JFP **31**, e19, 51 pages, 2021. © CSIRO and The Author(s), 2021. Published by Cambridge University Press.
doi:10.1017/S0956796821000174

Real-time MLton: A Standard ML runtime for real-time functional programs

BHARGAV SHIVKUMAR, JEFFREY MURPHY
AND LUKASZ ZIAREK
University at Buffalo, Buffalo, NY 14260, USA
(*e-mails:* bhargavs@buffalo.edu, jcmurphy@buffalo.edu, lziarek@buffalo.edu)

Abstract

There is a growing interest in leveraging functional programming languages in real-time and embedded contexts. Functional languages are appealing as many are strictly typed, amenable to formal methods, have limited mutation, and have simple but powerful concurrency control mechanisms. Although there have been many recent proposals for specialized domain-specific languages for embedded and real-time systems, there has been relatively little progress on adapting more general purpose functional languages for programming embedded and real-time systems. In this paper, we present our current work on leveraging Standard ML (SML) in the embedded and real-time domains. Specifically, we detail our experiences in modifying MLton, a whole-program optimizing compiler for SML, for use in such contexts. We focus primarily on the language runtime, reworking the threading subsystem, object model, and garbage collector. We provide preliminary results over a radar-based aircraft collision detector ported to SML.

1 Introduction

With the renewed popularity of functional programming, practitioners have begun re-examining functional programming languages as an alternative for programming embedded and real-time applications (Hammond, 2001; Wan *et al.*, 2001; Hammond, 2003; Li *et al.*, 2016). Recent advances in program verification (Audebaud & Paulin-Mohring, 2009; Kumar *et al.*, 2014) and formal methods (López *et al.*, 2002; Arts *et al.*, 2004) make functional programming languages appealing, as embedded and real-time systems have more stringent correctness criteria. Correctness is not based solely on computed results (logic) but also the predictability of execution (timing). Computing the correct result late is as serious an error as computing the wrong result.

Functional languages can provide a type-safe real-time implementation that, by nature of the language structure, prevents common errors such as buffer underflow/overflow and null pointer dereferencing from being expressed. Programmers can thus produce higher fidelity code with lower programmer effort (Hughes, 1989). Additionally, constructs like immutability and referential transparency make functional programming languages easier to analyze statically than their object-oriented counter parts, and significantly easier than C. As such, they purport to reduce time and effort from a validation and verification perspective. Since many embedded boards are now multicore, advances in parallel and concurrent programming models and language implementations for functional languages are also appealing as programs can avoid the need for locks if shared state is immutable in concurrent/parallel programs.

However, there are many challenges that need to be addressed prior to being able to leverage a functional language for developing a real-time system. Some of these challenges, according to Hammond (2003), are that functional languages must exhibit *deterministic* behavior under resource constraints, have runtimes that can be bounded in space and time, provide predictable and low latency asynchronous responsiveness, as well as provide a robust concurrency model. In our prior work, we surveyed the current state of the art in functional languages and their suitability for developing real-time systems (Murphy *et al.*, 2019). We assessed metrics like the predictability of the language runtime, threading and concurrency support, as well as support for expressing real-time constraints in the program. We observed that all of the languages exhibited unpredictable behavior once competition for resources was introduced, specifically in their runtime architectures. The major challenge in providing a predictable language runtime performance for the languages surveyed was their lack of a real-time garbage collection (RTGC) mechanism (predictable memory management).

In this paper, we introduce a predictable language runtime for Standard ML (SML) (Milner *et al.*, 1997) capable of executing real-time applications. We use MLton (2012), a whole-program optimizing compiler for SML, as a base to implement the constructs necessary for using SML in an embedded and real-time context. We discuss adding a new chunked object model for predictable allocation and nonmoving real-time garbage collector with a reservation mechanism. We leverage our previous experience with Multi-MLton (Sivaramakrishnan *et al.*, 2014) and the Fiji real-time virtual machine (Pizlo *et al.*, 2010*a*) in guiding our modifications to MLton. Our changes sit below the MLton library level, providing building blocks to explore new programming models. Our system supports running programs on RT-Linux, a real-time operating system (RTOS). On account of being a real-time version of the MLton compiler, we call our work RTMLton – short for Real-Time MLton. We present performance measurements, indicating the viability of RTMLton, which is publicly available for download at: `https://github.com/UBMLtonGroup`.

Our original PADL paper (Shivkumar *et al.*, 2020) was an extension of our previous short workshop paper (Li *et al.*, 2016), to which we had added a detailed description of the MLton runtime, the consequences of the design decisions adopted by MLton, and the details of our chunked, concurrent, reservation-based RTGC algorithm. We presented additional benchmarks, including a full evaluation of our system on a radar-based aircraft collision detector (CD_x). This paper is an extension of our PADL paper. Specifically, this version provides additional discussion on MLton's architecture and the consequences of its design decisions on predictability and suitability as an embedded system. We provide a detailed discussion of the stack-based representation, considering common optimizations performed by MLton and the challenges they pose when stacks are noncontiguous due to the RTGC. We add a new section on the implementation of stacklets (noncontiguous stacks) and discuss the challenges involved in moving from a contiguous stack model to the noncontiguous stacklet model. Additional implementation details and discussion are also provided for the threading model, noncontiguous arrays, and the RTGC. We extend the empirical evaluation to include the evaluation of the system with stacklets on the CD_x and additional benchmarks to compare raw performance with MLton. We expand the related work section to discuss other functional languages/domain-specific languages (DSLs) that have potential for use in real-time system development. Last but not least, we have made a conscious effort to keep this paper rich with implementation details to benefit others who would like to follow suit and modify another language to make it amenable to building real-time systems.

1.1 Real-time guarantees

While the use of functional languages to develop real-time systems is our long-term goal, we envision RTMLton as a major step toward achieving that goal. The work described in this paper positions RTMLton as a substrate for building a functional language for real-time systems. We focus on language runtime-specific features that are essential to *predictability*—the threading model and memory management systems. Predictability in the context of a real-time system is the assurance that all individual tasks in the system meet a predetermined deadline.[1] From the language runtime perspective, this would mean we need to achieve *predictable memory management* by bounding the time overheads of memory management. RTMLton aims to improve the MLton compiler to ensure a predictable language runtime and to this effect the current version of RTMLton provides the following guarantees:

- Predictable allocation–we present an object model and Garbage Collection (GC) strategy that ensures a bound on the worst-case allocation costs and as a result predictable performance as described in Section 4. Our reservation mechanism guarantees allocation by reserving the memory before it is allocated and eliminates any pause due to insufficient space at the point of allocation. Moreover, the chunked memory model ensures fragmentation never occurs thereby eliminating GC work involved in de-fragmenting the heap.
- Shorter GC pause times—we reduce the amount of time an application can be paused for GC purposes to *O(size of its stack)* from *O(size of entire heap)*. Real-time systems perform a worst-case execution time (WCET) analysis to ascertain the maximum execution time for every task in the application. This analysis also needs to take into account the worst case for the GC pauses in order arrive at an accurate WCET for the application. A shorter pause time would translate to a lower WCET which can put the application (or any task of the application) within range of the deadline. We present incremental strategies to further optimize such pause times in Section 3.3.6.

In order to keep the discussion about how to prime RTMLton as a general purpose platform for specifying real-time systems, we make some assumptions that we state here. First, we assume that there exists a system specification and a WCET. RTMLton compiles SML code to highly optimized C code, and a WCET calculation can be performed on this generated C code using tools used for this purpose (Ballabriga *et al.*, 2010; Lisper,

[1] Deadlines are determined from system requirements.

2014; RapiTime, 2021). Secondly, we assume that a schedulability analysis[2] exists which ensures that the GC is scheduled often enough to free up space for tasks to run. It is more of a design choice to think about whether the GC needs to run during slack time (when no other tasks are running), as a high priority task which runs periodically, or to perform GC work incrementally whenever an allocation request is satisfied. Kalibera *et al.* (2009*a*) specifies some ways in which such an analysis can be done. For anyone developing with RTMLton, this is the process they would leverage. Such an analysis crucially relies on bounding the worst case of the GC, a process made easier by bounding allocation costs (and hence GC work) and the maximum GC pause time.

Finally, RTMLton targets newer embedded boards with more memory and not embedded microcontrollers. Keeping that in mind, for this version of RTMLton, we find it more important to focus on optimizing for a predictable runtime system as opposed to optimizing for memory size. We do, however, provide possible optimizations as part of future work in Section 3.3. RTMLton is currently limited to 32-bit deployments and supports a majority of features supported by MLton. The list of unsupported features is given in Section 4.1.

2 MLton architecture and consequences for embedded and real-time systems

MLton is an open-source, whole-program optimizing SML compiler that generates very efficient executables in both runtime performance and size. MLton has a number of features that are well suited for embedded systems and that make it an interesting target for real-time applications.

2.1 Whole-program optimization

MLton's approach to compilation can be summarized as whole-program optimization (WPO) using a simply typed, first-order intermediate language (IL). This approach is different from other compilers for functional languages and imposes significant constraints on the compiler but yields many optimization opportunities not available with other approaches. There are numerous issues that arise when translating SML into a simply typed IL.

First, how does one represent SML modules and functors, which utilize a rich type system, in a simply typed IL? MLton's answer: defunctorize the program (Reynolds, 1972; Elsman, 1999). This transformation turns an SML program with modules into an equivalent one without modules by duplicating each functor at every application and eliminating structures by renaming variables. Second, how does one represent SML's polymorphic types and polymorphic functions in a simply typed IL? MLton's answer: monomorphise the program (Tolmach & Oliva, 1993). This transformation eliminates polymorphism from an SML program by duplicating each polymorphic datatype and function at every type at which it is instantiated. Third, how does one represent SML's higher-order functions in a first-order IL? MLton's answer: defunctionalize the program. This transformation replaces higher-order functions with data structures to represent them and first-order functions to

[2] All the tasks in a real-time system are analyzed to see if they can be scheduled in a way such that none of the tasks miss their deadlines. Typically, the RTGC is included as a separate task in this analysis.

apply them; the resulting IL is static single assignment (SSA) form. Because each of the above transformations requires matching a functor, function definition, or type definition with all possible uses, MLton must be a whole-program compiler.

MLton's whole-program compilation strategy has a number of implications. Most importantly, MLton's use of defunctorization (Reynolds, 1972; Elsman, 1999) means that the placement of code in modules has no effect on performance. In fact, it has no effect on the generated code whatsoever. Modules are purely for the benefit of the programmer in structuring code. Also, because MLton duplicates functors at each use, no runtime penalty is incurred for abstracting a module into a functor. The benefits of monomorphisation are similar. Thus, with MLton, a programmer does not suffer the time and space penalties from an extra level of indirection in a list of doubles just because the compiler needs a uniform representation of lists. In MLton, whole-program control flow analysis based on Shivers (1988) is employed early in the compilation process, immediately after defunctorization and monomorphisation, and well before any serious code motion or representation decisions are undertaken. Information computed by the analysis is used in the defunctionalization pass to introduce dispatches at call sites to the appropriate closure. The possibility of reasoning over the entire program, including libraries, as well as being able to reason about computations precisely via eager evaluation, makes MLton an interesting compiler for real-time exploration.

2.1.1 Consequences for embedded systems

WPO is important for resource-constrained embedded systems, as aggressive optimization can improve resource usage, code size and runtime performance. For example, aggressive inlining coupled with dead code elimination can result in a smaller code footprint. As a result, MLton produces executables that are 50% smaller than Standard ML of New Jersey (SML/NJ) (Appel & MacQueen, 1987). MLton is able to achieve both a smaller footprint and good performance in comparison to SML/NJ thanks to its WPO strategy (MLton performance, 2012). This is important for an embedded system as the entire application along with all supporting software, including the operating system and libraries, will be packaged together into a single bootable executable. Many embedded systems offer limited storage for this image and achieving a small footprint is necessary.

2.1.2 Consequences for real-time systems

WPO is even more crucial for real-time platforms. For such deployments, predictability is paramount and being able to reason about the WCET for a given piece of code is highly valuable. WCET is used in an offline schedulability analysis that asserts that all tasks in the system will meet their deadlines. For many systems, static WCET gives an over approximation of the runtime for a given piece of code or task. If this over approximation exceeds the deadline target for the piece of code being analyzed, the real-time system cannot be scheduled. MLton's WPO approach and aggressive optimization makes it an interesting target for implementing WCET in a functional language, though we leave this to future work. While DSLs that allow for the specification of timing constraints exist (Timber Language, 2008; Hawkins, 2010), we are not aware of a WPO compiler for any functional language that targets real-time applications. A feature that is complementary to WPO is eager evaluation.

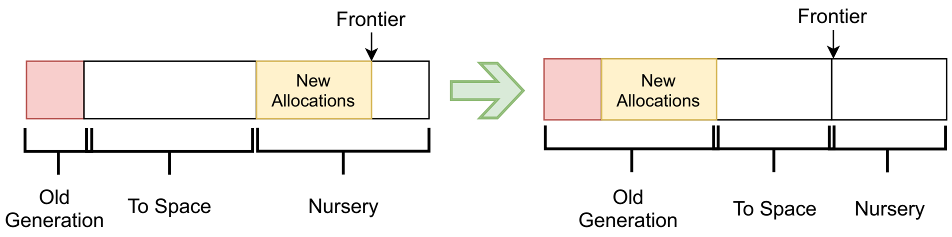

Fig. 1. MLton heap layout and minor GC strategy. New objects are allocated in the *nursery* and moved to *tospace* by the Cheney copy GC.

Whereas WPO involves analyzing the impact and relevance of statements on the overall correctness of a program at compile time, eager evaluation ensures that statements and parameters are evaluated fully and predictably at runtime. SML is an eagerly evaluated language. This is an important attribute when reasoning about timing critical systems because if we cannot predict which parameters and functions will be evaluated at runtime, we will be unable to easily predict if a program can satisfy its timing constraints.

2.2 GC architecture

MLton allocates all SML objects in a contiguous heap which is split into different sections as shown in Figure 1. It adopts a hybrid garbage collector that uses runtime memory utilization information to decide the strategy it needs to use for collection. All objects are initially allocated in the nursery section of the heap in bump pointer fashion, incrementing a pointer by the amount of space needed for the objects—until the nursery runs out of space or a *GC safepoint* is reached (discussed below), upon which the garbage collector is called. If the ratio of bytes live to nursery size is greater than a predetermined nursery ratio, the runtime uses a minor Cheney copy GC (Cheney, 1970). A minor GC (Figure 1) copies objects from nursery to a tospace, which starts at the end of the *oldgeneration*—by appending the objects to the end of the oldgeneration thus increasing the oldgeneration size and reducing tospace and nursery size. When there is no memory pressure, the tospace is unused and oldgeneration has the objects that have survived a collection. Therefore, the “generational" GC is not triggered until the memory utilization is fairly large, but the garbage collector can still be called for various other tasks like growing the stack. After multiple minor GCs, when the nursery space is exhausted and cannot support new allocations, a major GC is triggered.

Major garbage collection is performed in one of the two strategies, shown in Figure 2. If there is enough space to allocate a new heap, the same size of the current heap, then a Cheney copy GC is performed. In this strategy, the heap is split into two semi-spaces (Fenichel & Yochelson, 1969) and live objects are copied from one to the other during a GC. If there is not enough space for the second semi-space, a mark-compact GC (Jones *et al.*, 2016) is performed. The compaction aids in de-fragmenting the heap as well as freeing up more space. After the mark-compact phase, the GC falls back to a minor GC for subsequent collections, until it again needs to call a major GC.

A garbage collector needs to be invoked at specific places in code in order to ensure that the state of the code is safe for the GC to run. For example, all allocated objects need to be

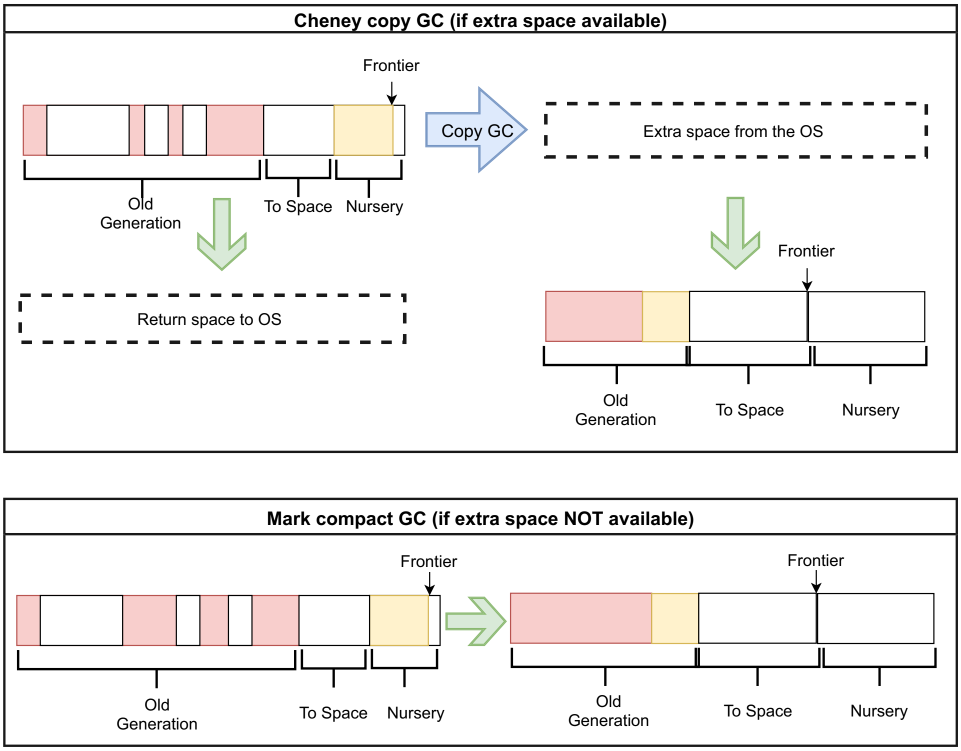

Fig. 2. MLton major GC strategy. A copying collection is performed if extra space (amount equal to the size of current heap) can be procured from the OS and an in-place compaction is performed if no extra space is available.

made reachable (from the program stack) before the GC runs, so that the GC knows these are live objects and must not be collected. Such safe places in the code are called GC safepoints. MLton performs control and data flow analysis that tracks the liveness properties of allocations over various code segments. Additionally, MLton also has access to allocation sizes at compile time. Thus, at compile time, MLton is able to identify safepoints in the code where it can invoke the GC. It relies on its effective analysis techniques to insert as many safepoints as needed to ensure that there are sufficient opportunities for the GC to free space for new allocations.

MLton's GC architecture implements a "Stop The World" (STW) approach to garbage collection, in which all computation threads are paused while the garbage collector runs. Given MLton's single computation model, the heap is prone to corruption if multiple threads access the heap when the GC is copying objects or doing a compaction, hence the need for a STW approach. Pause times vary depending on the strategy being used for collection; it follows that minor GC takes less time than a major GC. There are four kinds of MLton objects: *Normal* (fixed size) objects, *weak* objects, *arrays*, and *stacks*. The arrays and stacks are generally allocated in the oldgeneration as they are more likely to persist longer than the other two kinds of objects. Normal and weak objects are bump pointer allocated in the nursery and then moved to the oldgeneration based on their longevity. Arrays are allocated through a runtime function (`GC_arrayAllocate`) as opposed to bump pointer fashion. Arrays need to be checked for overflow and also have the potential to invoke a GC if contiguous space required to allocate the array is unavailable. Before a call to such a function, MLton needs to take necessary steps to ensure no object is wrongfully collected by the GC. For these reasons, a bump pointer allocation is unsuitable when it comes to allocating arrays. Stacks are allocated in a similar fashion as described in Section 2.3.

2.2.1 Consequences for embedded systems

In an embedded environment, having a robust GC that employs multiple strategies can be a boon. The main advantages include ensuring sufficient space is available and reduced fragmentation as compared to dynamic memory management using C. Embedded systems often utilize multiple cores to parallelize the programs to achieve speedups and the GCs must be capable of utilizing the multiple processors to perform collection faster than one processor could alone do. If the GC were purely sequential and the mutator[3] parallel, it would negate whatever speedup the mutator gains by spreading out across different processors. Some garbage collectors, like that of Marlow *et al.* (2008), have been able to address the problems with STW, allowing for better performance in a multicore environment. MLton inherently works on a single core and thus the GC is incapable of true parallelism. However, a multicore variant of MLton exists, called Multi-MLton. Multi-MLton has a per core heap and a per heap GC and is well suited for parallel programming where tasks are disjoint in the memory they use. Alternatively, to add parallelism to MLton, one can implement a garbage collector capable of working in parallel with the mutator.

2.2.2 Consequences for real-time systems

In a real-time setting, the use of a STW GC is a deal breaker. The cost of performing this GC is directly proportional to the utilization of the heap, and if done during the tasks that have a tight deadline, it could lead to deadline misses. Preempting the GC when it runs out of time could make it real-time compatible, but this will not suffice as collection could then not be guaranteed to always complete. This could be addressed by implementing incremental collection strategies (Nettles & O'Toole, 1993). The multiple GC strategies utilized by MLton further complicates the case by making the maximum pause time more unpredictable, as the strategy used for collection depends on the state of the heap when a collection is triggered. We perform a microbenchmark on MLton by allocating an array of 10 million elements consisting randomly of `NONE` or `SOME` option types. Figure 3 shows how the allocation time of the array objects jump by 2x with no obvious pattern, when the GC changes gears from Cheney copy to compacting the heap and copying objects to tospace. This makes it difficult to put a bound on the GC pause times and to formally prove the system correct, we would have to reason about the maximum GC pauses at all points the GC could run. A tighter bound on the maximum GC pause time can make the difference between whether the application is schedulable or not. Since WCET needs to

[3] Application threads that destructively update the heap, that allocate new objects or rearrange pointers to make objects unreachable, are also called mutators.

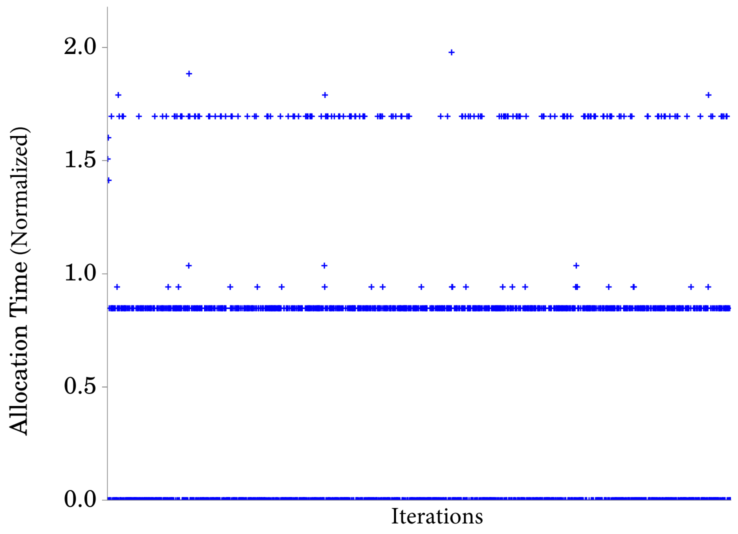

Fig. 3. Unpredictable object allocation in MLton. The Y-axis depicts the normalized allocation time, obtained by dividing the actual allocation time by the mean of all the nonzero allocation times. The benchmark randomly allocates large array objects and measures how the allocation time varies as the GC switches strategies.

assume that the worst case can occur at any allocation, we need to assign the overhead to *all* allocations. We want to reduce this variance and make allocation predictable.

2.3 MLton stacks

MLton leverages two different types of stack frames to represent its runtime stack, ML stack frames for SML code and C stack frames for native calls. ML stack frames are heap-allocated by MLton, while C stack frames are both heap-allocated and also allocated on the system stack (the reason for this double allocation is explained in Section 2.4). Heap-allocating stack frames is a design decision that is not uncommon in compilers that use a Continuation Passing Style (CPS) representation (Hieb *et al.*, 1990), since allocating stack frames on the heap is very efficient in such situations as demonstrated by Appel & Shao (1996). Figure 4 illustrates the MLton heap-allocated stack object.

The `markTop` and `markIndex` fields are used by the mark-compact GC (Jones *et al.*, 2016) for identifying the current pointer on the stack that is being followed. The `reserved` field indicates the number of bytes reserved for the stack that is the maximum size of the stack. The sequence of reserved bytes that follow hold a linear sequence of frames. The `used` field keeps track of how much of the reserved bytes has been actually utilized. During garbage collection, it is crucial to identify the live heap pointers on the stack. The `frameLayouts` structure holds information about the kind of frame (`C_FRAME` or `ML_FRAME`), size of frame, and an array of offsets that give the locations (relative to the bottom of the frame) of the live heap pointers in the frame. The `C_FRAME`s indicate a call to a C function and act only as a marker on the ML stack that this has occurred. The actual C call executes on the system stack. The compiler emits a static array

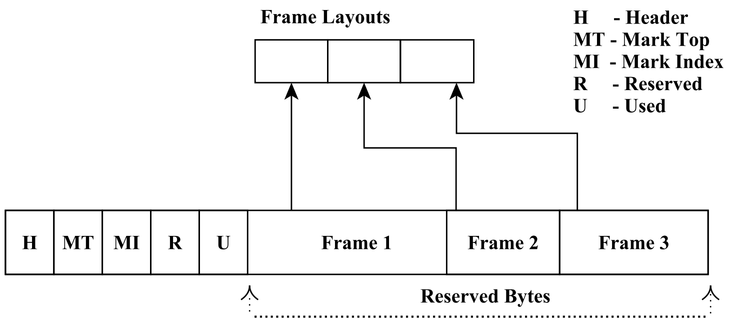

Fig. 4. MLton stack object composed of various fields to manage the stack and a contiguous sequence of *reserved* bytes that house the stack frames. The stack management fields are each of word width and the reserved bytes are increased/decreased by the GC.

of frame layouts for each compiled program, containing the location within the frame where pointers to local variables and function arguments are stored. Although objects themselves are generally allocated at runtime, much can be determined at compile time (e.g. the number of local variables) and space for those pointers can be pre-allocated on the stack frame.

Frame allocation and growth are performed directly by bump pointer allocation on the reserved bytes, but when the stack itself runs out of space it is grown by the GC via an allocation and copy. This stack invariant is checked at the GC safepoints that the compiler emits at precalculated points. Note that an actual GC is performed only when there is not enough free space in the heap to allocate a new stack during a stack grow operation, forcing a compaction to occur. We discuss more about the layout of MLton's stack frames and how they are managed in Section 3.4 to provide context for RTMLton's stacks.

2.3.1 Consequences for embedded systems

Embedded and real-time systems often leverage static allocation of the stack due to memory resource constraints. Maximum stack size, therefore, must be calculated for the system and specified upfront.[4] Precisely calculating the space required for the stack is important for such systems as underestimating leads to runtime fault and overestimating leads to wasting resources. However, in MLton's case, the system stack is infrequently used due to the use of heap-allocated stacks. This minimizes the likelihood of running out of system stack space. Finally, MLton's precise control of the stack layout allows for efficient and predictable memory utilization.

2.3.2 Consequences for real-time systems

Heap-allocated stacks, in MLton's case, have an important consequence for a real-time system: the use of a GC to manage growth and reclamation. We discussed the consequence of MLton's GC earlier in this section and will show how the system can be modified to leverage a real-time GC in Section 3. MLton's implementation of stacks in particular is very tightly bound to the MLton GC. Calling into the GC to allocate a larger stack works well

[4] The real-time OS we are integrating with requires static system stack specification for all threads in the system.

when there is a single thread of control but creates synchronization bottlenecks between threads and the GC when GC is concurrent (a requirement for most real-time GCs). Any synchronization point in a real-time system is costly, as synchronization must have priority inversion avoidance protocols built in to bound the time, a low priority thread can delay a high priority thread. A synchronization point that is shared by all threads is undesirable as it prevents reasoning about the execution of a given thread in isolation from others, or subsets of other threads. As such, a single allocation path for stack allocation is not feasible in a real-time system.

2.4 MLton lightweight concurrency and threads

MLton provides a concurrent, but not parallel, threading model with support for communication between threads. This communication can be either over shared memory or through message passing abstractions. Native MLton threads are green threads that are multiplexed over a single OS-level thread. MLton's thread API is well suited for implementing user-defined schedulers, including preemptive and cooperative threading models as well as Concurrent ML (CML) (Reppy, 1999) and Asynchronous CML (ACML) (Ziarek *et al.*, 2011). Since the low-level model assumes, ultimately, a single thread of control, synchronization in the runtime is minimized. Instead of heavy weight locking mechanisms, MLton will disable interrupts to achieve atomicity for critical regions of code.

MLton compiled programs consist of only a single OS-level thread, over which many green threads are multiplexed. There exists a monolithic global structure called `GC_state` which is used to track the state of the system across all these threads. This structure contains many fields (detailed in Appendix A), but we focus on only those fields relevant to our discussion. There is a set of three process-wide stack pointers, distinct from the system stack and stored in the `GC_state`. These stack pointers point to the stack bottom, top, and limit of the currently running computation. A thread in MLton is therefore a lightweight data structure that represents a paused computation consisting primarily of a pointer to the thread's stack as well as an index into the stack to allow for unwinding in the case of an exception.

Figure 5 depicts a simplified representation of the thread switching code in MLton's C runtime. The `switchToThread` function performs the switch from the currently running computation to the thread passed in as an argument. When a thread is paused, the amount of stack space in use—`used` field in the `Stack` structure—is saved from the current process-wide stack to the thread's stack structure. The other two fields in the `Stack` structure are essentially constants and would only change if the stack were to be moved or grown by the GC. When a thread is resumed, the stack pointers are restored to the process-wide stack fields and computation continues. Thus, thread context switching at its most basic level consists of a pointer swap. This is illustrated in Figure 6 where the application's entry point is depicted as Thread A. In the figure, we observe that when a switch to Thread B is required, Thread A causes the active stack pointer (tracked in `GC_state`) to be changed to point to the stack of Thread B. The same process is repeated when the execution needs to switch back to Thread A. Since the actual switching consists of a call into the C runtime (the code in Figure 5), a context switch from Thread A to B is recorded onto Thread A's stack as a separate C frame. When Thread A is eventually resumed, MLton knows

```
struct Stack {
   size_t  reserved;
   size_t  used;
   /*bottom is a pre-allocated, fixed amount of
    *space corresponding to 'reserved' */
   char    bottom[...];
}

struct Thread {
   /* offset to exception unwind point in stack */
   size_t  exnStack;
   Stack  *stack;
}

void switchToThread (GC_state s, Thread t) {

  /* save stack for the pausing thread */
  s->currentThread->stack->used = s->stackTop - s->stackBottom
  s->currentThread->stack->exnStack = s->exnStack

  /* swap in stack from thread being switched to */
  s->currentThread = t;
  s->exnStack      = t->exnStack;
  s->stackBottom   = t->stack->bottom;
  s->stackTop      = t->stack->bottom + t->stack->used;
  s->stackLimit    = t->stack->bottom + t->stack->reserved;
}
```

Fig. 5. Simplified representation of MLton's thread switching mechanism.

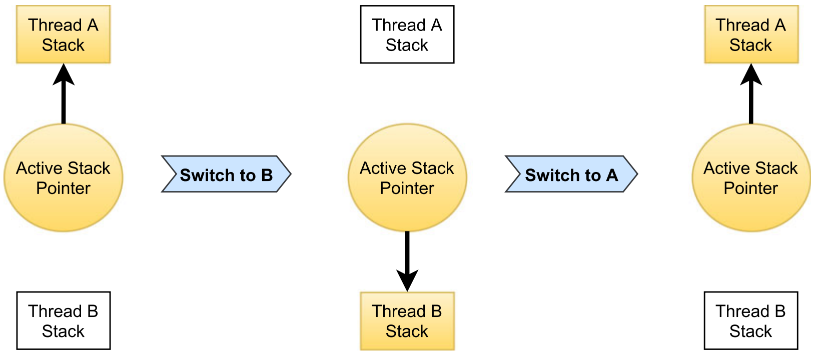

Fig. 6. High-level conceptualization of MLton threads. An active stack pointer keeps track of the currently executing thread.

that the frame to resume computation at is just below that C frame. An advantage of this implementation is that context switches occur rapidly, and SML stack operations, again being distinct from the system stack, are relatively cheap and facilitate deep recursion.

MLton provides a logical ready queue from which the next runnable thread is accessed by the scheduler. This is a regular First In First Out (FIFO) queue with no notion of priority; however, the structure is implicit, relying on continuation chaining and is embedded in the thread switching code[5] itself. What we mean by continuation chaining is that there is

[5] Threads in MLton are one-shot continuations (Bruggeman *et al.*, 1996).

no single data structure that governs threads nor is there an explicit scheduler, and the continuation for the given thread's code includes a call to the next thread thereby switching threads. Note that the code in Figure 5 is called as part of this thread switching process to update the state of the system in the runtime. Threading and concurrency libraries (e.g. CML and ACML) build on top of the MLton threading primitives. Therefore, they introduce their own threading primitives, scheduler policy and structures for managing ready, suspended, and blocked threads. This layering of low-level threading constructs and higher-level scheduling constructs opens up a variety of possibilities with respect to rapidly exploring different scheduling models without needing significant compiler retrofitting.

2.4.1 Consequences for embedded systems

Embedded systems are generally more resource-constrained and frequently have one, or only a few, cores. Such systems benefit from a concurrency model with a low overhead. MLton's lightweight concurrency model is well suited for single-core embedded boards as it presents an optimized and low cost (minimized synchronization, absence of heavy locking mechanisms) solution. MLton specializes in rapid context switching between threads on a shared heap and is able to achieve a lock-free context switch because of the assumption of executing in a single-core environment and disabling interrupts when executing atomic code.

While the majority of embedded boards are still single core, multicore boards are becoming increasingly common. When multiple cores are available, lightweight concurrency results in lower overall system utilization as the additional cores cannot be used to execute tasks. Being designed for a single core, MLton makes many assumptions in the architecture that are not thread safe. For example, as shown in Figure 5, the `GC_state` structure has a `currentThread` field which holds the location of the currently executing (green) thread on the heap. There is no synchronization mechanism when writing to this field as there is only one running thread at a given time. When there are multiple threads, simultaneous access to the `GC_state` structure would lead to obvious concurrency issues. Even if the embedded environment is single core, if multiple OS threads are introduced in MLton, then a preemptive context switch between the OS threads would lead to corruption of the `GC_state` fields and unpredictable behavior. Thus, MLton's single-core model of execution with lightweight threading is of limited utility when an opportunity to parallelize presents itself. Parallel implementations like Multi-MLton might seem better suited for such purposes, but they present overheads (see Section 3.1) whose benefits are realized when deployed on a large number of cores, whereas embedded boards support only a handful of cores.

2.4.2 Consequences for real-time systems

In a real-time setting, lightweight concurrency can minimize the effect of the operating system (e.g. context switches) on the overall timing constraints for a given task. This is desirable as a reduction in the number of components contributing to latency will simplify analyzing whether or not the application can meet its timing constraints. Context switches are included in the execution time of a task when performing schedulability analysis.

MLton's single computation model limits the types of real-time systems that can be defined to those systems that can be expressed as a cyclic executive[6] (Baker & Shaw, 1988) or a single periodic thread. More complex real-time systems consist of multiple threads that are scheduled based on priority. Such systems are difficult to express in MLton for two reasons. First, MLton lacks the infrastructure to schedule such systems. RTOSs provide constructs like high precision timers, priority-based scheduling, and periodic tasks which are essential functionality for such real-time systems. Building these systems on MLton would require providing support for these constructs at the language level. Second, a task that performs a blocking operation would prevent all other tasks from executing, since all of them are multiplexed onto a single OS thread. For example, if one of the green threads in the schedule attempts I/O, the underlying OS would pause the entire process (and so all the tasks in the schedule) until the I/O completes. One way to handle this is to implement a non-blocking system and reason about its effect on predictability. Alternatively, a real-time system can use a blocking mechanism in conjunction with OS-level threads as long as the time the task spends blocked is factored in when calculating the WCET. The real issue is preventing *all* tasks from being blocked when one task performs a blocking operation as this increases the WCET of all tasks instead of just the task which performs the blocking operation. Thus, the blocking approach has an adverse effect on the overall schedulability of the real-time system.

Our approach is to provide a mapping of green threads to OS threads. This would give MLton access to all the essential real-time functionalities an RTOS provides, removing the engineering task of implementing these constructs and services at the language level. It also removes the need to implement non-blocking I/O as it ensures that any blocking calls, including blocking I/O, can be reasoned about in isolation as other tasks can execute without being blocked. Additionally, such a mapping opens up the potential to explore hierarchical scheduling policies, like that of Lipari & Bini (2005), to schedule *multiple* real-time applications in the same system.

3 Real-time extensions to MLton

To create a version of MLton that supports real-time computations, we must address the limitations described in Section 2. At a high level, this includes moving concurrency to the OS level with potential to support parallelism, extending the MLton threading model to support priorities and multiplexing over OS threads, addressing runtime stacks and redesigning the GC to be real-time aware.

3.1 Thread model

Most embedded boards today have more than one core and this might make highly scalable implementations like Multi-MLton a preferred choice for embedded use. Such implementations, more often than not, go with thread-local heaps which have additional

[6] A *cyclic executive* is a simple deterministic scheme that consists, for a single processor, of the repeated execution of a series of *frames* (or *minor cycles* as they are often called). Each frame comprises a sequence of *jobs* that execute in a defined sequence and must complete by the end of the frame. The set of frames is called the *major cycle*.

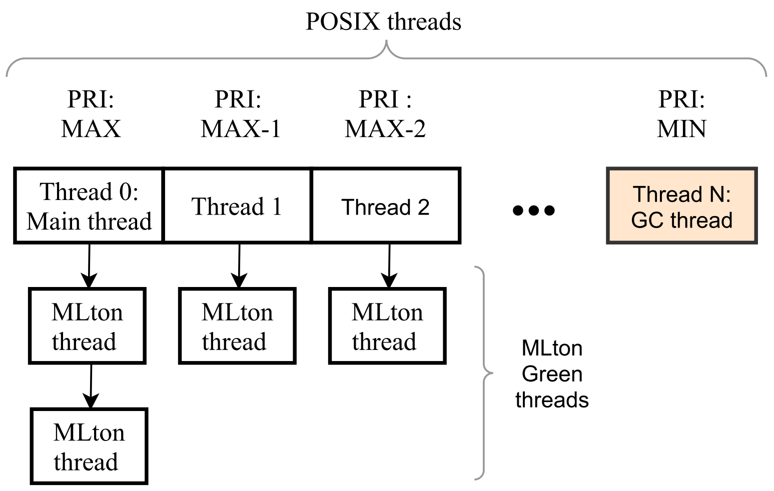

Fig. 7. Priority-based OS/green thread relationship model. This arranges OS threads as *priority buckets*. Computation to be run at a specific priority is associated with the corresponding priority bucket.

overheads like read/write barriers and synchronization for a global GC. Although Multi-MLton uses techniques such as procrastination (delaying writes that would cause eviction to the global heap) and cleanliness (copying of mutable state to avoid sharing of state) (Sivaramakrishnan *et al.*, 2012) to reduce some of these overheads, the true potential of these multicore optimizations are evident when the underlying architecture has 16 or more cores. While embedded boards of today support multiple cores, the number of cores supported is less than what it takes to make Multi-MLton efficient for use in such situations. It is also not clear what the blocking delays are for procrastination and synchronizations for a global GC, as they cannot be easily reasoned about for individual computations or tasks. We discuss additional implications of the Multi-MLton model on inter-thread communication in Section 3.5. Systems like MLton, on the other hand, are highly optimized to keep concurrency lightweight and fast. We would like to utilize this know-how to build a shared heap implementation and give it the ability to spread out across more than one core if needed. This model, we feel, is closer to the current state of embedded boards.

Figure 7 shows our concurrency model. The first step to having a threading model that supports OS-level concurrency is to split the green threads multiplexed over a single OS thread, over multiple OS-level threads. Moreover, to support real-time execution, we also must split green threads based on their priority. In the simplest case, there exists at most one green thread (computation) for any given priority supported by the system.[7] We use POSIX threads to expose OS-level threads and a POSIX thread is created for each priority the system supports. Note that the heavyweight POSIX threads are distinct from the lightweight green threads provided by MLton. Currently, we expose only the priorities that the underlying OS or RTOS expose. Real-time systems assign priorities based on a schedulability analysis. We provide a mapping from the priorities given by this analysis to POSIX thread priorities. POSIX provides primitives to assign thread priorities, which we use to

[7] Most real-time systems have a specific set of priorities they support.

assign a fixed priority to each POSIX thread we create. Each POSIX thread is also given a *thread ID* which serves to identify the thread as well as denote the fixed priority that thread is mapped to. For example, if the real-time system supports three priority levels, RTMLton creates three POSIX threads having thread IDs from 0 to 2. Internally, during creation, thread 0 is mapped to the highest priority the RTOS provides and thread 1 is mapped to the second highest priority and thread 2 is mapped to the lowest priority, which is also the default priority of the GC thread. Thus, we have a system where thread ID 0 and 2 map to the highest and lowest priority supported by the RTOS, respectively. Figure 7 depicts this idea where the green threads are multiplexed onto POSIX threads, each of which acts like a priority bucket. Scheduling issues like *priority inversions* are typically addressed by the schedulability analysis coupled with priority inversion avoidance protocals an RTOS would provide. In our implementation, we leverage the priority inversion avoidance protocols of the RTOS, which will temporarily boost the priority of any thread which accesses a shared resource that a higher priority thread requires. This priority boosting is taken into consideration in the schedulability analysis. There is also scope to use static priority inversion detection and avoidance techniques as in Muller *et al.* (2018).

We create all the OS threads the program can support at startup to save the overhead of creating a new thread during program execution. Before the threads are created, the first MLton OS thread (let us call it the *main-os-thread*) needs to do some initial setup which includes setting up necessary structures like `GC_state` and allocating and chunking the program heap, as described in Section 3.3.1. Once this setup is done, the main-os-thread will then create all the other OS threads (number equal to maximum priorities supported), which execute a *busy-wait* (spin loop) as they cannot call into SML code since they do not have an associated ML stack. Instead, they wait until main-os-thread has completed setting up the MLton *green thread* infrastructure. The main-os-thread trampolines[8] into SML code creates the first MLton thread (let us call it *main-green-thread*) and finishes any initialization code found in the MLton basis library. When this step is done, the main-os-thread calls back into the runtime to duplicate the main-green-thread's stack and associated data, once for each OS thread created. At this point, the OS threads can start executing SML code, when scheduled. However, the OS thread at the lowest priority is associated with the GC thread and the GC thread does not need an ML stack since it will not execute any SML code. Thus, the GC thread is kept busy-waiting until it is signaled to start running at the end of the system startup process, which is explained in detail in Section 3.3.4. Therefore, we view our system as having two distinct phases; The *setup phase*, where the program heap and associated infrastructure are initialized, and the *mission phase*, where all threads are executing SML code and GC is available for collection.

At its very core, a simple real-time system which is represented as a single periodic task, would be just the main-os-thread mapped to the task running at the highest priority and the GC running at the lowest priority. More complicated real-time systems with multiple tasks can map tasks to the other POSIX threads at the desired priority to ensure that all tasks are appropriately scheduled. Thus, this model allows for specification of different types of real-time systems.

[8] A trampoline (Steele, 1978) is a loop that iteratively calls an inner function. When the inner function wants to call another function, it returns the address to the trampoline, which then calls this new function (continuation-passing style).

```
void switchToThread (GC_state s, Thread t) {

  /* save stack for the pausing thread */
  s->currentThread[osthreadnumber]->currentFrame =
        s->currentFrame[osthreadnumber];
  s->currentThread[osthreadnumber]->exnStack =
        s->exnStack[osthreadnumber];

  /* swap in stack from thread being switched to */
  s->currentThread[osthreadnumber] = t;
  s->exnStack[osthreadnumber]      = t->exnStack;
  s->currentFrame[osthreadnumber]  = t->currentFrame;
  s->stackDepth[osthreadnumber]    = t->stackDepth;
}
```

Fig. 8. RTMLton switching green threads multiplexed on an OS thread identified by `osthreadnumber`.

3.2 Handling shared state

Migrating to a runtime system that leverages multiple OS-level threads requires re-engineering how MLton keeps track of the state of the system using the `GC_state` structure. This structure has fields that store the state of the system like the current executing green thread, pointers to the top and bottom of the current stack among others. All these fields are accessed at any time by offsetting a pointer to the `GC_state` structure. The decision to use one single structure for storing all the global state was to make the access fast by caching a pointer to the structure. When there is a single thread of execution, there is no need to worry about concurrent access to the `GC_state` and thus the integrity of the state is maintained. Introducing multiple threads of execution brings in a plethora of changes including the necessity to identify the thread of execution to which the value being stored belongs. Needless to say, threads must also have controlled access to the shared fields in this structure. In RTMLton, we have decided to keep `GC_state` as a single structure but implement arrays within it where appropriate. For example, to find the current green thread running within the OS thread, we would refer to the index `GC_state->currentThread[osthreadnumber]`.

Appendix A shows the changes made to the `GC_state` structure by RTMLton in red. `MAXPRI` is the maximum set of priorities supported by the real-time system, which can be set using a compile time argument. Using arrays where appropriate allows us to be more efficient when it comes to memory utilization—an important consideration when targeting embedded systems. Moreover, this method reduces the need for locking on frequently accessed fields, thus keeping concurrency as lightweight as possible. An example of this is evident in the thread switching code we saw in Figure 5. We noted that access to the `currentThread` field in the code would need to be controlled when there are multiple threads operating on the heap. Figure 8 shows the `switchToThread` function utilizing this array notation in RTMLton to perform a lock-free green thread context switch. Note that the handling of the stack is as per discussion in Section 3.4. It follows that every OS-level thread only modifies the index corresponding to its thread ID in the `currentThread` array (and other similar arrays). The `GC_state`, however, is still a single monolithic structure shared by all OS threads.

3.2.1 Discussion

As we saw in Section 2.4, MLton provides us with a lightweight thread data structure, or green threads, and a mechanism to choose which green thread is executed. MLton has no explicit scheduler but since threads are no different from any other continuation, we can build a user-defined scheduler to schedule these green threads. This scheduler can be preemptive (can de-schedule a thread before it finishes execution) or non-preemptive (lets every thread finish its execution) and even implement a notion of priority. But the fact remains that all these green threads are still multiplexed over the same OS thread. In RTMLton, we pull out multiple OS-level threads and then use the scheduling policy as dictated by the RTOS to schedule these threads. Since we still associate OS threads with these green threads, it opens up a plethora of opportunities to explore various thread scheduling mechanisms. We can explore hierarchical schedules (as noted in Section 2.4.2) which make use of the fact that we can still have user-defined schedules over green threads, while the OS threads continue to use either the default RTOS scheduler or a new one implemented in the runtime system. Such hierarchical models gives a very nice way to compose priorities where computation can be further prioritized (at the green thread level) when they have the same system priority (at OS thread level). Additionally, different scheduling idioms can be considered for threads that have same OS priority. For example, cooperative scheduling can be used to logically break a given task with a given OS priority, into multiple cooperative green threads multiplexed on the appropriate priority OS thread. Alternatively, its also possible to build a real-time system that has multiple tasks at the same OS priority, but choosing which one gets to run is determined based on the green thread priority. Instead of implementing our own scheduler, we can also explore the use of concurrency libraries like CML and ACML to further increase expressivity of our programs by adding threading primitives like message passing and making use of scheduling policies these libraries define. We leave such effort to future work as part of coming up with a better programming model for real-time systems.

3.3 Real-time GC

To implement a concurrent GC, it is necessary to have the garbage collector execute in its own thread so that it can work independently of mutator threads (application threads that mutate the heap). While it is possible to design an incremental real-time GC that is not concurrent, there are benefits to having the GC on its own thread. First, the system can now utilize the underlying RTOS' scheduler where the GC can be given a priority and scheduled along with other tasks. This also promotes exploration of different RTGC strategies like a *slack-based* GC or a *time-based* GC. Second, in platforms where the system can utilize more than one core, the GC can be parallelized to avoid being scheduled with all the other RT tasks in the system. Multicore implementations of SML like Multi-MLton take a different route in handling this separation. They use a per thread heap and thus have a per thread GC which stays coupled to the execution thread. Multiple heaps may pose other complexities (like read/write barrier overheads, global synchronization) in an embedded or real-time system, which is why we chose a single shared heap.

A shared heap implementation is easier but brings us back to the difficult task of refactoring the GC onto a separate thread. In doing so, we need to make sure each thread is responsible for growing its own stack and allocating objects that it requires. Although a concurrent GC can scan and collect while mutator threads execute, mutator threads must be paused to scan their stacks and construct a root set. This is necessary because MLton stores temporary variables on the stack and if the GC were to run before the stack frame had been fully reified, the results would be unpredictable. MLton will also write into a newly created stack frame before finalizing and recording the size of the frame. Without the identification of safepoints to pause the threads, the heap will be malformed with live objects being considered dead due to being temporarily unreachable. Fortunately, MLton identifies these safepoints for us. GC safepoints in MLton are points in the code where it is safe for the thread running the code to pause, allowing the GC to scan stacks.

Although GC safepoints are pre-identified for us, the code generated by the compiler assumes a single-threaded model and introduces problematic constructs, such as global variables, and relies on caching important pointers in registers for performance. We needed to rework these architectural decisions to make them compatible with a multi-threaded model. As discussed above, MLton tracks a considerable amount of global state using the `GC_state` structure so we must refactor this structure, in particular, to make it thread-aware. MLton also uses additional global state, outside of `GC_state` structure, to implement critical functionality.

3.3.1 Handling fragmentation

Heap Layout: the design of a real-time garbage collector should ensure predictability of memory management. To eliminate GC work induced by defragmentation and heap compaction, we make sure that objects are allocated as fixed size chunks so that objects will never need to be moved for defragmentation through the use of a hybrid-fragmenting GC like Pizlo *et al.* (2010*b*). This chunked heap is managed by a free list. In MLton, the size of normal objects, arrays, and stacks vary significantly. In RTMLton, since one objective of a unified chunked heap is to prevent moving objects during a GC, we need to have all chunks to be of the same size. We split the heap into fixed size chunks and initialize the free list to map all the free chunks. This does lead to space wastage in each chunk as object sizes vary. However, this opens up room for potential optimizations discussed later in this section. During collection, the GC first marks all fixed size chunks that are currently live. Then it sweeps the heap and returns all unmarked chunks to the free lists. This completely eliminates the need for compaction in order to handle fragmentation.

Object Layout: All RTMLton objects have a chunked layout with each chunk having the same fixed size. Generally speaking, each chunked object contains fields for chunk management, a payload portion which houses the original MLton object and pointers to link chunked objects. Multiple chunks need to be linked if the MLton object does not fit into the payload portion of one chunk. However, representation of various objects differ slightly.

A general strategy to arrive at a chunk size in RTMLton is to ensure that the biggest MLton objects can be fit into the least amount of chunks. This would ensure fast access times. MLton already tries to pack small objects into larger ones. In our empirical study of MLton's regression and benchmark suite, most normal objects are around 24 bytes and arrays are close to 128 bytes. The regression suite mimics real-world programs that stress

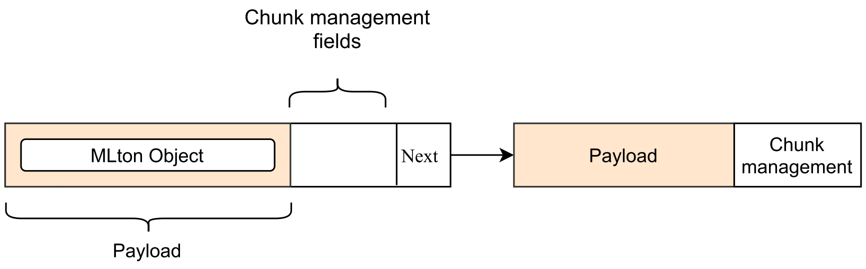

Fig. 9. Normal/weak objects represented as a list of chunks in RTMLton.

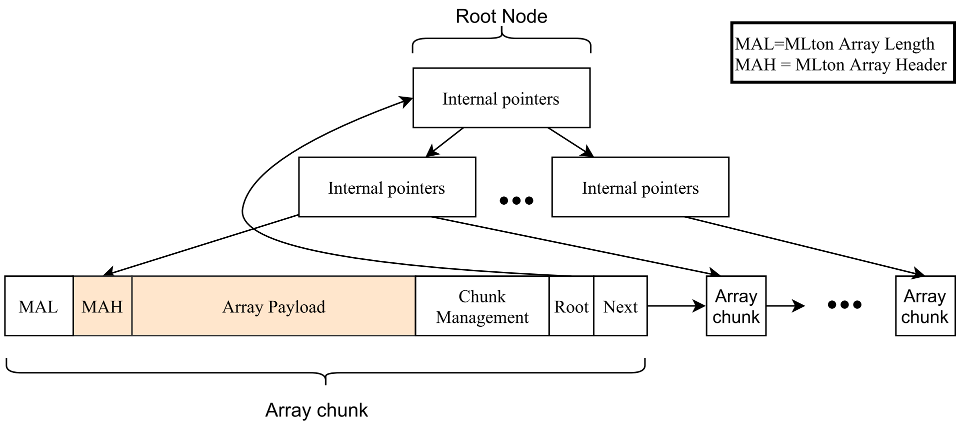

Fig. 10. Arrays represented as a tree of chunks in RTMLton. Leaf nodes hold the array data. Internal nodes hold pointers to their children, which are used to reach the leaf nodes from the root.

test the compiler and runtime system and is a good approximation of real-time system allocation workloads as well. We currently use this information to arrive at a fixed size for our chunks. Given that in MLton/RTMLton, we have access to object size information at compile time, future work on selecting an efficient chunk size at compile time is planned.

For results in this paper, normal and weak objects have a payload size of 318 bytes along with an extra 28 bytes associated with chunk management. These objects are represented as a linked list of chunks, ranging from just one chunk in the list to a maximum of two, as shown in Figure 9. Normal and weak objects larger than 318 bytes (the RTMLton chunked object payload for such objects) are split into multiple chunks. In our current implementation, we limit normal objects to two chunks although we have not noticed objects that are greater than 64 bytes in our empirical analysis, to account for programs that might allocate larger normal objects. Since object sizes in MLton are predictable at compile time, we achieve constant access time when allocating these objects by sizing our chunks to fit an object.

In MLton, arrays are passed around using a pointer to its payload. The header and length of an array are retrieved by subtracting the header size and array length size from current pointer. In RTMLton, we stick to this representation as much as possible. Arrays are represented as an *m*-ary tree as depicted in Figure 10 where each node is fixed size. The array payloads are 300 bytes with 46 bytes to manage array chunks. Internal nodes carry 75 pointers (maximum number that fit in the payload) to their children. We pass an array around via a pointer to the root of the tree. In the event that the array is small enough to fit in a single chunk, then we will pack the array contents into the root node. If a tree must be created, we will construct it bottom up. We first allocate the minimum number of leaves necessary to contain the array and then create a tree above those leaves with the minimum number of internal nodes needed to reach all leaves. A root pointer and a next pointer is embedded in each leaf node. The next pointer connects all leaves for linear traversal. The root pointer allows the garbage collector to quickly reach the root node when it is in its marking phase. When accessing an element of an array, we begin at the root and proceed top down, until we finally arrive at the leaf containing the slice of the array that contains the target index. Thus, for small arrays that can be packed into a single chunk, access is $O(1)$ and consists of pointer arithmetic, and for larger arrays, access time is $O(\log_{75} N)$.

3.3.2 Array limitations

Flattening refers to the multiple optimization passes in MLton that reduce the overhead for accessing nested objects by representing them as tuples. Unfortunately, it is difficult to reliably decide on an array element size after flattening that can be used at the time of allocation, since tuples can carry elements that differ in size. Our tree-structured array has no information about flattening and the access scheme generated from MLton after flattening cannot work with our chunked array model. Hence, we need to disable some of the flattening optimization passes. We first tried disabling all the flattening passes including local flatten and deep flatten. But in our later investigation, only deep flatten will try to flatten objects in arrays. The local flatten passes are compatible with our implementation.

3.3.3 Array discussion

One thing we do not currently do is optimize for linear traversal. Many array accesses, such as I/O, and common operations like `foldl` and `foldr`, traverse an array linearly. In our current implementation, if the array spans more than one chunk, we do a tree traversal each time we want to find an element. We envision that adding some memoization to the tree walking function could allow it to detect if the index being searched for is “close to" the most recent index that was looked up. If it is, we could calculate, relative to the previous index, where the current one would be and skip the tree traversal. Another optimization would be to implement *spines* as described by Pizlo *et al.* (2010*b*). All the internal nodes can be allocated as one contiguous spine in a separate heap managed by a fast copying GC. Since these spines are immutable, they can be safely accessed by the mutator even when being copied by the GC. Thus, we can access any element in the array in constant time through its spine. A lack of such optimizations is demonstrated in our measurements of the MLton benchmarks in Table 1.

3.3.4 GC model

For collection, our concurrent GC leverages a traditional nonmoving, mark-and-sweep with a Dijkstra’s incremental update write barrier (Dijkstra *et al.*, 1978). Our GC runs on its own OS thread and operates independently of the mutator, repeating the steps below:

1. Wait for synchronization
2. Start marking

3. Sweep
4. Cleanup and bookkeeping

Each loop consisting of these four steps/phases signifies a *GC cycle*.

Wait for synchronization: As mentioned earlier, MLton identifies GC safepoints where the mutator threads can be paused safely for a GC. In RTMLton, we use this opportunity to make our garbage collection incremental by having the mutator threads scan their own stacks at these GC safepoints. MLton performs complicated data flow and control flow analysis to estimate object lifetimes and amount (size) of objects allocated in various code segments. It uses this information to insert these safepoints to minimize the number of garbage collections needed, while ensuring the mutator has enough memory for allocations. However, the analysis assumes a single-heap model in which memory for objects is calculated by number of bytes required (rather than chunks), which is incompatible with our model. One solution is to patch up each path in the GC safepoints flow, redirecting all GC checks to our GC runtime function, and letting the C runtime function decide whether a garbage collection is needed (based on the state of the heap). This method has high overheads in the form of preparing the code to jump to a C call. Any calls to a C runtime function that has the potential to perform a garbage collection needs to be "made safe" so that the GC does not wrongfully collect any object. This involves saving all the temporaries currently live, onto the stack as local variables and adding a `C_FRAME` marker to that stack frame. This process increases the stack size and affects overall runtime of the program. Instead, in RTMLton, we add an optimization pass (*gc-check*) that sums up the allocations in a code block and inserts a check to see if there are adequate chunks left to satisfy allocations. If the code block does not allocate objects at all, we ignore it. Such a check only introduces a branch and an inlined integer comparison, which is much faster and more efficient than the former method. Since arrays are allocated by the C runtime (see Section 2.2), MLton ensures the stack is completely prepared before jumping into `GC_arrayAllocate`. We can thus safely make GC checks in the array allocation.

In this phase, the GC is waiting for all the mutator threads to synchronize at the GC safepoints so that it can continue with its work in a safe manner. Currently, each thread walks its own stacks and marks all object chunks that are immediately reachable from its stacks using a tricolor abstraction. All the chunks that are immediately reachable from the stack are marked black (meaning reachable and explored). The children of the black chunks are shaded gray (reachable but unexplored) and then put into a *worklist*. It follows that any chunk marked white, or unmarked, is not reachable and hence would be collected eventually. This model where each thread scans its own stack to construct a root set and the GC scans the rest and sweeps concurrently is different from that of MLton's monolithic GC model, in that the mutator does not have to wait until the entire heap is scanned.

When all the mutators have finished marking their own stacks at their GC checkpoints, they set a bit to indicate that they have synced. The last mutator to do so would signal the GC to start its process in parallel as all the mutators go about doing their respective jobs.

Start marking: The GC starts marking the heap when it receives the *all-synced* signal from mutators. All object chunks in the worklist are gray at this point and the GC starts by marking all reachable chunks from each worklist item. Each time a worklist object is picked up, it is marked black and when it has been fully explored, it is removed from the worklist. Marking proceeds as before with the chunk being marked black when reachable and all the chunks immediately reachable from it are shaded gray. The GC aims to collect all unreachable objects without wrongfully collecting objects that are in use. But with the mutator allocating while the GC is marking, it could lead to a rearrangement of the heap by the mutator that invalidates our marking. Which is why we make use of a Dijkstra-style incremental update barrier which enforces the strong tricolor invariant. The strong tricolor invariant states that there should be no pointers from black objects to white objects. The write barrier is inserted by the compiler on any pointer store on the heap and upholds the strong invariant by shading gray any pointer store that moves a white chunk into a black chunk. The write barrier is made to selectively perform this operation (turned on) only when the GC is running in parallel and at other times only does an atomic comparison to see if it the GC is running or not. When the write barrier is turned on, all new object chunks are allocated gray so as to protect them from collection. Marking phase ends when the worklist is empty.

Sweep: Once the marking is done, the GC traverses the heap contiguously and reclaims any unmarked chunks back to the free list. While it sweeps the heap, the GC also unmarks any chunk that is marked in order to prep it for the next GC cycle. Adding a chunk back to the free list is done atomically and involves minor work like clearing out the chunk headers. Since we are using a chunked heap, we do not need to perform any defragmentation and the addition of chunks back to free list makes them available for reuse almost instantly.

Cleanup and bookkeeping: Before the GC goes back to “wait for synchronization" phase, it does some cleanup and bookkeeping work such as clearing out the sync bits and waking up any mutator that is blocked while waiting for the GC to complete its cycle. In a typical scenario no mutator will be paused, while the GC is running except very briefly to scan its own stack. RTGCs rely on efficient scheduling policies to ensure that the GC runs enough to make sure that no task runs out of memory, but in the absence of such policies we block the mutator if it does not have enough chunks free and the GC is running. The GC throws an *insufficient memory* message when it has made no progress (all mutators are blocked) in two consecutive GC cycles. This blocking mechanism can prove useful when designing mixed criticality real-time systems where non-RT threads that can afford to be blocked can do so while ensuring higher priority RT threads have enough space to run.

3.3.5 Memory reservation mechanism

MLton generated C code is split into basic blocks of code with each block containing multiple statements and ending with a transfer to another code block. These code blocks are translated from the SML functions, and an SML function can span multiple C code blocks. If a C code block performs an allocation and if the transfer out of the block has the potential to perform a GC, the allocated objects are pushed into stack slots to prevent the GC from wrongfully collecting it.

An example of such a transfer is a call to a C runtime function, which has the potential to GC, or a transfer to a GC safepoint which is inserted by the compiler. MLton is very efficient at inserting GC safepoints to ensure that allocations between two safepoints can proceed without requiring the GC to intervene. But to achieve this, it makes assumptions which it can afford to do in a single computation model. In RTMLton, since there are multiple threads running concurrently, there is always the possibility that an allocation request of a thread, between two GC safepoints, cannot be fulfilled because another thread utilized all available memory chunks. Such a case will require the mutator to be paused, while the GC frees up more memory chunks. This becomes a source of unpredictability for a real-time system.

Moreover, when the there are not enough free chunks at the point of allocation in RTMLton, it leads to an edge case where any previously allocated chunk might be wrongfully collected, because they were not pushed into stack slots. Consider the scenarios in Figure 11 which examine a code block which does two allocations before these are pushed into stack slots (before a GC safepoint). Scenario 1 and 2 show the cases when the GC is running (write barrier is turned ON), and Scenario 3 and 4 show the cases when the GC is not running (write barrier turned OFF). In Scenario 1, there are more than two free chunks available during the allocation and hence the allocation requests are satisfied without any issue. Since the write barrier is turned ON, the objects are allocated with the default gray marking to prevent them from being wrongfully collected by the GC. In Scenario 2, there is just one free chunk available. So the first object is allocated (with gray marking) and when the second allocation happens, the mutator needs to wait for GC to free up space. But since the GC was already running, the first object is shaded gray and is not wrongfully collected by the GC. When the GC frees up one more chunk, the second allocation request can be satisfied. Scenario 3 shows the execution when the write barrier is OFF and there are more than two free memory chunks available. Both objects are allocated with a default white marking and since there is enough free chunks to satisfy both requests, GC intervention is not required and both objects are safe from collection (despite being unmarked) until they are pushed into stack slots. However, in Scenario 4, there is only one free chunk which is enough to satisfy the first allocation request (Alloc1). When the second allocation request is processed, the GC needs to intervene to make more space and since Alloc1 was allocated unmarked (white), it would result in the GC wrongfully collecting it.

One possible solution to the edge case is to convert all allocations into calls to C runtime functions and then let MLton appropriately protect all previous allocations by pushing them into stack slots before the next allocation happens. This would involve splitting up each of the C basic blocks further into multiple blocks with each block containing only allocation. We found that this involves a considerable overhead in terms of the code size as well as the stack space since the number of allocations done by a program is not trivial. Moreover, this does not address the issue of unpredictability in allocation for RTMLton.

What we need is a way to guarantee that when a block of code is being executed, it will receive all the memory chunks it requests for allocation before it begins executing that block. Thus, guaranteeing that the mutator will not be paused at an allocation point. This can be achieved with the help of a reservation mechanism which reserves the chunks that the following block of code needs, from the free list, before it is actually allocated. MLton already has information about the number of objects allocated (except allocations by runtime functions) in every block at compile time. We can use this to our advantage by leveraging the gc-check pass we put into do a little more than insert the GC safepoints. At the point where we insert the GC check, we reserve the number of memory chunks

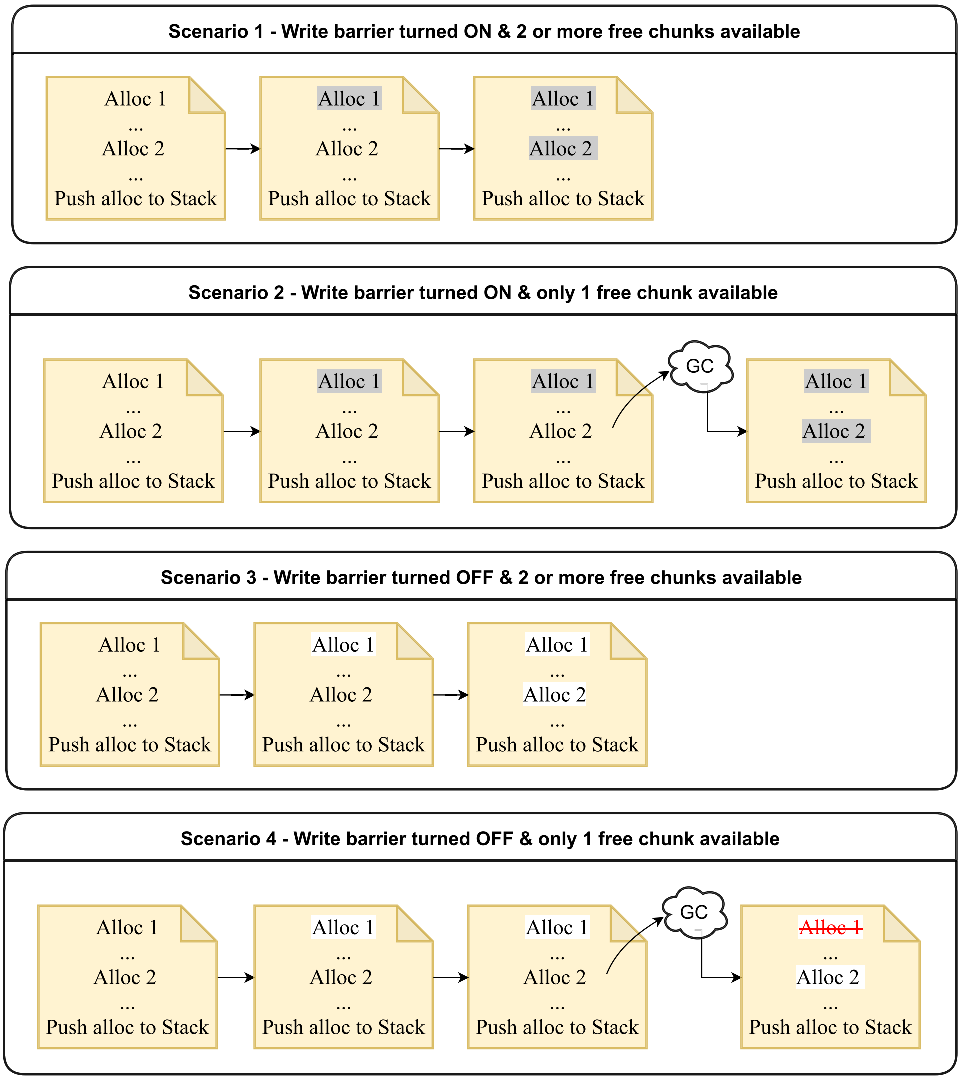
Scenario 1 - Write barrier turned ON & 2 or more free chunks available
Alloc 1 ... Alloc 2 ... Push alloc to Stack
Scenario 2 - Write barrier turned ON & only 1 free chunk available
GC
Scenario 3 - Write barrier turned OFF & 2 or more free chunks available
Scenario 4 - Write barrier turned OFF & only 1 free chunk available
~~Alloc 1~~

Fig. 11. Allocation scenarios depicting an edge case where GC can wrongfully collect an object. Code segment performs two allocations, and scenarios show various combinations of the write barrier (ON/OFF) and free chunks available (< 2 or ≥ 2). The color the “Alloc" statement is highlighted represents the color the object allocated is marked (gray—reachable, white—unreachable, red strikethrough—wrongfully collected).

the next code block needs. Reservation is done by atomically incrementing a counter before executing the block and then decrementing it when the object is actually allocated. Figure 12 summarizes the logic involved in reserving allocations before a block is executed.

Thus, at each safepoint, if the number of free chunks available (excluding those reserved by other threads) is less than what is required by the next code block, the reservation mechanism prepares the mutator for synchronizing with the GC thread and suspends the

```
LOCK;
while (free_chunks < (reserved_chunks + reqd_chunks))
{
   UNLOCK;
   GC_collectAndBlock;
   LOCK;
}
reserved_chunks += reqd_chunks;
UNLOCK;
```

Fig. 12. Reservation mechanism code snippet.

mutator until woken up by the GC. Once again, such a scenario (where the mutator is suspended) is avoided for high-priority RT threads with the help of a schedulability analysis of the real-time system. If there are enough free chunks available, we simply increment the reserved count by the number of chunks the mutator will need and any subsequent mutator that tries to allocate will know that those many chunks have already been reserved from the free list. The pass does not consider array allocations which is done by a C runtime function and other allocations like a new thread object or new stack frames which are decided at runtime. But as discussed before, MLton appropriately manages the stack before transferring into such runtime functions, making it safe to have the runtime do the reservation in these cases thus adhering to the policy of "*No allocation without reservation.*"

3.3.6 Discussion

Our implementation of the real-time GC opens up many possibilities for optimizations which would improve the performance of the GC. Some areas for optimization are

- Stack scanning at the end of thread period: currently, the mutator is responsible for scanning its own stack, which—albeit performing incremental GC work—induces a small pause affecting the overall mutator performance. An efficient scheduling policy is key to most real-time systems. The threads (or tasks) are scheduled so as to perform a bulk of their execution by the end of their period. By having each thread scan its own stack, at the end of its period, it contributes to making the GC work incrementally which would give good mutator performance. Secondly, one can also argue that the stack would be at its shallowest at the end of its period because the thread would be done with the bulk of its work, therefore reducing the time the mutator spends marking its stack at the end of its period as opposed to the middle of its period. In fact, such a policy coupled with our reservation mechanism could ensure allocation guarantees for each release of a task and efficient reclamation of garbage once the task finishes its execution.
- Object packing based on lifetime: one limitation of our chunked object model—as compared to a traditional object model—is space wastage because we allocate one MLton object per fixed size memory chunk. We define memory wastage as the memory in the chunk that is unused due to the object size being less than the chunk size. Note that objects that are bigger than the fixed size chunk may waste memory if they are split into two or more chunks, though only the last chunk would contain

wasted memory. One obvious optimization is the packing of multiple MLton objects into one fixed size chunk, which can reduce memory wastage. We could pack objects based on many criteria like object sizes or those allocated by the same function but we are particularly interested in exploring packing of multiple MLton objects into chunks based on their lifetimes. This would make the GC much more efficient by facilitating the collection of all objects in the same chunk in one go. This is similar to techniques that combine region-based memory management with GC (Hallenberg *et al.*, 2002*a*; Elsman & Hallenberg, 2020), where the similarity lies in the fact that our memory chunks would act like small regions. Instead of our current mapping of one MLton object to one RTMLton chunk, we could pack multiple MLton objects into chunks (acting as regions) based on their lifetime or even temporal information like a predetermined schedule of tasks. Such work has not been implemented in this paper but is being currently studied.

- Efficient GC check insertion: as described in the section above, the gc-check pass does the check and reservation at a per-block (generated C basic blocks) level. There is potential to incorporate some of MLton's control and data flow analysis to find a better place to reserve memory chunks as part of future work.

3.4 Stack model

Section 2.3 gave an insight into the MLton stack object which required contiguous space and was very closely coupled with MLton's GC. While there are many different representations of stacks that could be used (Clinger *et al.*, 1999), since RTMLton employs a chunked object model for ensuring predictable memory management, it naturally leads us to a *stacklet* implementation (Goldstein, 1997; Cheng, 2001) to handle stacks. In this model, each stack frame is placed in a separate chunk, and the chunks are linked together with forward and reverse pointers to facilitate traversal. By moving to a stacklet model, and allowing the mutator threads to manage their own stack size without necessitating intervention from the GC, we have addressed the concerns outlined in Section 2.3.2. Furthermore, stacklets allow us to maintain our chunked memory model and avoid copy and compaction when a stack must grow.

The organization of our stacklet model follows MLton's stack layout very closely. This allows us to migrate to stacklets with a minimum of changes to any stack specific optimizations that the MLton compiler does. In order to put RTMLton's implementation and design decisions into context, it is beneficial to understand MLton's stack operations in detail.

3.4.1 Managing call stacks : Stack versus stacklets

An application is organized as a collection of functions. If a function F calls another function G, upon completion, G must determine how to return control and any associated computations, back to F. This transfer of control between functions is managed by a *call stack*, which we will refer to as a *stack* in this section. The stack itself is organized into *frames* (also called *activation records*) where each frame contains information pertaining to an invocation of a function. So in our example, the first frame on the stack would correspond to F and the second frame to G. Each of those frames contain state information, such

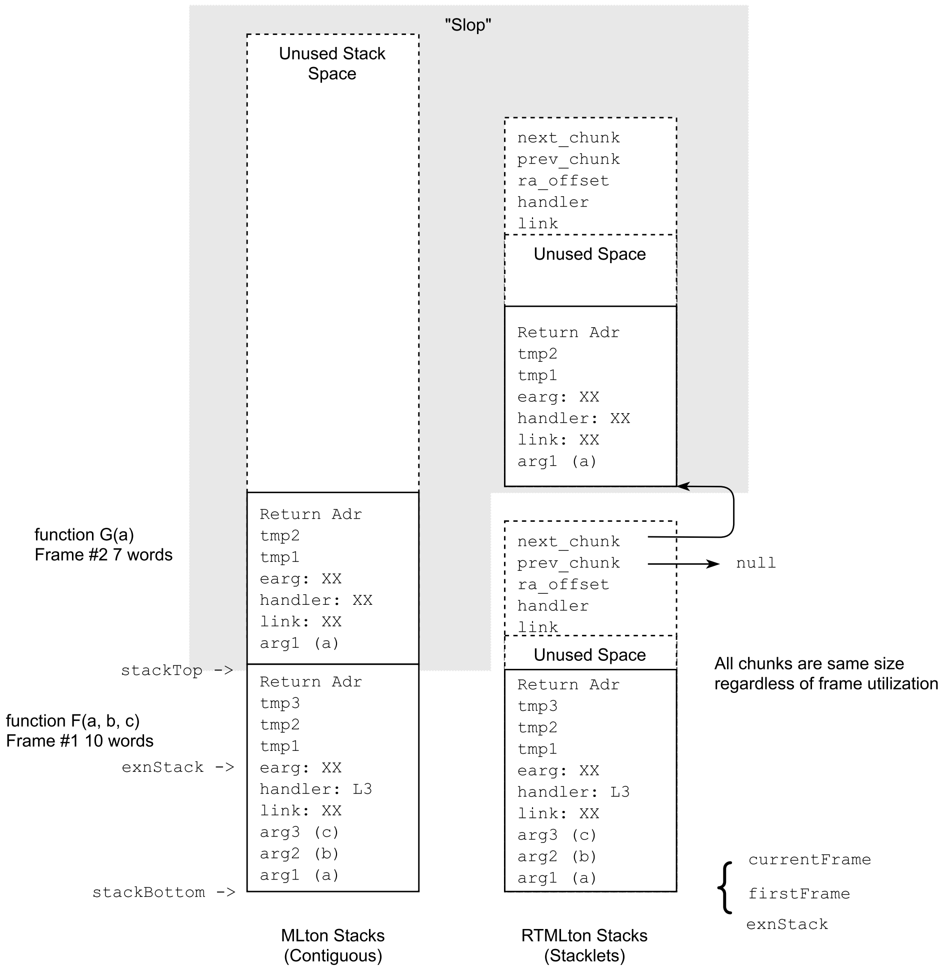

Fig. 13. Stacks: MLton versus RTMLton. MLton stacks are contiguous memory segments, whereas RTMLton stacks are a linked list of memory chunks.

as arguments, temporary variables, the return address of the calling function, and exception handling information if such a handler was installed by the function. Figure 13 shows a diagram of a stack (on the left) which is a contiguous block of memory into which frames are consecutively written. As a function calls another function, a pointer is used to indicate which frame is the currently executing function. When a function is called, the pointer moves forward one frame, when that function returns, the pointer moves back a frame. Exception handlers are implemented as pointers that allow control to jump backward an arbitrary number of frames rather than having to adhere to the strict push/pop semantics of the stack data structure.

A stacklet structure, in contrast, breaks the frames into a linked list of heap memory blocks. The right side of Figure 13 shows a stacklet structure. There are GC-related implications to selecting a stacklet structure rather than a stack structure which we will discuss later in this section.

3.4.2 MLton stack frames

Frame layout : We saw in Figure 4 how each stack object has contiguous reserved bytes, which MLton uses to allocate stack frames in bump pointer fashion. Figure 13 gives a more detailed picture of the stack frames and what they contain. MLton organizes stack frames by “slots." A slot contains a value or a pointer. Starting from the bottom of the frame in Figure 13, the `arg` slots hold the arguments to the function. There can be more than one `arg` slot if the function takes many arguments and no `arg` slot if the function takes none. The `earg`, `handler`, and `link` slots are used for exception handling (discussed below). The `tmp` slots in the figure represent temporary values the function might allocate, but once again they need not always be present. Finally, the slot closest to the top of the frame holds the return address. The return address is a numeric value (as opposed to a pointer), that is assigned by the compiler.

Frame access and management: The `stackTop` pointer tracks the frame at the top of the stack (note that this is stored in the `GC_state` structure). A new frame can be pushed onto the stack (and so calling a function) by advancing the `stackTop` pointer by the size of the new frame. Similarly, a function return would pop a frame from the MLton stack by decrementing the `stackTop` pointer by the size of the popped off frame. MLton maintains a portion of memory at the top of the stack called the *slop*, which is the unused space in reserved bytes. This portion is written to when a function is preparing to call another function. This is indicated in Figure 13 by the `stackTop` pointer which is pointing to the top of the first frame, while the slop space contains active data. This helps MLton setup the function call by copying required data from the current frame to the next frame. For example, if the current function passes arguments to the function being called, MLton can write this data in its respective slot prior to the actual function call. Once the function call is made, the `stackTop` pointer is advanced, growing the active stack and shrinking the slop. Similarly, when the function returns from the call, `stackTop` is moved down the stack, returning space to the slop. The caller would access the callee’s return values when present. These values are stored in the callee’s frame which is now, again, part of the slop.

When accessing a slot in a frame, MLton needs only a slot width and offset into the current frame in order to read/write any stack slot via pointer arithmetic.

3.4.3 RTMLton stacklets

As mentioned earlier, the stacklet implementation allocates every frame on a new fixed size chunk obtained from the *freelist*. This removes the necessity to have contiguous space allocated to maintain the reserved bytes portion of the MLton stack. Thus, we are able to keep the heap fragmentation free, while allocating stacks on the heap. As is the case with normal objects in Section 3.3.1, the payload portion of the stacklet memory chunk will hold the MLton stack frame. The slots within each frame remain the same (as MLton) since they are generated by the compiler. For example, the return address is still placed in the slot closest to the top of the frame. However, since the size of the chunk is fixed irrespective of the frame size, it follows that there may be some unused space (between the top of the frame and end of the payload space) in every chunk’s payload. This means that the return address slot is no longer at a known location accessible with simple pointer arithmetic. Thus, we add metadata in each chunk to manage the stacklets as shown in

Figure 13. The `ra_offset` field points to the location of the `Return Adr` slot within the MLton stack frame. The `next_chunk` and `prev_chunk` pointers assist with chunk traversal. The `handler` and `link` fields are copies (discussed below) of the exception handling fields found within the frame.

In RTMLton, `stackTop`, which was an offset from the bottom of the stack, is replaced with a pointer to the `currentFrame`. The `currentFrame` pointer points to the frame that is conceptually on the top of the stack, but in terms of stacklets, it is the frame that corresponds to the currently executing function. This pointer is advanced by setting it to `currentFrame->next_chunk` when we want to push a frame, thereby calling a function. Similarly, stack frames can be popped by adjusting `currentFrame` so that it points to `currentFrame->prev_chunk`. When calling a function (and so pushing a frame), we make an additional store operation to record where in the current frame the return address is stored. We also track the current depth of the stack (number of live frames) in order to implement a heuristic grow/shrink model for stacklets. Thus, our push/pop operations are less efficient than MLton's which is able to derive all of that information from just an offset from `stackTop`. However, in return, we get predictability guarantees around stack management that are important to our goal of providing a real-time capable compiler.

A challenge in implementing stacklets was the handling of MLton's function call mechanism. As we saw in the previous section, MLton writes to the slop area when preparing for a function call. In our stacklet model, this means that our minimum stack size is two chunks. We must be sure to have one additional chunk past the currently active one, in order to allow the compiler to write into that chunk in preparation for making a function call. This additional chunk represents the slop area for RTMLton's stacklets. While in MLton, this was accessible via an offset to `stackTop`, in RTMLton, that area is only accessible by traversing `currentFrame->next_chunk`. This means that all of the stack read/write operations (that access slop space) inserted by the compiler become incompatible with RTMLton's stacklets. One way to handle this would be to modify the compiler to remove any such read/writes to the slop space before a function call and after a return from a function. Not only does this require a large engineering effort, but it also takes away from our mission of staying as close to MLton's stack layout as possible in order to utilize any optimizations the compiler may perform. Another way to handle this is at the C codegen level. MLton finalizes all of its stack reads and writes at the machine Intermediate Representation (machine IR) level of the compilation process. This is done so that all the different backends it supports can utilize the same stack operations. In RTMLton, since we only support the C codegen backend, we can intercept these stack operations that write to a location beyond the current frame and simply emit a C preprocessor macro that references the next stacklet frame. In effect, we “translate" (in the C codegen backend) the stack operations that assume a contiguous set of frames into a stack operations that can now work on the stacklet model.

In terms of the GC, stacklet chunks are treated similarly to other chunks, in that they are collected when they are no longer reachable. The first frame in a chain of stacklets (tracked by the `firstFrame` pointer in Figure 13) is reachable via the thread object to which the stacklet belongs. Therefore, as long as the thread is alive, the stacklets are alive too. As mentioned, in order to grow a stacklet, the thread can request additional chunks per our

```
exception E
fun H n = if n > 0 then raise E else 0
fun G n = H n handle _ => (print "exn\n"; 0)
fun F n = G n
val _ = F(1)
```

Fig. 14. SML exception example. *G* installs a handler for the exception raised by *H*.

reservation model (Section 3.3.5). If the thread decides to shrink its stacklets, it can simply break the chain by NULLing the `next_chunk` and `prev_chunk` pointers. This makes the latter portion of the stacklet unreachable, and the GC will collect it. In this release of RTMLton, we have taken a heuristic approach to stack growth and reduction: a stacklet will grow by 25% once it reaches 90% utilization and will shrink by 10% once it falls below 50% utilization. Unlike the segmented stack model explored in Rust and Go, we do not allocate and deallocate frames upon function calls and returns. This mitigates the hot split problem observed in Go (Go-Lang, 2013; Morsing, 2014) and the stack thrashing problem in Rust (Anderson, 2013). It is still possible that if a code path cycled between 90% and 50% stacklet utilization, we might encounter an allocation/deallocation hot spot, but our avoidance of copy and compaction further mitigates the problem they encountered in their more traditional memory model. As noted in Regehr *et al.* (2005) calculating the worst-case stack depth is undecidable, but it can be approximated through program analysis. Recursion's effect on stack depth is particularly difficult to estimate, and so they conclude that the developer should specify a maximum iteration count on recursive functions. We intend to explore how such annotations can be incorporated into a programming model in future work.

3.4.4 Exceptions

Referring to Figure 13, when specifying an exception handler (example in Figure 14) MLton saves the current exception stack offset (if there is one) in the `link` slot, the function number of the handler being specified in the `handler` slot and then sets the exception stack offset (stored in the thread-global `GC_state` structure) to point to the slot after (explained below) the handler slot. The `XX` field values in the figure represent either unused fields (e.g. in MLton's second frame where an exception handler was not set in contrast to where one was set in the first frame) or fields that will be filled in at a later time (e.g. `earg`—exception argument—in the first frame which will be filled in when an exception is raised). When an exception occurs, the `stackTop` is reset using the exception stack offset, and then the return address immediately before that location (the handler slot) is used to jump to the exception handling function. When that function returns, the `stackTop` is again adjusted downward to point to the bottom of the frame where the exception occurred. Control then resumes using the return address just before that location (the function that called the function where the exception occurred). This sequence of events relies on storing the return address at the very top of the frame (as discussed previously). This is different from the layout discussed in Hieb *et al.* (1990) where the return address is stored at the base of the frame.

In our stacklet implementation, we move the exception handling fields to dedicated fields in the chunk's metadata (shown as `handler` and `link` just below `ra_offset`). This has the downside of being less space efficient than the MLton implementation for two reasons. First, we duplicate the fields in the header but do not remove them from the interior of the frame. We intend to address this in a future release of the compiler. Second, the `handler`, `earg`, and `link` fields will not be allocated in the MLton frame unless a handler is actually installed by the function.

3.4.5 Exceptions implementation

In the following discussion, a MLton "push" is defined as a C preprocessor macro. There is no explicit "pop" operation and instead MLton uses push with a negative argument to represent pop:

```
#define push(bytes) stackTop += (bytes)
```

A MLton "return" is defined as a C preprocessor macro that loads the word *before* `stackTop`. Since MLton stores the return address at the top of each frame, accessing the word just before the top of the stack will yield the return address for the previous function:

```
#define return()
      nextFun = *(uintptr_t *)(stackTop - sizeof(void*));
      (and then jump to the function given in nextFun)
```

An RTMLton "push" moves forward or backward one frame based on the sign of *bytes*:

```
#define push(bytes)
      if (bytes > 0)
          currentFrame = currentFrame->next_chunk;
      else if (bytes < 0)
          currentFrame = currentFrame->prev_chunk;
      else (error);
```

An RTMLton "return" loads the word from the return address slot in the previous chunk with the assistance of a chunk metadata field (`ra_offset`). This is done because as discussed earlier, the return address can be at an arbitrary location within the chunk and we save that location in the `ra_offset` field:

```
#define return()
      nextFun = *(uintptr_t *)(currentFrame->prev_chunk +
          currentFrame->prev_chunk->ra_offset);
```

In addition to `stackTop` and `stackBottom`, which specify the location of the currently live frame and the bottom most frame, respectively, MLton also tracks where in the stack the most recent exception handler was set using `exnStack`. MLton will set `exnStack` to point to the slot just after the `handler` slot. This allows MLton to use "return" to load the function to jump to when an exception occurs.

MLton's IR code has four main operations associated with exception handling. These operations are translated into C code and preprocessor macros by the compiler's C codegen. We now discuss these four operations, their meaning, and their corresponding C code.

`SetExnStackLocal` points `exnStack` to a specific slot on the stack, the handler slot. However, in order to maintain the semantics of "return," MLton points to just after

```
Push(+20): F calls G
SetHandler LV_426: G installs a handler
Push(+20): G calls H
Push(-8): H raises an exception
SetExnStackLocal: LV_426 handles the exception
Push(-12): LV_472 returns to G
Push(-20): G Returns to F
```

Fig. 15. Simplified IR exception example showing stack frame advancement (positive pushes) and partial pops (negative pushes) when an exception is raised.

that slot, and so MLton will subtract one word from `exnSlot` before dereferencing it and extracting the exception handler's function number:

```
exnStack = stackTop + handlerOffset - StackBottom;
```

In MLton, a raise will do the following:

```
stackTop = stackBottom + exnStack;
return();
```

This causes control to be transferred to the exception handler function, which then does a `push(-X)`, where X is the distance from the start of the current frame to the slot just above the the handler slot is located within the current frame (`handlerOffset`), causing `stackTop` to be moved back to the start of the current frame. Since `stackTop` is pointing into the middle of a frame, X winds up being smaller than that frame's total size. This is shown in Figure 15. Note that the example shown is greatly simplified to highlight the exception mechanics. Details shown in Figure 14 such as parameter passing, allocation, conditionals, I/O, etc, are not reflected in Figure 15.

In RTMLton, `currentFrame` (the equivalent of `stackTop` in MLton) is a pointer to a chunk of memory, not an offset relative to the bottom of the stack. We felt changing the name to `currentFrame` more clearly indicates its role, whereas `stackTop` in MLton indicates the distance from `stackBottom` to the current frame. That is to say that in MLton, `stackTop` is an an offset from the bottom of the stack, so in order to find the memory location corresponding to the top of the stack, MLton adds the offset to the bottom. In RTMLton, since we employ a stacklet model, pointer arithmetic like that is not possible, and so we directly track the memory location as a pointer.

To record an exception handler in RTMLton, the following code is used:

```
exnStack = currentFrame;
currentFrame->handler = (function number);
```

where the function number is assigned by the compiler and placed in the slot determined by `currentFrame+handlerOffset`. The `SetHandler` operation (discussed below) performs the `currentFrame->handler` assignment step for us. A raise in MLton sets the `stackTop` to point to the slot after the `handler` slot and then uses “return" to load the function number from the `handler` slot and then jumps to that function.

A raise in RTMLton does almost the same thing, except we set the `currentFrame` to the chunk *after* the chunk that contains the `handler`. Since we directly load the function number from the metadata's handler function, we do not use the "return" macro:

```
currentFrame = exnStack->next_chunk;
nextFun = currentFrame->prev_chunk->handler; /* the exception
    handler */
(jump to the function given in nextFun)
```

The reason we do this is to preserve the sequence of operations that occur in MLton when handling an exception. MLton sets stackTop to the slot after the handler slot and then adjusts stackTop downward (via a negative push) so that stackTop winds up pointing to the bottom of the frame that the handler slot is in. The word below stackTop is then Return Adr which can be used to continue execution once the handler has executed. MLton therefore performs two negative pushes: the first is when it sets stackTop above the handler slot and the second is when it sets stackTop to the bottom of that frame. Both of those pushes, in MLton's case, are fractional pushes in the sense that they sum to the frame size. This can be seen on lines 4 and 6 of Figure 15 which, when taken together, are the inverse of line 3 and restore the stackTop pointer to the correct location. In RTMLton's case, since pushes always move forward and backward by one full frame, we are unable to do easily implement these fractional pushes in the manner that MLton does. Instead, we load the handler field from the exception handler frame into nextFun and then set our currentFrame to be one frame *ahead* of the frame that has the handler. Then, the negative push that MLton's codegen emits causes us to move our currentFrame pointer back one frame, to the frame that contains handler, and then the second push that MLton emits causes currentFrame to move back to the previous frame, resulting in currentFrame pointing to same frame as stackTop does in MLton once the exception has been handled.

SetHandler creates an exception handler by placing a label corresponding to the code block that implements the exception handler into the handler slot. It is the inverse of SetExnStackLocal:

```
*(uint *)(stackTop + handlerOffset) = handler;
```

In RTMLton, we additionally store the handler in a dedicated field in the chunk (as discussed previously this is an inefficient duplication that will be addressed in the future). This is not space optimal but allows us to quickly find the handler. The reason for this is because return() (which is called following a raise()) grabs the function to return to by setting nextFun to be *(exnStack - sizeof(void*)). However, our exnStack points to the start of the chunk (frame) where that slot is, but we do not know the specific slot location in that frame at runtime:

```
*(uint *)(stackTop + handlerOffset) = handler;
currentFrame->handler = handler;
```

SetExnStackSlot extracts a function number stored in the link slot of the stack frame. The link slot holds a previous value of exnStack that was swapped out due to a new handler being installed. After the handler is no longer needed, the original exnStack can be restored from the link slot value. This allows for nested exceptions. SetSlotExnStack saves the original exnStack value into the link slot.

MLton executes the following code:

```
exnStack = *(uint *)(stackTop + linkOffset);
```

it takes the value from the `link` stack slot and places it into `gcState->exnStack`.

In RTMLton, `exnStack` is a pointer and not an offset value, so we just load the *pointer* stored in `link` into `exnStack`:

```
exnStack = *(pointer)(currentFrame + linkOffset);
```

`SetSlotExnStack` will stash the current exception stack offset into the `link` slot. This operation is used to install a new exception handler when one is already in place. This operation is the inverse of `SetExnStackSlot` (above):

```
*(uint *)(stackTop + linkOffset) = exnStack;
```

In RTMLton, we stash the `exnStack` chunk pointer into the `link` slot in the current frame:

```
*(pointer)(currentFrame + linkOffset) = (pointer)exnStack;
```

3.4.6 Exception example

In Figure 16, we see a representation of the MLton stack corresponding to the example program in Figure 14. The diagram, and example code, assume no optimizations such as inlining in order to illustrate how exception handling occurs on the stack. The code in Figure 15 is the IR showing some of the operations discussed above. The `SetExnStackSlot` and `SetSlotExnStack` operands are not used because the example only installs one signal handler. In the figures, the numbers in brackets represent byte offsets from the bottom of the stack. Initially, the stack contains only one frame (F), but when F calls G, a new frame is added to the stack, and `stackTop` is moved forward 20 bytes. Since F does not install an exception handler, the slots related to exception handling are unused as indicated by the “XX" values. G, however, installs an exception handler. Since the handler is part of G, MLton take the handler's code and assigns it a label (`LV_426`) so that it can accessed via a goto/jump. This label is recorded in G's `handler` slot. G then calls H, which moves `stackTop` ahead another 20 bytes, resulting in the stack having three frames on it, one per function. H installs no exception handler and so its exception slots are unused. H instead raises an exception. This causes `stackTop` to be adjusted by -8 bytes, so that it points above the `handler` slot. Control then jumps to the code at label `LV_426`; once that code finishes executing, `stackTop` is then adjusted by -12 bytes, which leaves it pointing to the bottom of G's frame. At this point, G returns control to F, which then causes `stackTop` to be adjusted by -20 bytes.

Figure 17 shows the resulting RTMLton stack layout for the same program. The offsets are relative to the start of the chunk, not the bottom of the stack. The RTMLton metadata fields are not given offsets since they are not part of the stack frame. `exnStack` points to the start of the chunk that has the exception handler value in it, not to a particular slot. The exact handler value is in the chunk's metadata area. The only significant change from the MLton figures is that `currentFrame` remains pointing to H's frame after the exception is raised. For clarity, we show the `next_chunk` pointer changing to `NULL` in the figures,

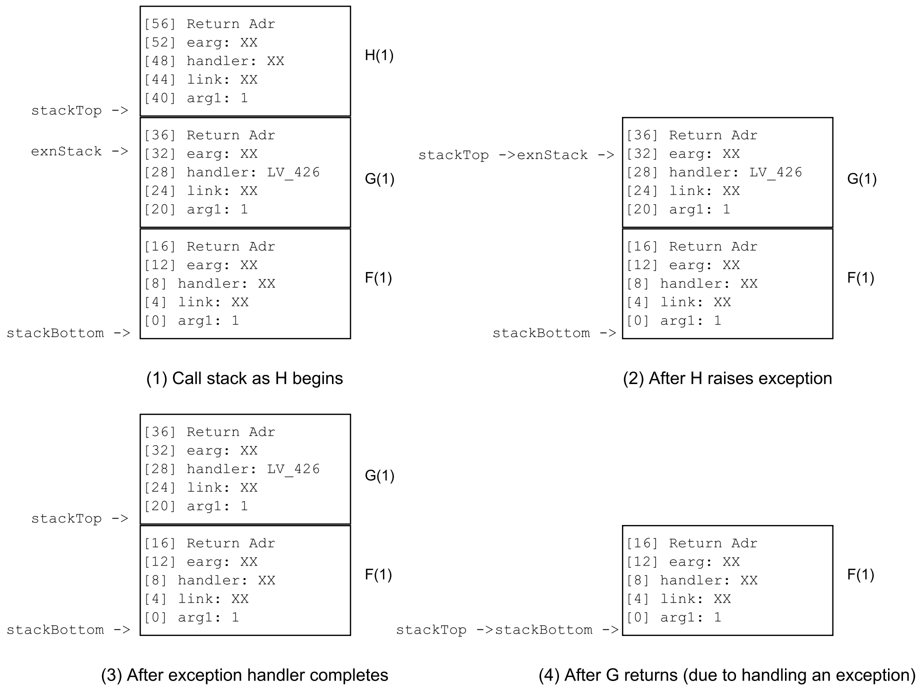

Fig. 16. Exception timeline: MLton showing `exnStack` tracking where the exception handler is installed. `stackTop` points to the `exnStack` location after a `raise`.

but the actual implementation does not immediately unlink the unused chunks in order to avoid over working the GC during function calls and returns as mentioned in Section 3.4.

3.4.7 Discussion

While we are less space efficient, access time is consistent with MLton. Both MLton and RTMLton must check for overflow in order to grow the stack, but in our worst case, growth is a matter of linking additional chunks to the existing stack, whereas MLton's worst case involves obtaining a memory segment large enough to accommodate the growth (which might lead to a GC compaction phase) and then making a copy of the stack. For exceptions, we examined if there was overhead introduced by our changes. The expectation is that there is no difference between returning a value from a function or returning a value via the exception mechanism, since both involve following a single pointer to locate the frame to which we are returning. We used MLton's `exn` regression and measured the time it took to throw 1000 exceptions, we then modified it to return values instead of using exceptions and repeated the measurements. We found the runtime, averaged over 100 executions, to be the same (1.69 seconds). The standard deviation for the exception application was 0.014 seconds and for the non-exception application it was 0.0099 seconds.

3.5 Generalizing our solution to other languages

All of our efforts in this paper were directed toward taking an optimized compiler for a functional language (MLton) and making changes to ensure that we have a predictable

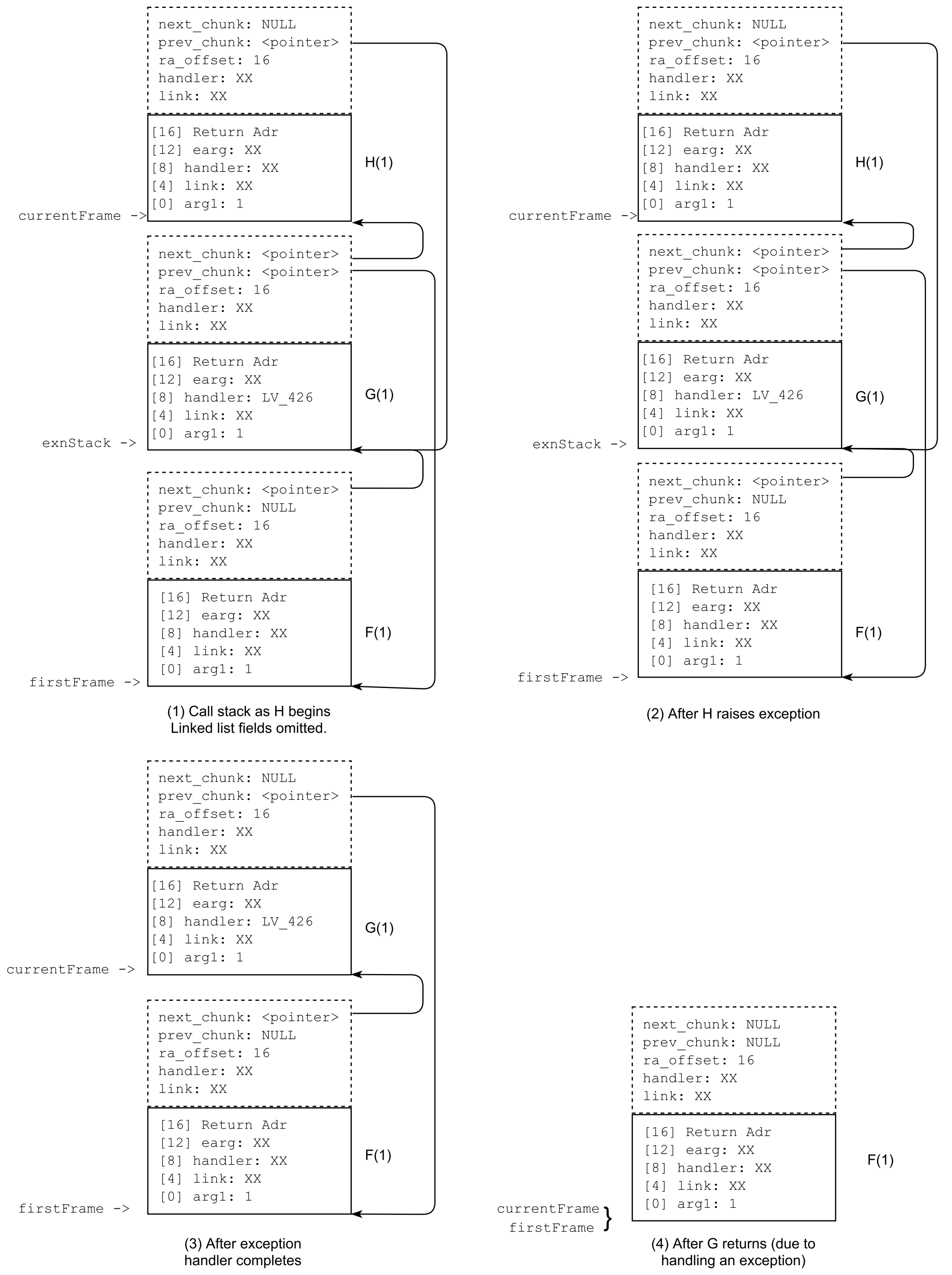

Fig. 17. Exception timeline: RTMLton sets `exnStack` to the memory chunk that holds the exception handler. `currentFrame` does not point to `exnStack` as one might expect.

runtime which then simplifies the building of real-time systems. This approach can be applied to any similar language which may have beneficial properties (like a type system that helps in expression of real-time systems) but does not have a predictable runtime system. A chunked object model helps in building a heap which can be efficiently utilized without requiring the GC to move around objects to create space when there is memory pressure. This chunking can be realized by splitting the heap into fixed size chunks at the start and then allocating objects into these chunks, as we do when we allocate MLton objects into chunk payloads in RTMLton. We find that this is an effective way to provide an abstraction over the compiler's existing object model, without having to redefine it and potentially lose any optimizations that it may already perform. Moreover, this opens up the potential to further optimize object allocations when coupled with packing strategies that pack objects into chunks. However, the GC must now be modified to work with these chunks as opposed to the original objects allocated.

All managed languages specify GC safepoints, which are safe locations in code where a garbage collection can occur. Typically, these are added on function entry/return, back-edges of loops, and before suspending a thread, to name a few. Each language implementation chooses how to add these safepoints so as to allow the garbage collection to proceed without having to wait a long time until the next safepoint is reached. Our memory reservation mechanism can be added as an additional check at an existing safepoint or as a new safepoint and can provide guarantees on allocation granularities. For example, if the reservation mechanism is inserted at every function entry, we can guarantee that all the allocation requests in that function are satisfied.

Languages which only support a single OS thread are not fully disqualified from use in building real-time systems. However, they restrict the type of real-time systems that can be defined using them. For example, if the system comprises just one periodic thread, such a system can be specified in a language with a single computation model (a real-time memory management strategy is still necessary). Such systems can even be run on bare metal without the necessity of an RTOS when packaged as a cyclic executive (Baker & Shaw, 1988). However, not all real-time systems can be reduced to such a loop and when it comes to defining real-time systems which cannot be specified as cyclic executives, you need to have a scheduler and a schedulability analysis. One can choose to build a scheduler at the language level to define various tasks and switch between them, but handling other OS constructs like timers and interrupts at that level to ensure these tasks perform as intended is a difficult venture. In fact, moving from such a model to one using the RTOS to provide all the lower-level constructs is more appealing as we can leverage existing features, and it is unclear what the benefit of doing this at the language level would be.

Compilers like MLton perform many optimizations assuming a single computation model. When moving to a multi-threaded model in such languages, one needs to deal with shared resources in a thread-safe manner. We chose to use thread-local copies of much of the global state and minimize the use of locks as much as possible. In light of the effort required to migrate the compiler's runtime to support OS threads, existing multicore versions of the compiler might seem like a better substrate to implement a real-time GC. For example, Multi-MLton already has the necessary framework to support multiple OS threads. However, as identified in Section 3.1, Multi-MLton is built to work best on hardware that has a lot of cores. While embedded boards used for real-time systems do contain multiple cores, they are nowhere near the number required to make Multi-MLton a viable substrate. In order to build an RTGC that explores different GC scheduling strategies, we would still need to place the GC in its own OS thread. Since Multi-MLton has a per-thread heap, the design of a GC that manages many heaps would be much more complicated and would make a schedulability analysis in Multi-MLton harder. Multi-MLton supports only message passing communication between threads. Shared heap inter-process communication primitives are much more expensive in Multi-MLton due to the overhead involved in shared data access. On the other hand, a shared heap implementation like MLton gives us the flexibility of choosing to explore either shared heap communication or a message passing communication (using CML libraries). MLton's WPO compilation strategy and elegant support for green threads make it an interesting target for real-time systems, and the multicore variant, although not an ideal choice in our case, provides insight into how a single-threaded implementation can be expanded to support multiple threads. The same reasoning applies to other language implementations with runtime architectures similar to MLton.

4 Evaluation

In order to ensure that the changes we have described in this paper translates to expected behavior in terms of predictability of applications, we evaluate how RTMLton fares as compared to MLton, when it is used to run a real-time application. Although our focus has been on achieving predictability, we provide raw performance numbers of RTMLton and MLton to give a more holistic view of the system.

4.1 Completeness and Performance versus MLton

Our current implementation of RTMLton supports the majority of MLton features. However, there are a number of notable exceptions. Our implementation currently does not support infinite precision numbers, weak objects, signal handling, finalizers, booting from a cached "world" and is limited to 32-bit deployments. As a measure of completeness, RTMLton passes 75% of the MLton regression suite, 10% of the failures represent unsupported features, and 15% represent known bugs still to be resolved.

Although predictability is our primary concern, we have also tested the performance of RTMLton versus MLton. Table 1 shows the performance of RTMLton versus MLton on some computationally intensive programs. The slow down column indicates the ratio of the runtime of a benchmark test, compiled with MLton versus RTMLton. A slow down value more than 1 indicates that RTMLton is slower than MLton and a value less than 1 indicates it is faster than MLton. We see comparable performance on most benchmarks but notice performance slow downs in primarily two cases: (1) programs that require deep stacks (Fib, tak) and (2) programs that leverage arrays heavily (Flat Array and Matrix). Tests (barnes hut) that use both deep stacks and large arrays are affected depending on how extensively they use these constructs. We expect to be able to address this overhead by adjusting our array offset method to skip tree traversal in the common linear traversal case as discussed in Section 3.3.3. MLton exhibits a higher variance in performance for the Mandelbrot test runs when compared to RTMLton. The overall result is that MLton is marginally slower than RTMLton. While there is no obvious difference in how the code is executed in both systems, we notice MLton's GC copying objects. This benchmark is not memory intensive, but the GC is invoked for increasing the stack size and performing a minor GC based on heuristics. The calls to the GC result in MLton's high variance compared to RTMLton. In a run where MLton's GC heuristics are not triggered,

Table 1. Performance of MLton and RTMLton on computationally intensive programs. Each benchmark was run five times. The slow down column depicts the ratio of runtimes of MLton and RTMLton

Test name	MLton (s)	RTMLton (s)	Slow down
Fib	15.53	34.16	2.19
TailFib	0.28	0.31	1.10
Mandelbrot	19.96	18.59	0.93
MD5	7.22	13.40	1.85
BarnesHut	0.00038	0.0019	5.00
Even-Odd	7.72	10.23	1.32
Flat Array	0.014	1.41	100.00
Imp-For	0.046	0.048	1.04
Peek	0.012	0.012	1.00
Psdes-Random	1.97	7.98	4.05
Tak	9.24	33.21	3.59
Matrix	0.33	99.72	302.18

performance between RTMLton and MLton are similar. Raw performance, however, is not how real-time systems are evaluated. Predictability is paramount in the system, and overheads, as long as they can be accounted for, are acceptable if the system can meet its target deadlines.

4.2 Predictability

We evaluate the predictability of RTMLton on an SML port of a real-time benchmark, the CD$_x$ introduced in Kalibera *et al.* (2009*b*). CD$_x$ is an airspace analysis algorithm that detects potential collisions between aircrafts based on simulated radar frames and is used to evaluate the performance of C- and Java-based real-time systems. CD$_x$ consists of two main parts, namely the air traffic simulator (ATS) and the collision detector (CD). The ATS generates radar frames, which contain important information about aircraft, such as their callsign and position in 3D space. The ATS produces a user-defined number of frames. The CD analyzes frames periodically and it detects a collision in a given frame whenever the distance between any two aircrafts is smaller than a predefined proximity radius. The algorithm for detecting collisions is given in detail in Kalibera *et al.* (2009*b*). The CD performs complex mathematical computations to discover potential collisions and benchmarks various properties of the system like deadline misses and response time for operation. CD processes frames differently based on how far apart planes are in the frames. It does a simple 2D analysis when planes are further away and does a more complicated 3D calculation of relative positions when a collision is imminent. At its core, the benchmark is a single periodic task that repeats the collision detection algorithm over the subsequent radar frames.

We run the CD$_x$ benchmark using a deadline of 50 ms for the CD task and leverage a workload that has heavy collisions. We measure the computation time for each release of the CD thread, gather numbers over 2000 releases of the CD thread, and graph out the

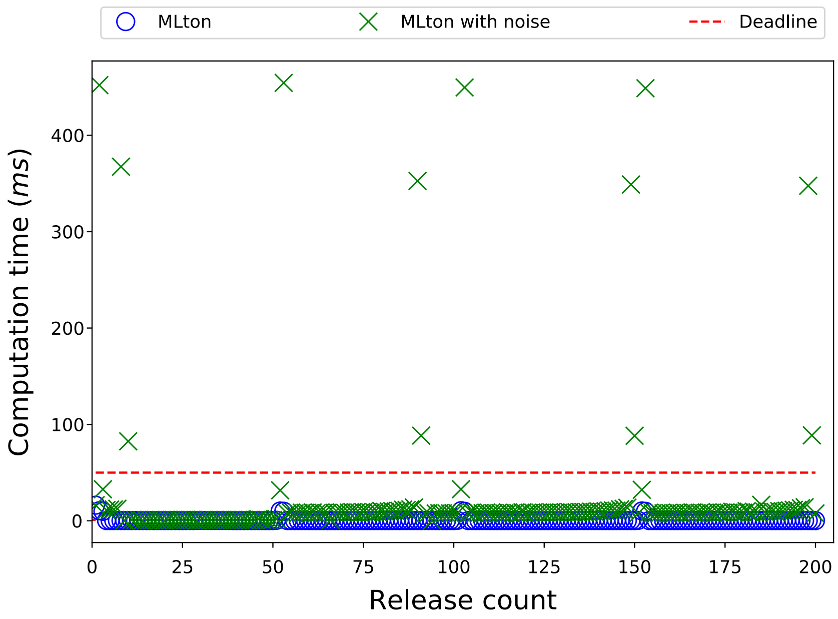

Fig. 18. Performance of MLton on CD_x. The red line marks the deadline for the task according to system requirements. The green X and blue circle signify the computation time for each release of MLton with and without noise, respectively. Many releases of the CD task miss the deadline.

distribution of the computation times. These computation times are then compared with the deadline for the CD task. For readability, we highlight a representative 200 releases. To measure the predictability of each system, we rerun the same benchmark with a noise-making thread, which runs a computation that allocates objects on the same heap as the CD thread. In RTMLton, the noise-making thread is executed in a separate POSIX thread which allows the OS real-time scheduler to schedule threads preemptively and based on their priority. In MLton, the noise-making thread is just a green thread that is scheduled non-preemptively (cooperatively) with the CD thread. Thus, in MLton, all jitter in the numbers is isolated to the runtime itself as the noise-making thread can never interrupt the computation of the CD thread. If the noise-making thread would be scheduled preemptively, the jitter would increase further since MLton does not have a priority mechanism for threads. All benchmarks are run on an Intel i7-3770 (3.4GHz) machine with 16GB of RAM running 32-bit Ubuntu Linux (16.04) with RT-Kernel 4.14.87.

We expect RTMLton to perform more predictably than MLton under memory pressure as the RTMLton GC is concurrent and preemptive. Figures 18 and 19 show the results of running the benchmark on RTMLton and MLton, respectively. As expected, the computation time in RTMLton does not exceed the deadline even when the noise thread is running but does exhibit overhead compared to MLton as we saw in the regular benchmarks. In RTMLton, the computation time with the noise thread is a little more than without noise due to the increase in frequency of the CD thread having to mark its own stack, but it is never exceeds the task deadline of 50 ms. When used with a scheduling policy which does incremental GC work, by forcing the mutator to mark its own stack at the end of every

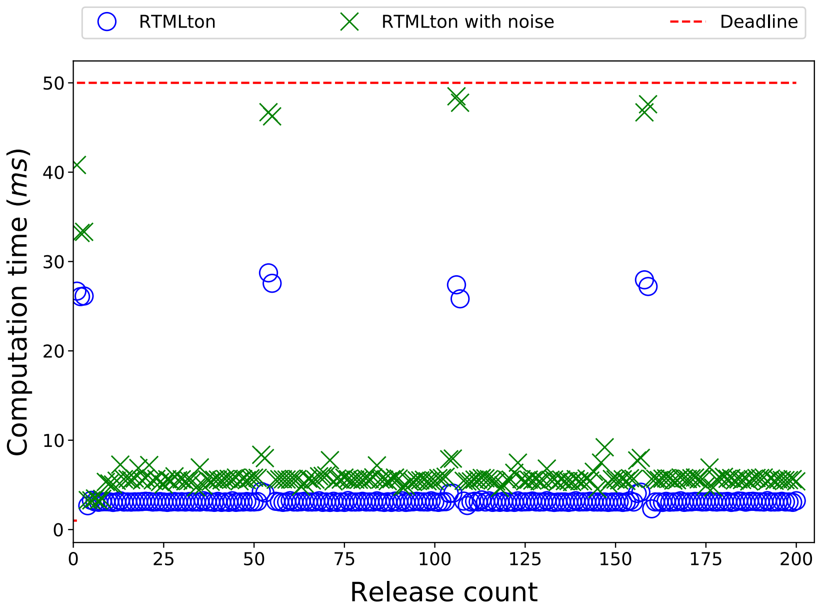

Fig. 19. Performance of RTMLton on CD_x. All releases of the CD task complete within the deadline.

period, we expect the runtime to be more uniform irrespective of noise. We leave exploration of such scheduling policies as part of future work. In the case of MLton, we can see that the computation time varies up to a maximum of over 400 ms, when it has to compact the heap in order to make space for CD to run. Such unpredictability is undesirable and leads to missed deadlines. Worse it causes jitter on subsequent releases. The graphs also show that with no memory pressure, MLton performs better than RTMLton. This is expected as our system does induce overhead for leveraging chunked objects. Similarly, we have not yet modified MLton's aggressive flattening passes to flatten chunked objects. Operations that span over whole arrays are implemented in terms of array random access in MLton's basis library. In MLton's representation, this implementation is fast; accessing each element incurs $O(1)$ cost. But this implementation induces overhead in RTMLton due to $O(\log(n))$ access time to each element. In this case, the logarithmic access time is a trade-off—predictable performance for GC for slower, but still predictable, array access times.[9] Another source of overhead for RTMLton is the per-block GC check and reservation mechanism. In comparison, MLton performs its GC check more conservatively, as discussed in Section 3.3, but crucially relies on a lack of OS-level concurrency for the correctness of this optimized GC check. Figure 19 shows some frames in RTMLton taking a lot more time than the others even under no memory pressure; these computations represent the worst-case performance scenario for RTMLton on the CD_x benchmark as they are computationally more intensive (due to imminent collisions in the frame) and do

[9] Almost all dynamically allocated arrays are small and fit into one chunk making them $O(1)$ access and large arrays are statically allocated and their size known up front, so the $O(\log(n))$ access time can be taken into consideration when validating the system.

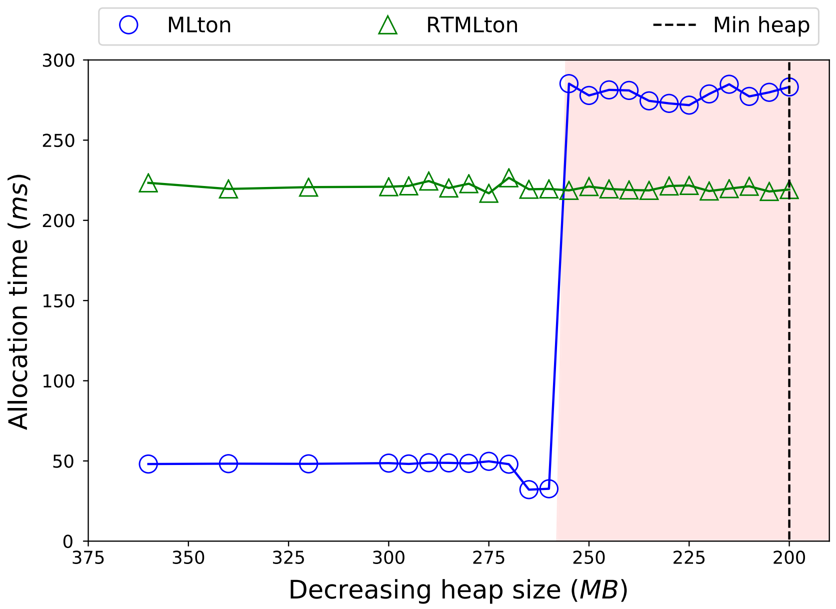

Fig. 20. Fragmentation tolerance of MLton versus RTMLton. As the size of the heap given to the program is reduced, MLton takes drastically longer to allocate objects on a fragmented heap as compared to RTMLton, which is more predictable (takes the same time) in allocatine objects, irrespective of size of the heap.

significantly more allocations as well, thereby increasing the number of times the mutator needs to scan its stack. Although the benchmark triggers the worst case, RTMLton is still able to meet the task deadline for CD.

To better understand the predictability of object allocation in RTMLton, we implemented a classic fragmentation tolerance benchmark. In this test, we allocate a large array of refs (largest size that will fill the minimum heap), deallocate half of it, and then time the allocation of another array which is approximately the size of holes left behind by the deallocated objects. Figure 20 shows that when we move closer to the minimum heap required for the program to run, MLton starts takes a lot more time for allocating on the fragmented heap whereas RTMLton, with its chunked model, is more predictable. Since we are allocating arrays in the fragmentation benchmark, we expect the high initial overhead of RTMLton as multiple heap objects are allocated for every user-defined array since they are chunked. Another reason for the default overhead is because we portray the worst-case scenario for RTMLton by having our mutator scan the stacks on every GC checkpoint, irrespective of memory pressure. Despite these overheads, RTMLton manages to perform predictably when heap space is constricted and limited. MLton, however, is inherently optimized for the average case and so the allocation cost degrades when heap pressure is present. We note that most embedded systems run as close to the minimal heap as possible to maximize utilization of memory. Predictable performance as available heap approaches an application's minimum heap is crucial and is highlighted in the shaded region of the Figure 20.

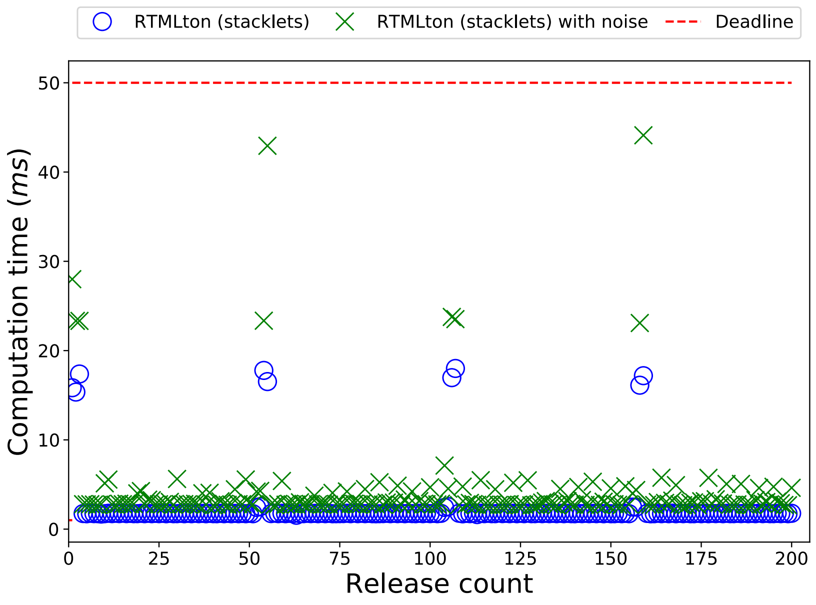

Fig. 21. The performance of RTMLton is not distorted by the addition of stacklets and it is still able to process CD_x frames well within the deadline as seen in Figure 19.

4.3 Impact of stacklets

Addition of stacklets to RTMLton puts the mutator threads in charge of their own stacks. This means they now are responsible for increasing the size of their stacks and also responsible for shrinking them when their utilization is low. Although this model entrusts more work to the mutator, it is better than having the GC do it and incurring additional overheads in the form of synchronization with the GC thread. Our implementation is efficient because the cost of shrinking is linear in the number of frames to reduce, as compared to MLton where shrinking involves procuring (contiguous) memory equal to the size of the new stack and copying the current stack into the new space. Stack growing is also inexpensive as it involves having to get the required chunks from the free list, a constant time operation if they are available. As a trade-off to having an efficient stacklet implementation, we had to bump up the size of the fixed chunks to accommodate the size of the maximum frame. Before stacklets were added, empirical size of arrays decided our chunk size (please see Section 3.3.1) but in order to have one stack frame per chunk, we now need a chunk size that can hold the biggest frame in the program. For the CD_x benchmark that translates to a payload size of 302 bytes per chunk, almost double the previous payload size. One can estimate that this increase in chunk size would also increase memory pressure on the benchmark, but RTMLton is able to manage this overhead. Figure 21 shows that we are still able to process the frames in the CD_x benchmark well within the deadline even with the noise thread allocating an equivalent amount of objects on the heap as in the pre-stacklet version of RTMLton. Future work will include having the compiler determine the maximum frame size at compile time and then tune the chunk size to suit the application. It can also be observed that the overall performance of RTMLton with stacklets in Figure 21 has slightly improved as compared to Figure 19. This can be attributed to a minor optimization that was added, which involved efficiently caching POSIX thread-specific data that avoided multiple calls to `pthread_getspecific()`.

5 Related work

Real-Time Garbage Collection: There are roughly three classes of RTGC: (i) *time-based*, like Bacon *et al.* (2003), where the GC is scheduled as a task in the system, (ii) *slack-based*, like Pizlo *et al.* (2010*b*), where the GC is the lowest priority real-time task and executes in the times between release of higher priority tasks, and (iii) *work-based*, like Siebert (2007), where each allocation triggers an amount of GC work proportional to the allocation request. In each of these RTGC definitions, the overall system designer *must* take into consideration the time requirements to run the RTGC. We currently have adopted a slack-based approach in the context of real-time MLton, though a work-based approach is also worth exploring.

Region-Based Memory Management: This is an alternative to garbage collection where an allocated object is placed in a program-pecified region and the memory is reclaimed by freeing the entire region, thereby reclaiming all the objects within it. Traditional region-based techniques (Ross, 1967; Hanson, 1990; Gay & Aiken, 2001) require programmers to explicitly annotate programs with region annotations, which specify the regions in which objects would be allocated. This places the burden of analyzing which objects go into which regions on the programmer. This leads to the desire to automate region identification with the help of *region inference* techniques. Tofte & Talpin (1997) introduced region inference which performs a static analysis on the un-annotated code to arrive at object lifetimes to identify appropriate locations to introduce region annotations, which would group objects into regions. We envision utilizing such techniques for packing MLton objects into RTMLton chunks as described in Section 3.3.6. Perhaps the closest region-based work to what we envision is that of Hallenberg *et al.* (2002*b*), which combines a Tofte & Talpin (1997) like region inference with a garbage collector. In fact, Elsman & Hallenberg (2020) shows how a region-based memory scheme with support for generational garbage collection can be a benefit to garbage collection, which furthers our motivation to combine these two techniques. However, this work does not focus on making memory management predictable like ours does.

Other languages with real-time capabilities: Our survey of existing functional languages and their real-time adaptability (Murphy *et al.*, 2019) showed us that most languages we reviewed were found to be lacking in at least one of the key areas we identified in order to provide a predictable runtime system. However, some functional DSLs were found to be very suitable for hard real-time applications. DSLs, by their nature, offer a reduced set of language and runtime functionality and so are not suitable for general purpose real-time application development. Also notable are efforts such as the real-time specification for Java (RTSJ) (Gosling & Bollella, 2000) and safety critical Java (SCJ) (Henties *et al.*, 2009), which provide a general purpose approach but burden the developer with having to manage memory directly. For example, both provide definitions for scoped memory (Hamza & Counsell, 2012), a region-based automatic memory management scheme where the developer manages the regions. Deters & Cytron (2002) show how to lessen the burden by automatically discovering how to infer scoped regions. Finally, Tofte & Talpin (1997) apply a region-based memory management approach, while avoiding the use of a GC, in the context of SML.

5.1 Languages with real-time potential

A short summary of some of the languages we found to have potential for use to build real-time systems, with or without modifications, follows:

Atom (Hawkins, 2010) (a derivative of Haskell) can target hard real-time systems but does so by eliminating automatic memory management from the runtime. The DSL models a state machine linked to clock cycles for triggering functions. The schedule is validated at compile time. A limitation to its use in other domains is its lack of dynamic memory allocation (and GC). Atom produces C code that has variables predeclared with a minimal set of types supported to facilitate low-level hardware control and measurement of simple systems. Additionally, and perhaps critically, it does not instrument the scheduler and so does not alert on missed deadlines.

Hume (Hammond *et al.*, 2007) is a language based on concurrent finite-state automata (FSA). The focus of Hume is formal analysis, and so we found that its scheduler is cyclic and memory management is focused on static cost space utilization. However, a notable feature was the inclusion of automatic memory management—Hume uses static analysis to limit space usage. Within the scheduling model, one can set timeouts for computation durations.

Timber (Timber Language, 2008) includes concurrency, strong timing constraints that influence the output of its scheduler, event-driven reactions, and object-oriented modeling. Timber includes a mini POSIX-based RTOS with threading, a garbage collector, and a cyclic executive. The garbage collector allows dynamic allocations with support for basic types in addition to arrays and tuples but is not slack-based and instead executes when heap utilization exceeds a certain threshold. This is not ideal for hard real-time applications.

Erlang (Erlang, 2021) was built with soft real-time applications in mind and so has many real-time features built in. The processes of Erlang already have priorities built into them and they are even scheduled according to priorities. However, it has little or no support for expressing periodic tasks and resorts to timeouts to express periodicity which do not even provide real-time guarantees. There has been a recent project to implement a hard real-time version of Erlang (Nicosia, 2007), which has support for periodic tasks and a real-time scheduler, but does not include any changes to the existing Erlang GC. It acknowledges the fact that actual hard real-time cannot be achieved in Erlang until it has a Schism (Pizlo *et al.*, 2010*b*) like GC.

6 Conclusion

In this paper, we discussed the challenges of bringing real-time systems programming to a functional language and presented the GC-specific implementation challenges we faced while adapting MLton for use on embedded and real-time systems. Specifically, we discussed our chunked model and how it leads to more predictable performance, which is critical for real-time applications, when heap utilization is high. We used CD_x to benchmark the predictability of our system relative to general purpose MLton and show in our evaluation section that our worst-case GC impact is constant which is an important objective to achieve in a real-time language. We observe that while we are slower than generic MLton, it is due to conservative design decisions that can be addressed in future revisions of our system. We believe our biggest contribution in this paper is the integration of a real-time suitable garbage collector into a general purpose, functional language to allow for the targeting of real-time systems.

Acknowledgments

This work has been support in part by National Science Foundation grants: CCF 1749539 and CNS 1823230.

Conflicts of Interest

None.

References

Anderson, B. (2013) Abandoning segmented stacks in Rust. Available at: https://mail.mozilla.org/pipermail/rust-dev/2013-November/006314.html.

Appel, A. W. & MacQueen, D. B. (1987) A standard ML compiler. In *Functional Programming Languages and Computer Architecture*, Kahn, G. (ed). FPCA 1987. Lecture Notes in Computer Science, vol. 274. Berlin, Heidelberg: Springer. https://doi.org/10.1007/3-540-18317-5_17.

Appel, A. W. & Shao, Z. (1996) An empirical and analytic study of stack vs. heap cost for languages with closures. *J. Funct. Program.* **6**, 47–74. https://doi.org/10.1017/S095679680000157X.

Arts, T., Benac Earle, C. & Derrick, J. (2004) Development of a verified Erlang program for resource locking. *Int. J. Softw. Tools Technol. Transf.* **5**(2), 205–220.

Audebaud, P. & Paulin-Mohring, C. (2009) Proofs of randomized algorithms in Coq. *Sci. Comput. Program.* **74**(8), 568–589. Special Issue on Mathematics of Program Construction (MPC 2006).

Bacon, D. F., Cheng, P. & Rajan, V. T. (2003) Controlling fragmentation and space consumption in the Metronome, a real-time garbage collector for Java. In Proceedings of the 2003 ACM SIGPLAN Conf. on Language, Compiler, and Tool for Embedded Systems. LCTES'03. ACM, pp. 81–92.

Baker, T. P. & Shaw, A. (1988) The cyclic executive model and Ada. In Proceedings. Real-Time Systems Symposium, pp. 120–129.

Ballabriga, C., Cassé, H., Rochange, C. & Sainrat, P. (2010) OTAWA: An open toolbox for adaptive WCET analysis. In *Software Technologies for Embedded and Ubiquitous Systems*, Min, S. L., Pettit, R., Puschner, P. & Ungerer, T. (eds). Berlin, Heidelberg: Springer, pp. 35–46.

Bruggeman, C., Waddell, O. & Dybvig, R. K. (1996) Representing control in the presence of one-shot continuations. *SIGPLAN Not.* **31**(5), 99–107.

Cheney, C. J. (1970) A nonrecursive list compacting algorithm. *Commun. ACM* **13**(11), 677–678.

Cheng, P. (2001) *Scalable Real-Time Parallel Garbage Collection for Symmetric Multiprocessors*. PhD thesis, USA. AAI3179039.

Clinger, W. D., Hartheimer, A. H. & Ost, E. M. (1999) Implementation strategies for first-class continuations. *Higher-Order Symb. Comput.* **12**(1), 7–45.

Deters, M. & Cytron, R. K. (2002) Automated discovery of scoped memory regions for real-time Java. In Proceedings of the 3rd Int'l Symposium on Memory Management. ISMM'02. ACM, pp. 132–142.

Dijkstra, E. W., Lamport, L., Martin, A. J., Scholten, C. S. & Steffens, E. F. M. (1978) On-the-fly garbage collection: An exercise in cooperation. *Commun. ACM* **21**(11), 966–975.

Elsman, M. (1999) *Program Modules, Separate Compilation, and Intermodule Optimisation*. DIKU.

Elsman, M. & Hallenberg, N. (2020) On the effects of integrating region-based memory management and generational garbage collection in ML. In *Practical Aspects of Declarative Languages*, Komendantskaya, E. & Liu, Y. A. (eds). Springer International Publishing, pp. 95–112.

Erlang. (2021) Erlang Programming language official website. Available at: http://www.erlang.org/.

Fenichel, R. R. & Yochelson, J. C. (1969) A lisp garbage-collector for virtual-memory computer systems. *Commun. ACM* **12**(11), 611–612.

Gay, D. & Aiken, A. (2001) Language support for regions. *SIGPLAN Not.* **36**(5), 70–80.

Go-Lang. (2013) Contiguous stacks. Available at: https://docs.google.com/document/d/1wAaf1rYoM4S4gtnPh0zOlGzWtrZFQ5suE8qr2sD8uWQ/pub.

Goldstein, S. C. (1997) *Lazy Threads Compiler and Runtime Structures for Fine-Grained Parallel Programming*. PhD thesis, University of California–Berkeley, Berkeley, CA.

Gosling, J. & Bollella, G. (2000) *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc.

Hallenberg, N., Elsman, M. & Tofte, M. (2002a) Combining region inference and garbage collection. In ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02). Berlin, Germany: ACM Press.

Hallenberg, N., Elsman, M. & Tofte, M. (2002b) Combining region inference and garbage collection. In Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation. PLDI'02. ACM, pp. 141–152.

Hammond, K. (2001) The dynamic properties of hume: A functionally-based concurrent language with bounded time and space behaviour. In Implementation of Functional Languages: 12th Int'l Workshop, IFL 2000 Aachen, Germany, 4–7 September 2000 Selected Papers. Berlin, Heidelberg: Springer, pp. 122–139.

Hammond, K. (2003) Is it time for real-time functional programming? In Revised Selected Papers from the Fourth Symposium on Trends in Functional Programming, TFP 2003, Edinburgh, UK, 11–12 September 2003, Gilmore, S. (ed), vol. 4. Intellect, pp. 1–18.

Hammond, K., Michaelson, G. & Pointon, R. (2007) The Hume Report, Version 1.1. Available at: http://www-fp.cs.st-andrews.ac.uk/hume/report/hume-report.pdf.

Hamza, H. & Counsell, S. (2012) Region-based RTSJ memory management: State of the art. *Sci. Comput. Program.* **77**(5), 644–659.

Hanson, D. R. (1990) Fast allocation and deallocation of memory based on object lifetimes. *Software Pract. Exp.* **20**(1), 5–12.

Hawkins, T. (2010) Atom: A Synchronous Hard Real-Time EDSL for GHC. Available at: https://github.com/tomahawkins/atom.

Henties, T., Hunt, J. J., Locke, D., Nilsen, K., Schoeberl, M. & Vitek, J. (2009) Java for safety-critical applications. In 2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert 2009).

Hieb, R., Dybvig, R. K. & Bruggeman, C. (1990) Representing control in the presence of first-class continuations. In Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation. PLDI'90. Association for Computing Machinery, pp. 66–77.

Hughes, J. (1989) Why functional programming matters. *Comput. J.* **32**(2), 98–107.

Jones, R., Hosking, A. & Moss, E. (2016) *The Garbage Collection Handbook: The Art of Automatic Memory Management*. CRC Press.

Kalibera, T., Hagelberg, J., Pizlo, F., Plsek, A., Titzer, B. & Vitek, J. (2009b) CDx: A family of real-time java benchmarks. In Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems. JTRES'09. ACM, pp. 41–50.

Kalibera, T., Pizlo, F., Hosking, A. & Vitek, J. (2009a) Scheduling hard real-time garbage collection. In 30th IEEE Real-Time Systems Symposium, 2009, RTSS 2009, pp. 81–92.

Kumar, R., Myreen, M. O., Norrish, M. & Owens, S. (2014) CakeML: A verified implementation of ML. In Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL'14. ACM, pp. 179–191.

Li, M., McArdle, D. E., Murphy, J. C., Shivkumar, B. & Ziarek, L. (2016) Adding real-time capabilities to a SML compiler. *SIGBED Rev.* **13**(2), 8–13.

Lipari, G. & Bini, E. (2005) A methodology for designing hierarchical scheduling systems. *J. Embedded Comput.* **1**(2), 257–269.

Lisper, B. (2014) SWEET – A tool for WCET flow analysis (extended abstract). In *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications*, Margaria, T. & Steffen, B. (eds). Berlin, Heidelberg: Springer, pp. 482–485.

López, N., Núñez, M. & Rubio, F. (2002) Stochastic process algebras meet Eden. In Proceedings of the Third International Conference on Integrated Formal Methods. IFM'02. Springer-Verlag, pp. 29–48.

Marlow, S., Harris, T., James, R. P. & Peyton Jones, S. (2008) Parallel generational-copying garbage collection with a block-structured heap. In *ISMM'08: Proceedings of the 7th International Symposium on Memory Management*. ACM.

Milner, R., Tofte, M. & Macqueen, D. (1997) *The Definition of Standard ML*. MIT Press.

MLton. (2012) The MLton compiler and runtime system. Available at: `http://www.mlton.org`.

MLton Performance. (2012) MLton performance benchmarks. Available at: `http://mlton.org/Performance`.

Morsing, D. (2014) How Stacks are Handled in Go. Available at: `https://blog.cloudflare.com/how-stacks-are-handled-in-go/`.

Muller, S. K., Acar, U. A. & Harper, R. (2018) Competitive parallelism: Getting your priorities right. *Proc. ACM Program. Lang.* **2**(ICFP).

Murphy, J. C., Shivkumar, B., Pritchard, A., Iraci, G., Kumar, D., Kim, S. H. & Ziarek, L. (2019) A survey of real-time capabilities in functional languages and compilers. *Concurr. Comput. Pract. Exp.* **31**(4), e4902.

Nettles, S. & O'Toole, J. (1993) Real-time replication garbage collection. *SIGPLAN Not.* **28**(6), 217–226.

Nicosia, V. (2007) Towards hard real-time Erlang. In Proceedings of the 2007 SIGPLAN Workshop on ERLANG Workshop. ERLANG'07. ACM, pp. 29–36.

Pizlo, F., Ziarek, L., Blanton, E., Maj, P. & Vitek, J. (2010a) High-level programming of embedded hard real-time devices. In Proceedings of the 5th European Conference on Computer systems. EuroSys'10. ACM, pp. 69–82.

Pizlo, F., Ziarek, L., Maj, P., Hosking, A. L., Blanton, E. & Vitek, J. (2010b) Schism: Fragmentation-tolerant real-time garbage collection. In Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI'10. ACM, pp. 146–159.

RapiTime. (2021) Rapita Systems Ltd homepage. Available at: `https://www.rapitasystems.com/`.

Regehr, J., Reid, A. & Webb, K. (2005) Eliminating stack overflow by abstract interpretation. *ACM Trans. Embed. Comput. Syst.* **4**(4), 751–778.

Reppy, J. H. (1999) *Concurrent Programming in ML*. Cambridge University Press.

Reynolds, J. C. (1972) Definitional interpreters for higher-order programming languages. In Proceedings of the ACM Annual Conference, vol. 2. ACM'72. ACM, pp. 717–740.

Ross, D. T. (1967) The AED free storage package. *Commun. ACM* **10**(8), 481–492.

Shivers, O. (1988) Control flow analysis in scheme. In Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation. PLDI'88. ACM, pp. 164–174.

Shivkumar, B., Murphy, J. C. & Ziarek, L. (2020) RTMLton: An SML runtime for real-time systems. In Practical Aspects of Declarative Languages - 22nd International Symposium, PADL 2020, New Orleans, LA, USA, 20–21 January, 2020, Proceedings, Komendantskaya, E. and Liu, Y. A. (eds). Lecture Notes in Computer Science, vol. 12007. Springer, pp. 113–130.

Siebert, F. (2007) Realtime garbage collection in the JamaicaVM 3.0. In Proceedings of the 5th Int'l Workshop on Java Technologies for Real-time and Embedded Systems. JTRES'07. ACM, pp. 94–103.

Sivaramakrishnan, K. C., Ziarek, L. & Jagannathan, S. (2014) MultiMLton: A multicore-aware runtime for standard ML. *J. Funct. Program.* **24**, 613–674.

Sivaramakrishnan, K., Ziarek, L. & Jagannathan, S. (2012) Eliminating read barriers through procrastination and cleanliness. In Proceedings of the 2012 Int'l Symposium on Memory Management. ISMM'12. ACM, pp. 49–60.

Steele, G. L. (1978) *Rabbit: A Compiler for Scheme*. Tech. rept. USA.

Timber Language. (2008) Timber: A Gentle Introduction. Available at: http://www.timber-lang.org/index_gentle.html.

Tofte, M. & Talpin, J.-P. (1997) Region-based memory management. *Inf. Comput.* **132**(2), 109–176.

Tolmach, A. & Oliva, D. P. (1993) From ML to Ada: Strongly-typed language interoperability via source translation. *J. Funct. Program.* **8**, 367–412.

Wan, Z., Taha, W. & Hudak, P. (2001) Real-time FRP. In Proceedings of the Sixth ACM SIGPLAN Int'l Conference on Functional Programming. ICFP'01. ACM, pp. 146–156.

Ziarek, L., Sivaramakrishnan, K. & Jagannathan, S. (2011) Composable asynchronous events. In Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI'11. ACM, pp. 628–639.

Appendices A The global state structure `GC_state`

```
struct GC_state {
  /* These fields are at the front because they are the most commonly
   * referenced, and having them at smaller offsets may decrease code
   * size and improve cache performance.
   */
  pointer frontier; /* heap.start <= frontier < limit */
  pointer limit; /* limit = heap.start + heap.size */
  /* stackTop : Top of stack in current thread.
   * stackLimit :stackBottom + stackSize - maxFrameSize
   * stackBottom : Bottom of stack in current thread.*/
  pointer stackTop;          \
  pointer stackLimit;         }  =>currentFrame[MAXPRI]
  pointer stackBottom;       /
  size_t exnStack; => exnStack[MAXPRI]
  /* Alphabetized fields follow. */
  size_t alignment;
  bool amInGC;
  bool amOriginal;
  /* Initial @MLton args, processed before command line. */
  char **atMLtons;
  int atMLtonsLength;
  uint32_t atomicState;
  /* Handler for exported C calls (in heap). */
  objptr callFromCHandlerThread; => callFromCHandlerThread[MAXPRI]
  struct GC_callStackState callStackState;
  bool canMinor; /* TRUE iff there is space for a minor gc. */
  struct GC_controls controls;
  struct GC_cumulativeStatistics cumulativeStatistics;
   /* Currently executing thread (in heap). */
  objptr currentThread; => currentThread[MAXPRI]
  struct GC_forwardState forwardState;
  GC_frameLayout frameLayouts; /* Array of frame layouts. */
  uint32_t frameLayoutsLength; /* Cardinality of frameLayouts array. */
  struct GC_generationalMaps generationalMaps;
  objptr *globals;
  uint32_t globalsLength;
  bool hashConsDuringGC;
  struct GC_heap heap; => struct GC_UM_heap umheap (chunked heap)
  struct GC_lastMajorStatistics lastMajorStatistics;
  pointer limitPlusSlop; /* limit + GC_HEAP_LIMIT_SLOP */
  int (*loadGlobals)(FILE *f); /* loads the globals from the file. */
  uint32_t magic; /* The magic number for this executable. */
  uint32_t maxFrameSize;
  bool mutatorMarksCards;
  GC_objectHashTable objectHashTable;
  GC_objectType objectTypes; /* Array of object types. */
  uint32_t objectTypesLength; /* Cardinality of objectTypes array. */
  struct GC_profiling profiling;
  GC_frameIndex (*returnAddressToFrameIndex) (GC_returnAddress ra);
  objptr savedThread; => savedThread[MAXPRI]
  int (*saveGlobals)(FILE *f); /* saves the globals to the file. */
  bool saveWorldStatus;
  struct GC_heap secondaryHeap; /* Used for major copying collection. */
  /* Handler for signals (in heap). */
  objptr signalHandlerThread; => signalHandlerThread[MAXPRI]
  struct GC_signalsInfo signalsInfo;
  struct GC_sourceMaps sourceMaps;
  struct GC_sysvals sysvals;
  struct GC_translateState translateState;
  struct GC_vectorInit *vectorInits;
  uint32_t vectorInitsLength;
  GC_weak weaks; /* Linked list of (live) weak pointers */

   Other RTMLton specific additions (omitted for brevity) .....
};
```