because every function call goes through the slow path unless a function requires no stack frame. The optimized segmented stack implementation incorporates the hot-split alleviation algorithm, which boosts its performance to be comparable with contiguous stack. The overflow-check-only version allocates a huge initial stacklet prior to running the workload, using the prologue only to guard against overflows but not intending to allocate a new chunk. The overhead is incurred by only the function prologue.

Memory efficiency. The segmented stack increases the maximum stack usage by a task because it adds metadata and reserves extra space in each stacklet to support existing call convention and interrupt mechanism. Our implementation introduces a 60-byte overhead per stacklet: 36 for the interrupt frame and its alignment padding, 12 for the stacklet metadata, and 12 for saving spilled callee-saved registers. Thus, the maximum memory usage of the segmented stack is greater than that of the contiguous stack, as shown in Table 3 (Optimized).

This overhead, however, can be substantially reduced if two changes to the ABI are allowed (Ideal): (*i*) `%r4` and `%r5` are made caller-saved; and (*ii*) interrupt frames are saved to the kernel stack, which is actually the case with x86_64, or banked in register, as in ARM Cortex-A or Cortex-R but unfortunately not Cortex-M. With this ideal ABI, the memory overhead of each stacklet can be reduced to 12 bytes: 8 for stacklet metadata and 4 for spilled `%lr` register. We calculate the numbers for the hypothetical cases in the right columns, when we make `%r4` and `%r5` caller-saved, and ideally when the interrupt frame is also moved out of the user stack.

More importantly, when tasks are bursty or scheduled properly such that they do not reach the maximum stack usage at the same time, the overall memory usage by all stacks can still be much lower than that by statically allocated stacks. For example, 500 bytes is enough to hold segmented stacks for the four Crazyflie tasks in

Contiguous Stack		Segmented Stack		
Static	RWPI	Naive	Optimized	OF Check Only
379	379	147	357	361

Table 2: CoreMark Iterations per Second

Contiguous Stack	Segmented Stack		
Static	Optimized	Caller Save `%r4` `%r5`	Ideal
468 B	920 B	856 B	536 B

Table 3: CoreMark Stack Maximum Size

Table 1, almost 20x smaller than the current usage with contiguous stacks. This suggests an interesting research opportunity into joint task scheduling and memory management.

8 RELATED WORKS

Stack Memory Safety. Stack overflows can be caught by either hardware or software-based approaches. Some hardware based approaches employ the MPU to catch stack overflows, as in Tock OS [15] and the MPU port of FreeRTOS [8]. Others, e.g., Drone OS [27], uses a single stack for all tasks, places it at the lower end of the SRAM region, and lets it grow downwards.

Software-based approaches can be further categorized into unreliable and reliable ones. Unreliable ones allow some stack overflows to go undetected. FreeRTOS without MPU can optionally perform unreliable overflow detection by examining the stack pointer of the running task at each system tick. An overflow goes undetected if its duration does not span across a system tick. Reliable software-based approaches employ instrumented code to check the stack pointer at runtime. The instrumentation can be done either by source-to-source code transformation as in Capriccio [28] or through compiler as in MTSS [21] and our work. To reduce the runtime overhead, Capriccio and MTSS analyze the call graph statically to eliminate the instrumented code in some functions and merge the check into their caller functions. In contrast, our approach learns the function call pattern at runtime, in order to reduce the runtime overhead for functions experiencing hot-split. It can support dynamically loaded code, whereas those based on static analysis cannot. Moreover, when merging stacklet allocations to alleviate hot-splits, our approach can achieve higher temporal memory efficiency than those static approaches, thanks to runtime information.

Stack Memory Efficiency. Efficient stack memory usage is achieved by allowing multiple tasks to share stack memory. Drone OS employs a single system stack and runs all tasks with it. A higher priority task may preempt the running task and place its stack frame on top of the preempted task's frames. However, the preempted task may only resume after the high priority task yields and relinquishes its stack frames. Capriccio and our approach allocate memory for the stack on-demand from the heap, allowing the most flexible time-multiplexing of memory among stacks and dynamic memory. MTSS allows a stack to overflow into unused space of another stack, which essentially allows all stacks to share memory.

Other works advocate eliminating the stack in the thread. Protothreads [6] introduces the notion of local continuation so that each protothread needs not a separate stack but shares a single one. However, their implementation based on C macros prohibits the normal use of stack variables because they cannot restore their values across blocking points. UnStacked C [20] performs static call-graph analysis and saves function activation records in global static buffers. These approaches trade global static data for saved stack memory, but since the memory for global static data are not time-multiplexed, they are less memory efficient than Capriccio, MTSS, and our work.

Segmented Stack Overhead. Farvardin and Reppy [7] recently demonstrated that the performance of the segmented stack can be close to that of the contiguous stack. They implemented a parallel and concurrent subset of Standard ML, using different ways of implementing the call stack. They reported the performance of the segmented stack as within 10% of that of the contiguous stack most of the time and even occasionally better, which matches what we observe about our embedded Rust implementation, despite the vastly different system contexts.

9 RESEARCH OPPORTUNITIES

Resource-aware scheduling. Dynamic memory allocation may cause runtime memory exhaustion, which a robust system should handle gracefully or prevent if possible. Tasks should be blocked on allocation requests if memory has been exhausted. The possibility of deadlock can be reduced by resource-aware scheduling, e.g., selecting the task that is likely to release memory very soon while under memory pressure. Capriccio [28], a cooperatively scheduled thread library, implements resource-aware scheduling by learning at runtime a state machine for a thread with each state representing a blocking point for the thread and annotated with its resource usage. It updates the probability of each state transition as the thread runs. This technique can be readily incorporated into preemptive multi-task embedded software with the segmented stack: a state can represent a point where memory is allocated or freed. Using this state machine, the scheduler can know which tasks will soon release or acquire memory by tracking their states. It might also be possible to prevent memory exhaustion by improving Banker's algorithm [5], where the tasks are scheduled to avoid any resource deadlock.

Compiler optimization of stack. Machine code generated by compilers, e.g., GCC and Clang, keeps dead data in the stack frame rather than immediately compacts the stack frame when variables go out of their scopes. We believe the compilers choose to trade memory efficiency for runtime performance because updating the stack pointer as soon as a variable is dead will slow down the program. However, with the segmented stack, having compact stack frames will allow more tasks to run concurrently. Thus, there is a trade-off between the task runtime speed and the maximum number of concurrent tasks.

10 CONCLUDING REMARKS

We demonstrate the potential of the segmented stack to guarantee memory safety solely by the compiler and to improve memory efficiency by time-multiplexing the memory used as stack. We report an implementation of the segmented stack for embedded C and Rust, with runtime performance close to that of the contiguous stack. The current implementation has noticeable memory overhead, which can be greatly reduced by a re-design of ABI. Nevertheless, we believe the segmented stack can still significantly save memory under workloads with multiple bursty tasks.

ACKNOWLEDGMENTS

This work is supported in part by NSF Award #2130257. The authors are grateful for the constructive feedback from the reviewers and paper shepherd Dr. Pat Pannuto.

REFERENCES

[1] Ambiq. 2020. *Apollo4 Blue.* https://ambiq.com/apollo4-blue/
[2] ARM. 2016. *ARMv7-M Architecture Reference Manual.* https://developer.arm.com/documentation/ddi0403/latest
[3] ARM. 2019. *Application Binary Interface for the Arm Architecture - The Base Standard - ABI 2019Q4 documentation.* https://developer.arm.com/documentation/ihi0036/d
[4] CoreMark. 2009. *Embedded Microprocessor Benchmark Consortium.* https://www.eembc.org/coremark/
[5] Edsger W Dijkstra. 1964. Een algorithme ter voorkoming van de dodelijke omarming. *http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD108.PDF* (1964).
[6] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. 2006. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems.
[7] Kavon Farvardin and John Reppy. 2020. From folklore to fact: comparing implementations of stacks and continuations.
[8] FreeRTOS. 2022. https://www.freertos.org.
[9] GCC. 2022. *Segmented Stack Implementation.* https://github.com/gcc-mirror/gcc/blob/0c7bce0ac184c057bacad9c8e615ce82923835fd/libgcc/config/i386/morestack.S
[10] GCC. 2022. *Support for nested functions.* https://gcc.gnu.org/onlinedocs/gccint/Trampolines.html
[11] Wojciech Giernacki, Mateusz Skwierczyński, Wojciech Witwicki, Paweł Wroński, and Piotr Kozierski. 2017. Crazyflie 2.0 quadrotor as a platform for research and education in robotics and control engineering. In *Proc. IEEE MMAR.*
[12] Golang. 2014. *Switch to contiguous stacks.* https://docs.google.com/document/d/1wAaf1rYoM4S4gtnPh0zOlGzWtrZFQ5suE8qr2sD8uWQ/pub
[13] Golang. 2014. *Switch to contiguous stacks.* https://agis.io/post/contiguous-stacks-golang/
[14] Robert Hieb, R Kent Dybvig, and Carl Bruggeman. 1990. Representing control in the presence of first-class continuations. *ACM SIGPLAN Notices* (1990).
[15] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64KB computer safely and efficiently.
[16] Libgcc. 2022. *Missed return address prediction in segmented stack implementation.* https://github.com/gcc-mirror/gcc/blob/releases/gcc-12.2.0/libgcc/config/i386/morestack.S#L216
[17] Libgcc. 2022. *Reserve 64KiB in stacklet for conventional code.* https://github.com/gcc-mirror/gcc/blob/releases/gcc-12.2.0/libgcc/config/i386/morestack.S#L83
[18] LLVM. 2022. *Segmented Stack Implementation.* https://github.com/llvm/llvm-project/blob/8f2ec974d1cf0ecde8b1b047db2e1bf0cb3a6c7f/llvm/lib/Target/X86/X86FrameLowering.cpp#L3089
[19] Miguel Masmano, Ismael Ripoll, Alfons Crespo, and Jorge Real. 2004. TLSF: A new dynamic memory allocator for real-time systems. In *Proc. IEEE Euromicro Conf. Real-Time Systems.*
[20] William P McCartney and Nigamanth Sridhar. 2011. Stackless preemptive multithreading for TinyOS. In *Proc. IEEE DCOSS.*
[21] Bhuvan Middha, Matthew Simpson, and Rajeev Barua. 2008. MTSS: Multitask stack sharing for embedded systems. *ACM Trans. Embedded Computing Systems (TECS)* (2008).
[22] Nordic. 2015. *nRF52 Series.* https://infocenter.nordicsemi.com/topic/struct_nrf52/struct/nrf52.html
[23] Qualys Security Advisory. 2017. *The Stack Clash.* https://www.qualys.com/2017/06/19/stack-clash/stack-clash.txt
[24] Rust. 2013. *Abandoning segmented stacks.* https://mail.mozilla.org/pipermail/rust-dev/2013-November/006314.html
[25] Stratify OS. 2022. *Issue of -fstack-check.* https://github.com/StratifyLabs/StratifyOS/issues/173
[26] System V. 2014. *Application Binary Interface: AMD64 Architecture Processor Supplement.* https://www.uclibc.org/docs/psABI-x86_64.pdf
[27] Valentine Valyaeff. 2017. *Drone OS.* https://www.drone-os.com/.
[28] Rob Von Behren, Jeremy Condit, Feng Zhou, George C Necula, and Eric Brewer. 2003. Capriccio: scalable threads for internet services. *ACM SIGOPS Operating Systems Review* (2003).