Safe Stack Allocation in Standard ML

Thomas Schollenberger
Department of Computer Science
Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, NY 14586
tss2344@cs.rit.edu

***Abstract*—Standard ML is a garbage-collected functional programming language. Fully garbage-collected programming languages abstract away memory management from the programmer. Garbage collection guarantees memory safety, but can also introduce unpredictable latency due to garbage-collection pauses. This paper extends Standard ML with a modal type system that enables safe, optional stack allocation of data. By annotating value declarations and pattern bindings with allocation modes, programmers gain safe, manual memory management in specific sections of code while retaining the safety and convenience of the garbage collector elsewhere.**

We implement this system in MLton, a whole-program optimizing compiler for Standard ML. Our approach includes a modal type system that enables programmers to safely stack-allocate values. We evaluate the implementation using benchmarks that stress-test high allocations with short lifetimes. The results demonstrate that stack allocation can virtually eliminate garbage-collection overhead, yielding performance gains of over 30% and improved cache locality.

***Index Terms*—Standard ML; Modal Type Systems; Garbage Collection; MLton; Memory Safety; Region-based Memory Management; Functional Programming**

I. Introduction

Standard ML (SML) is a garbage-collected functional programming language that supports higher-order functions and impure operations. Languages with garbage collection give the programmer the freedom to avoid manual memory management by inserting a runtime that handles the creation and deletion of allocated memory. These languages provide tangible benefits to programmers: memory safety is guaranteed, all pointers created by the runtime are valid, and the programmer need not manage memory manually.

There are also noticeable concessions when programming in languages such as Standard ML. Programs experience unpredictable latency due to garbage-collection pauses, which occur when the runtime reaches a threshold that triggers a cleanup of unused memory before allocating additional memory. These stalls can be unacceptable in performance-critical applications. Additionally, delayed memory reuse can lead to poorer cache locality than stack allocation, which places data directly in the executing function's activation record, allowing the memory to be recycled immediately after the function returns.

The balance between high-level language safety and low-level performance control is a central theme in modern programming language design. For instance, Rust provides a borrow checker and lifetimes that guarantee memory safety without a garbage collector. Go's garbage collector is engineered for extremely low latency. Its concurrent collector runs alongside application code, minimizing 'stop-the-world' pauses, making the language suitable for latency-sensitive services. Both approaches mitigate the performance unpredictability of traditional garbage collection while maintaining safety. Standard ML is an ideal programming language for exploring a third path by combining the convenience of GC with optional, programmer-directed manual memory management. Furthermore, the MLton compiler's whole-program optimization strategy provides a unique foundation for implementing a global inference system, as it can analyze the entire program data flow to ensure safety.

One might envision a language that defaults to GC when performance is not a concern, while still allowing programmers to opt into manual memory management when necessary. In this paper, we add a modal type system to Standard ML that enables the programmer to safely stack-allocate data. This type system annotates data declarations, such as `val` declarations, function parameters, and case expression patterns, with their allocation location. Modes are optional, inferable from the surrounding context, and are backward-compatible with existing Standard ML code by defaulting to heap-allocated data.

There has been significant research into combining the safety of managed memory with the performance benefits of explicit allocation. Rust's borrow checker, for example, achieves this through a sophisticated system of ownership, borrowing, and lifetimes that are pervasively checked by the compiler [1]. While extremely powerful, this requires the programmer to manage lifetime annotations throughout their code. Our modal system, in contrast, is fully inferable, providing a more gradual on-ramp for performance optimization. MLKit's region-based memory system offers another alternative, allocating memory into hierarchically organized regions [2]. MLKit's strategy avoids the need for a garbage collector but can sometimes result in longer-than-necessary memory lifetimes when objects in a region need to be preserved. Our stack-based approach offers more fine-grained, lexical lifetimes for allocated data. Escape analysis, which is used extensively in the Go compiler, automatically determines if a value can be safely stack-allocated [3]. While effective, this is an automatic optimization beyond the programmer's direct control. Our work builds directly on the modal type system proposed by Lorenzen et al. [4], which gives the

programmer explicit control over allocation decisions, offering greater simplicity than lifetimes and greater predictability than automatic escape analysis.

We integrate the system into MLton, a whole-program optimizing compiler for Standard ML, providing parsing, constraint solving, and code generation that emit stack allocations. This system augments Standard ML binding and type constraint occurrences with optional mode parameters. When no annotation is present, the compiler performs mode inference alongside type inference, generating constraints from data flow, closure captures, and returns. Functions can, in tail position, allocate within their caller's stack frame through use of the `exclave` construct. If any path could leak a stack-allocated value from its allocation region, the compiler demotes that binding to heap mode or throws a mode-checking error, depending on the value's surrounding context. This is the basis of our no-escape invariant for the modal type system, which prohibits storing stack-allocated values in longer-lived structures or closures that may outlive the stack frame. If such a case arises, the compiler emits a type error, flagging that the value escapes its lexical scope and suggesting that the `exclave` construct may be applicable. With this system, programmers can achieve predictable performance in hot code paths involving short-lived stack-allocated values and closures, while maintaining the same memory-safety guarantee as baseline MLton.

We evaluate the implementation by simulating "hot paths" with high allocation pressure and short object lifetimes. This isolates the cost of region management versus heap allocation. We compare the performance of our modal stack allocation system against the baseline MLton compiler using recursive list generation benchmarks. Results demonstrate that stack allocation can eliminate garbage collection overhead in intensive loops. In our highest-iteration stress test, GC overhead dropped from approximately 41% of the total runtime to less than 0.1%. Our results also show an order-of-magnitude reduction in cache misses. This reduction in pause times and cache misses contributed to an overall execution speedup of 1.85x, confirming that the system effectively mitigates latency and improves performance without sacrificing memory safety.

The remainder of this paper is organized as follows. Section II establishes the background of our work. Section III explains the modal type system, formally introducing the concepts of modes, the modified syntax, and describing the safety of the type system. We then dedicate subsections to detailing the `exclave` construct, which enables safe caller-frame allocation, the mode inference process, and a direct comparison with traditional escape analysis to motivate our design choices. With the theory established, Section IV presents the implementation, describing the practical engineering effort in MLton with reproducibility in mind. We describe the necessary modifications to the parser and abstract syntax tree, the challenges of propagating mode information through MLton's numerous intermediate representations, the target-specific considerations for code generation, and the required changes to the MLton runtime to manage the new allocation kind. Section VI presents our evaluation, first detailing the experimental setup and then presenting the analysis on programs that would most benefit from our work. Finally, in Section VII, we compare this work to other approaches to memory safety and discuss future work in Section VIII.

II. Background

A. OxCaml

This work directly builds on the work of Lorenzen et al. [4], who extended OCaml with additional modern language features. Their work primarily focuses on introducing Rust-style ownership, improved support for parallelism, and the allocation locality of values through a modal type system. Their paper was then formalized into an open-source fork of the OCaml compiler, called OxCaml.

Our work aims to connect the formalism of the Oxidizing OCaml paper with a reproducible demonstration of local modal allocation, and to provide much-needed performance benchmarks that report concrete numbers.

B. MLton

MLton is a whole-program optimizing compiler for Standard ML that performs aggressive ahead-of-time compilation. Unlike compilers that break programs into separate compilation units, MLton considers the entire program, including external libraries, which enables cross-module optimizations (extensive inlining and aggressive dead-code elimination). These attributes are critical for inferring and validating stack versus heap allocation, as soundness requires global knowledge of how and where values are consumed or captured.

The MLton compilation pipeline lowers the source program through a series of increasingly explicit intermediate representations (IRs), such as CoreML, XML, SXML, SSA, SSA2, RSSA, and finally, the Machine representation[1]. Alongside the generated code, MLton links to a C runtime library that provides program initialization and garbage collection functionality. MLton supports various garbage collection algorithms, including copying, mark-sweep, and generational. Each is susceptible to performance penalties due to GC pauses.

Why Standard ML and MLton for this? Mode constraints require knowledge of the entire program, such as whether a closure escapes or if a value `e` is consumed by a heap-allocated structure. MLton considers the whole program during typechecking and optimization, providing the visibility required to infer and validate stack vs. heap decisions soundly. Modes affect how closures are allocated. MLton's explicit closure environments and representation choices in SSA/RSSA make it feasible to track which variables are captured, thereby bounding the closure's lifetime and allocation mode. This is necessary for closures that capture stack-allocated data, as they must also be stack-allocated.

Standard ML differentiates tail expressions from the declarations within expressions. This allows our mode analysis to

[1]XML and SXML are *explicit* ML and *simplified explicit* ML [5]. All IRs are explained in detail in Section IV

verify that no stack data escapes the allocating function and that `exclaves` are used only in the tail position.

Finally, MLton's runtime already supports precise GC, stack scanning, and multiple targets. Introducing a second allocation class (stack) with strict no-escape guarantees can be integrated without compromising GC invariants, since stack values never enter GC-managed memory regions.

III. Type System

A. Modes

The modal type system for memory allocation introduced in this work distinguishes between two programmer-visible allocation modes, `stack` and `heap`, and relies on two additional internal modes: `constant` and `undetermined`. These annotations are applied to value bindings and patterns, enabling programmers to have fine-grained control over memory lifetimes and storage classes without sacrificing safety.

In this framework, a binding declared with the `stack` mode is allocated in the current function's region, such that its lifetime is strictly limited to the lexical scope of that function. Languages like C allocate values within a function's stack frame, but do not provide any static guarantees that these values do not escape their defining function. Values allocated on the `heap` have their lifetimes determined by the garbage collector and can outlive the function frame in which they were created. This is the default allocation strategy in most functional languages, like Standard ML, OCaml, Go, and Python.

The `constant` mode exists only for immediate values (numerical literals, booleans, string constants), which require no dynamic allocation, and is used internally by the compiler. Finally, the `undetermined` mode is the initial state for any value without a mode annotation and must be inferred from usage, similar to how Standard ML infers types for bindings without explicit type annotations.

Compared to systems such as the region-based memory management used in MLKit, modal typing grants per-binding granularity rather than grouping into lifetime regions. Unlike escape analysis, which operates as a compiler optimization with no surface-level guarantee, mode specifications in this system are verified and enforced as part of the static semantics, giving programmers and tooling the confidence and explicitness of types.

Modal allocation enables programmers to gain performance speedups that may not have been possible without stack allocation. For example, examine the `sumPairs` code[2] in Fig. 1.

Fig. 1. Sum Pairs

```
fun sumPairs xs =
 let
   fun doit (acc, xs) =
     case xs of
       [] => a c
     | a :: b :: rest =>
       let
          (* short-lived pair on stack *)
          val p :- stack_ =
            if a < b
              then (a, b)
              else (b, a)
       in
          (* no heap traffic for p *)
          doit (acc + #1 p + #2 p, rest)
       end
     | _ => acc
 in
   doit (0, xs)
 end
```

Without allocating `p` on the stack, these values often allocate on the heap, creating GC pressure and cache churn. `p` is explicitly annotated with `:- stack_`, which is a contractual demand from the programmer that this value must be stack-allocated. In `sumPairs`, the compiler proves that `p` is non-escaping and safe to stack-allocate. However, the compiler would return a diagnostic error if `p` were invalid for stack allocation. This results in predictable latency and lower allocation volume.

[2]We include the if statement to avoid optimization, as MLton often eliminates naive tuple allocations.

B. Exclave

Functions delineate regions within which stack values may be allocated. A function cannot safely return a value that was stack-allocated in its own region, because the callee's frame is deallocated on return. To enable producer functions to construct results that the caller can still use without heap allocation, we introduce `exclave`.

Our informal semantics are:

- `exclave e` evaluates `e` with the intent that any allocation for the expression's result occurs in the caller's region.
- Exclusively allowed in tail position.
- `e` may only reference values whose lifetimes are valid in the caller's region. In particular, `e` must not capture stack values from the current frame, as this would return dangling pointers.

To demonstrate the benefits of exclave, examine the list code example Fig. 2, which illustrates one potential use case. This code defines two functions that operate on lists, `map_exclave` and `tabulate_exclave`. These are both variants of existing list functions that return stack-allocated lists. This code then uses these two functions to construct a list, map each element by multiplying it by 2, and then sum the resulting list. This demonstrates that temporary lists can be allocated on the stack.

Fig. 2. List Functions

```
fun map_exclave f [] = []
  | map_exclave f (x :: xs) =
      exclave_ (f x :- stack_)
        :: (map_exclave f xs :- stack_)

fun tabulate_exclave n f =
  if n <= 0 then []
  else
    exclave_
      (f n :- stack_)
        :: tabulate_exclave (n - 1) f

val list1 :- stack_ =
    tabulate_exclave n (fn x => x)
val list2 :- stack_ =
    map_exclave (fn x => x * 2) list1
val sum = List.foldl op+ 0 list2
```

C. Inference

Inference enables existing SML programs to typecheck correctly with this modal extension. Without inference, every binding and pattern would require its own mode annotation. Standard ML does not require this for type annotations, using the Hindley-Milner type system to infer types from expressions, and our modal type system must do the same. Unlabeled modes are initialized to `undetermined`; each closure is also assigned a mode. We then define a `subsumes` function that determines whether modes are used in the correct context. For example, a stack-allocated list can contain heap-allocated elements, but a heap-allocated list cannot contain stack elements.

Modes are left as `undetermined` throughout the *AST* and *CoreML*, and are filled in progressively as the compiler approaches the defunctorize pass. Constraints must be evaluated at the following positions:

- Returns: expressions in return positions must not contain stack data that is defined within the function.
- Function Calls: argument passing imposes that the argument pattern mode constraint must subsume the passed parameter.
- Closure Captures: if a closure captures a stack-allocated variable, its environment must also be stack-allocated.
- Stores: writing a value into a heap-allocated field restricts that the value must be heap-allocated.

Annotations in this system are assertions. An explicit stack annotation on a binding or pattern is treated as a bound. If constraints require a heap, the type system reports a mode error and terminates compilation. Explicit heap forces heap regardless of otherwise stack-safe data flows.

To illustrate how inference and type checking work, we examine several examples. In Fig. 3, we have a function that `exclaves` back into some context that has two values, `inferred` and `error`. Our type system correctly infers two things here: `exclave_func` returns a stack-allocated optional type, and the value `inferred`'s mode must be `stack`. However, this code is not entirely correct. The value `error` is annotated with the mode `heap`, but is assigned to the application of `exclave_func`. Our type system knows that `exclave_func` may only return stack-allocated data (we do not support mode polymorphism), therefore a type error will occur, as the programmer who wrote this code improperly constrained `error` to `heap`, when it may only be `stack`.

Fig. 3. Mode Inference and Type Checking

```
fun exclave_func () = exclave_ (SOME 1)

val inferred = exclave_func ()
    ^^^^^^^^ inferred to stack allocated
val error :- heap_ = exclave_func ()
           ^^^^^^^^ modal type error
```

IV. Implementation

MLton's pipeline must be modified across all intermediate representations:

- The front-end is modified to parse optional mode annotations, defaulting to `undetermined` when mode annotations are not provided.
- During the middle-end, mode constraints must be generated.
  - We guarantee that all modes are solved for by the time we lower to the *XML* representation.
  - When lowering from *Core ML* to *XML*, we remove the `exclave` and insert a region pop. This is discussed more in Section IV-B.
- Codegen and the runtime must also be modified. Within the backend IRs, instructions are inserted to allocate new values onto the stack properly when necessary.
- In the C runtime library, new space is allocated for the stack, and partitioned into regions that are pushed and popped during runtime.

A. Parsing and Abstract Syntax Tree

The implementation of our modal type system begins by modifying the MLton front-end, which requires extensions to both the concrete grammar and the Abstract Syntax Tree (AST). These modifications are designed to accommodate the new `stack` and `heap` memory allocation modes, without altering the semantics of existing Standard ML programs. Our goal is to integrate mode annotations as optional constraints that operate alongside standard type annotations.

To support programmer-directed allocation, we modified the MLton parser definition, which utilizes ML-Yacc. We introduced a new non-terminal, `mode`, to parse the keywords `stack_` and `heap_`, which are post-fixed with an underscore

to avoid collisions within the MLton codebase. To syntactically distinguish mode constraints from standard type constraints (which use the `:`), we introduced the token COLONDASH (`:-`) [3]. This choice prevents ambiguity in the grammar, allowing mode annotations to coexist with type annotations on the same binding.

New production rules were added to allow these constraints in three specific contexts:

- **Expressions**: The rule `exp COLONDASH mode` enables annotations on values, such as `val x = e :- stack`.
- **Patterns**: The rule `cpat COLONDASH mode` allows pattern matching to enforce allocation modes of both the matched value and its components.
- **Functions**: We extended function declarations to support optional result constraints. This allows a syntax of `fun foo (x: int): int list :- stack`, enabling developers to explicitly specify the allocation mode of a function's return value.

These grammar extensions are optional. If the `:-` token is absent, the parser defaults to existing production rules. This ensures that the modified compiler maintains full backward compatibility with legacy SML codebases.

Once parsed, these syntactic constructs must be preserved for semantic analysis. We extended the core AST definition to represent mode information.

We extend the expression and pattern datatypes with a new variant, `ModeConstraint`. This node wraps an underlying expression or pattern along with its associated mode value. This structural encapsulation enables the compiler's subsequent passes to traverse the AST and peel off mode constraints, thereby separating the allocation intent from the expression's logic. This design mirrors MLton's existing handling of type constraints, ensuring that mode constraints are propagated reliably through the front-end pipeline. Functions are extended to include a "result mode" option, in addition to the existing "result type" option.

Finally, to support modes, we introduce the Mode datatype, consisting of four variants:

- Stack: Explicitly requested stack allocation.
- Heap: Explicitly requested or defaulted garbage-collected allocation.
- Undetermined: The initial state for all unannotated bindings, acting as a variable to be solved during inference.
- Constant: A specialized mode for immediate values that require no allocation.

The Mode datatype and corresponding module also provide the necessary algebra for mode analysis, equality checks, and, most importantly, subsumption logic. The subsumption functions defined here are critical to the inference engine described in Section III-C, enabling the compiler to determine whether a specific allocation mode satisfies the constraints of a given context, such as ensuring that a stack-allocated list does not flow into a heap-allocated structure.

[3]Lorenzen et al. [4] use an `@` to represent mode annotations, but this symbol is reserved for list concatenation in SML.

B. Lowering Through Intermediate Representations

The central engineering challenge in implementing safe stack allocation within MLton is to preserve high-level allocation intent through a compilation pipeline that aggressively restructures code. MLton transforms programs through a sequence of intermediate representations (IRs). The pipeline starts at the high-level AST, becomes fully typed within CoreML, then from CoreML to XML, SXML, SSA, RSSA, and finally Machine IR. At each boundary, the compiler restructures control flow and applies various IR-specific optimizations. Consequently, the modal information captured in the AST must be handled with care as it propagates through these IRs and is eventually lowered to concrete runtime primitives.

The first phase of lowering ensures that stack or heap mode annotations survive the transition from the front-end to the middle-end. In CoreML, mode inference results are stored directly on patterns and expressions. This allows the type checker to easily verify our no-escape invariant. For instance, we check that the tail position of functions does not leak stack-allocated data, as in Fig. 4. This function allocates a list within its region and then attempts to return it. CoreML must analyze all tail positions of functions to verify that pointers to deallocated data are not returned.

Fig. 4. Temporary Value Escaping Defining Function

```
fun make_list_bad n =
  let
    val lst :- stack_ = [n, n+1, n+2]
  in
    (* lst will outlive the region *)
    lst
  end
```

Standard ML also supports impure `ref` cells, which can be updated mutably. This poses another issue to guarantee. If a `ref` cell is heap-allocated, at no point may stack-allocated data be stored within it. Fig. 5 demonstrates this problem. In this example, `global_cache` is heap-allocated at the top level of the program. Storing stack-allocated data within it would result in a reference to an invalid pointer once the `update_cache` function exits.

As the compiler proceeds into the defunctorization and monomorphization passes, the abstract syntax trees are repeatedly rebuilt. Monomorphization, which duplicates polymorphic functions for specific types, poses a particular risk. As code is cloned, the associated mode metadata must also be cloned. We modified the lowering passes to replicate the modes during these duplication events. Similarly, optimization passes that introduce new intermediate variables, such as shrink or simplify-types passes, needed to be updated to con-

Fig. 5. Ref Cell Escaping

```
val global_cache = ref NONE

fun update_cache (n) =
  let
    val temp_data :- stack_ = SOME n
  in
    (* temp_data will outlive the region
     * if stored in global_cache
     *)
    global_cache := temp_data
end
```

sult the surrounding context and assign coherent modes, preventing the reintroduction of undetermined states that would invalidate the inference done in the front-end.

Additionally, strict consistency must be maintained when generating intermediate variables for desugared constructs. For instance, prepending to a list is represented internally as a list `Cons` constructor applied to a tuple of the head and tail. A critical pitfall arises if the `Cons` application is annotated for the stack, but the implicitly generated tuple defaults to the heap. Since subsequent optimizations often flatten constructor applications, the Cons node and its mode annotation may be erased. If the underlying tuple has not correctly inherited the stack mode, the operation will silently revert to heap allocation, violating the allocation strategy requested by the programmer.

Once the modes are fully determined, the compiler must translate the abstract concept of ”stack allocation” into concrete region management operations. This translation begins during the defunctorization pass, which translates CoreML to XML. For standard stack allocations, the mode remains a property of the allocation expression. However, for the exclave construct, which permits allocation in a caller's stack region, the compiler must make the region boundaries explicit. We modified the defunctorization pass to lower exclave expressions by first popping the function's current region, using a newly added region pop primitive, then assigning the expression's mode to the stack. Popping the region effectively allocates it in the caller. These primitives are side-effecting operations that cannot be folded or removed by standard dead-code elimination, ensuring that the region lifetime is preserved.

The target of this defunctorization process is XML, MLton's high-level polymorphic intermediate representation [6]. This IR features explicit type annotations on every node, fully expanded primitives, and flattened pattern matching. It retains the lexical scoping and higher-order function structure of the source language. At the same time, XML simplifies complex CoreML constructs by resolving flexible records into ordered tuples and lifting datatype declarations to the top level. This structural retention makes it the ideal stage for enforcing region invariants before lowering the program into the control-flow-heavy SXML and SSA representations.

The most complex logic resides in the Single Static Assignment (SSA) backend, where control flow is explicit, high-level data type structures are absent, and all variables are assigned only once. Here, allocation modes on constructors and tuples are finally used to generate region-aware code. We implemented two dedicated passes to manage this:

1) Insert Regions Pass: This pass constructs the physical stack-frame logic for standard function calls. It scans the control flow graph for stack-mode allocations and performs a dataflow analysis on predecessor edges to determine which basic blocks execute within an active region. Based on this analysis, it inserts a `Region_push` at the function entry (if stack allocation is required). It ensures `Region_pop` instructions are inserted before all exit points, including tail calls and exception handlers. Exception handlers are a vital edge case, as exceptions cannot be stack-allocated at all due to the nature of the call stack and the difficulty of determining where exception handlers may be located.

2) Remove Regions Pass: The insertion region passes occur at the beginning of the SSA optimizations. After this pass, many additional SSA optimizations are performed, including inlining, global extraction, and dead-code elimination. This can result in regions that were initially required but now contain no stack allocations. This optimization pass acts as dead-code elimination for the regions themselves after all optimization passes have been run. It identifies regions that were created but contain no stack allocations. It eliminates these empty push/pop pairs, ensuring zero-cost overhead for unused features. Furthermore, this pass performs inter-procedural analysis to determine whether a callee requires the caller to open a region (as is the case with exclave usage). This analysis allows the compiler to prune upstream region operations if the call graph does not strictly require them.

C. Code Generation & Runtime

The final phase of the compilation pipeline lowers the SSA IR to Machine IR, the final IR with simple operations that map to the various architecture targets MLton supports. To support stack allocation, we extended the MLton runtime with a secondary memory arena and modified the code generator to manipulate this arena directly. This integration ensures that the abstract primitives `Region_push` and `Region_pop` operations are lowered into machine instructions that manage memory without the overhead of the system garbage collector.

The underlying mechanism relies on a stack of regions implemented within a contiguous block of raw memory, shown in Figure 6. The C runtime manages this state via four state fields:

- `regionBuffer`: A pointer to the start of a pre-allocated 4MB memory arena.
- `regionBufferSize`: The total size of the arena.
- `regionBase`: A pointer to the start of the current region within the buffer.
- `regionTop`: The allocation frontier, pointing to the next available byte for new objects.

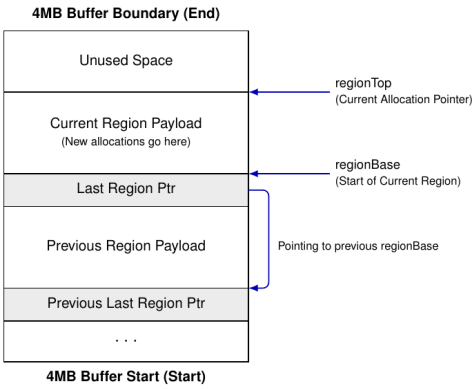

Fig. 6. Segmented Region Buffer

Visually, this buffer acts as a linear stack of regions. While the buffer is contiguous, the regions effectively form a linked list. The header of each region contains a pointer to the previous region's base, enabling the runtime to traverse the stack when a scope is exited. The headers have the same size as MLton metadata headers, allowing existing GC algorithms to traverse the entire region buffer to scan for GC roots.

When the backend transforms SSA2 into the final SSA form, RSSA, it must decide how to implement region transitions. We provide two distinct implementations to balance performance requirements against verification needs.

1) C Runtime Prototype: The runtime library includes C functions `GC_regionPush` and `GC_regionPop` that manipulate the state variables described above. These functions serve as the reference implementation. However, relying on foreign function calls for every scope change introduces significant overhead due to context switching and register saving. These are used primarily for debugging and validating the correctness of the backend logic.

2) Inline Pointer Arithmetic: For performant code, we bypass the C runtime. While lowering the SSA2 IR to RSSA, we translate region primitives directly into inline pointer-arithmetic sequences.

The region push primitive enters a new scope. To enter a new scope, the compiler emits instructions to save the current `regionBase` into the address currently pointed to by `regionTop`. It then advances both `regionTop` and `regionBase` by one machine word. This effectively embeds the "return address" of the stack frame immediately preceding the payload, an $O(1)$ operation requiring no loops or checks.

The region pop primitive leaves the current scope. To exit a scope, the compiler calculates the address immediately preceding the current `regionBase` to retrieve the saved pointer. It then restores `regionBase` to that saved value and retracts `regionTop` to the old base, instantly reclaiming the memory used by the popped region.

V. Garbage Collector Integration

The introduction of a secondary memory arena necessitates careful integration with MLton's existing garbage collector. The region buffer is situated outside the managed heap, meaning its lifecycle is independent of the GC. However, stack-allocated objects often contain pointers to heap-allocated objects, effectively acting as roots that must be visible to the collector to prevent dangling pointers.

We modified the runtime's root generation phase to include the active portion of the region buffer in the root set. When a garbage collection cycle is triggered, the runtime scans this memory area. Any pointers found within the stack that reference objects in the heap are treated as live roots.

During the copy phase, the collector treats the heap objects referenced by these roots as live data, but it makes no attempt to move, copy, or reclaim the stack objects themselves. This integration ensures that the manual stack management remains entirely orthogonal to the automatic heap management, interacting only to preserve the heap data referenced by the stack.

VI. Evaluation

To assess both the correctness and the speed of the modal type system and stack allocation, we conducted a series of performance benchmarks. Our evaluation focuses on three key metrics: total garbage-collection time, number of cache misses, and overall program execution time. We aim to demonstrate that stack allocation reduces pressure on the garbage collector without incurring significant runtime overhead due to the additional pointer arithmetic required for region management.

A. Setup

All benchmarks were executed on a Framework Laptop 13 equipped with an AMD Ryzen 5 7640U processor (Zen 4 architecture, 6 cores, 12 threads, with a max boost clock of 4.97 GHz) and 32 GiB of RAM. The operating system was Fedora Linux 43 (Workstation Edition).

We compared two configurations of the MLton compiler. The baseline is the unmodified MLton compiler (master branch) before the introduction of modal types, and the modal stack is our modified version of MLton with the stack allocation backend enabled.

B. Benchmarking

Properly evaluating performance wins of these additions can be difficult. Many existing tests in the MLton benchmark suite do not run long enough to trigger meaningful garbage-collection cycles, or lack the hot path loops that would benefit from stack allocation.

Therefore, we provide an additional testing suite to simulate different hot paths that allocate temporary data structures and use them to compute a result. This provides a clear picture of the allocator's efficiency by isolating the cost of region management from that of heap allocation and collection. These scenarios, with high allocation pressure and short object lifetimes, which are the theoretically ideal use cases for stack allocation, can set the upper bound on the system's performance gains.

1) Runtime & Garbage Collection Performance: A typical pattern in programming, both functional and imperative, is to allocate a list of data (e.g., reading a line from a file), perform operations on that list, and then reduce it to a final value. This benchmark defines a recursive function that generates N lists of N for a given iteration count. This stresses the `Cons` cell allocation and tests the interaction between data of different sizes and the region buffer.

Table I demonstrates substantial overhead reduction in this test. The stack-allocated version maintained a near-zero garbage-collection time across all iteration counts, with just 1 ms of GC time for the 50,000- and 100,000-iteration stress tests. The heap-allocated version had its GC time scale linearly with the number of input iterations. At 50,000 iterations, the standard MLton runtime was 31.6 seconds, and at 100,000 iterations, 117 seconds, both of which are $\approx 41\%$ of the total execution time spent on garbage collection.

By allocating all temporary allocations within our region buffers, we observe a speedup of approximately 1.85x at our highest-iteration test. This performance improvement results from a two-fold reduction in GC overhead. Traditionally, we view the work performed by a GC as the pauses required to free memory. However, additional work is underway to ensure sufficient memory is available, even if no pause is necessary. We observe this extra work by subtracting the GC time from the total runtime of the heap-allocating tests. This does not yield the stack-allocated runtime. The stack-allocated runtime is even faster, thanks to reduced overhead from GC checks and improved cache locality.

TABLE I
LIST PERFORMANCE COMPARISON

Iterations	Allocation	Total (ms)	GC (ms)	Overhead
1,000	Stack	17	0	0.0%
	Heap	27	7	25.9%
5,000	Stack	404	0	0.0%
	Heap	702	250	35.6%
10,000	Stack	1,619	0	0.0%
	Heap	2,939	1,120	38.1%
50,000	Stack	41,826	1	<0.1%
	Heap	77,681	31,664	40.7%
100,000	Stack	146,985	1	<0.1%
	Heap	284,801	117,301	41.1%

2) Cache Locality Analysis: In addition to reducing garbage-collection pauses, our results indicate improvements in memory access patterns. Modern processor performance is affected by the efficiency of the cache hierarchy. Depending on the CPU architecture and model, a miss in the L1 or L2 cache can cost hundreds of CPU cycles [7], stalling the instruction pipeline. The stack allocation model increases temporal locality by keeping the same memory regions in cache longer. When a function returns or a region is popped, the `regionTop` pointer decrements to the previous region. Future allocations immediately overwrite the same memory addresses that were just released.

The baseline MLton allocator follows a standard generational nursery pattern. Generational allocation provides fast object creation by allocating objects sequentially forward through the nursery until the nursery is exhausted. Even if the nursery fits in the L3 cache, the constant forward progression means the processor will frequently access cold memory. Furthermore, when the garbage collector triggers, it must traverse live objects and copy them. This process is memory-bandwidth intensive and pollutes the cache, forcing evictions of the user's data to make room for the collector's structures and the live data it copies.

We profiled the benchmark using the Linux `perf stat -e cache-misses` command to justify this claim. Shown in Table II, our results demonstrate an order of magnitude difference in memory traffic. At 100,000 iterations, the standard heap allocator incurred over 1.1 billion cache misses. The stack-allocated version, performing the same logic, incurred only 30.5 million misses, a roughly 36x reduction.

These results explain the performance difference noted in the previous section. Even after subtracting the GC pause time from the heap-allocated runtime, the stack version remains faster. We attribute this speedup to the processor spending less time waiting for data to be fetched from main memory.

TABLE II
CACHE MISS ANALYSIS

Iterations	Stack Allocation	Heap Allocation	Reduction
1,000	40,220	114,038	2.8x
5,000	117,237	4,064,441	34.6x
10,000	345,307	16,255,877	47.0x
50,000	7,867,401	323,486,010	41.1x
100,000	30,583,492	1,101,361,423	36.0x

VII. RELATED WORK

Our work sits at the intersection of region-based memory management, structural type systems, and escape analysis.

The MLKit compiler [2] pioneered the use of region inference to manage memory without a garbage collector. In MLKit, values are assigned to regions based on their computed lifetimes. This is powerful, but because region inference is automatic, memory may be retained longer than necessary. Cyclone [8] built on these concepts for the C programming language, allowing explicit allocation regions. Haskell has a similar feature, the ST Monad [9], which utilizes the Haskell type system to have safe stateful memory regions.

Wadler [10] proposed linear types (values that must be used exactly once) as a novel way to change how mutation can be performed in functional languages. Rust [1] expanded on the idea of linear types by utilizing affine types (used at most once) and a borrow checker that enforces these properties. Rust's model is highly expressive but requires the programmer to manage ownership and lifetime annotations manually, incurring significant cognitive overhead. We examined these works and offer a "gradual" path.

Escape analysis is a compiler optimization that determines whether a dynamically allocated object can safely be allocated on the stack. Blanchet [11] demonstrated a practical escape analysis specifically for ML, and modern managed languages

such as Go [3] rely heavily on it. However, these are compiler heuristics that are invisible to the programmer. A small code change can silently disable the optimization, leading to performance regressions. Our system makes allocation explicit in the type system. If a programmer annotates a value as stack and it escapes, the compiler emits an error rather than silently degrading to heap allocation.

Our implementation builds on the work of Lorenzen et al. [4]. We adapt their modal type system to Standard ML. Introducing this into MLton enables whole-program optimization, which is not available in the OCaml compiler. This helped us to validate mode constraints globally, ensuring that stack pointers never dangle and that the interaction between the stack regions and the garbage collector is safe.

VIII. Future Work

There are several promising directions for further research. The most immediately impactful is to extend programmer-driven allocation to the Array and Vector primitives. These primitives are baked into the runtime and will require an extensive overhaul of both the compiler internals and the runtime code. This also requires that operations on the Array and Vector types are polymorphic with respect to container allocation. For instance, an `Array.update` would require that the container's mode subsume the modes of the replacement values.

Further work is needed to test and validate this system. While microbenchmarks demonstrate the potential of this system in a vacuum, extending existing SML programs to incorporate modal allocation is the next step.

Finally, another area of research is to extend our modal type system to incorporate other use cases proposed by Lorenzen et al. [4]. Tracking allocation uniqueness would add an ownership system to Standard ML, enabling memory reuse and in-place updates to immutable data structures.

IX. Conclusion

We demonstrated the feasibility of integrating a modal type system into Standard ML and the MLton compiler. We provided constructs that enable the everyday Standard ML programmer to have fine-grained control over where values are allocated. We validated that the core mechanics of our type system and runtime function as a prototype. Further work to move this towards production readiness will require more extensions and testing.

We also recognize that in a language as complex as Standard ML, our type system may not be fully understood. While our inference engine passes all syntax and runtime tests, applying it to real-world applications may reveal edge cases that our type system does not handle correctly.

We have shown that garbage-collected functional programming languages can be extended with modern manual memory management concepts. We prove that a garbage-collected language can be retrofitted with region-based manual management without violating the safety guarantees that functional programmers rely. While not suitable for production-ready code, this serves as a foundation for further SML extensions or for integration into other languages.

REFERENCES

[1] N. D. Matsakis and F. S. Klock, "The Rust language," in *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, ser. HILT '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 103–104. [Online]. Available: https://doi.org/10.1145/2663171.2663188

[2] M. Tofte, L. Birkedal, M. Elsman, and N. Hallenberg, "A retrospective on region-based memory management," *Higher-Order and Symbolic Computation*, vol. 17, no. 3, pp. 245–265, Sep 2004. [Online]. Available: https://doi.org/10.1023/B:LISP.0000029446.78563.a4

[3] J. Meyerson, "The go programming language," *IEEE Software*, vol. 31, no. 5, pp. 104–104, 2014.

[4] A. Lorenzen, L. White, S. Dolan, R. A. Eisenberg, and S. Lindley, "Oxidizing OCaml with Modal Memory Management," *Proc. ACM Program. Lang.*, vol. 8, no. ICFP, Aug. 2024. [Online]. Available: https://doi.org/10.1145/3674642

[5] R. Harper and J. C. Mitchell, "On the type structure of Standard ML," *ACM Trans. Program. Lang. Syst.*, vol. 15, no. 2, p. 211–252, Apr. 1993. [Online]. Available: https://doi.org/10.1145/169701.169696

[6] S. Weeks, "Whole-program compilation in MLton," p. 1, invited talk.

[7] J. Stokes, "Understanding CPU caching and performance," Ars Technica, Jul. 2002. [Online]. Available: https://arstechnica.com/gadgets/2002/07/caching/

[8] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of c," in *2002 USENIX Annual Technical Conference (USENIX ATC 02)*. Monterey, CA: USENIX Association, Jun. 2002. [Online]. Available: https://www.usenix.org/conference/2002-usenix-annual-technical-conference/cyclone-safe-dialect-c

[9] J. Launchbury and S. L. Peyton Jones, "Lazy functional state threads," *ACM SIGPLAN Notices*, vol. 29, no. 6, pp. 24–35, 1994.

[10] P. Wadler, "Linear types can change the world!" in *Programming concepts and methods*, vol. 3, no. 4. North-Holland, Amsterdam, 1990, p. 5.

[11] B. Blanchet, "Escape analysis: correctness proof, implementation and experimental results," in *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '98. New York, NY, USA: Association for Computing Machinery, 1998, p. 25–37. [Online]. Available: https://doi.org/10.1145/268946.268949