RTMLton : An SML runtime for real-time systems

Bhargav Shivkumar[0000−0002−8430−9229], Jeffrey Murphy, and Lukasz Ziarek

SUNY - University at Buffalo , New York, USA
`https://ubmltongroup.github.io/`

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 in the embedded and real-time domains. Specifically we detail our experiences in modifying MLton, a whole program, optimizing compiler for Standard ML, for use in such contexts. We focus primarily on the language runtime, re-working the threading subsystem and garbage collector. We provide preliminary results over a radar-based aircraft collision detector ported to SML.

Keywords: Real-time systems · Predictable GC · Functional programming.

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 [13, 7, 25, 8]. Recent advances in program verification [2, 12] and formal methods [1, 14] 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 and bugs from being expressed, such as buffer under/over flow and null pointer dereferencing. Programmers can thus produce higher fidelity code with lower programmer effort [**?**]. Additionally, functional programming languages are 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 multi-core, advances in parallel and concurrent programming models and language implementations for functional languages are also appealing as lack of mutable state often results in simpler reasoning about concurrency and parallelism.

There are, however many challenges that need to be addressed prior to being able to leverage a functional language for developing a real-time system. 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, to name a few [8]. We surveyed the current state of the art of functional languages and their suitability for developing real-time systems [17], by assessing metrics like the predictability of the language runtime, threading and concurrency support, as well as the ability for the programmer to express real-time constraints. We observed that all of the languages exhibited unpredictable behavior once competition for resources was introduced, specifically in their runtime architectures. The major challenges 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) capable of executing real-time applications [15]. We use MLton [16], 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 non-moving real-time garbage collector with a reservation mechanism. We leverage our previous experience with Multi-MLton [22] and the Fiji real-time virtual machine [19] 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 built using this system on RT-Linux, a real-time operating system. We present performance measurements, indicating the viability of our prototype, which is publicly available for download at: `https://github.com/UBMLtonGroup`. This paper is an extension of our previous short workshop paper [13], to which we have 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 real-time GC algorithm. We present additional benchmarks, including a full evaluation of our system on a radar based aircraft collision detector.

2 MLton Architecture and Consequences for Real-time

MLton is an open-source, whole-program optimizing SML compiler that generates very efficient executables. 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 MLton Threads

MLton compiled programs consist of only a single OS level thread, over which many green threads are multiplexed. There is a set of three process-wide stack pointers, distinct from the system stack and stored in a monolithic global structure called `GC_State`, that 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.

When a thread is paused, the amount of stack space in use is saved from the current process-wide stack to the thread's stack structure (the other two fields are essentially constants and would only change if the stack was 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. 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. However, this comes at a cost when trying to move to a parallel implementation. The thread runtime semantics are deeply embedded into the compiler and many assumptions are made that are unsafe in a parallel architecture.

MLton provides a logical ready queue from which the next runnable thread is accessed by the scheduler. This is a regular FIFO queue with no notion of priority, however the structure is implicit, relying on continuation chaining and is embedded in the thread switching code itself. This means that there is no single data structure that governs threads nor is there an explicit scheduler. Threading and concurrency libraries (e.g. CML and ACML) build on top of the MLton threading primitives, therefore, introduce their own threading primitives, scheduler, policy, as well as 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.

Consequences for Real-time The concept of prioritization is useful for ensuring high priority tasks execute accordingly. When adding prioritization to thread scheduling, one approach is to utilize the underlying OS for scheduling. However, as noted in the section above, mapping pools of green threads to OS threads leads to concurrency issues due to MLton's use of a single global structure for state tracking. Another approach is to implement prioritization at the green thread layer. This is not preferable for two reasons. First, there is no notion of pre-emption at the green thread layer. As noted above, MLton threads are essentially chained continuations, and so a thread switch is entirely at the discretion of the currently running thread. While one might argue that this could open the way to the compiler generating a very finely calculated schedule, it would also lead to unacceptable pauses due to I/O.

For example, if syntax is available for specifying timing constraints, then a predetermined (and validated) schedule can be generated [23], obviating the need for specifying priorities. However, if one of the green threads in the schedule attempts I/O, the underlying OS would pause the entire process until the I/O completes. Therefore, we believe that it is necessary for the compiler's runtime to offer a clean, and safe, mapping of green threads to OS threads so that, for example, I/O operations can be isolated onto a separate OS thread without affecting high priority computations.

2.2 GC Architecture

MLton adopts a hybrid garbage collector that calls upon the runtime memory utilization to decide the strategy it needs to use for collection. All SML objects are allocated in a contiguous heap. All objects are initially allocated in the Nursery section of the heap in bump pointer fashion until the nursery runs out of space, 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 [**?**]. A minor GC copies objects from nursery to the beginning of the To space (i.e. appending to end of old generation) thus increasing the old generation size and reducing To space and nursery size. When there is no space in the nursery to allocate a new object, a major GC is triggered. It is worthy to note that when there is no memory pressure, the To space is zero size and old generation has the objects that have survived a collection. Therefore, the generational GC isn't triggered until the memory utilization is fairly large, but the garbage collector can still be called for various other tasks like growing the stack.

Major garbage collection is performed in one of the two strategies. If there is enough space to allocate a new heap, the same size of the current heap, then a Cheney copy GC is performed. If there is not enough space for the second semi space, a mark compact GC 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 the minor GC for subsequent collections, until it again needs to call a major GC.

MLton's GC architecture is one that implements a "stop the world" (STW) approach. In this approach, all computation threads pause while the garbage collector runs. This design decision was made keeping in mind the single computation model of MLton, that the heap is more prone to corruption if multiple threads are accessing the heap when the GC is copying objects or doing a compaction. 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 ML objects: Normal (fixed size) objects, weak objects, arrays and stacks. The arrays and stacks are generally allocated in the old generation 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 old generation based on their longevity.

Consequences for Real-time 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. Pre-empting the GC when it runs out of time could make it real-time compatible but it will be useless as it could not be guaranteed that collection would always complete. This could be addressed by implementing incremental collection [18] strategies, but would require the GC to run as a separate thread which is contrary to MLton's single threaded computation model. 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.

3 Real-time Extensions to MLton

To create a version of MLton that supports a real-time computation, we must address the limitations described in the 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, and redesigning the GC to be real-time aware.

3.1 Concurrency and Threading

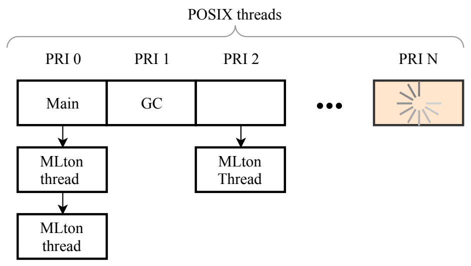

Fig. 1: Priority Based OS/Green Thread Relationship Model

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. More over, to support real-time execution we also must split green threads based on their priority. In the most simple case there exists at most one green thread (computation) for any given priority supported by the system [1]. Figure 1 shows our concurrency model. An OS-level thread is created for each priority the system supports. Currently we expose only the priorities that the underlying OS or Real-time Operating System (RTOS) expose.

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 numerous fields that store the current position of frontier, current executing green thread, current StackTop/StackBottom among others and all these values are accessed at any time by offsetting a pointer to this structure. The decision to use one single structure for storing all the global state was to make the access fast by caching the entire structure on a register. 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 differentiate between 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've decided to keep `GC_State` as a single structure, but implement arrays within it where appropriate. This allows us to be more efficient when it comes to memory utilization – an important consideration when targeting embedded systems. For example, finding the current green thread running within the OS thread, we would refer to the index `GC_State->currentThread[osthreadnumber]`.

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

3.2 Creating a Real-time GC

To implement concurrent GC, it is necessary to have the garbage collector execute in its own thread so that it can work independently to mutator threads (program threads). Multi-core implementations of SML like MultiMLton 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 pulling out 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 it requires. Although the 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 is fully reified, the results would be unpredictable. MLton also will write into a newly created stack frame before finalizing and recording the size of the frame. Without the identification of safe points to pause the threads, the heap will be malformed with potentially live objected considered dead. Fortunately, MLton identifies these safe points for us. GC safe points in MLton are points in code where it is safe for the thread running the code to pause allowing the GC to scan stacks.

Although GC safe points are pre-identified for us, the code generated by the compiler assumes a single threaded model and so we found problematic constructs such as global variables and reliance on caching important pointers in registers for performance. We needed to rework these architectural decisions. 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.

Handling Fragmentation The design of a real-time garbage collector should ensure predictability. To eliminate GC work induced by defragmentation and compacting the heap, 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 [20]. This chunked heap is managed by a free list.

Normal and weak objects are represented as linked lists. 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. Arrays are represented as trees, in which each node is fixed-size. Internal nodes have a large number of branches (32 in our implementation), which keeps access time $log_{32}(n)$ and is close to constant. MLton constantly allocates small sized arrays and even zero sized arrays. We represent such arrays as a single leaf to eliminate the overhead of finding the immediate child of the root. 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.

Heap Layout: In MLton, the size of normal objects, arrays and stacks vary significantly. Since one objective of a unified chunked heap is to prevent moving during GC, we need to have all chunks be of the same size. This does lead to space wastage in each chunk as object sizes vary. However, this opens up room for potential optimizations where we can further explore packing of multiple MLton objects into chunks either based on object sizes or their lifetimes, making the GC much more efficient.

Object Layout: MLton already tries to pack small objects into larger ones. In our empirical study, most normal MLton objects are around 24 bytes and arrays are close to 128 bytes. We choose 154 bytes as the chunk payload that carries MLton object along with an extra 12 bytes overhead associated with normal object chunk management and an extra 56 bytes to manage array chunks. Normal objects that are larger than 154 bytes are split into multiple chunks. In our current implementation, we limit normal objects to two chunks each even though we haven't noticed objects that are greater than 64 bytes. The object layout is depicted in Figure 2.

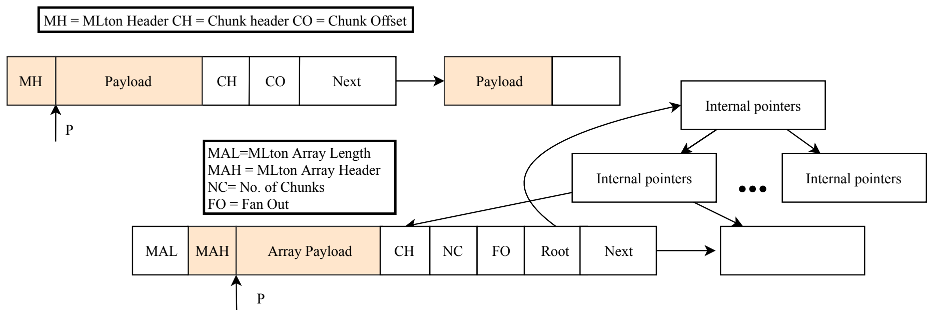

Fig. 2: Chunked object layout

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. We stick to this representation as much as possible. Array nodes are represented in Figure 2. Internal nodes carry 32 pointers to their children. We pass an array around via a pointer to its first leaf. A root pointer and a next pointer is embedded in the leaf node. The leaf pointer connects all leaves that actually carry payloads for potential linear traversal optimization. For an array that is 128 bytes or less, we can fit it into 1 leaf chunk. For arrays that span multiple chunks, we construct trees. When accessing an element of an array, we first follow the root pointer to retrieve the root node and then access the array in a top down manner, in which we determine the branch in current node by `index % FO`, then we follow the branch to an alternative internal node. The process is repeated until we finally arrive at a leaf.

Array Limitations Flattening refers to the multiple optimization passes in MLton that reduces the overhead for accessing nested objects. 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 totally compatible with our implementation.

GC model For collection, our concurrent GC leverages a traditional non-moving, mark and sweep scheme with a Dijkstra's incremental update write barrier [5]. It is needless to say that our GC thread runs on its own OS thread and operates independently of the mutator, repeating the steps below. Each loop signifies a *GC Cycle* :
Wait for synchronization- In this phase, the GC is waiting for all the mutator threads to synchronize at the GC checkpoints so that it can continue with its work in a safe manner. MLton performs complicated data flow and control flow analysis to insert GC checkpoints to minimize the number of garbage collections needed. However, the data flow and control flow analysis assumes a single heap model and objects are calculated by number of bytes required (and not chunks), which is incompatible with our model. One solution is to patch up each path in the GC check flow, redirecting all GC checks to our GC runtime function and let the C runtime function decide whether a garbage collection is needed. This method has high overheads in the form of preparing the code to jump to a C call which involves having to save all the temporaries currently live onto the stack as local variables and adding a `C_FRAME` marker onto the stack all of which not only increases the stack size but also affects overall runtime of the program. In RTMLton, we currently add an optimization pass (*gc-check*) which sums up the allocations in a block and inserts a check to see if there are adequate chunks left. If the 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, MLton ensures the stack is completely prepared before jumping into `GC_arrayAllocate`. We can thus safely make GC checks in the array allocation.

Currently, each thread walks its own stacks and marks all 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) and 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 eventually collected. This model where each thread scans only the root set from the stacks and the GC scans the rest and sweeps concurrently, is different from that of MLton's monolithic GC model, in that the mutator doesn't have to wait till the entire heap is scanned. By having each thread scan its own stack, at the end of its period, also contributes to making the GC work incrementally which would give good mutator performance. When all the mutators have finished marking their own stacks at their GC Checkpoints they set a bit to indicate that they have synced and 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 reachable objects without wrongfully collecting objects 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 some small book keeping 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 it available for reuse almost instantly.

Cleanup and book keeping- Before the GC goes back to waiting for synchronization phase, it does some clean up and book keeping like 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 initially to scan its own stack but when the memory is very constrained it may so happen that the mutator does not have enough free chunks to satisfy its allocation requests. Ideally, RTGCs rely on efficient scheduling policies to ensure that the GC runs enough to make sure these scenarios are avoided, but in the absence of such policies we currently block the mutator if it doesn't have enough chunks free and the GC is running. The GC decides to die with an *Insufficient memory* message when it has made no progress(all mutators are blocked) in 2 consecutive GC cycles.

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. When an allocation is done, the allocated objects are pushed into stack slots if the transfer out of the code block has the potential to invoke the GC, failing which may result in the newly allocated object being wrongfully collected. In vanilla MLton, GC can be invoked only from GC safepoints, which ensure that the allocated objects are pushed into appropriate stack slots before the GC runs. In RTMLton however there are two possible places where the GC can

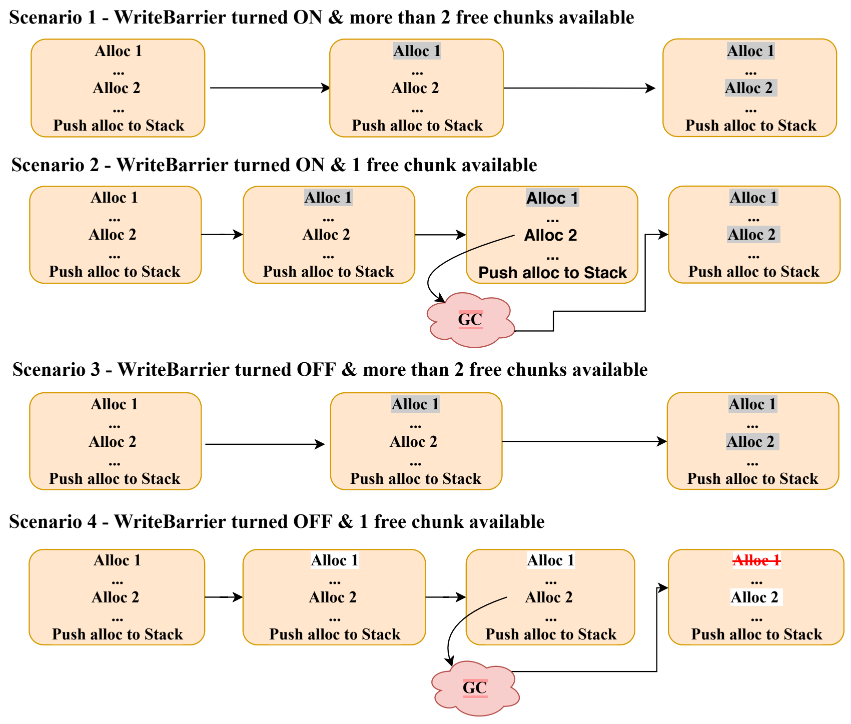
Scenario 1 - WriteBarrier turned ON & more than 2 free chunks available
Alloc 1
...
Alloc 2
...
Push alloc to Stack
Scenario 2 - WriteBarrier turned ON & 1 free chunk available
GC
Scenario 3 - WriteBarrier turned OFF & more than 2 free chunks available
Scenario 4 - WriteBarrier turned OFF & 1 free chunk available
~~Alloc 1~~

Fig. 3: Allocation scenarios

be invoked: One, at GC safepoints and two, during allocation when there are no free memory chunks available.

When the GC is invoked at the point of allocation in RTMLton, it leads to an edge case where any previously allocated chunk might be wrongfully collected in very tight memory scenarios, because they were not pushed into stack slots. An allocation statement is not a transfer in MLton's design and therefore it does not expect a GC to happen at that point. Consider the scenarios in Figure 3:

Scenario 1 and 2 show the cases when the GC is running (i.e. write barrier is turned ON) and Scenario 3 and 4 show the cases when the GC is not running (i.e. write barrier turned OFF) . In Scenario 1, you have a code block with 2 sequential allocations and there are more than 2 free chunks available. Since the write barrier is ON, both objects are allocated gray and since there is more than 2 free chunks available, no GC is triggered. In Scenario 2, there is just 1 free chunk available. So the first object is allocated gray and when the second allocation happens, the GC is triggered. But since the GC was already running, the first object in shaded and is not wrongfully collected by the GC. Scenario 3 shows the execution when there are more than 2 free memory chunks available. Since the write barrier is turned off, both objects are allocated white. Since there are enough memory chunks available, GC isn't triggered and execution completes normally. When there is only one free chunk available in memory, it leads to the case as in Scenario 4. Since the write barrier is turned OFF (GC isn't running) , Alloc1 is allocated white. When control reaches Alloc2, there is a need to invoke the GC as there are no free chunks left. But this time, Alloc1 is not shaded as in the case of Scenario 2 and is therefore wrongfully collected by the GC.

One possible solution to this issue is to convert all allocation into transfers (calls to C 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 however 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 isn't trivial.

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

Fig. 4: Reservation mechanism snippet

Another way, which we find more efficient, is to guarantee that when a basic block is being executed, it will either receive all the memory chunks it requests for allocation or it will not execute at all. Thus guaranteeing that the GC won't be invoked from an allocation point. MLton already has information about the number of chunks allocated (except allocations by runtime methods) in every block at compile time. We can use this to our advantage by leveraging the *gc-check* pass we put in to do a little more than insert the GC checkpoints. At the point where we insert the GC check, we reserve the number of memory chunks 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 4 summarizes the logic involved in reserving allocations before a block is executed.

If the number of free chunks available is lesser than what is already reserved and what is required by the next block, a GC checkpoint is inserted by the optimization pass and the mutator is blocked preventing the execution of the next block until woken up by the GC. 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. It is to note that the pass does not consider array allocations and other allocations like a new thread object or new stack frames since these 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*". In contrast to the other method of splitting each block to have only one allocation, this method incurs no specific overhead except that of the statements to increment and decrement the reserved count. Currently this check and reservation is done at a per block level but it opens up potential to incorporate some of MLton's complex control and dataflow analysis to find a better place to reserve memory chunks as part of future work.

4 Evaluation

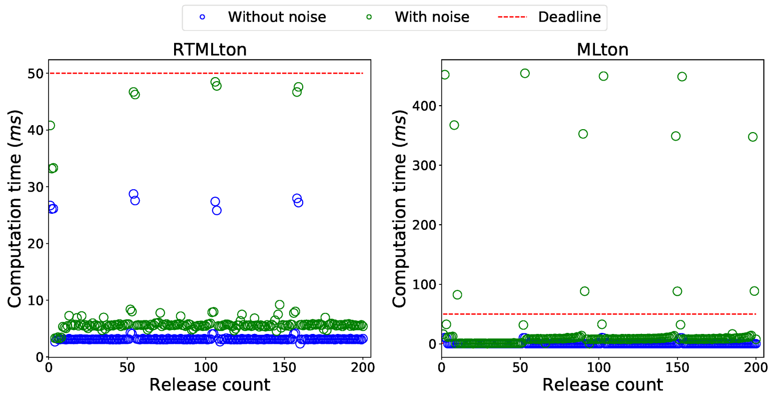

Fig. 5: Performance of RTMLton and MLton on CD_x.

Comparing RTMLton to MLton using the MLton benchmark suite, indicates that RTMLton performs similarly to MLton in tests that do not employ arrays extensively (less than 15% overhead) and performance degrades in tests, like matrix multiplication, that make use of arrays heavily without amortizing these costs. Thus the matrix multiplication benchmark exhibits the worst case performance of our system due to array access overhead (roughly 2x slow down) [13]. We expect to be able to address this overhead by adjusting our array layout in future revisions of our system and reworking MLton's optimization strategies to be real-time compliant. 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 ok if the system can meet its target deadlines. We evaluate the predictability of RTMLton on an SML port of a real-time benchmark, the aircraft Collision Detector (CD_x) [11]. CD_x is an airspace analysis algorithm that detects potential collisions between aircrafts based on simulated radar frames and 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, like 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 the original paper [11]. 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 period of 50ms for the CD and leverage a workload that has heavy collisions and measure the computation time for each release of the CD thread. We gather numbers over 2000 releases of the CD thread and graph out the distribution of the computation times and compare it with the deadline for the task. For readability and due to space constraints 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. The noise thread is scheduled along with 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 vanilla MLton the noise making thread is just a green thread that is scheduled non preemptively (co-operatively) 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 preemptible. Figure 5 shows the results of running the benchmark on RTMLton and MLton respectively. As expected, RTMLton does not distort the computation time by more than the deadline when the noise thread is running, but does exhibit overhead compared to MLton as we saw in the regular benchmarks. The computation time with the noise thread is a little more than without noise in RTMLton 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 period, we expect to the runtime 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 a huge impact in terms of missing deadlines and consequently jitter on subsequent releases. The graphs also show that with no memory pressure vanilla 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 [2]. 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 3.2, but crucially relies on a lack of OS level concurrency for the correctness of this optimized GC check. Figure 5 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 benchmark as they are computationally more intensive (due to imminent collisions in the frame) and do 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, de-allocate 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 6 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 6.

5 Related Work

Real-Time Garbage Collection: There are roughly three classes of RTGC: (i) *time based* [3] where the GC is scheduled as a task in the system, (ii) *slack based* [20] where

[2] 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.

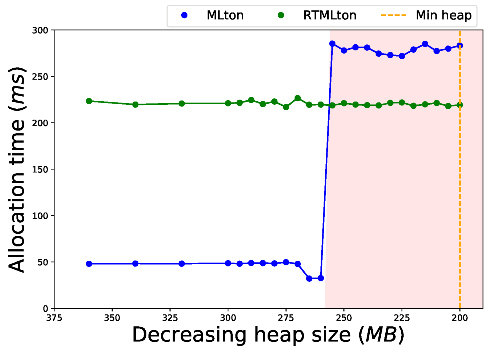

Fig. 6: Fragmentation tolerance

the GC is the lowest priority real-time task and executes in the times between release of higher priority tasks, and (iii) *work based* [21] 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.

Other languages with real-time capabilities: Our survey [17] of existing functional languages and their real-time adaptability 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 Domain Specific Languages (DSL) 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) [6] and safety critical Java (SCJ) [10], which provide a general purpose approach but burden the developer with having to manage memory directly. For example, both provide definitions for scoped memory [9], a region based automatic memory management scheme where the developer manages the regions. There is also research available on how to lessen the burden by automatically discovering how to infer scoped regions [4]. Finally, there is research in applying a region based memory management approach, while avoiding the use of a GC, in the context of SML [24].

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 an aircraft Collision Detector (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.

Acknowledgements

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

References

1. Arts, T., Benac Earle, C., Derrick, J.: Development of a verified erlang program for resource locking. Int. J. Softw. Tools Technol. Transf. **5**(2), 205–220 (Mar 2004)
2. Audebaud, P., Paulin-Mohring, C.: Proofs of randomized algorithms in coq. Science of Computer Programming **74**(8), 568 – 589 (2009), special Issue on Mathematics of Program Construction (MPC 2006)
3. Bacon, D.F., Cheng, P., Rajan, V.T.: 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. pp. 81–92. LCTES '03, ACM, New York, NY, USA (2003)
4. Deters, M., Cytron, R.K.: Automated discovery of scoped memory regions for real-time Java. In: Proceedings of the 3rd Int'l symposium on Memory management. pp. 132–142. ISMM '02, ACM, New York, NY, USA (2002)
5. Dijkstra, E.W., Lamport, L., Martin, A.J., Scholten, C.S., Steffens, E.F.M.: On-the-fly garbage collection: An exercise in cooperation. Commun. ACM **21**(11), 966–975 (Nov 1978)
6. Gosling, J., Bollella, G.: The Real-Time Specification for Java. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2000)
7. Hammond, K.: Implementation of Functional Languages: 12th Int'l Workshop, IFL 2000 Aachen, Germany, September 4–7, 2000 Selected Papers, chap. The Dynamic Properties of Hume: A Functionally-Based Concurrent Language with Bounded Time and Space Behaviour, pp. 122–139. Springer Berlin Heidelberg, Berlin, Heidelberg (2001)
8. Hammond, K.: Is it time for real-time functional programming? In: Gilmore, S. (ed.) Revised Selected Papers from the Fourth Symposium on Trends in Functional Programming, TFP 2003, Edinburgh, United Kingdom, 11-12 September 2003. Trends in Functional Programming, vol. 4, pp. 1–18. Intellect (2003)
9. Hamza, H., Counsell, S.: Region-based RTSJ memory management: State of the art. Sci. Comput. Program. **77**(5), 644–659 (May 2012)
10. JSR 302: Safety Critical Java Technology (2007)
11. Kalibera, T., Hagelberg, J., Pizlo, F., Plsek, A., Titzer, B., Vitek, J.: Cdx: A family of real-time java benchmarks. In: Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems. pp. 41–50. JTRES '09, ACM, New York, NY, USA (2009)
12. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: Cakeml: A verified implementation of ml. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 179–191. POPL '14, ACM, New York, NY, USA (2014)

13. Li, M., McArdle, D.E., Murphy, J.C., Shivkumar, B., Ziarek, L.: Adding real-time capabilities to a sml compiler. SIGBED Rev. **13**(2), 8–13 (Apr 2016)
14. López, N., Núñez, M., Rubio, F.: Stochastic process algebras meet eden. In: Proceedings of the Third International Conference on Integrated Formal Methods. pp. 29–48. IFM '02, Springer-Verlag, London, UK, UK (2002)
15. Milner, R., Tofte, M., Macqueen, D.: The Definition of Standard ML. MIT Press, Cambridge, MA, USA (1997)
16. MLton. `http://www.mlton.org`
17. Murphy, J.C., Shivkumar, B., Pritchard, A., Iraci, G., Kumar, D., Kim, S.H., Ziarek, L.: A survey of real-time capabilities in functional languages and compilers. Concurrency and Computation: Practice and Experience **31**(4), e4902 (2019)
18. Nettles, S., O'Toole, J.: Real-time replication garbage collection. In: Proceedings of the ACM SIGPLAN 1993 Conf. on Programming Language Design and Implementation. pp. 217–226. PLDI '93, ACM, New York, NY, USA (1993)
19. Pizlo, F., Ziarek, L., Blanton, E., Maj, P., Vitek, J.: High-level programming of embedded hard real-time devices. In: Proceedings of the 5th European Conf. on Computer systems. pp. 69–82. EuroSys '10, ACM, New York, NY, USA (2010)
20. Pizlo, F., Ziarek, L., Maj, P., Hosking, A.L., Blanton, E., Vitek, J.: Schism: fragmentation-tolerant real-time garbage collection. In: Proceedings of the 2010 ACM SIGPLAN Conf. on Programming language design and implementation. pp. 146–159. PLDI '10, ACM, New York, NY, USA (2010)
21. Siebert, F.: 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. pp. 94–103. JTRES '07, ACM, New York, NY, USA (2007)
22. Sivaramakrishnan, K.C., Ziarek, L., Jagannathan, S.: MultiMLton: A multicore-aware runtime for standard ML. Journal of Functional Programming **24**, 613–674 (2014)
23. Timber: A gentle introduction. `http://www.timber-lang.org/index_gentle.html`
24. Tofte, M., Talpin, J.P.: Region-based memory management. Inf. Comput. **132**(2), 109–176 (Feb 1997)
25. Wan, Z., Taha, W., Hudak, P.: Real-time frp. In: Proceedings of the Sixth ACM SIGPLAN Int'l Conf. on Functional Programming. pp. 146–156. ICFP '01, ACM, New York, NY, USA (2001)