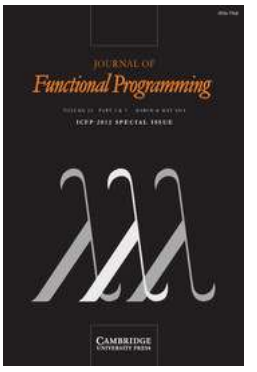


MultiMLton: A multicore-aware runtime for standard ML

K. C. SIVARAMAKRISHNAN, LUKASZ ZIAREK and SURESH JAGANNATHAN

Journal of Functional Programming / *FirstView* Article / July 2014, pp 1 - 62
DOI: 10.1017/S0956796814000161, Published online: 18 June 2014

JFP: page 1 of 62. © Cambridge University Press 2014
doi:10.1017/S0956796814000161

MultiMLton: A multicore-aware runtime for standard ML

K.C. SIVARAMAKRISHNAN
Purdue University, West Lafayette, IN, USA
(*e-mail:* chandras@purdue.edu)

LUKASZ ZIAREK
SUNY Buffalo, NY, USA
(*e-mail:* lziarek@buffalo.edu)

SURESH JAGANNATHAN
Purdue University, West Lafayette, IN, USA
(*e-mail* suresh@cs.purdue.edu)

Abstract

MULTIMLTON is an extension of the MLton compiler and runtime system that targets scalable, multicore architectures. It provides specific support for ACML, a derivative of Concurrent ML that allows for the construction of composable *asynchronous* events. To effectively manage asynchrony, we require the runtime to efficiently handle potentially large numbers of lightweight, short-lived threads, many of which are created specifically to deal with the implicit concurrency introduced by asynchronous events. Scalability demands also dictate that the runtime minimize global coordination. MULTIMLTON therefore implements a split-heap memory manager that allows mutators and collectors running on different cores to operate mostly independently. More significantly, MULTIMLTON exploits the premise that there is a surfeit of available concurrency in ACML programs to realize a new collector design that completely eliminates the need for read barriers, a source of significant overhead in other managed runtimes. These two symbiotic features - a thread design specifically tailored to support asynchronous communication, and a memory manager that exploits lightweight concurrency to greatly reduce barrier overheads - are MULTIMLTON's key novelties. In this article, we describe the rationale, design, and implementation of these features, and provide experimental results over a range of parallel benchmarks and different multicore architectures including an 864 core Azul Vega 3, and a 48 core non-coherent Intel SCC (Single-Cloud Computer), that justify our design decisions.

1 Introduction

Functional languages have long been viewed as particularly well-suited for expressing concurrent computation, and there has been much work over the years on building parallel implementations (McKay & Shapiro, 1980; Goldman & Gabriel, 1988; Miller, 1988; Feeley & Miller, 1990; Raymond, 2000) of these languages. This is because (a) first-class procedures and expression-oriented syntax make it convenient to express concurrent threads of control, (b) the availability of advanced control structures like continuations make it straightforward to define and experiment with different kinds of user-level thread

schedulers and concurrency structures, and (c) the lack of pervasive side-effects simplifies reasoning, facilitates optimizations, and reduces synchronization costs. On the other hand, because it is so easy to create and manipulate threads, the cost of concurrency management is often obscured and hidden from programmers. Implementations are typically burdened with having to deal with large numbers of dynamically created threads, with little guarantees that the cost to create these threads is amortized by the computation they perform.

In this article, we consider two specific design strategies to enable more efficient and scalable runtime systems for such languages. The first defines a thread management and scheduling infrastructure geared towards the creation of new threads of control only when the computation to be performed is sufficiently substantial to warrant it. The second considers a new memory management design intended to reduce coordination costs between mutators and collectors operating in a multicore environment. Our design exploits the observation that functional programs often reveal a surfeit of concurrency, which can be exploited to eliminate memory management overheads introduced by mechanisms such as read barriers to track forwarding pointers.

The context of our investigation is MULTIMLTON, a multicore-aware extension of the MLton (2012) whole-program optimizing compiler and runtime for Standard ML. MULTIMLTON supports a programming model that exposes opportunities for fine-grained concurrency in which threads primarily communicate via message-passing. This model assumes an underlying logical shared heap that allows references (rather than deep copies) of heap-allocated data to be directly communicated in messages. A functional programming discipline, combined with explicit communication via messages (rather than implicit communication via shared-memory) offers an enticingly attractive programming model, but, as alluded to above, also introduces numerous challenges for realizing efficient implementations.

In the ML family of languages, Concurrent ML (CML) (Reppy, 2007), a synchronous message-passing dialect of ML, is a well-studied instantiation that provides event combinators, which can be used to construct sophisticated synchronous communication protocols. MULTIMLTON additionally provides support for ACML (Ziarek *et al.*, 2011), an extension of CML that supports *asynchronous* events and the construction of *heterogeneous* protocols, which mix synchronous and asynchronous communication. Asynchrony can be used to mask communication latency by splitting the creation of a communication action from its consumption. An asynchronous action is *created* when a computation places a message on a channel (e.g., in the case of a message send), and is *consumed* when it is matched with its corresponding action (e.g., the send is paired with a receive); the creating thread may perform arbitrarily many actions before the message is consumed. In contrast, synchronous message-passing obligates the act of placing a message on a channel and the act of consuming it to take place as a single atomic action. Supporting composable asynchronous protocols in the spirit of CML's synchronous event combinators necessarily entails the involvement of two distinct, abstract threads of control – the thread that creates the asynchronous action, and the thread that discharges it. The latter thread is implicit, and requires careful treatment by the runtime to ensure that efficiency is not compromised.

Having to contend with a large number of threads also imposes additional strain on memory management. The scalability demands on multicore processors suggest that the task of memory management should be distributed across all of the available cores. One

way to achieve this is to split the program heap among the different cores, allowing each such heap to be managed independently. However, messages sent by threads from one core to another may include references to heap-allocated data accessible to the sender, but not the receiver. To retain SML semantics, such data must be copied to a global heap whose objects are directly available to all threads. The cost of checking whether a reference is local or global is non-trivial, requiring a barrier on every read.

To overcome these challenges, MULTIMLTON provides a number of new language and runtime abstractions and mechanisms, whose design, rationale, and implementation are the primary focus and contribution of this paper. Specifically,

1. We present ACML, an extension of CML that allows the definition of composable asynchronous operators, which is the primary programming model supported by MULTIMLTON.
2. We introduce the design and implementation of *parasites*, a runtime thread management mechanism intended to reduce the cost of supporting asynchrony. As the name suggests, parasites live on host threads, and enable efficient sharing of thread resources. In the fast path, when parasites do not block (i.e., when they perform synchronous communication where a partner exists, perform asynchronous communication, or purely functional computation), they only incur the overhead of a non-tail function call, and do not entail the creation of a new thread object. In the slow path, a parasite can be inflated to a full-blown thread if the computation or communication actions it performs would merit their execution as a separate thread of control.
3. A new garbage collector design tuned for ACML that obviates the need for read barriers. Conceptually, a thread performing an operation that would trigger lifting a local object to the shared heap is stalled, an operation we call *procrastination*, with the garbage collector suitably informed. Existing references to this object are properly repaired during the next local collection, thereby eliminating forwarding pointers, and the read barriers that check for them. A new object property called *cleanliness* enables a broad class of objects to be moved from a local to a shared heap without requiring a full traversal of the local heap to fix existing references; cleanliness serves as an important optimization that achieves the effect of procrastination without actually having to initiate a thread stall.
4. A detailed performance study is provided to quantify the impact of the MULTIMLTON runtime, especially with respect to the use of parasites and procrastination. This study is undertaken across a number of widely different multicore designs, including an 864 core Azul Vega 3, a 48 core Intel non-coherent Single-chip Cloud Computer (SCC), and a 48 core AMD-based coherent NUMA system.

The paper is organized as follows. In the next section, we give an informal description of ACML. Section 3 discusses the implementation of asynchrony in terms of parasites. Section 4 describes MULTIMLTON's GC design, providing details of procrastination and cleanliness. Section 5 quantifies the effectiveness of the runtime over a range of parallel benchmarks. Section 6 presents related work. Conclusions are given in Section 7.

```
spawn      : (unit -> 'a) -> threadID
sendEvt    : 'a chan * 'a -> unit Event
recvEvt    : 'a chan -> 'a Event
alwaysEvt  : 'a -> 'a Event
never      : 'a Event
sync       : 'a Event -> 'a
wrap       : 'a Event * ('a -> 'b) -> 'b Event
guard      : (unit -> 'a Event) -> 'a Event
choose     : 'a Event list -> 'a Event
```

Fig. 1. CML event operators.

2 ACML

In this section, we will provide an overview of the source language of MULTIMLTON, ACML, which allows asynchronous computations to be managed composably. This section motivates the need to effectively manage asynchrony in the runtime system on scalable multicore architectures.

2.1 Background: concurrent ML

Context: The programming model supported by MULTIMLTON is based on message-passing, and is heavily influenced by Concurrent ML (CML), a concurrent extension of Standard ML that utilizes synchronous message passing to enable the construction of synchronous communication protocols. Threads perform `send` and `recv` operations on typed channels; these operations block until a matching action on the same channel is performed by another thread.

CML also provides first-class synchronous *events* that abstract synchronous message-passing operations. An event value of type `'a Event` when synchronized on yields a value of type `'a`. An event value represents a potential computation, with latent effect until a thread synchronizes upon it by calling `sync`. The following equivalences thus therefore hold: `send(c, v)` $\equiv$ `sync(sendEvt(c,v))` and `recv(c)` $\equiv$ `sync(recvEvt(c))`. Notably, thread creation is *not* encoded as an event – the thread `spawn` primitive simply takes a thunk to evaluate as a separate thread, and returns a thread identifier that allows access to the newly created thread's state.

Besides `sendEvt` and `recvEvt`, there are other base events provided by CML. The `never` event, as its name suggests, is never available for synchronization; in contrast, `alwaysEvt` is always available for synchronization. These events are typically generated based on the (un)satisfiability of conditions or invariants that can be subsequently used to influence the behavior of more complex events built from the event combinators described below. Much of CML's expressive power derives from event combinators that construct complex event values from other events. We list some of these combinators in Figure 1. The expression `wrap (ev, f)` creates an event that, when synchronized, applies the result of synchronizing on event `ev` to function `f`. Conversely, `guard(f)` creates an event that, when synchronized, evaluates `f()` to yield event `ev` and then synchronizes on `ev`. The `choose` event combinator takes a list of events and constructs an event value that represents the non-deterministic choice of the events in the list; for example:

```
sync(choose[recvEvt(a),sendEvt(b,v)])
```

will either receive a unit value from channel `a`, or send value `v` on channel `b`. Selective communication provided by `choose` motivates the need for first-class events. We cannot, for example, simply build complex event combinators using function abstraction and composition because function closures do not allow inspection of the encapsulated computations, a necessary requirement for implementing combinators like `choose`.

2.2 Asynchronous events

While simple to reason about, synchronous events impose non-trivial performance penalties, requiring that both parties in a communication action be available before allowing either to proceed. To relax this condition, we wish to allow the expression of *asynchronous* composable events.

An asynchronous operation initiates two temporally distinct sets of actions. The first defines *post-creation* actions – these are actions that must be executed after an asynchronous operation has been initiated, without taking into account whether the effects of the operation have been witnessed by its recipients. For example, a post-creation action of an asynchronous send on a channel might initiate another operation on that same channel; the second action should take place with the guarantee that the first has already deposited its data on the channel. The second are *post-consumption* actions – these define actions that must be executed only after the effect of an asynchronous operation has been witnessed. For example, a post-consumption action might be a callback that is triggered when the client retrieves data from a channel sent asynchronously. These post-consumption actions take place within an implicit thread of control responsible for completing the asynchronous operation.

ACML introduces first-class *asynchronous* events with the following properties: *(*i) they are extensible both with respect to pre- and post-creation as well as pre- and post-consumption actions; *(*ii) they can operate over the same channels that synchronous events operate over, allowing both kinds of events to seamlessly co-exist; and, *(*iii) their visibility, ordering, and semantics is independent of the underlying runtime and scheduling infrastructure.

2.2.1 Base events

In order to provide primitives that adhere to the desired properties outlined above, we extend CML with a new asynchronous event type `('a,'b) AEvent` and the following two base events: `aSendEvt` and `aRecvEvt`, to create an asynchronous send event and an asynchronous receive event, respectively. The differences in their type signature from their synchronous counterparts reflect the split in the creation and consumption of the communication action they define:

```
sendEvt  : 'a chan * 'a -> unit Event
recvEvt  : 'a chan -> 'a Event
aSendEvt : 'a chan * 'a -> (unit, unit) AEvent
aRecvEvt : 'a chan -> (unit, 'a) AEvent
```

An `AEvent` value is parameterized with respect to the type of the event's post-creation and post-consumption actions. In the case of `aSendEvt`, both actions yield `unit`: when synchronized on, the event immediately returns a `unit` value and places its `'a` argument value on the supplied channel. The post-consumption action also yields `unit`. When synchronized on, an `aRecvEvt` returns `unit`; the type of its post-consumption action is `'a` reflecting the type of value read from the channel when it is paired with a send.

The semantics of both asynchronous send and receive guarantees that successive communication operations performed by the same thread get witnessed in the order in which they were issued. In this respect, an asynchronous send event shares functionality with a typical non-blocking send of the kind found in languages like Erlang (Armstrong *et al.*, 1996; Svensson *et al.*, 2010) or libraries like MPI (Li *et al.*, 2008). However, an asynchronous receive does not exhibit the same behavior as a typical non-blocking receive. Indeed, CML already provides polling methods that can be used to implement polling loops found in most non-blocking receive implementations. The primary difference between the two is that an asynchronous receive places itself on the channel at the point where it is synchronized (regardless of the availability of a matching send), while a non-blocking receive typically only queries for the existence of a value to match against, but does not alter the underlying channel structure. By actually depositing itself on the channel, an asynchronous receive thus provides a convenient mechanism to implement *ordered* asynchronous buffers or streams – successive asynchronous receives are guaranteed to receive data from a matching send in the order in which they were synchronized.

2.2.2 Event combinators

Beyond these base events, ACML also provides a number of combinators that serve as asynchronous versions of their CML counterparts. These combinators enable the extension of post-creation and post-consumption action of asynchronous events to create more complex events, and allow transformation between the synchronous and asynchronous events.

```
wrap  : 'a Event * ('a -> 'b) -> 'b Event
sWrap : ('a, 'b) AEvent * ('a -> 'c) -> ('c, 'b) AEvent
aWrap : ('a, 'b) AEvent * ('b -> 'c) -> ('a, 'c) AEvent

guard  : (unit -> 'a Event) -> 'a Event
aGuard : (unit -> ('a, 'b) AEvent) -> ('a, 'b) AEvent

choose  : 'a Event list -> 'a Event
aChoose : ('a, 'b) AEvent list -> ('a, 'b) AEvent
sChoose : ('a, 'b) AEvent list -> ('a, 'b) AEvent

aTrans : ('a, 'b) AEvent -> 'a Event
sTrans : 'a Event -> (unit, 'a) AEvent
```

Similar to CML `wrap` combinator, `sWrap` and `aWrap` extend the post-consumption and post-creation actions of an asynchronous event, respectively. `aGuard` allows creation of a guarded asynchronous event. `sChoose` is a blocking choice operator which blocks until one of the asynchronous base events has been consumed. `aChoose` is a non-blocking variant, which has the effect of non-deterministically choosing one of the base

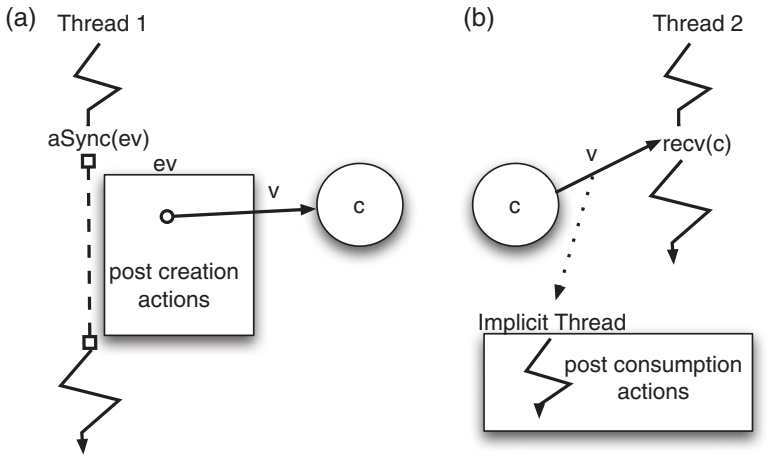


Fig. 2. The figure shows a complex asynchronous event `ev`, built from a base `aSendEvt`, being executed by Thread 1. When the event is synchronized via `aSync`, the value `v` is placed on channel `c` and post-creation actions are executed (see (a)). Afterwards, control returns to Thread 1. When Thread 2 consumes the value `v` from channel `c`, an implicit thread of control is created to execute any post-consumption actions (see (b)).

asynchronous events if none are available for immediate consumption. Finally, `aTrans` and `sTrans` allow transformation between the synchronous and asynchronous variants.

In order to keep this paper focused on the runtime system aspects of MULTIMLTON, the further details of these combinators have been elided. Full description of these combinators along with their formal semantics is available in Ziarek *et al.* (2011).

2.2.3 Event synchronization

```
sync  : 'a Event -> 'a
aSync : ('a, 'b) AEvent -> 'a
```

We also introduce a new synchronization primitive: `aSync`, to synchronize asynchronous events. The `aSync` operation fires the computation encapsulated by the asynchronous event of type `('a, 'b) AEvent`, returns a value of type `'a`, corresponding to the return type of the event's post-*creation* action. Unlike their synchronous variants, asynchronous events do *not* block if no matching communication is present. For example, executing an asynchronous send event on an empty channel places the value being sent on the channel and then returns control to the executing thread (see Figure 2(a)). In order to allow this non-blocking behavior, an *implicit* thread of control is created for the asynchronous event when the event is paired, or *consumed* as shown in Figure 2(b). If a receiver is present on the channel, the asynchronous send event behaves similarly to a synchronous event; it passes the value to the receiver. However, a new implicit thread of control is still created to execute any post-consumption actions.

Similarly, the synchronization of an asynchronous receive event does not yield the value received (see Figure 3); instead, it simply enqueues the receiving action on the channel.

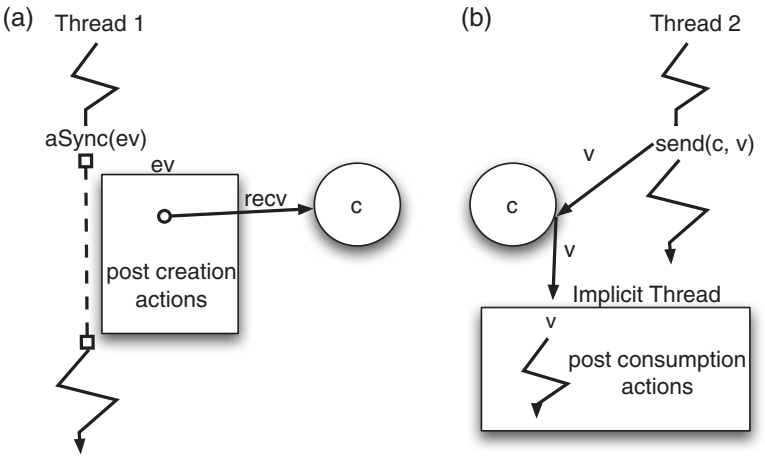


Fig. 3. The figure shows a complex asynchronous event `ev`, built from a base `aRecvEvt`, being executed by Thread 1. When the event is synchronized via `aSync`, the receive action is placed on channel `c` and post-creation actions are executed (see (a)). Afterwards, control returns to Thread 1. When Thread 2 sends the value `v` to channel `c`, an implicit thread of control is created to execute any post-consumption actions passing `v` as the argument (see (b)).

Therefore, the thread that synchronizes on an asynchronous receive always gets the value `unit`, even if a matching send exists. The actual value consumed by the asynchronous receive can be passed back to the thread which synchronized on the event through the use of combinators that process post-consumption actions. This is particularly well suited to encode reactive programming idioms: the post-consumption actions encapsulate a reactive computation.

To illustrate the differences between synchronous and asynchronous primitive events, consider the two functions `f` and `af` shown below:

```
1 fun  f () =
2   (spawn (fn () => sync (sendEvt(c, v)));
3    sync (sendEvt(c, v'));
4    sync (recvEvt(c)))
5 fun af () =
6   (spawn (fn () => sync (sendEvt(c, v)));
7    aSync (aSendEvt(c, v'));
8    sync (recvEvt(c)))
```

The function `f` will block if there is no recipient available for the send on line 3. Suppose there was a receiver thread, say `rt`, available to receive on channel `c`. Let us also assume that the program has no additional threads. Due to non-determinism, the sends on line 2 and 3 are both equally viable to match with the receive on `rt`. If the send on line 3 matched with the receive on `rt`, then the send on line 2 can match with the receive on line 4, and the control returns from the function `f`. On the other hand, if the receive on `rt` matches with the send on line 2, the function `f` will block at line 3, waiting for a matching receive.

Unlike the function `f`, the function `af` will not block, even if a receiver thread `rt` is not available. The receive on line 8 may see either the value `v` or `v'`, since the asynchronous

send event (line 7) *only* asserts that the value `v'` has been placed on the channel and *not* that it has been consumed. If we swapped lines 6 and 7, the receive operation on line 8 is guaranteed to read `v'`. While asynchronous events do not block, they still enforce *ordering* constraints that reflect the order in which they were synchronized.

The definition of ACML's base events involves the creation of an implicit thread of control used to handle the completion of the asynchronous action, evaluating any post-consumption actions associated with the asynchronous event. A scalable and efficient runtime system for ACML must ensure that the cost to manage these implicit threads is not prohibitive, especially given the likely possibility that there may be *no* post-consumption actions to execute. In the following section, we describe *parasites*, a runtime-managed data structure that can be used to implement extremely cheap threads, useful for precisely this purpose.

3 Parasitic threads

In this section, we focus on the parasitic threading system of MULTIMLTON (Sivaramakrishnan *et al.*, 2010), which is specifically suited for implementing the implicit thread creation involved in the synchronization of an asynchronous event. We first describe the baseline threading system of MULTIMLTON– an *n* over *m* threading system that leverages potentially many lightweight (language level) threads multiplexed over a single kernel thread – and illustrate why lightweight threads might not be best suited to implement the implicit threads created by asynchronous events. We then describe parasitic threads and their implementation in MULTIMLTON, focusing on their utility in the realization of implicit thread creation and management.

3.1 Lightweight threads and scheduler

MULTIMLTON's runtime system is built on top of lightweight, user-level threads. The user-level thread scheduler is in turn implemented using the `MLton.Thread` (MLton, 2012) library, which provides one-shot continuations. `MLton.Thread` uses a variation of Bruggeman *et al*.'s (1996) strategy for implementing one-shot continuations. A `MLton.Thread` is a lightweight data structure that represents a paused computation, and encapsulates the metadata associated with the thread as well as a stack. The stack associated with the lightweight thread is allocated on the heap, and is garbage collected when the corresponding thread object is no longer reachable.

Unlike Bruggeman *et al.*'s use of linked stack segments, the stack layout is contiguous in MULTIMLTON. We utilize contiguous stack layout in order to avoid the hot-split problem, where an out of stack function call may be invoked repeatedly in a tight-loop, forcing allocation of a new segment, which is immediately freed upon return. Even if the stack segments were cached, the performance tends to be unpredictable as seen in other concurrent, garbage collected language runtimes such as Go (Hot-Split, 2013) and Rust (Stack Thrashing, 2013). On stack overflow, a larger stack segment is allocated on the heap, the old stack frames are copied to this new segment, and the old stack segment is garbage collected.

As such, `MLton.Thread` does not include a default scheduling mechanism. Instead, MULTIMLTON builds a preemptive, priority supported, run-queue based, multicore-capable scheduler using `MLton.Thread`. Building multicore schedulers over continuations in this way is not new, first described by Wand (1980), and successfully emulated by a number of modern language implementations (Auhagen *et al.*, 2011; GHC, 2014).

Implementing MULTIMLTON's threading system over one-shot continuation as opposed to full-fledged (multi-shot) continuations greatly reduces the cost of the thread and scheduler implementation. In particular, if full-fledged continuations were used to implement the scheduler, then during every thread switch, a *copy* of the current thread would have to be made to reify the continuation. This is, in our context, unnecessary since the current stack of the running thread (as opposed to the saved stack in the continuation), will never be accessed again. One-shot continuations avoid copying the stack altogether; during a thread switch, a reference to the currently running thread is returned to the programmer. The result is a very efficient baseline scheduler.

Lightweight threads are garbage collected when no longer reachable. MULTIMLTON's threading system multiplexes many lightweight thread on top of a few operating system threads. Each kernel thread represents a virtual processor and one kernel thread is pinned to each processor. The number of kernel threads is determined statically and is specified by the user; they are not created during program execution.

3.2 Asynchrony through lightweight threads

Lightweight threads provide a conceptually simple language mechanism for achieving asynchrony. Threads, unlike specialized asynchronous primitives, are general purpose: they act as vessels for *arbitrary* computation. Unfortunately, harnessing threads for asynchrony typically comes at a cost. Instead of utilizing threads where asynchrony could be *conceptually* leveraged, one must often reason about whether the runtime cost of managing the thread may outweigh the benefit of performing the desired computation asynchronously. We can loosely categorize the cost of running lightweight threads into three groups:

- **Synchronization costs**: The creation of a burst of lightweight threads within a short period of time increases contention for shared resources such as channels and scheduler queues.
- **Scheduling costs**: Besides typical scheduling overheads, the lightweight threads that are created internally by the asynchronous primitive might not be scheduled *prior* to threads explicitly created by the programmer. In such a scenario, the completion of a primitive, implicitly creating threads for asynchrony, is delayed. In the presence of tight interaction between threads, such as synchronous communication, if one of the threads is slow, progress is affected in all of the transitively dependent threads.
- **Garbage collection costs**: Creating a large number of threads, especially in bursts, increases allocation burden and might subsequently trigger a collection. This problem becomes worse in a parallel setting, when multiple mutators might be allocating in parallel.

It is precisely for this reason that specialized primitives for asynchrony are typically the *de facto* standard for most programming languages.

3.3 Characteristics of implicit threads

It is worthwhile to first consider the unique properties of implicit post-consumption actions in ACML, which are created as a result of synchronizing on an asynchronous event (referred to as *implicit threads* in the sequel). These characteristics are the reason why full-fledged lightweight threads are not the best vehicle for implementing implicit threads. From these characteristics, we derive a set of requirements, which motivates our design.

Communication-intensive: Implicit threads arising out of asynchronous events are typically communication intensive i.e., the computations encapsulated in post-consumption actions are small when compared to the communication actions performed by threads. The typical case is synchronization on a base asynchronous event that is readily available to match, without any post-consumption action wrapped to it. In such a scenario, it would be completely unnecessary to create the implicit thread in the first place.

> ***Requirement 1.*** *The cost of creation of an implicit thread should be much cheaper than explicit threads.*

High probability of blocking: However, there is some chance that the implicit thread will block on a communication. In general, it is difficult to assert if an asynchronous action will block when synchronized. If we polled for satisfiability, the state of the system might change between polling and actual synchronization. In addition, the asynchronous action might be composed of multiple blocking sub-actions, not all of which might expose polling mechanisms.

> ***Requirement 2.*** *Implicit threads should have the ability to be efficiently blocked and resumed.*

Might be long-lived: Post-consumption actions do not impose any restriction on the encapsulated computation, and might well be a very long-lived, computation-intensive activity. In this case, we need to be able to exploit the availability of multiple processing cores at our disposal. Thus, the new threading mechanism must be able to adapt for long running computations and multi-core exploitation.

> ***Requirement 3.*** *Implicit threads should have the ability to be reified into full-fledged threads.*

3.4 Parasitic threads

With the aim of satisfying these requirements, the MULTIMLTON runtime system supports a new threading mechanism that is well suited to encapsulate the implicit threads created by asynchronous actions. Our runtime supports two kinds of threads: hosts and parasites. Host threads map directly to lightweight threads in the runtime. Parasitic threads can encapsulate arbitrary computation, just like host threads. However, unlike a regular thread,

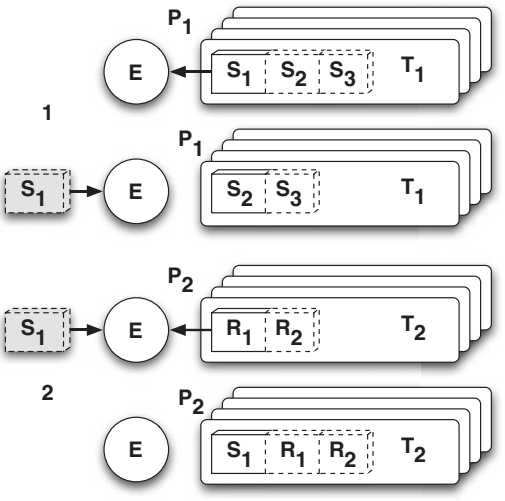


Fig. 4. Blocking and unblocking of parasitic threads.

a parasitic thread executes using the execution context of the host that creates the parasite; it is intended primarily to serve as the execution vehicle for asynchronous actions.

Parasitic threads are implemented as raw frames living within the stack space of a given host thread. A host thread can hold an arbitrary number of parasitic threads. In this sense, a parasitic thread views its host in much the same way as a user-level thread might view a kernel-level thread that it executes on. A parasite is suspended when it performs a blocking action (e.g., a synchronous communication operation, or I/O). Such a suspended parasite is said to have been *reified*. Reified parasites are represented as stack objects on the heap. Reified parasites can resume execution once the conditions that had caused it to block no longer hold. Thus, parasitic threads are not scheduled using the language runtime; instead they *self-schedule* in a demand-driven style based on flow properties dictated by the actions they perform.

Figure 4 shows the steps involved in a parasitic communication, or blocking event, and we illustrate the interaction between the parasitic threads and their hosts. The host threads are depicted as rounded rectangles, parasitic threads are represented as blocks within their hosts, and each processor as a queue of host threads. The parasite that is currently executing on a given host and its stack is represented as a block with solid edges; other parasites are represented as blocks with dotted edges. Reified parasites are represented as shaded blocks. Host threads can be viewed as a collection of parasitic threads all executing within the same stack space. When a host thread is initially created it contains one such computation, namely the expression it was given to evaluate when it was spawned.

Initially, the parasite S_1 performs a blocking action on a channel or event, abstractly depicted as a circle. Hence, S_1 blocks and is reified. The thread T_1 that hosted S_1 continues execution by switching to the next parasite S_2. S_1 becomes runnable when it is unblocked. Part 2 of the figure shows the parasite R_1 on the thread T_2 invoking an unblocking action.

```
type 'a par
type ready_par
val spawnParasite : (unit -> unit) -> unit
val reify         : ('a par -> unit) -> 'a
val prepare       : ('a par * 'a) -> ready_par
val attach        : ready_par -> unit
val inflate       : unit -> unit
```

Fig. 5. (Colour online) Parasitic thread management API.

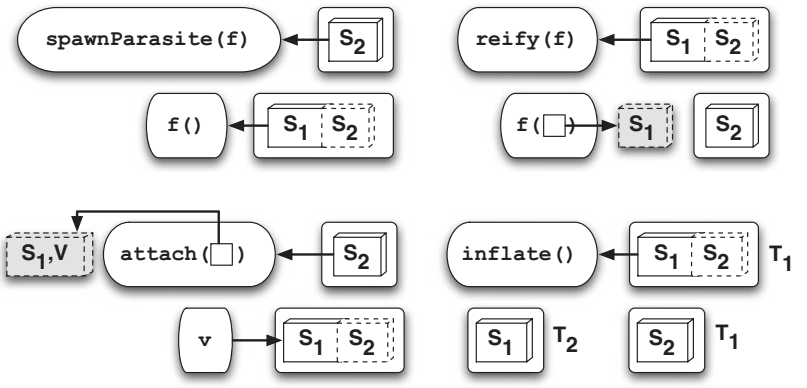


Fig. 6. Behavior of the parasite API.

This unblocks S_1 and schedules it on top of R_1. Thus, the parasitic threads implicitly migrate to the point of synchronization.

3.5 An API for parasitic threads

Although parasites are a runtime mechanism, a parasite management API is exposed to the programmer as a library (see Figure 5) that can be used to build higher-level asynchronous abstractions. In Section 3.8, we will illustrate how ACML primitives are constructed using this API. The primitives exposed by the parasite API are similar to continuations, but differ primarily in the way they are scheduled and migrated. Figure 6 illustrates the behavior of the primitives.

Unlike host threads, parasitic threads are implemented as raw stack frames. The expression `spawnParasite(f)` pushes a new frame to evaluate expression `f`, similar to a function call. We record the stack top at the point of invocation. This corresponds to the caller's continuation and is a *de facto* boundary between the parasite and its host (or potentially another parasite). If the parasite does not block, the computation runs to completion and control returns to the caller, just as if the caller made a non-tail procedure call.

A parasite can voluntarily block by invoking `reify(f)`. The function `f` is applied to the current parasite, similar to the invocation of `call-with-current-continuation`. Once the parasite has been reified, control immediately switches to the next parasite on the

host stack. Reified parasites are represented as stack objects on the heap, reachable from the resource (e.g. , a channel) the parasite is blocked on.

A reified parasitic computation is a value of type `'a par`, which expects a value of type `'a` for the parasitic computation to be resumed. Such a parasite typically is waiting for an event, and the value associated with the result of the event. For example, a parasite might be blocked on a channel receive action which, when resumed, is expected to continue with the value received over the channel. For this purpose, a `prepare` primitive is provided. A parasite prepared with a value `v` resumes supplying `v` as the return value of the `reify` call that caused it to block.

Prepared parasites can be resumed with the `attach` primitive. A host invoking `attach` on a prepared parasite, copies the parasitic stack on top of its current stack and switches to the newly attached parasite. The newly attached parasite resumes execution with the value it was prepared with. When a parasite runs to completion, control returns to the parasite residing below it as if a non-tail call has returned.

Although parasites are envisioned to be used as short-lived threads, they can encode arbitrarily long computations since the programming model places no restrictions on the computation they encapsulate. Executing multiple long running computations on the same stack is unfavorable as it serializes executions. To address this concern, the API provides an `inflate` primitive to convert a parasitic thread into a host thread, which can then be scheduled by MULTIMLTON's multicore scheduler.

Let the parasite invoking `inflate` primitive be `pt`. The inflate primitive works by creating a new host thread `ht`, and copying the `pt`'s stack frames to this `ht`'s stack segment. `pt` is killed by popping its frames off its host stack, which switches control to the parasite residing below on this host. The new host thread `ht` is then added to the host thread scheduler. The overall effect of these operations is such that the original parasite `pt`'s computation is now resumed in the host thread `ht`.

Finally, relating this API to the requirements listed in Section 3.3, `spawnParasite` satisfies the first requirement, `reify`, `prepare`, and `attach` satisfies the second, and `inflate` satisfies the third.

3.6 Formal semantics

We define an operational semantics that models host threads in terms of a stack of parasites, and parasites as a stack of frames. Transitions in our formalism are defined through stack-based operations. Our semantics is given in terms of a core call-by-value functional language with threading and communication primitives. New threads of control (host threads) are explicitly created through a `spawnHost` primitive. We can view a host thread as a *container* for computations, which are represented by parasitic threads.

To model parasitic thread creation, we extend this language with two additional primitives - `spawnParasite` to create a parasite, and `inflate` to inflate a parasite into a host thread. Computation in our system is thus split between host threads, which behave as typical threads in a language runtime, and parasitic threads, which behave as asynchronous operations with regard to their host. In addition, we provide primitives to `reify` and `attach` parasites, which behave similarly to `callcc` and `throw`, except that `reify`

additionally removes the parasite from the execution stack, implicitly transferring control to the next parasitic or host thread.

We formalize the behavior of parasitic threads in terms of an operational semantics expressed using a CEK machine (Felleisen & Friedman, 1986) formulation. A CEK machine is small-step operational definition that operates over program states. A state is composed of an expression being evaluated, an environment, and the continuation (the remainder of the computation) of the expression. The continuation is modeled as a *stack* of frames.

3.6.1 Language

In the following, we write $\overline{\mathtt{v}}$ to denote a sequence of zero or more elements. In our semantics, we deal with two kinds of sequences – a sequence (stack) of parasites modeling a host thread, and a sequence (stack) of frames modeling a parasite. For readability, we use distinct notations for each sequence. For the sequence of frames, we write . for the empty sequence, and : for sequence concatenation. For the sequence of parasites, we write $\emptyset$ for an empty sequence, and $\triangleleft$ for concatenation. We assume that : has higher precedence than $\triangleleft$. Relevant domains and meta-variables used in the semantics are shown in Figure 7.

In our semantics, we use stack frames to capture intermediate computation state, to store environment bindings, to block computations waiting for synchronization, and to define the order of evaluation. We define ten unique types of frames: return frames, argument frames, function frames, receive frames, send frames, send value frames, receive and send blocked frames, and prepare and prepare value frames. The return frame pushes the resulting value from evaluating an expression on to the top of the stack. The value pushed on top of the stack gets propagated to the frame beneath the return frame (see Figure 9 RETURN TRANSITIONS).

The receive and send blocked frames signify that a parasite is blocked on a send or receive on a global channel. They are pushed on top of the stack to prevent further evaluation of the given parasitic computation. Only a communication across a global channel can pop a blocked frame. Once this occurs, the parasite can resume its execution. Argument and function frames enforce left-to-right order of evaluation. Similarly, the send and send value frames define the left to right evaluation of the `send` primitive. The prepare and prepare value frames are used for order of evaluation of the prepare primitive and to prepare a reified parasite respectively.

Our semantics is defined with respect to a program state that is composed of two maps – a thread map $\mathcal{T}$ that maps thread identifiers (`t`) to thread states (`s`) and a channel map $\mathcal{C}$. The channel map $\mathcal{C}$ maps channels (`c`) to a pair $\langle \overline{\mathtt{l_s}}, \overline{\mathtt{l_r}} \rangle$, where $\overline{\mathtt{l_s}}$ ($\overline{\mathtt{l_r}}$) is the sequence of parasites blocked on a channel send (receive) operation, with the top frame necessarily being the send (receive) block frame.

A thread is a pair composed of a thread identifier and a thread state. A (host) thread state (`s`) is a CEK machine state extended with support for parasitic threads. Therefore, a host thread is a collection of stacks, one for each parasitic thread. A concrete thread state can be in one of three configurations: a control state, a return state, or a halt state. A *control state* is composed of an expression (`e`), the current environment (`r`) — a mapping between variables and values, the current stack (`k`), as well as a stack of parasitic computations ($\overline{\mathtt{k}}$). A *return state* is simply a stack of parasitic computations ($\overline{\mathtt{k}}$). The *halt state* is reached

$$
\begin{array}{rcl}
\mathtt{e} \in \mathit{Exp} & ::= & \mathtt{x} \mid \mathtt{v} \mid \mathtt{e(e)} \mid \mathtt{spawnHost(e)} \mid \mathtt{spawnParasite(e)} \\
& \mid & \mathtt{attach(e)} \mid \mathtt{prepare(e,e)} \mid \mathtt{reify(e)} \mid \mathtt{inflate(e)} \\
& \mid & \mathtt{chan()} \mid \mathtt{send(e,e)} \mid \mathtt{recv(e)} \\
\mathtt{v} \in \mathit{Val} & ::= & \mathtt{unit} \mid (\lambda \mathtt{x.e,r}) \mid \mathtt{c} \mid \kappa
\end{array}
$$

$$
\begin{array}{rcrcl}
\kappa & \in & \mathtt{Constant} & & \\
\mathtt{c} & \in & \mathtt{Channel} & & \\
\mathtt{x} & \in & \mathtt{Var} & & \\
\mathtt{t} & \in & \mathtt{ThreadID} & & \\
\mathtt{r} & \in & \mathtt{Env} & = & \mathtt{Var} \to \mathtt{Value} \\
\mathtt{k} & \in & \mathtt{Par} & = & \mathtt{Frame}* \\
\mathtt{v} & \in & \mathtt{Value} & = & \mathtt{Unit} + \mathtt{Closure} + \mathtt{Channel} + \mathtt{Constant} + \mathtt{Par} \\
(\lambda \mathtt{x.e,r}) & \in & \mathtt{Closure} & = & \mathtt{LambdaExp} \times \mathtt{Env}
\end{array}
$$

$$
\begin{array}{rcrcl}
\mathit{ret}\lceil \mathtt{v} \rceil & \in & \mathtt{RetFrame} & = & \mathtt{Value} \\
\mathit{arg}\lceil \mathtt{e,r} \rceil & \in & \mathtt{ArgFrame} & = & \mathtt{Exp} \times \mathtt{Env} \\
\mathit{fun}\lceil \mathtt{v} \rceil & \in & \mathtt{FunFrame} & = & \mathtt{Value} \\
\mathit{recv}\lceil \rceil & \in & \mathtt{RecvFrame} & = & \mathtt{Empty} \\
\mathit{sval}\lceil \mathtt{e,r} \rceil & \in & \mathtt{SValFrame} & = & \mathtt{Exp} \times \mathtt{Env} \\
\mathit{send}\lceil \mathtt{v} \rceil & \in & \mathtt{SendFrame} & = & \mathtt{Value} \\
\mathit{rblk}\lceil \mathtt{v} \rceil & \in & \mathtt{RblkFrame} & = & \mathtt{Value} \\
\mathit{sblk}\lceil \mathtt{v}_1, \mathtt{v}_2 \rceil & \in & \mathtt{SblkFrame} & = & \mathtt{Channel} \times \mathtt{Value} \\
\mathit{pval}\lceil \mathtt{e,r} \rceil & \in & \mathtt{PValFrame} & = & \mathtt{Exp} \times \mathtt{Env} \\
\mathit{prep}\lceil \mathtt{v} \rceil & \in & \mathtt{PrepFrame} & = & \mathtt{Value}
\end{array}
$$

$$
\begin{array}{rcrcl}
 & & \mathtt{ProgramState} & = & \mathcal{T} \times \mathcal{C} \\
\mathcal{T} & \in & \mathtt{ThreadMap} & = & \mathtt{ThreadID} \to \mathtt{ThreadState} \\
\mathcal{C} & \in & \mathtt{ChannelMap} & = & \mathtt{Channel} \to \mathtt{ChannelState} \\
\mathtt{s} & \in & \mathtt{ThreadState} & = & \mathtt{ControlState} + \mathtt{ReturnState} + \mathtt{HaltState} \\
 & & \mathtt{ChannelState} & = & \mathtt{Par}* \times \mathtt{Par}* \\
\langle \mathtt{e,r,k,\overline{k}} \rangle & \in & \mathtt{ControlState} & = & \mathtt{Exp} \times \mathtt{Env} \times \mathtt{Par} \times \mathtt{Par}* \\
\langle \mathtt{\overline{k}} \rangle & \in & \mathtt{ReturnState} & = & \mathtt{Par}* \\
\mathtt{halt(v)} & \in & \mathtt{HaltState} & = & \mathtt{Value}
\end{array}
$$

Fig. 7. Domains for the CEK machines extended with host threads and parasites.

when all parasitic threads in a given host thread have completed. A host thread, therefore, is composed of a stack of parasitic threads executing within its stack space.

Evaluation proceeds by non-deterministically choosing one of the host threads in the thread map $\mathcal{T}$. When a thread transitions to a control state, the parasite on top of the stack is evaluated. Once the parasite transitions to a return state, evaluation is free to choose the next host thread in the thread map $\mathcal{T}$ to evaluate subsequently.

3.6.2 CEK machine semantics

The rules given in Figures 8 and 9 define the transitions of the CEK machine. There are three types of transitions: control transitions, return transitions, and global transitions. Control and return transitions are thread local actions, while global transitions affect global state. We utilize the two types of local transitions to distinguish between states in which an

GLOBAL TRANSITIONS

(LocalEvaluation)
$$\frac{\mathtt{s} \longrightarrow \mathtt{s}'}{\langle \mathcal{T}[\mathtt{t} \mapsto \mathtt{s}], \mathcal{C}\rangle \longrightarrow \langle \mathcal{T}[\mathtt{t} \mapsto \mathtt{s}'], \mathcal{C}\rangle}$$

(Channel)
$$\frac{\mathtt{c}\ \textit{fresh}}{\begin{array}{c}\langle \mathcal{T}[\mathtt{t} \mapsto \langle \mathtt{chan}(), \mathtt{r}, \mathtt{k}, \overline{\mathtt{k}}\rangle], \mathcal{C}\rangle \longrightarrow \\ \langle \mathcal{T}[\mathtt{t} \mapsto \langle \mathit{ret}\lceil \mathtt{c}\rceil : \mathtt{k} \lhd \overline{\mathtt{k}}\rangle], \mathcal{C}[c \mapsto \langle \emptyset, \emptyset\rangle]\rangle\end{array}}$$

(SpawnHost)
$$\frac{\mathtt{t}'\ \textit{fresh}}{\begin{array}{c}\langle \mathcal{T}[\mathtt{t} \mapsto \langle \mathtt{spawnHost}(\mathtt{e}), \mathtt{r}, \mathtt{k}, \overline{\mathtt{k}}\rangle], \mathcal{C}\rangle \longrightarrow \\ \langle \mathcal{T}[\mathtt{t} \mapsto \langle \mathit{ret}\lceil \mathtt{unit}\rceil : \mathtt{k} \lhd \overline{\mathtt{k}}\rangle, \mathtt{t}' \mapsto \langle \mathtt{e}, \mathtt{r}, .., \emptyset\rangle], \mathcal{C}\rangle\end{array}}$$

(Inflate)
$$\frac{\mathtt{t}'\ \textit{fresh}}{\begin{array}{c}\langle \mathcal{T}[\mathtt{t} \mapsto \langle \mathtt{inflate}(\mathtt{e}), \mathtt{r}, \mathtt{k}, \overline{\mathtt{k}}\rangle], \mathcal{C}\rangle \longrightarrow \\ \langle \mathcal{T}[\mathtt{t} \mapsto \langle \overline{\mathtt{k}}\rangle, \mathtt{t}' \mapsto \langle \mathtt{e}, \mathtt{r}, \mathtt{k}, \emptyset\rangle], \mathcal{C}\rangle\end{array}}$$

(SendReify)
$$\begin{array}{c}\langle \mathcal{T}[\mathtt{t} \mapsto \langle \mathit{sblk}\lceil \mathtt{c}, \mathtt{v}\rceil : \mathtt{k} \lhd \overline{\mathtt{k}}\rangle], \mathcal{C}[\mathtt{c} \mapsto \langle \overline{\mathtt{l_s}}, \emptyset\rangle]\rangle \longrightarrow \\ \langle \mathcal{T}[\mathtt{t} \mapsto \langle \overline{\mathtt{k}}\rangle], \mathcal{C}[\mathtt{c} \mapsto \langle \overline{\mathtt{l_s}} \lhd \mathit{sblk}\lceil \mathtt{c}, \mathtt{v}\rceil : k, \emptyset\rangle]\rangle\end{array}$$

(RecvReify)
$$\begin{array}{c}\langle \mathcal{T}[\mathtt{t} \mapsto \langle \mathit{rblk}\lceil \mathtt{c}\rceil : \mathtt{k} \lhd \overline{\mathtt{k}}\rangle], \mathcal{C}[\mathtt{c} \mapsto \langle \emptyset, \overline{\mathtt{l_r}}\rangle]\rangle \longrightarrow \\ \langle \mathcal{T}[\mathtt{t} \mapsto \langle \overline{\mathtt{k}}\rangle], \mathcal{C}[\mathtt{c} \mapsto \langle \emptyset, \overline{\mathtt{l_r}} \lhd \mathit{rblk}\lceil \mathtt{c}\rceil : \mathtt{k}\rangle]\rangle\end{array}$$

(SendMatch)
$$\begin{array}{c}\langle \mathcal{T}[\mathtt{t} \mapsto \langle \mathit{sblk}\lceil \mathtt{c}, \mathtt{v}\rceil : \mathtt{k}_1 \lhd \overline{\mathtt{k}_1}\rangle], \mathcal{C}[\mathtt{c} \mapsto \langle \overline{\mathtt{l_s}}, \mathit{rblk}\lceil \mathtt{c}\rceil : \mathtt{k}_2 \lhd \overline{\mathtt{l_r}}\rangle]\rangle \longrightarrow \\ \langle \mathcal{T}[\mathtt{t} \mapsto \langle \mathit{ret}\lceil \mathtt{v}\rceil : \mathtt{k}_2 \lhd \mathit{ret}\lceil \mathtt{unit}\rceil : \mathtt{k}_1 \lhd \overline{\mathtt{k}_1}\rangle], \mathcal{C}[\mathtt{c} \mapsto \langle \overline{\mathtt{l_s}}, \overline{\mathtt{l_r}}\rangle]\rangle\end{array}$$

(RecvMatch)
$$\begin{array}{c}\langle \mathcal{T}[\mathtt{t} \mapsto \langle \mathit{rblk}\lceil \mathtt{c}\rceil : \mathtt{k}_1 \lhd \overline{\mathtt{k}_1}\rangle], \mathcal{C}[\mathtt{c} \mapsto \langle \mathit{sblk}\lceil \mathtt{c}, \mathtt{v}\rceil : \mathtt{k}_2 \lhd \overline{\mathtt{l_s}}, \overline{\mathtt{l_r}}\rangle]\rangle \longrightarrow \\ \langle \mathcal{T}[\mathtt{t} \mapsto \langle \mathit{ret}\lceil \mathtt{unit}\rceil : \mathtt{k}_2 \lhd \mathit{ret}\lceil \mathtt{v}\rceil : \mathtt{k}_1 \lhd \overline{\mathtt{k}_1}\rangle], \mathcal{C}[\mathtt{c} \mapsto \langle \overline{\mathtt{l_s}}, \overline{\mathtt{l_r}}\rangle]\rangle\end{array}$$

Fig. 8. Global evaluation rules defined in terms of thread states ($\mathcal{T}$).

expression is being evaluated from those in which an expression has already been evaluated to a value. In the latter case, the value is propagated to its continuation. Global transitions are transitions that require global coordination, such as the creation of a new channel or thread, or a communication action.

Global Transitions. There are eight rules that define global transitions given in Figure 8. Rule `LocalEvaluation` states that a thread with thread state `s` can transition to a new state $\mathtt{s}'$ if it can take a local transition from `s` to $\mathtt{s}'$. This rule subsumes non-deterministic thread scheduling, and defines global state change in terms of operations performed by individual threads. The second rule, `Channel`, defines the creation of a new global channel. The channel map is extended with a new binding corresponding to the new channel, with empty sequences for the list of blocked parasites. The `SpawnHost` rule governs the creation of a new host thread; this rule generates a unique thread identifier and begins the evaluation

of the spawned expression (`e`) in the parent thread's (`t`) environment (`r`). Inflation (rule `Inflate`) is similar to the rule for spawning a thread, except the stack associated with the parasite that calls `inflate` is removed from its current host thread and added to the newly created thread.

The last four rules are concerned with channel communication. Rule `SendReify` handles the case when a parasite performing a send on a channel does not find a matching receive. In this case, the parasite is reified and appended to the back of the list of blocked sender parasites. Control switches to the parasite residing below the reified parasite. The rule `RecvReify` is the dual of the `SendReify` rule and handles a parasite blocking on a receive operation.

The rule `SendMatch` handles the case when a parasite performing a send finds a waiting receiver on the channel. In this case, the receiver parasite on the front of the sequence is removed, and its stack is prepared so that the receiver parasite resumes with the value from the sender. The receiver parasite is attached to the top of the host thread that performed the send. Thus, the blocked receiver parasite migrates to the same host as the sender parasite. Rule `RecvMatch` is the dual of `SendMatch`.

Notice that the communication rules preserve the property that parasites continue to evaluate until they are blocked on a channel communication. As a consequence, if two parasitic computations are spawned one after the other, each performing a single `send` operation on the same channel, the order in which the sends deposit values on the channel is the same as the order in which the parasites were spawned. We utilize this property of parasites to ensure the ordering invariants required by ACML (Section 2.2.3).

Control Transitions. There are ten rules that define local control transitions. Because the definition of these rules are standard, we omit their explanation here, with the exception of the rules that explicitly deal with parasites (`Reify`, `Prepare`, `Attach`, `SpawnParasite`). The `Reify` rule depicts the execution of the reify primitive, which, like `callcc` takes a function as an argument. This rule rewrites the expression into a function call, passing the continuation of parasite as the argument, and discards the parasite's continuation on the current host. As a result, once the function call returns, the control implicitly switches to the next parasite on the stack.

The `Prepare` rules evaluates the first argument, which should evaluate to a parasite, and pushed the second expression onto the stack. The `Attach` rule installs a parasite into the current host thread and switches execution to this parasite. The `SpawnParasite` rule models the creation of a new parasitic thread within the current thread. The currently evaluating parasitic thread is added back to the set of parasites with a unit return value pushed on its stack. The expression is evaluated in a new parasitic thread constructed with the environment of the parent and an empty stack. Thread execution undertakes evaluation of the expression associated with this new parasite.

Return Transitions. There are nine rules that define local return transitions. These rules, like local control transitions, are mostly standard. We comment on the rules that involve thread and parasite management. Rule `ThreadHalt` defines thread termination via a transition to a halt state. A thread transitions to a halt state if it has no active parasites and its stack is empty except for a return frame. Parasites themselves are removed by the `ParasiteHalt` rule. The return value of a thread is thus defined as the last value produced

CONTROL TRANSITIONS

$$
\begin{array}{lrcl}
(\texttt{Constant}) & \langle \kappa, \mathtt{r}, \mathtt{k}, \overline{\mathtt{k}} \rangle & \longrightarrow & \langle \mathit{ret}\lceil \kappa \rceil : \mathtt{k} \triangleleft \overline{\mathtt{k}} \rangle \\
(\texttt{Variable}) & \langle \mathtt{x}, \mathtt{r}, \mathtt{k}, \overline{\mathtt{k}} \rangle & \longrightarrow & \langle \mathit{ret}\lceil \mathtt{r}(\mathtt{x}) \rceil : \mathtt{k} \triangleleft \overline{\mathtt{k}} \rangle \\
(\texttt{Closure}) & \langle \lambda \mathtt{x}.\mathtt{e}, \mathtt{r}, \mathtt{k}, \overline{\mathtt{k}} \rangle & \longrightarrow & \langle \mathit{ret}\lceil \lambda \mathtt{x}.\mathtt{e}, \mathtt{r} \rceil : \mathtt{k} \triangleleft \overline{\mathtt{k}} \rangle \\
(\texttt{Application}) & \langle (\mathtt{e}_1\,\mathtt{e}_2), \mathtt{r}, \mathtt{k}, \overline{\mathtt{k}} \rangle & \longrightarrow & \langle \mathtt{e}_1, \mathtt{r}, \mathit{arg}\lceil \mathtt{e}_2, \mathtt{r} \rceil : \mathtt{k}, \overline{\mathtt{k}} \rangle \\
(\texttt{Reify}) & \langle \mathtt{reify}(\mathtt{e}), \mathtt{r}, \mathtt{k}, \overline{\mathtt{k}} \rangle & \longrightarrow & \langle (\mathtt{e}\ \mathtt{k}), \mathtt{r}, .., \overline{\mathtt{k}} \rangle \\
(\texttt{Send}) & \langle \mathtt{send}(\mathtt{e}_1, \mathtt{e}_2), \mathtt{r}, \mathtt{k}, \overline{\mathtt{k}} \rangle & \longrightarrow & \langle \mathtt{e}_1, \mathtt{r}, \mathit{sval}\lceil \mathtt{e}_2, \mathtt{r} \rceil : \mathtt{k}, \overline{\mathtt{k}} \rangle \\
(\texttt{Prepare}) & \langle \mathtt{prepare}(\mathtt{e}_1, \mathtt{e}_2), \mathtt{r}, \mathtt{k}, \overline{\mathtt{k}} \rangle & \longrightarrow & \langle \mathtt{e}_1, \mathtt{r}, \mathit{pval}\lceil \mathtt{e}_2, \mathtt{r} \rceil : \mathtt{k}, \overline{\mathtt{k}} \rangle \\
(\texttt{Receive}) & \langle \mathtt{recv}(\mathtt{e}), \mathtt{r}, \mathtt{k}, \overline{\mathtt{k}} \rangle & \longrightarrow & \langle \mathtt{e}, \mathtt{r}, \mathit{recv}\lceil \rceil : \mathtt{k}, \overline{\mathtt{k}} \rangle \\
(\texttt{SpawnParasite}) & \langle \mathtt{spawnParasite}(\mathtt{e}), \mathtt{r}, \mathtt{k}, \overline{\mathtt{k}} \rangle & \longrightarrow & \langle \mathtt{e}, \mathtt{r}, .., (\mathit{ret}\lceil \mathtt{unit} \rceil : \mathtt{k}) \triangleleft \overline{\mathtt{k}} \rangle \\
(\texttt{AttachParasite}) & \langle \mathtt{attach}\ \mathtt{k}', \mathtt{r}, \mathtt{k}, \overline{\mathtt{k}} \rangle & \longrightarrow & \langle \mathtt{k}' \triangleleft (\mathit{ret}\lceil \mathtt{unit} \rceil : \mathtt{k}) \triangleleft \overline{\mathtt{k}} \rangle
\end{array}
$$

RETURN TRANSITIONS

$$
\begin{array}{lrcl}
(\texttt{ThreadHalt}) & \langle \mathit{ret}\lceil \mathtt{v} \rceil : . \triangleleft \emptyset \rangle & \longrightarrow & \mathtt{halt}(\mathtt{v}) \\
(\texttt{ParasiteHalt}) & \langle \mathit{ret}\lceil \mathtt{v} \rceil : . \triangleleft \overline{\mathtt{k}} \rangle & \longrightarrow & \langle \overline{\mathtt{k}} \rangle \\
(\texttt{Argument}) & \langle \mathit{ret}\lceil \mathtt{v} \rceil : \mathit{arg}\lceil \mathtt{e}, \mathtt{r} \rceil : \mathtt{k} \triangleleft \overline{\mathtt{k}} \rangle & \longrightarrow & \langle \mathtt{e}, \mathtt{r}, \mathit{fun}\lceil \mathtt{v} \rceil : \mathtt{k}, \overline{\mathtt{k}} \rangle \\
(\texttt{Function}) & \langle \mathit{ret}\lceil \mathtt{v} \rceil : \mathit{fun}\lceil \lambda \mathtt{x}.\mathtt{e}, \mathtt{r} \rceil : \mathtt{k} \triangleleft \overline{\mathtt{k}} \rangle & \longrightarrow & \langle \mathtt{e}, \mathtt{r}[\mathtt{x} \mapsto \mathtt{v}], \mathtt{k}, \overline{\mathtt{k}} \rangle \\
(\texttt{PrepareValue}) & \langle \mathit{ret}\lceil \mathtt{k}' \rceil : \mathit{pval}\lceil \mathtt{e}, \mathtt{r} \rceil : \mathtt{k} \triangleleft \overline{\mathtt{k}} \rangle & \longrightarrow & \langle \mathtt{e}, \mathtt{r}, \mathit{prep}\lceil \mathtt{k}' \rceil : \mathtt{k}, \overline{\mathtt{k}} \rangle \\
(\texttt{PrepareFinalize}) & \langle \mathit{ret}\lceil \mathtt{v} \rceil : \mathit{prep}\lceil \mathtt{k}' \rceil : \mathtt{k} \triangleleft \overline{\mathtt{k}} \rangle & \longrightarrow & \langle \mathit{ret}\lceil \mathit{ret}\lceil \mathtt{v} \rceil : \mathtt{k}' \rceil : \mathtt{k} \triangleleft \overline{\mathtt{k}} \rangle \\
(\texttt{SendValue}) & \langle \mathit{ret}\lceil \mathtt{c} \rceil : \mathit{sval}\lceil \mathtt{e}, \mathtt{r} \rceil : \mathtt{k} \triangleleft \overline{\mathtt{k}} \rangle & \longrightarrow & \langle \mathtt{e}, \mathtt{r}, \mathit{send}\lceil \mathtt{c} \rceil : \mathtt{k}, \overline{\mathtt{k}} \rangle \\
(\texttt{SendBlock}) & \langle \mathit{ret}\lceil \mathtt{v} \rceil : \mathit{send}\lceil \mathtt{c} \rceil : \mathtt{k} \triangleleft \overline{\mathtt{k}} \rangle & \longrightarrow & \langle \mathit{sblk}\lceil \mathtt{c}, \mathtt{v} \rceil : \mathtt{k} \triangleleft \overline{\mathtt{k}} \rangle \\
(\texttt{RecvBlock}) & \langle \mathit{ret}\lceil \mathtt{c} \rceil : \mathit{recv}\lceil \rceil : \mathtt{k} \triangleleft \overline{\mathtt{k}} \rangle & \longrightarrow & \langle \mathit{rblk}\lceil \mathtt{c} \rceil : \mathtt{k} \triangleleft \overline{\mathtt{k}} \rangle
\end{array}
$$

Fig. 9. Local evaluation defining both control and return transitions.

by its last parasite. The lifetime of a thread is bounded by the parasites that inhabit it. The `PrepareFinalize` rule, prepares a reified parasite by pushing a return frame on the stack of the reified parasite. This prepared and reified parasite is then returned to the currently executing parasite.

3.7 Unifying the host and parasitic APIs

As mentioned before, host threads in MULTIMLTON are built on top of the `MLton.Thread` library, which is agnostic to scheduling policy, while the parasitic thread API implicitly encodes the thread management API with scheduling and migration services. With two underlying threading systems, we need to abstract away differences in thread creation and scheduling, so as to ease the development of higher-level libraries such as ACML.

With this in mind, we have implemented a scheduler that exposes both one-shot continuations and parasites, and unifies other scheduler idioms such as suspending the execution of current thread, preparing a suspended thread with a value to make it runnable, scheduling a runnable thread, etc. The core primitives in the unified scheduler interface are shown below.

```
signature UNIFIED_SCHEDULER =
sig

type 'a host     (* A host (one-shot) continuation *)
type ready_host  (* A runnable host *)
type 'a par      (* A parasitic thread *)
type ready_par   (* A runnable parasite *)

datatype 'a thrd = H_THRD of 'a host
                 | P_THRD of 'a par

(* parasite management *)
val spawnParasite   : (unit -> unit) -> unit
val reifyPar              : ('a par -> unit) -> 'a
val preparePar            : ('a par * 'a) -> ready_par
val attachPar             : ready_par -> unit
val inflatePar            : unit -> unit

(* host management *)
val spawnHost   : (unit -> unit) -> unit
val newHost                      : (unit -> unit) -> unit host
val prepareHost : 'a host * 'a -> ready_host

(* host + parasites *)
val attachParToHost  : 'a par * ready_host -> 'a host
val reifyParFromHost : 'a host -> 'a par * ready_host
val inflateParToHost : 'a par -> 'a host

(* switch *)
val switch: ('a host -> ready_host) * ('a par -> unit) -> 'a

(* scheduler queue operations *)
val enqueue : ready_host -> unit
val dequeue : unit -> ready_host option

end
```

The `UNIFIED_SCHEDULER` is built using the `MLton.Thread` library and the parasitic primitives presented in Section 3.5. The parasite management primitives are the same as the ones described in Section 3.5, with the names of the primitives suitably aliased so as to avoid conflict with host thread management primitives. The scheduler provides a way to instantiate parallel execution of a host thread using the `spawnHost` primitive. A suspended host thread (a one-shot host continuation) can be created using the `newHost` primitive. The primitive `prepareHost` takes a suspended host thread, along with a value and produces a *ready* host thread, which can then be scheduled. The readied host thread resumes execution with the prepared value (return value of the `switch` primitive).

The most interesting primitives are the ones that combine parasites with host threads. Primitive `attachParToHost` attaches the given parasite to a runnable host thread, and produces a new host continuation. When this host continuation is prepared with a value

and scheduled, the attached parasite is first evaluated, followed by the original host computation.

The parasite attached to a host continuation can be separated using the `reifyParFromHost` primitive. The behavior of the primitive is the dual of `attachParToHost`. The function `inflateParToHost` converts a parasite into a host continuation. Such a host continuation can be scheduled in parallel, after suitably preparing it with a value.

The `inflatePar` and `inflateParToHost` functions are implemented with the help of other library primitives as follows:

```
fun inflatePar () : unit =
  reifyPar (fn p : 'a par =>
    let
      val readyPar : ready_par = preparePar (p, ())
      val h : unit host = newHost (fn () => attachPar
        readyPar)
      val rh : ready_host = prepareHost (h,())
    in
      enqueue rh
    end)

fun inflateParToHost (p : 'a par) : 'a host =
  let
    val h : unit host = newHost (fn () => ())
    val rh : ready_host = prepareHost (h, ())
    val rh' : 'a host = attachParToHost (p, rh)
  in
    rh'
  end
```

The `switch` primitives enables the control to switch away from the currently running thread. Since the currently running thread can either be a host or a parasite, the target is either some host thread (if the currently running thread is a host) or the next parasite on the current stack (if the currently running thread is a parasite). To this end, the `switch` primitive takes two arguments: one to handle the case when the currently running computation is a host thread, and another to handle parasitic switching. If the thread being switched is a host, then the programmer is expected to return a runnable host thread, to which the control returns. If the thread being switched is a parasite, then we know that there is a host or more parasitic threads residing on the current stack. Hence, control simply switches to the next parasite or the host computation on this stack.

Finally, the scheduler queue operations enqueue and dequeue add and remove a ready host thread from the scheduler queue.

3.8 Implementing ACML primitives

In this section, we illustrate how ACML primitives can be encoded using parasitic threads. For clarity, we elide the definition of simple helper functions. However, when not obvious, we explain in prose how they are implemented. ACML exposes to the programmer a single `spawn` primitive. We implement this spawn primitive by instantiating a new host thread on

the scheduler. In particular, we do not expose the `spawnParasite` primitive to the ACML programmer. All parasitic threads are instantiated internally by the ACML library.

3.8.1 Implementing asynchronous send

The basic asynchronous primitive in ACML is an asynchronous send - `aSend (c,v)` that sends the value v on channel c. An asynchronous send has the ordering guarantee that the continuation of the asynchronous send is evaluated only after discharging the send. In other words, either the send completes (matches with a receiver) or deposits the value on the channel, before evaluating the continuation. Most importantly, there is equal opportunity for the asynchronous send to successfully match with a receive immediately, or wait until a value is picked up by a receiver thread. It would be unwise to create a full-fledged thread to capture the asynchronous behavior for a send that will immediately complete.

Locking and polling the channel to check whether the send can be completed immediately may work in the case of an isolated send. But such a solution is not directly applicable to synchronization over choices, where multiple channels may need to be polled atomically. It is this very fact that motivated Reppy *et al.* (2009) to build a completely new synchronization protocol for parallel CML.

Luckily, parasitic threads can be used to preserve desired ordering guarantees without compromising efficiency. We implement asynchronous send by evaluating a *synchronous* send primitive in a new parasitic thread. Since the `spawnParasite` primitive is guaranteed to evaluate before the continuation, we capture the desired ordering guarantee. In addition, if the asynchronous send immediately matches with a receive, then the cost of the whole operation is on the order of a non-tail function call.

The implementation of the synchronous send primitive is shown below:

```
val send : 'a chan * 'a -> unit
fun send (c, v) =
  case dequeuePendingReceivers c of
      SOME (H_THRD ht) =>
        enqueue (prepareHost (ht, v))
    | SOME (P_THRD pt) =>
        attachPar (preparePar (pt, v))
    | NONE =>
      let
        fun handleHost t =
          (enqueuePendingSenders (c, H_THRD t);
           dequeue ())
        fun handlePar t =
           enqueuePendingSenders (c, P_THRD t)
      in
        switch (handleHost, handlePar)
      end
```

Each ACML channel is represented by a pair of queues, one for the list of pending senders and another for the list of pending receivers. At any point in time, one of these queues is empty. In the above code snippet, the functions `dequeuePendingReceivers : 'a chan -> 'a thrd option` and `enqueuePendingSenders : 'a chan * 'a`

`thrd -> unit` are internal functions utilized in the channel implementation, and are used to manipulate the channel queues.

During a synchronous send on a channel `c`, if there are pending receiver threads on the channel, then the first one is dequeued. Notice that this receiver thread can either be a host or a parasitic thread, and each case must be handled separately. If the receiver is a host thread, then we prepare the receiver host thread with the value being sent, and add it to the scheduler. However, if the receiver is a parasite, we prepare the parasite with a value, and attach it to the current thread. In this case, the parasitic computation `pt` is first evaluated, followed by the current continuation.

If there are no pending receiver threads blocked on the channel, then the current thread has to be enqueued on the channel. We use the `switch` primitive to perform the necessary blocking. If the current thread is a host, we resume the next thread from the scheduler, after enqueueing the current thread to the channel. If the current thread is a parasite, then we simply enqueue the parasite to the channel. The next parasite on this stack, or the underlying host computation is scheduled next.

Now, the `aSend` primitive can simply be implemented as:

```
fun aSend (c,v) = spawnParasite (fn () => send (c,v))
```

The asynchronous receive primitive `aRecv` is implemented in a similar fashion, by spawning a parasitic thread, which performs the synchronous `recv`. Thus, parasites allow asynchronous computations to be encoded efficiently in a straight forward manner.

3.8.2 Implementing base asynchronous events

Recall that in ACML, every synchronization of an asynchronous event creates an implicit thread to evaluate a post-consumption action. Most of these post-consumption actions typically involve only a few communication actions, and do not perform any computation intensive tasks. Parasitic threads are an ideal vehicle for realizing these implicit threads.

MULTIMLTON represents asynchronous events simply as a tuple with two synchronous events or a guarded asynchronous event as follows:

```
datatype ('a,'b) AEvent =
    PAIR of ('a Event (* post-creation *) *
              'b Event (* post-consumption *))
  | AGUARD of unit -> ('a, 'b) AEvent
```

The first and second components of the pair represent post-creation and post-consumption actions. Since the post-creation action of an asynchronous send event primitive `aSendEvt` is empty, `aSendEvt` is represented by pairing together an always event `alwaysEvt`, and the *synchronous* send event `sendEvt` as follows:

```
val aSendEvt : 'a chan -> (unit, unit) AEvent
fun aSendEvt c = PAIR (alwaysEvt (), sendEvt c)
```

3.8.3 Implementing asynchronous event synchronization

An ACML event is synchronized through `aSync : ('a,'b) AEvent -> 'a` primitive, which is analogous to the `sync : 'a Event -> 'a` primitive for synchronizing on a synchronous event. The code for `aSync` is shown below:

```
val aSync : ('a,'b) AEvent -> 'a
fun aSync aevt
let
  fun force (PAIR p) = PAIR p
    | force (GUARD g) = force (g ())
  val PAIR (postCreation, postConsumption) = force aevt
  val _ = spawnParasite (fn _ => ignore (sync
    postConsumption))
in
  sync postCreation
end
```

We first force execution of the guard to get the pair consisting of post-creation and post-consumption synchronous events. ACML semantics dictates that post-consumption actions should only be evaluated after successful match of the base asynchronous event. In addition, a post-consumption action has to be run on an implicit thread. Our implementation achieves this by spawning a parasite that synchronizes on the synchronous event corresponding to a post-consumption action.

Since implicit thread computation is expected to be short-lived, parasitic threads alleviate the cost to create such threads. Most importantly, the fact that parasitic threads are evaluated immediately after spawning, and before the continuation of the `spawnParasite` primitive, ensures that the ordering guarantees of ACML event synchronization is preserved. Finally, the post-creation event is synchronized in the thread performing the synchronization. This captures the intended behavior of a post-creation action.

3.9 Runtime issues

While we have so far focused on the implementation and utility of parasitic threads, in this section, we focus on the interaction of parasitic threads with other components of the runtime system. In particular, we discuss issues related to long-running parasites, exception safety, and interaction with the garbage collector.

3.9.1 Long running parasites

Similar to asserting the blocking behavior of asynchronous computations, it is difficult to estimate the running time of an arbitrary asynchronous computation. If a long lived parasite continues to run on a host thread without ever being blocked, then we lose the opportunity for parallel execution of parasite and its host. In MULTIMLTON, long running host threads are preempted on a timer interrupt, so that other host threads have a chance to run. We extend this scheme to handle long running parasites.

By default, the implicit parasitic threads created during synchronization of an asynchronous event have preemption disabled. Preemption is enabled as a part of the `aWrap`

primitive, which is the only way to extend parasitic computation after the base asynchronous action. It is important to enable preemption only after the base asynchronous event in order to preserve the ordering property of the asynchronous events.

On a timer interrupt, if the current host thread has a parasite attached to it, on which preemption is enabled, we inflate this parasite into a new host thread, which is then enqueued to the scheduler. Control returns to the host or parasitic thread below the inflated parasite. On the other hand, if the interrupted thread was a host thread, this thread is enqueued, and control switches to the next available thread returned by the scheduler.

3.9.2 Exception safety in parasites

Exceptions in MLton are intertwined with the structure of the stack. In Concurrent ML, exceptions are stack local, and are not allowed to escape out of the host thread. Since parasitic operations copy frames across hosts stacks, there is a potential for parasitic exceptions to be raised in a context that is not aware of it. Therefore, it is important to ensure exception safety in the presence of parasites.

Exception handling in host threads. In MULTIMLTON, every host thread has a pointer to the handler frame on top of the current host stack. Every handler frame has a link to the next handler frame underneath it on the stack, represented as an offset. With this formulation, when exceptions are thrown, control can switch to the top handler in constant time, with subsequent handlers reached through the links. During program execution, whenever handler frames are popped, the thread's top-handler pointer is updated appropriately.

Exception handling in parasitic threads. How do we ensure that this structure is maintained for parasitic operations? Just like host threads, exceptions are not allowed to escape out of parasitic threads, and are handled by the default exception handler (Biagioni *et al.*, 1998), which typically logs an error to the standard output. As such, from an exception handling point of view, parasites are treated similar to host threads.

In MULTIMLTON, exceptions are implemented using exception frames, which are part of the thread's stack. When an exception is raised, the current stack is unwound till the first exception frame (*top-handler*), and control is transferred to the exception handler. The address of top-handler frame of the currently running stack is stored in a register for fast access. During a host thread switch, the offset of the top-handler from the bottom of the stack is saved to the suspending host thread's metadata, and restored during resumption.

Similar to host threads, every reified parasite stores the offset of the top handler frame from the bottom of the parasitic stack as part of its metadata. Since exception frames are a part of the parasite stack, all exception handlers installed by the parasite are also captured as a part of the reified parasitic stack. Recall that when a parasite is reified, control returns to the parasite residing below the reified parasite as if a non-tail call has returned. As a part of the return procedure, MULTIMLTON updates the top-handler address register to point to the top handler frame in the current host stack. When the reified parasite is attached to another host stack, the handler offset stored in the parasite metadata is used to set the attached thread's top-handler pointer to the top-most handler in the attached parasite. In this way, we ensure that parasites maintain expected exception semantics.

3.9.3 Interaction with the garbage collector

Just like host threads in the runtime system, the garbage collector (GC) must be aware of parasitic threads and the objects they reference. Parasites either exist as a part of a host stack or are reified as a separate heap object. If the parasites are a part of the host, the garbage collector treats the parasites as regular stack frames while tracing. Reified parasites are represented in the GC as stack objects, and the GC treats them as regular host stacks.

4 Garbage collection

In this section, we will consider the impacts of ACML and the MULTIMLTON threading system with both, lightweight and parasitic threads, on GC. MULTIMLTON's primary goal is scalability and multi-core performance. From a GC perspective, this obligates our primary design to focus on optimizing for throughput instead of reducing pause times and the memory footprint.

We will first describe the baseline MLton collector. We will then introduce a split-heap memory manager that allows mutators and collectors running on different cores to operate mostly independently. We then refine this collector based on the properties of ACML programs and functional programs in general with throughput in mind. Specifically we will focus on the premise that there is an abundance of available concurrency in ACML programs, introduced through *both* lightweight and parasitic threads, to realize a new collector that eliminates the need for read barriers and forwarding pointers. This naturally leads to a GC design that focuses on *procrastination* (Sivaramakrishnan *et al.*, 2012), delaying writes that would necessitate establishing forwarding pointers until a GC, where there is no longer a need for such pointers. The GC leverages the mostly functional nature of ACML programs and a new object property called *cleanliness*, which enables a broad class of objects to be moved from a local to a shared heap without requiring a full traversal of the local heap to fix existing references; cleanliness enables an important optimization that achieves the effect of procrastination without actually having to initiate a thread stall.

4.1 MLton's GC (Stop-the-world)

The base MULTIMLTON GC design uses a single, contiguous heap, shared among all cores. In order to allow local allocation, each core requests a page-sized chunk from the heap. While a single lock protects the chunk allocation, objects are allocated within chunks by bumping a core-local heap frontier.

In order to perform garbage collection, all the cores synchronize on a barrier, with one core responsible for collecting the entire heap. The garbage collection algorithm is inspired from Sansom's (1991) collector, which combines Cheney's two-space copying collector and Jonker's single-space sliding compaction collector. Cheney's copying collector walks the live objects in the heap just once per collection, while Jonker's mark-compact collector performs two walks. But Cheney's collector can only utilize half of memory allocated for the heap. Sansom's collector combines the best of both worlds. Copying collection is performed when heap requirements are less than half of the available memory. The runtime system dynamically switches to mark-compact collection if the heap utilization increases beyond half of the available space.

Since ML programs tend to have a high rate of allocation, and most objects are short-lived temporaries, it is beneficial to perform generational collection. The garbage collector supports Appel-style generational collection (Appel, 1989) for collecting temporaries. The generational collector has two generations, and all objects that survive a generational collection are copied to the older generation. Generational collection can work with both copying and mark-compact major collection schemes.

MULTIMLTON enables its generational collector only when it is profitable, which is determined by the following heuristic. At the end of a major collection, the runtime system calculates the ratio of live bytes to the total heap size. If this ratio falls below a certain (tunable) threshold, then generational collection is enabled for subsequent collections. By default, this ratio is 0.25.

4.2 Design rationale for a procrastinating GC

Splitting a program heap among a set of cores is a useful technique to exploit available parallelism on scalable multicore platforms: each core can allocate, collect, and access data locally, moving objects to a global, shared heap only when they are accessed by threads executing on different cores. This design allows local heaps to be collected independently, with coordination required only for global heap collection. In contrast, stop-the-world collectors need a global synchronization for every collection. As such, we begin our design exploration with a split-heap GC.

In order to ensure that cores cannot directly or indirectly access objects on other local heaps, which would complicate the ability to perform independent local heap collection, the following invariants need to be preserved:

- No pointers are allowed from one core's local heap to another.
- No pointers are permitted from the shared heap to the local heap.

Both invariants are necessary to perform independent local collections. The reason for the first is obvious. The second invariant prohibits a local heap from transitively accessing another local heap object via the shared heap. In order to preserve these invariants, the mutator typically executes a *write barrier* on every store operation. The write barrier ensures that before assigning a local object reference (source) to a shared heap object (target), the local object along with its transitive object closure is lifted to the shared heap. We call such writes *exporting writes* as they export information out of local heaps. The execution of the write barrier creates *forwarding pointers* in the original location of the lifted objects in the local heap. These point to the new locations of the lifted objects in the shared heap. Since objects can be lifted to the shared heap on potentially any write, the mutator needs to execute a *read barrier* on potentially every read. The read barrier checks whether the object being read is the actual object or a forwarding pointer, and in the latter case, indirects to the object found on the shared heap. Forwarding pointers are eventually eliminated during local collection.

Because the number of reads are likely to far outweigh the number of writes, the aggregate cost of read barriers can be both substantial and vary dramatically based on underlying architecture characteristics (Blackburn & Hosking, 2004). Eliminating read barriers, however, is non-trivial. Abstractly, one can avoid read barriers by eagerly *fixing* all references

that point to forwarded objects at the time the object is lifted to the shared heap, ensuring the mutator will never encounter a forwarded object. Unfortunately, this requires being able to enumerate all the references that point to the lifted object; in general, gathering this information is very expensive as the references to an object might originate from any object in the local heap.

We consider an alternative design that completely eliminates the need for read barriers *without* requiring a full scan of the local heap whenever an object is lifted to the shared heap. The design is based on the observation that read barriers can be clearly eliminated if forwarding pointers are never introduced. One way to avoid introducing forwarding pointers is to *delay* operations that create them until a local garbage collection is triggered. In other words, rather than executing a store operation that would trigger lifting a thread local object to the shared heap, we can simply *procrastinate*, thereby stalling the thread that needs to perform the store. The garbage collector must simply be informed of the need to lift the object's closure during its next local collection. After collection is complete, the store can take place with the source object lifted, and all extant heap references properly adjusted. As long as there is sufficient concurrency to utilize existing computational resources, in the form of available runnable threads to run other computations, the cost of procrastination is just proportional to the cost of a context switch.

Moreover, it is not necessary to always stall an operation that involves lifting an object to the shared heap. We consider a new property for objects (and their transitive object closures) called *cleanliness*. A clean object is one that can be safely lifted to the shared heap without introducing forwarding pointers that might be subsequently encountered by the mutator: objects that are immutable, objects only referenced from the stack, or objects whose set of incoming heap references is known, are obvious examples. The runtime analysis for cleanliness is combined with a specialized write barrier to amortize its cost. Thus, procrastination provides a general technique to eliminate read barriers, while cleanliness serves as an important optimization that avoids stalling threads unnecessarily.

The effectiveness of our approach depends on a programming model in which (a) most objects are clean, (b) the transitive closure of the object being lifted rarely has pointers to it from other heap allocated objects, and (c) there is a sufficient degree of concurrency in the form of runnable threads; this avoids idling available cores whenever a thread is stalled performing an exporting write that involves an unclean object. We observe that conditions (a) and (b) are common to functional programming languages and condition (c) follows from the ACML runtime model. Our technique does not rely on programmer annotations, static analysis or compiler optimizations to eliminate read barriers, and can be completely implemented as a lightweight runtime technique.

4.3 Local collector (Split-heap)

As mentioned earlier, the *local collector*[1] operates over a single shared (global) heap and a local heap for each core. The allocations in the shared heap is performed similar to allocations in the stop-the-world collector, where each core allocates a page-sized chunk in the

[1] Other terms have been used in the literature to indicate similar heap designs, notably *private nursery*, *local heap collector*, *thread-local* or *thread-specific heap*, and *on-the-fly collection*.

shared heap and performs object allocation by bumping its core-local shared heap frontier. Allocations in the local heaps do not require any synchronization. Garbage collection in the local heaps is similar to the baseline collector, except that it crucially does not require global synchronization.

Objects are allocated in the shared heap only if they are to be shared between two or more cores. Objects are allocated in the shared heap because of exporting writes and remote spawns (Section 4.6.3). Apart from these, all globals are allocated in the shared heap, since globals are visible to all cores by definition. For a shared heap collection, all of the cores synchronize on a barrier and then a single core collects the heap. Along with globals, all the live references from local heaps to the shared heap are considered to be roots for a shared heap collection. In order to eliminate roots from dead local heap objects, before a shared heap collection, local collections are performed on each core to eliminate such references.

The shared heap is also collected using Sansom's dual-mode garbage collector. However, we do not perform generational collection on the shared heap. This is because shared heap collection is expected to be relatively infrequent when compared to the frequency of local heap collections, and objects that are shared between cores, in general, live longer than a typical object collected during a generational collection.

4.4 Remembered stacks

In MULTIMLTON threads can synchronously or asynchronously communicate with each other over first-class message-passing communication channels. If a receiver is not available, a sender thread, or in the case of asynchronous communication the implicitly created thread, can block on a channel. If the channel resides in the shared heap, the thread object, its associated stack and the transitive closure of all objects reachable from it on the heap would be lifted to the shared heap as part of the blocking action. Since channel communication is the primary mode of thread interaction in our system, we would quickly find that most local heap objects end up being lifted to the shared heap. This would be highly undesirable.

Hence, we choose never to move stacks to the shared heap. We add an exception to our heap invariants to allow thread $\rightarrow$ stack pointers, where the thread resides on the shared heap, and references a stack object found on the local heap. Whenever a thread object is lifted to the shared heap, a reference to the corresponding stack object is added to the set of remembered stacks. This remembered set is considered as a root for a local collection to enable tracing of remembered stacks.

Before a shared heap collection, the remembered set is cleared; only those stacks that are reachable from other GC roots survive the shared heap collection. After a shared heap collection, the remembered set of each core is recalculated such that it contains only those stacks, whose corresponding thread objects reside in the shared heap, and have survived the shared heap collection.

Remembered stacks prevent thread local objects from being lifted to the shared heap, but require breaking the heap invariant to allow a thread object in the shared heap to refer to a stack object on the local heap. This relaxation of heap invariant is safe. The only object that can refer to thread-local stacks is the corresponding thread object. The thread objects are completely managed by the scheduler, and are not exposed to the programmer. As a

result, while the local heap objects can point to a shared-heap thread object, whose stack might be located on a different local heap, the only core that can modify such a stack (by running the thread) is the core that owns the heap in which the stack is located. Thus, there is no possibility of direct references between local heaps. Hence, the remembered stack strategy is safe with respect to garbage collection.

4.5 Cleanliness analysis

Although ACML provides an abundance of concurrency, with the procrastination mechanism, many of the threads in a program may end up blocked on exporting writes, waiting for a local garbage collection to unblock them. If all of the threads on a particular core have procrastinated, then a local garbage collection is needed in order to make progress. Such *forced* local garbage collections make the program run longer, and hence subdue the benefit of eliminating read barriers. Hence, it is desirable to avoid procrastination whenever possible.

In this section, we describe our cleanliness analysis, which identifies objects on which exporting writes do not need to be stalled. We first present auxiliary definitions that will be utilized by cleanliness checks, and then describe the analysis.

4.5.1 Heap session

Objects are allocated in the local heap by bumping the local heap frontier. In addition, associated with each local heap is a pointer called `sessionStart` that always points to a location between the start of the heap and the frontier. This pointer introduces the idea of a *heap session*, to capture the notion of recently allocated objects. Every local heap has exactly two sessions: a *current session* between the `sessionStart` and the heap frontier and a *previous session* between the start of the heap and `sessionStart`. Heap sessions are used by the cleanliness analysis to limit the range of heap locations that need to be scanned to test an object closure[2] for cleanliness. Assigning the current local heap frontier to the `sessionStart` pointer starts a new session. We start a new session on a context switch, a local garbage collection and after an object has been lifted to the shared heap.

4.5.2 Reference count

We introduce a limited reference counting mechanism for local heap objects that counts the number of references from other local heap objects. Importantly, we do not consider references from ML thread stacks. The reference count is meaningful only for objects reachable in the current session. For such objects, the number of references to an object can be one of four values: `ZERO`, `ONE`, `LOCAL_MANY`, and `GLOBAL`. We steal 2 bits from the object header to record this information. A reference count of `ZERO` indicates that the object only has references from registers or stacks, while an object with a count of `ONE`

[2] In the following, we write *object closure* to mean the set of objects reachable from some root on the heap; to avoid confusion, we write *function closure* to mean the representation of an SML function as a pair of function code pointer and static environment.

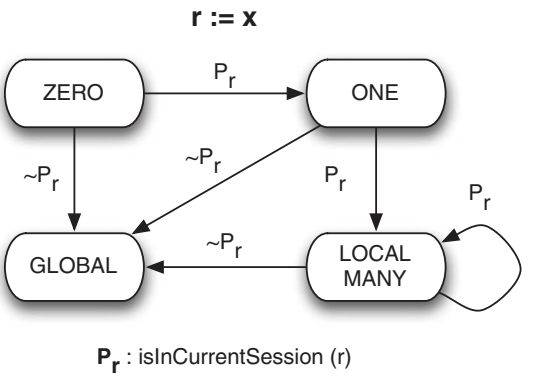


Fig. 10. State transition diagram detailing the behavior of the reference counting mechanism with respect to object `x` involved in an assignment, `r := x`, where P_r = `isInCurrentSession(r)`.

has exactly one pointer from the current session. A count of `LOCAL_MANY` indicates that this object has more than one reference, but that all of these references originate from the current session. `GLOBAL` indicates that the object has at least one reference that originates from outside the current session.

The reference counting mechanism is implemented as a part of the write barrier (Lines 13–22 in Figure 13). Figure 10 illustrates the state transition diagram for the reference counting mechanism. Observe that reference counts are non-decreasing. Hence, the reference count of any object represents the maximum number of references that pointed to the object at any point in its lifetime.

4.5.3 Cleanliness

An object closure is said to be clean, if for each object reachable from the root of the object closure,

- the object is immutable or in the shared heap. Or,
- the object is the root, and has `ZERO` references. Or,
- the object is not the root, and has `ONE` reference. Or,
- the object is not the root, has `LOCAL_MANY` references, and is in the current session.

Otherwise, the object closure is not clean.

Figure 11 shows an implementation of an object closure cleanliness check. Since the cleanliness check, memory barriers, and the garbage collector are implemented in low-level code (C, assembly and low-level intermediate language in the compiler), this code snippet, and others that follow in this section are in pseudo-C language, to better represent their implementation. If the source of an exporting assignment is immutable, we can make a copy of the immutable object in the shared heap, and avoid introducing references to forwarded objects. Standard ML does not allow the programmer to test the referential equality of immutable objects. Equality of immutable objects is always computed by structure. Hence, it is safe to replicate immutable objects. If the object is already in the shared heap, there is no need to move this object.

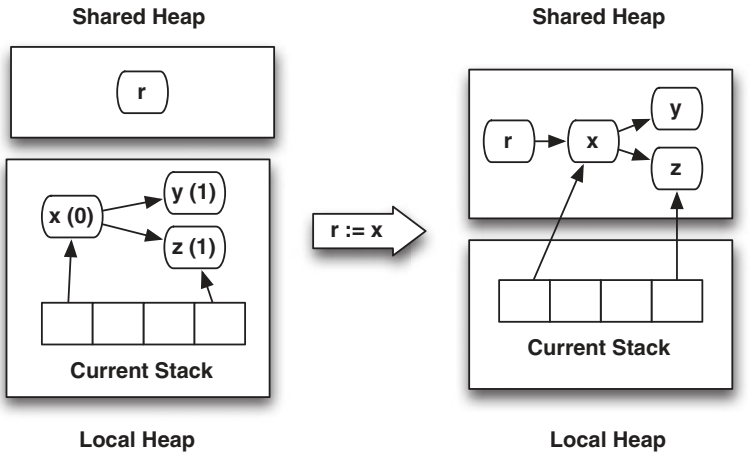
```
1  bool isClean (pointer p, bool* isLocalMany) {
2    clean = true;
3    foreach o in reachable(p) {
4      if (!isMutable(o) || isInSharedHeap(o))
5        continue;
6      nv = getRefCount(o);
7      if (nv == ZERO)
8        clean &&= true;
9      else if (nv == ONE)
10       clean &&= (o != p);
11     else if (nv == LOCAL_MANY) {
12       clean &&= (isInCurrentSession(o));
13       *isLocalMany = true;
14     }
15     else
16       clean = false;
17   }
18   return clean;
19 }
```

Fig. 11. (Colour online) Cleanliness check.

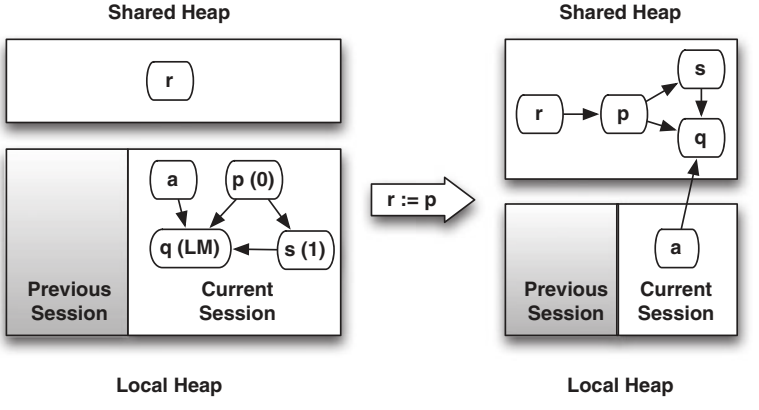
If the object closure of the source of a exporting write is clean, we can move the object closure to the shared heap and quickly fix all of the forwarding pointers that might be generated. For example, consider an object that defines a tree structure; such an object is clean if the root has `ZERO` references and all of its internal nodes have `ONE` reference from their parent. A root having `ZERO` references means it is accessed only via the stack; if it had a count of `ONE`, the outstanding reference may emanate from the heap. Internal nodes having a reference count of `ONE` implies they are reachable only via other nodes in the object being traced. Figure 12(a) shows such an object closure. In this example, we assume that all objects in the object closure are mutable. The reference count of relevant nodes is given in the brackets. Both the root and internal nodes can have pointers from the current stack not tracked by the reference count. After lifting the object closure, the references originating from the current stack are fixed by walking the stack.

Object closures need not just be trees and can be arbitrary graphs, with multiple incoming edges to a particular object in the object closure. How do we determine if the incoming edges to an object originate from the object closure or from outside the object closure (from the local heap)? We cannot answer this question without walking the local heap. Hence, we simplify the question to asking whether all the pointers to an object originate from the current session. This question is answered in the affirmative if an object has a reference count of `LOCAL_MANY` (lines 11–13 in Figure 11).

Figure 12(b) shows an example of a object closure whose objects have at most `LOCAL_MANY` references. Again, we assume that all objects in the object closure are mutable. In the transitive object closure rooted at `p`, object `q` has locally many references. These references might originate from the object closure itself (edges `p` $\rightarrow$ `q` and `s` $\rightarrow$ `q`) or from outside the object closure (edge `a` $\rightarrow$ `q`). After lifting such object closures to the shared heap, only the current session is walked to fix all of the references to forwarded



(a) Tree-structured object closure



(b) Session-based cleanliness

Fig. 12. Utilizing object closure cleanliness information for exporting writes to avoid references to forwarded objects.

objects created during the copy. In practice (Section 5.13), current session sizes are much smaller than heap sizes, and hence exporting writes can be performed quickly.

Finally, in the case of `LOCAL_MANY` references, the object closure is clean, but unlike other cases, after lifting the object closure to the shared heap, the current session must be walked to fix any references to forwarded objects. This is indicated to the caller of `isClean` function by assigning `true` to `*isLocalMany`, and is used in the implementation of lifting an object closure to the shared heap (Figure 14).

4.6 Write barrier

In this section, we present the modifications to the write barrier to eliminate the possibility of creating references from reachable objects in the local heap to a forwarded object. The

```
1  Val writeBarrier (Ref r, Val v) {
2    if (isObjptr(v)) {
3      //Lift if clean or procrastinate
4      if (isInSharedHeap(r) &&
5          isInLocalHeap(v)) {
6        isLocalMany = false;
7        if (isClean(v, &isLocalMany))
8          v = lift(v, isLocalMany);
9        else
10         v = suspendTillGCAndLift(v);
11     }
12     //Tracking cleanliness
13     if (isInLocalHeap (r) &&
14         isInLocalHeap(v)) {
15       n = getRefCount(v);
16       if (!isInCurrentSession (r))
17         setNumRefs(v, GLOBAL);
18       else if (n == ZERO)
19         setNumRefs(v, ONE);
20       else if (n < GLOBAL)
21         setNumRefs(v, LOCAL_MANY);
22     }
23   }
24   return v;
25 }
```

Fig. 13. (Colour online) Write barrier implementation.

implementation of our write barrier is presented in Figure 13. A write barrier is invoked prior to a write and returns a new value for the source of the write. The check `isObjptr` at line 2 returns true only for heap allocated objects, and is a compile time check. Hence, for primitive valued writes, there is no write barrier. Lines 4 and 5 check whether the write is exporting. If the source of the object is clean, we lift the transitive object closure to the shared heap and return the new location of the object in the shared heap.

4.6.1 Delaying writes

If the source of an exporting write is not clean, we suspend the current thread and switch to another thread in our scheduler. The source of the write is added to a queue of objects that are waiting to be lifted. Since the write is not performed, no forwarded pointers are created. If programs have ample amounts of concurrency, there will be other threads that are waiting to be run. However, if all threads on a given core are blocked on a write, we move all of the object closures that are waiting to be lifted to the shared heap. We then force a local garbage collection, which will, as a part of the collection, fix all of the references to point to the new (lifted) location on the shared heap. Thus, the mutator never encounters a reference to a forwarded object.

Parasitic threads and delayed writes. Recall that MULTIMLTON has two different threading systems – host and parasites. If the exporting write were performed in a parasitic thread, should the parasite be reified and the host thread allowed to continue? The answer

```
Set imSet;
void liftHelper (pointer* op,
                 pointer* frontierP) {
  frontier = *frontierP;
  o = *op;
  if (isInSharedHeap(o)) return;
  copyObject (o, frontier);
  *op = frontier + headerSize(o);
  *frontierP = frontier + objectSize(o);
  if (isMutable(o)) {
    setHeader(o, FORWARDED);
    *o = *op;
  }
  else
    imSet += o;
}

pointer lift (pointer op, bool isLocalMany) {
  start = frontier = getSharedHeapFrontier();
  imSet = {};
  //Lift transitive object closure
  liftHelper (&op, &frontier);
  foreachObjptrInRange (start, &frontier, liftHelper);
  setSharedHeapFrontier(frontier);
  //Fix forwarding pointers
  foreachObjptrInObject (getCurrentStack(), fixFwdPtr);
  foreach o in imSet
    foreachObjptrInObject(o, fixFwdPtr);
  frontier = getLocalHeapFrontier();
  if (isLocalMany)
    foreachObjptrInRange (getSessionStart(), &frontier,
                          fixFwdPtr);
  setSessionStart(frontier);
  return op;
}
```

Fig. 14. (Colour online) Lifting an object closure to the shared heap.

turns out to be no. Recall that parasitic threads are scheduled cooperatively i.e., they run until reified. In particular, ACML uses this property to preserve the order of asynchronous actions performed on a channel. For example, if an `aSend` operations, which is simply a synchronous send operation wrapped in a parasite, were to be reified before the `aSend` was able to deposit a value on the channel, the subsequent `aSend` performed by the host thread might be enqueued on the channel before the previous `aSend`. This violates the ordering property of `aSend`. Hence, during an exporting write, the host thread and any parasitic thread attached to it are suspended till the next local garbage collection.

4.6.2 Lifting objects to the shared heap

Figure 14 shows the pseudo-C code for lifting object closures to the shared heap. The function `lift` takes as input the root of a clean object closure and a Boolean representing whether the object closure has any object that has `LOCAL_MANY` references. For simplicity

```
1  ThreadID spawn (pointer closure, int target) {
2    ThreadID tid = newThreadID();
3    Thread t = newThread(closure, tid);
4    isLocalMany = false;
5    if (isClean(t, &isLocalMany)) {
6      t = lift(t, isLocalMany);
7      enqueThread(t, target);
8    }
9    else
10     liftAndReadyBeforeGC(t, target);
11   return tid;
12 }
```

Fig. 15. (Colour online) Spawning a thread.

of presentation, we assume that the shared heap has enough space reserved for the transitive object closure of the object being lifted. In practice, the lifting process requests additional shared heap chunks to be reserved for the current processor, or triggers a shared heap collection if there is no additional space in the shared heap.

Objects are transitively lifted to the shared heap, starting from the root, in the obvious way (Lines 22–23). As a part of lifting, mutable objects are lifted and a forwarding pointer is created in their original location, while immutable objects are copied and their location added to `imSet` (Lines 10–15). After lifting the transitive object closure to the shared heap, the shared heap frontier is updated to the new location.

After object lifting, the current stack is walked to fix any references to forwarding pointers (Line 27–28). Since we do not track references from the stack for reference counting, there might be references to forwarded objects from stacks other than the current stack. We fix such references lazily. Before a context switch, the target stack is walked to fix any references to forwarded objects. Since immutable objects are copied and mutable objects lifted, a copied immutable object might point to a forwarded object. We walk all the shared heap copies of immutable objects lifted from the local heap to fix any references to forwarded objects (Lines 27–28).

Recall that if the object closure was clean, but has `LOCAL_MANY` references, then it has at least one pointer from the current session. Hence, in this case, we walk the current session to fix the references to any forwarded objects to point to their shared heap counterparts (lines 30–32). Finally, session start is moved to the current frontier.

4.6.3 Remote spawns

Apart from exporting writes, function closures can also escape local heaps when threads are spawned on other cores. For spawning on other cores, the environment of the function closure is lifted to the shared heap and then, the function closure is added to the target core's scheduler. This might introduce references to forwarding pointers in the spawning core's heap. We utilize the techniques developed for exporting writes to handle remote spawns in a similar fashion.

Figure 15 shows the implementation of thread spawn. If the function closure is clean, we lift the function closure to the shared heap, and enqueue the thread on the target scheduler.

Otherwise, we add it to the list of threads that need to be lifted to the shared heap. Before the next garbage collection, these function closures are lifted to the shared heap, enqueued to target schedulers, and the references to forwarded objects are fixed as a part of the collection. When the target scheduler finds this new thread (as opposed to other preempted threads), it allocates a new stack in the local heap. Hence, except for the environment of the remotely spawned thread, all data allocated by the thread is placed in the local heap.

4.6.4 Barrier implementation

In our local collector, the code for tracking cleanliness (Lines 13–24 in Figure 13) is implemented as an RSSA pass, one of the backend intermediate passes in our compiler. RSSA is similar to Static Single Assignment (SSA), but exposes data representations decisions. In RSSA, we are able to distinguish heap allocated objects from non-heap values such as constants, values on the stack and registers, globals, etc. This allows us to generate barriers only when necessary.

The code for avoiding creation of references to forwarded objects (Lines 4–11 in Figure 13) is implemented in the primitive library, where we have access to the lightweight thread scheduler. `suspendTillGCAndLift` (line 11 in Figure 13) is carefully implemented to not contain an exporting write, which would cause non-terminating recursive calls to the write barrier.

5 Experimental results

5.1 Target platforms

We quantify the benefits of parasitic threads and our runtime design by considering a set of benchmarks executing on a 48 core AMD Opteron 6176 server (AMD). In addition to the AMD machine, garbage collection results were obtained on a 48 core Intel Single-chip Cloud Computer (SCC), and an 864 core Azul's Vega 3 (AZUL) machine. Our choice of architectures is primarily to study the robustness of our techniques across a diverse set of platforms rather than exploiting specific architectural characteristics in our design.[3]

The AMD Opteron machine used in our experiments has 48 cores that share 256 GB of main memory. The cores are arranged into 4 NUMA regions with 12 cores each with 64 GB of local memory per NUMA region. Access to non-local memory is mediated by a hyper-transport layer that coordinates memory requests between processors.

The Azul machine used in our experiments has 16 Vega 3 processors, each with 54 cores per chip; each core exhibits roughly 1/3 the performance of an Intel Core2-Duo. Out of the 864 cores, 846 are application usable while the rest of the cores are reserved for the kernel. The machine has 384 GB of cache coherent memory split across 192 memory modules. Uniform memory access is provided through a passive, non-blocking interconnect mesh. The machine has 205 GB/s aggregate memory bandwidth and 544 GB/s

[3] The source code and benchmarks used in our evaluation are available at `https://github.com/kayceesrk/multiMLton`. Each branch in the repository represents a variant of MultiMLton on a particular architecture.

aggregate interconnect bandwidth. Each core has a 16KB, 4-way L1 data and instruction caches.

Intel's Single-chip Cloud Computer (SCC) (Intel, 2012) is an experimental platform from Intel labs with 48 P54C Pentium cores. The most interesting aspect of SCC is the complete lack of cache coherence and a focus on inter-core interactions through a high speed mesh interconnect. The cores are grouped into 24 tiles, connected via a fast on-die mesh network. The tiles are split into 4 quadrants with each quadrant connected to a memory module. Each core has 16KB L1 data and instruction caches and 256KB L2 cache. Each core also has a small message passing buffer (MPB) of 8KB used for message passing between the cores.

Since the SCC does not provide cache coherence, coherence must be implemented in software if required. From the programmer's perspective, each core has a private memory that is cached and not visible to other cores. The cores also have access to a shared memory, which is by default not cached to avoid coherence issues. The cost of accessing data from the cached local memory is substantially less when compared to accessing shared memory. It takes 18 core cycles to read from the L2 cache; on the other hand, it takes 40 core cycles to request data from the memory controller, 4 mesh cycles for the mesh to forward the request and 46 memory cycles for the memory controller to complete the operation. Hence, in total, the delay between a core requesting data from the memory controller is $40\,k_{core} + 4*2*n\,k_{mesh} + 46\,k_{ram}$ cycles, where k_{core}, k_{mesh} *and* k_{ram} are the cycles of core, mesh network and memory respectively. In our experimental setup, where 6 tiles share a memory controller, the number of hops n to the memory controller could be $0 < n < 5$. Hence, shared heap accesses are much more expensive than local heap accesses.

MULTIMLTON uses 64-bit pointer arithmetic, and was extended from the base MLton version 20100608. By default, the MULTIMLTON backend uses C codegen, and the generated C files were compiled with GCC 4.8.2.

5.2 Establishing a performance baseline

Before we present an evaluation of the impact of parasitic threads and new garbage collector design on overall performance, we first establish a baseline for MULTIMLTON through comparisons against well-known multicore language runtimes on well-studied programs. For this purpose, we chose the `k-nucleotide`, `spectral-norm`, and `mandelbrot` benchmarks from the Computer Language Benchmarks Game[4]. These particular benchmarks were chosen since they were particularly amenable to parallel execution. We caution that the chosen programs are small, and neither particularly highly concurrent nor memory intensive. Our goal here is to provide anecdotal evidence that MULTIMLTON is competitive in terms of performance with other mature optimized systems, and thus the evaluation study presented later in this section are not simply reporting results from a straw man implementation. The benchmarks themselves were chosen primarily because they have implementations in several languages, and performance numbers were readily available.

We compare the performance of these benchmarks against implementations in C + POSIX threads, Java, Haskell, and MLton. The C programs were compiled with GCC

[4] http://benchmarksgame.alioth.debian.org/

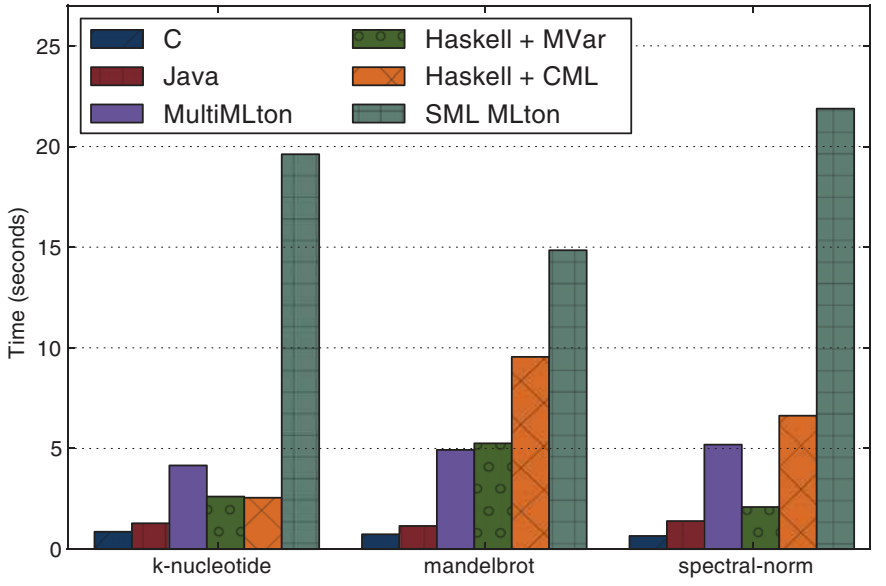


Fig. 16. (Colour online) Comparison of running times of MULTIMLTON programs against other language implementations on benchmarks taken from the Computer Language Benchmarks.

4.8.2. Java version was 1.6.0_34 with the programs running on 64-bit HotSpot server whose version was 20.9. Haskell programs were compiled with the Glasgow Haskell Compiler (GHC) 7.6.3. For each benchmark, we choose two versions of the Haskell programs for comparison; one that uses MVars, and another built using a Concurrent ML implementation written in Concurrent Haskell (Chaudhuri, 2009)[5].

MLton programs were compiled with version 20100608 of the compiler with the native backend. Though the MLton programs are not parallel, they have been included in the comparison to illustrate the performance overhead of MULTIMLTON's parallel runtime over MLton's sequential runtime, which has been extended to realize MULTIMLTON . The version of MULTIMLTON used for the comparison utilized the stop-the-world garbage collector 4.1 with support for parallel allocation. While the MLton programs use the native backend, the MULTIMLTON programs use the C backend; currently, MULTIMLTON does not compile directly to native code.

Except for the MLton and MULTIMLTON programs that were implemented by us, the other programs were the fastest versions in their respective languages in the benchmarks suite at the time this article was written. All the programs were compiled with their optimal flags indicated in the corresponding benchmarks. The experiments were performed on the AMD machine on 8 cores, and the results are shown in Figure 16. Haskell CML programs are slower than the corresponding Haskell MVar programs since the Haskell CML programs create implicit threads for each communication action to accommodate CML style composable events. The results indicate that the performance of MULTIMLTON is comparable to Haskell programs on GHC, which is the state-of-the-art parallel functional language compiler and runtime.

[5] Implementation available at http://hackage.haskell.org/package/cml-0.1.3

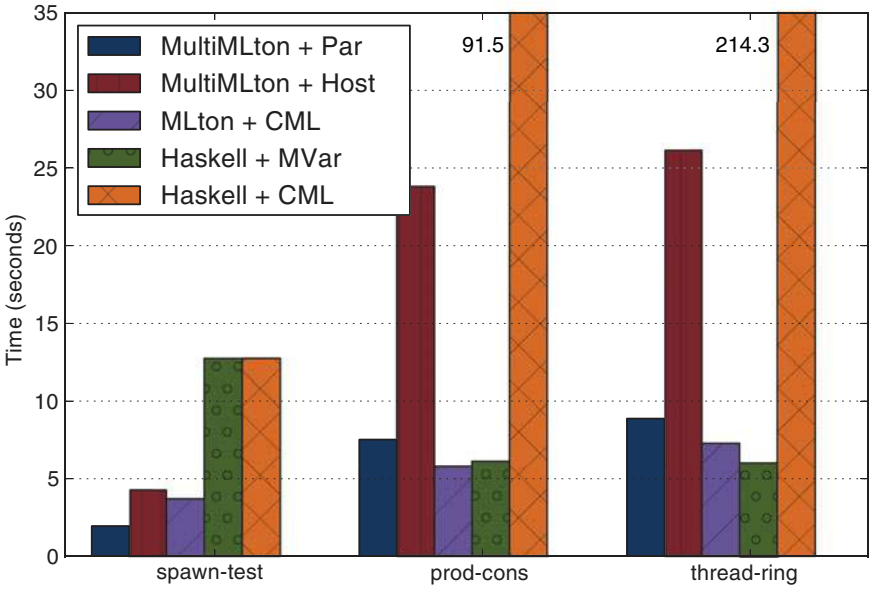


Fig. 17. (Colour online) Performance results of various threading systems.

In addition, we compared the performance of the threading and communication infrastructure using following microbenchmarks:

- **spawn-test**: Spawning 10 million threads.
- **prod-cons**: A single producer, single consumer benchmark where the producer sends 10 million integer tokens to the consumer.
- **thread-ring**: Thread-ring benchmark from Computer Language Benchmarks Game, where 503 threads are arranged in a ring, and an integer token is passed through the ring for 10 million hops.

The threading microbenchmarks are designed to gauge the benefit of parasitic threads in a highly concurrent setting. The results are presented in Figure 17. The results show that parasitic threads are significantly faster than a host threads-based implementation. We also see that MLton CML programs tend to be faster than the MultiMLton host thread programs. This is because MLton does not support parallel execution, and hence does not have the synchronization overheads associated with the scheduler and channel data structures. Haskell CML programs are significantly slower than the corresponding MVar based Haskell programs due to the cost of implicit thread creation in the CML implementation of Haskell.

5.3 Local collector with read barriers

In order to validate the effectiveness of our new GC design that eliminates read barriers (RB-), we have implemented a baseline local collector that utilizes read barriers (RB+). Unlike the RB- collector, the read barriers in the RB+ collector ensure that the necessary indirection is performed when accessing objects that have been lifted to the shared heap.

```
1 pointer readBarrier (pointer p) {
2   if (!isPointer(p)) return p;
3   if (getHeader(p) == FORWARDED)
4     return *(pointer*)p;
5   return p;
6 }
```

Fig. 18. (Colour online) Read barrier.

Hence, the RB+ collector allows us to precisely quantify the advantages of eliminating the read barriers, which is the motivation for the RB- collector.

5.3.1 Read barrier

Figure 18 shows the pseudo-C code for our read barrier. Whenever an object is lifted to the shared heap, the original object's header is set to `FORWARDED`, and the first word of the object is overwritten with the new location of the object in the shared heap. Before an object is read, the mutator checks whether the object has been forwarded, and if it is, returns the new location of the object. Hence, our read barriers are conditional (Blackburn & Hosking, 2004; Baker, 1978).

MLton represents non-value carrying constructors of (sum) datatypes using non-pointer values. If such a type additionally happens to have value-carrying constructors that reference heap-allocated objects, the non-pointer value representing the empty constructor will be stored in the object pointer field. Hence, the read barrier must first check whether the presumed pointer does in fact point to a heap object. Otherwise, the original value is returned (line 2). If the given pointer points to a forwarded object, the current location of the object in the shared heap is returned. Otherwise, the original value is returned.

5.3.2 Optimizations

MultiMLton performs a series of optimizations to minimize heap allocation, thus reducing the set of read barriers actually generated. For example, references and arrays that do not escape out of a function are flattened. Combined with aggressive inlining and simplification optimizations enabled by whole-program compilation, object allocation on the heap can be substantially reduced.

The compiler and runtime system ensure that entries on thread stacks never point to a forwarded object. Whenever an object pointer is stored into a register or the stack, a read barrier is executed on the object pointer to get the current location of the object. Immediately after an exporting write or a context switch, the current stack is walked and references to forwarded objects are updated to point to the new location of lifted objects in the shared heap.

Additionally, before performing an exporting write, register values are saved on the stack, and reloaded after exit. Thus, as a part of fixing references to forwarding pointers from the stack, references from registers are also fixed. This ensures that the registers never point to forwarded objects either. Hence, no read barriers are required for dereferencing object pointers from the stack or registers. This optimization is analogous to "eager" read

Table 1. *Comparison of cost of aSendEvt implemented over hosts and parasites*

Configuration	Normalized runtime
sendEvt	1
aSendEvt on Host	4.96
aSendEvt on Parasite	1.05

barriers as described in (Bacon *et al.*, 2003). Eager read barrier elimination has marked performance benefits for repeated object accesses, such as array element traversals in a loop, where the read barrier is executed once when the array location is loaded into a register, but all further accesses can elide executing the barrier.

5.4 Microbenchmarks

In this section, we present micro-benchmark measurements for common patterns observed in the use of parasites to implement ACML primitives.

5.4.1 Lazy thread creation

The most common use case of parasitic threads is to model asynchronous short-lived computation, which might potentially block on a global resource, and be resumed at a later time. In ACML, this is observed in the base asynchronous event constructs `aSendEvt` and `aRecvEvt`, where the implicit thread created during event synchronization may block on the channel, even though the thread synchronizing on the asynchronous event never blocks. For a short-lived computation that does not block, the overhead of parasites is the cost of a non-tail call and the overheads of mechanisms in place for parasitic reification and inflation. On the other hand, spawning a host thread for a short-lived, non-blocking computation is the cost of allocating and collecting the host thread object, typically enqueueing and dequeueing from the scheduler, and finally switching to the thread.

To illustrate the differences in a potentially blocking computation, we implemented a single producer, single consumer program, where the producer repeatedly sends messages to the consumer. We compared the performance of sendEvt (synchronous) versus aSendEvt (asynchronous) implemented over host and parasitic threads. The results, presented in Table 1, show that `aSendEvt` encoded with parasites have a worst-case overhead of only 5% over their synchronous counterparts. Not surprisingly, in this micro-benchmark, there was no profitable parallelism to extract.

5.4.2 Inflation

In this section, we study the impact of inflating long running parasites (Section 3.9.1). We implemented a micro-benchmark that spawns a large number of long running computations. We compare the runtime performance of spawning the computation as a host thread and parasite with and without inflation.

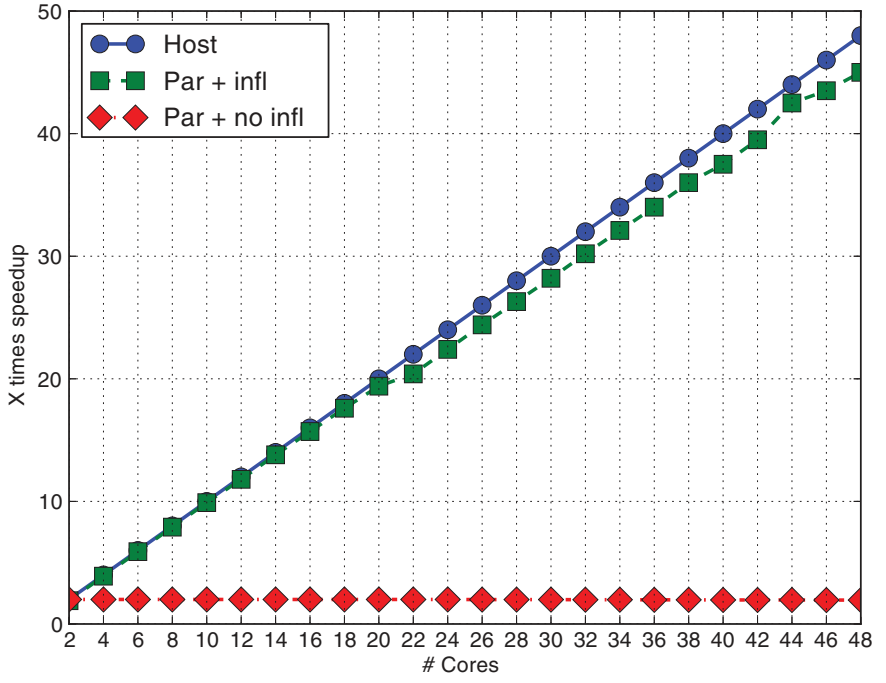


Fig. 19. (Colour online) Speedup of parasitic threads with and without inflation compared to host threads

Figure 19 shows the speedup of different configurations relative to the runtime of host threads on a single core. We see that, without inflation, parasites offer no parallelism and hence do not scale. With inflation, parasites perform identically to host threads, and incur no perceptible additional overhead for inflation. This benchmark shows that for a scalable implementation, it is necessary to inflate long running parasites to take advantage of parallel execution.

5.4.3 Read barrier overhead

In this section, we quantify the cost/benefit of read barriers in our system, which primarily motivates our new garbage collector design (RB-) that eliminates read barriers.

We evaluated a set of 8 benchmarks (described in Section 5.9) running on a 16 core AMD64, a 48 core Intel SCC and an 864 core Azul Vega 3 machine to measure read barrier overheads. Figure 20 shows these overheads as a percentage of mutator time. Our experiments reveal that, on average, the mutator spends 20.1%, 15.3% and 21.3% of time executing read barriers on the AMD64, SCC and Azul architectures, respectively, for our benchmarks.

While our read barrier implementation (Figure 18) is conditional (Baker, 1978), there exist unconditional variants (Brooks, 1984), where all loads unconditionally forward a pointer in the object header to get to the object. For objects that are not forwarded, this pointer points to the object itself. Although an unconditional read barrier, would have avoided the

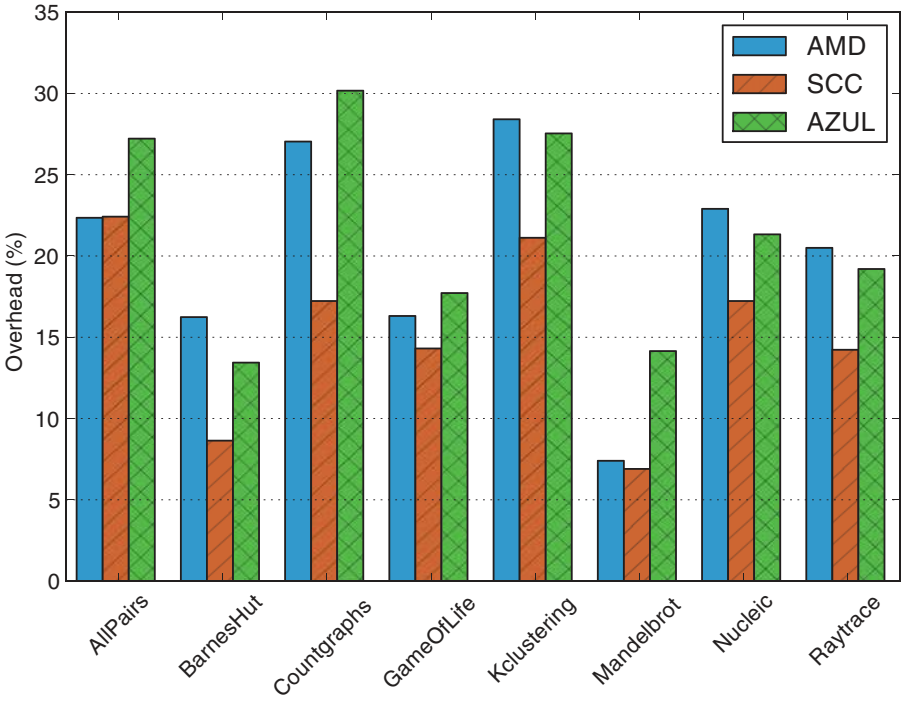


Fig. 20. (Colour online) Read barrier overhead as a percentage of mutator time.

Benchmark	AllPairs	BarnesHut	Countgraphs	GameOfLife	Kclustering	Mandelbrot	Nucleic	Raytrace
Checks (X 10^6)	9,753	2,864	2,584	4,858	3,780	2,980	2,887	2,217
Forwarded	123	52702	0	2143	101	23	328	0

Fig. 21. Effectiveness of read barrier checks: Checks represents the number of read barrier invocations and forwarded represents the number of instances when the read barrier encountered a forwarded object.

cost of the second branch in our read barrier implementation, it would necessitate having an additional address length field in the object header for an indirection pointer.

Most objects in our system tend to be small. In our benchmarks, we observed that 95% of the objects allocated were less than 3 words in size, including a word-sized header. The addition of an extra word in the object header for an indirection pointer would lead to substantial memory overheads, which in turn leads to additional garbage collection costs. Moreover, trading branches with loads is not a clear optimization as modern processors allow speculation through multiple branches, especially ones that are infrequent. Hence, we choose to encode read barriers conditionally rather than unconditionally.

The next question to ask is whether the utility of the read barrier justifies its cost. To answer this question, we measure the number of instances the read barrier is invoked and the number of instances the barrier finds a forwarded object (see Figure 21). We see

that read barriers find forwarded objects in less than one thousandth of a percent of the number of instances they are invoked. Thus, in our system, the cost of read barriers is substantial, but only rarely do they have to perform the task of forwarding references. These results motivated our interest in a memory management design that eliminates read barriers altogether.

5.5 Parasite benchmarks

Our benchmarks are parallel version of programs in MLton's benchmark suite. The benchmarks were made parallel with the help of ACML. The details of the benchmarks are provided below:

- **Mandelbrot**: a Mandelbrot set generator.
- **K-clustering**: a k-means clustering algorithm, where each stage is spawned as a server.
- **Barnes-Hut**: an n-body simulation using the Barnes-Hut algorithm.
- **Count-graphs**: computes all symmetries (automorphisms) within a set of graphs.
- **Mergesort**: merge-sort algorithm to sort one million numbers.
- **TSP**: a divide-and-conquer solution for the traveling salesman problem.
- **Raytrace**: a ray-tracing algorithm to render a scene.
- **Mat-mult**: a dense matrix multiplication of two 500 X 500 matrices.
- **Sieve-primes**: a streaming implementation of the sieve of Eratosthenes that generates first 3000 prime numbers. Each stage in the sieve is implemented as a lightweight thread.
- **Par-tak**: The highly recursive function Tak function, where every recursive call is implemented as a lightweight threads that communicates the result back to the caller over a channel. This is designed as a stress test for the runtime system.
- **Par-fib**: Parallel Fibonacci function implemented with lightweight threads to stress test the runtime system. Results are presented for calculating the 27th Fibonacci number.
- **Swerve**: A highly parallel concurrent web-server entirely written in CML, fully compliant with HTTP/1.1. Parts of Swerve were rewritten using ACML to allow for greater concurrency between interacting sub-components. Workloads were generated using Httperf — a standard tool for profiling web-server performance. For our results, 20,000 requests were issued at 2500 connections per second.

In order to capture the interaction of parasites and garbage collection, we evaluated the benchmarks in three different GC configurations:

- **STW**: A single shared heap, stop-the-world garbage collector.
- **RB+**: A thread-local garbage collector with split heaps which uses read barriers to preserve local heap invariants.
- **RB-**: A thread-local garbage collector with split heaps which uses procrastination and cleanliness to preserve local heap invariants.

The parasitic versions of the benchmarks utilize asynchronous communication between the threads, as well as parasites for implicit threads created by ACML. The non-parasitic

Table 2. *Thread and communication statistics for Stop-the-world GC running on 8 cores*

Benchmark	# Host (Par)	# Host (NPar)	# Par.	# Comm.	Avg host stack size (bytes)	Avg par. stack size (bytes)	# Force stack growth	Avg par. closure (bytes)
Mandelbrot	4083	6161	3306	4083	3463	155	1	328
K-clustering	771	1098	51402	51402	1804	136	0	240
Barnes-hut	8327	8327	24981	8075	1782	140	1	288
Count-graphs	153	153	148	516	3982	148	0	278
Mergesort	300	200260	2894734	2698766	1520	184	0	290
TSP	2074	4122	6143	4095	1304	303	1	528
Raytrace	580	590	24	256	4997	NA	0	NA
Matrix-multiply	528	528	512017	251001	2082	120	0	238
Sieve-primes	10411	12487	4872669	4946374	1738	139	0	243
Par-tak	122960	297338	175423	297361	1120	178	0	371
Par-fib	1	NA	466253	635621	1402	302	45087	463
Swerve	83853	110534	195534	367529	2692	160	2	405
Mean	19503	58327	766886	772090	2324	179	3758	334
Median	1423	6161	113413	151202	1793	155	0	290
SD	40211	101780	1526043	1516267	1217	64	13015	97

versions use synchronous communication, since asynchronous communication over host threads is expensive, as previously shown in Section. 5.4.1. The synchronous version of these benchmarks were appropriately restructured to eliminate the possibility of deadlocks. The benchmarks were run with same maximum heap size for each of the configurations.

5.6 Threads and communication

We have collected thread and communication statistics that provide insights into the behavior of our programs, and also illustrate the key to the benefits of parasites. Thread and communication statistics for parasitic (Par) and non-parasitic (NPar) versions are tabulated in Table. 2. These statistics were collected for stop-the-world GC version running on 48 cores. The statistics collected here were similar on the shared-heap parallel GC as well as running on different number of cores.

The number of hosts (# Hosts) show that our benchmarks are highly concurrent and amenable to parallel execution. The number of hosts created in parasitic versions is smaller than the non-parasitic versions. This is due to the fact many of the original host threads are created as parasites in the parasitic versions. On average, parasitic stacks are also much smaller than the host stacks. Also, not all parasites are ever reified and simply run to completion as a non-tail call. The cumulative effect of these on the parasitic version is the decreased cost of scheduling and management (allocation and collection).

Parasitic stacks are almost always smaller than the reserved space available in the host stacks. Hence, when parasites are attached to hosts, the hosts need not grow the stacks in order to fit in the parasites. We measured the number of instances where parasite being attached happened to be larger than the reserved space (# Force stack growth), and forces stack growth on the host. Apart from Par-fib, none of the benchmarks need to force stack

growths more than twice. In Par-fib, except the proto-thread, all of the threads created are parasitic, a few of which get inflated during the execution. Since all computation is encapsulated in parasitic threads, we see an increased number of stack growths corresponding to parasite migration. Parasitic stack measurements are unavailable for Raytrace since no parasites were reified. Parasitic closure size is the size of the parasitic stack including the transitive closure. This shows the average bytes copied for inter-processor parasitic communication.

The number of communications performed (# Comm) shows that the benchmarks are also communication intensive. For channel communication, a sender matching with a receiver is considered as one communication action. The number of communications performed is the same for the parasitic versions of the benchmarks, and only the nature of the communication varies (Synchronous versus Asynchronous). Since the benchmarks are communication intensive, asynchronous communication allows for greater concurrency between the communicating threads, and hence, exhibits better parallel performance.

The number of hosts for the non-parasitic version of Par-fib could not be obtained, since the program did not complete execution as it runs out of memory. The Par-fib program creates a very large number of threads, most of which are alive throughout the lifetime of the program. We observed that even for smaller Fibonacci numbers, the non-parasitic version of Par-fib spent most of its time performing GC. Not only did the parasitic version of Par-fib run, but also scaled as the number of cores were increased.

5.7 Scalability

Speedup results are presented in Figure 22. Baseline is a version of the programs optimized for sequential execution, without the overhead of threading wherever possible. Each data point represents the mean of the speedups of benchmarks presented in Section 5.5. In general, we see that the parasitic version perform better than their corresponding non-parasitic versions, and thread-local GCs (RB+ and RB- configurations) show better scalability and graceful performance degradation with increasing number of cores.

At peak scalability, `Par STW` is 23% faster than `Host STW`, `Par RB+` is 11% faster than `Host RB+` and `Par RB-` is 12% faster than `Host RB-`. In thread-local GCs, parasitic threads blocked on channels residing on shared heap must be lifted to the shared heap. Subsequently, the parasite must be copied back to the target local heap before being attached to or reified into a host thread. Hence, the improvement of parasitic versions over the host versions under thread-local GC scheme is subdued when compared to the stop-the-world version.

Parasitic threads also exhibited good scalability in the highly concurrent, communication intensive Par-tak and Par-fib benchmarks. These benchmarks were specifically intended to test the cost of thread creation and communication with very little computation. A large number of threads are created in these benchmarks with almost all of them being alive concurrently.

While the host version of Par-tak ran about between 1.1X to 1.4X faster than the baseline on different GC configurations, the parasitic versions were 1.6X to 2.2X faster than the baseline. Host version of Par-fib failed to run to completion due to the lack of space in the heap to accommodate all of the live host threads. Parasitic thread creation being cheap

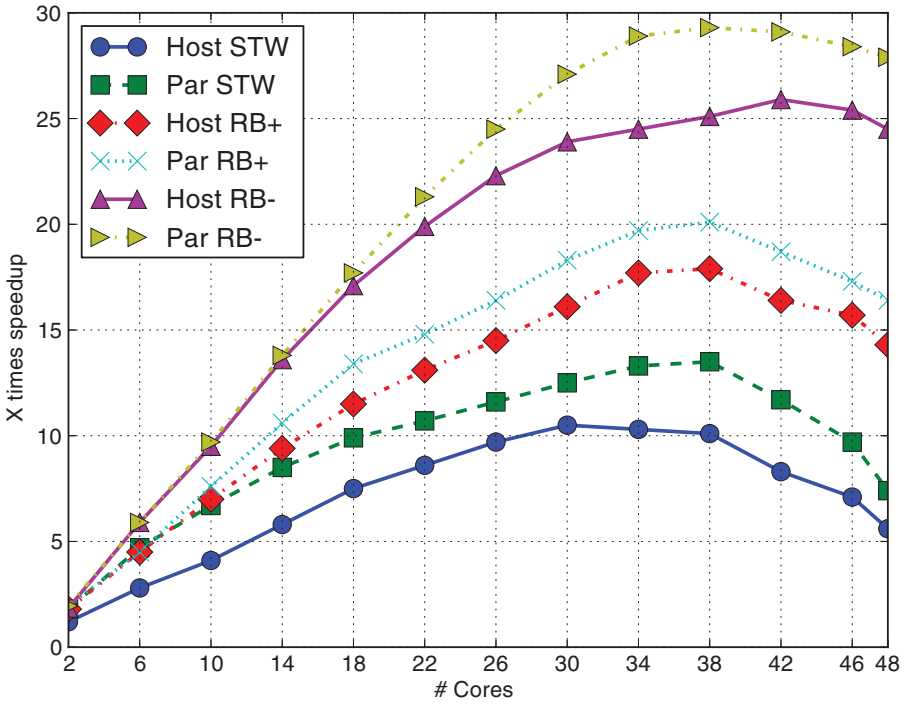


Fig. 22. (Colour online) Mean speedup of host and parasitic version of MULTIMLTON benchmarks under different GC configurations. Baseline is optimized sequential version.

allows the parasitic versions of Par-fib to run to completion. The parasitic versions of Par-fib ran between 1.2X to 1.8X faster than the baseline on different GC configurations.

5.8 Swerve

Swerve is designed as a collection of sub-components such as file processor, network processor, connection manager, etc. CML communication abstractions are utilized for inter-component interaction. This construction easily lends itself for multi-core execution. We evaluated the performance of Swerve on three configurations; a parallel CML implementation, an ACML implementation using host threads, and an ACML implementation using parasites. The parallel CML version uses synchronous communication for interaction between different sub-components. The experiments were performed in stop-the-world GC configuration. By utilizing asynchronous communication, overall concurrency in the system is increased.

The ACML version of Swerve improved upon the parallel CML implementation by eliminating lock-step file and network I/O, parallely packetizing file chunks, implementing non-blocking I/O, and rolling log. The details of the improvements along with the ACML code snippets is available is available as a case study in our earlier work (Ziarek *et al.*, 2011).

Benchmark	Allocation Rate (MB/s)			Bytes Allocated (GB)				# Threads		
	AMD	SCC	AZUL	AMD	SCC	AZUL	% Sh	AMD	SCC	AZUL
AllPairs	817	53	1505	16	16	54	11	256	512	32768
Barneshut	772	70	1382	20	20	876	2	512	1024	32768
Countgraphs	2594	144	4475	24	24	1176	1	128	256	16384
GameOfLife	2445	127	4266	21	21	953	13	256	1024	8192
Kclustering	3643	108	8927	32	32	1265	3	256	1024	8192
Mandelbrot	349	43	669	2	2	32	8	128	512	8192
Nucleic	1430	87	4761	13	14	609	1	64	384	16384
Raytrace	809	54	2133	11	12	663	4	128	256	2048

Fig. 23. Benchmark characteristics. %Sh represents the average fraction of bytes allocated in the shared heap across all the architectures.

We utilized Httperf to measure the reply rate of different configurations by repeatedly accessing a 10KB file. We issued connections at a rate of 2500 connections per second. In each case, Swerve was run on 48 cores. The PCML implementation scaled up to 1610 replies per second. As expected, naively spawning lightweight threads for asynchrony in the ACML version using host threads, suppressed the benefits of increased concurrency in the program due to communication overheads. Swerve with ACML implemented on hosts scaled up to 1890 replies per second. By mitigating the overheads of asynchrony, Swerve with parasites was able to scale better, and offered up to 3230 replies per second.

5.9 GC Benchmarks

The benchmarks shown in Figure 23 were designed such that the input size and the number of threads are tunable; each of these benchmarks were derived from a sequential Standard ML implementation, and parallelized using our lightweight thread system and CML-style (Reppy, 2007) and ACML message-passing communication. More specifically these benchmarks were chosen because they have interesting allocation and data sharing patterns.

- **AllPairs**: an implementation of Floyd-Warshall algorithm for computing all pairs shortest path.
- **BarnesHut**: an n-body simulation using Barnes-Hut algorithm.
- **CountGraphs**: computes all symmetries (automorphisms) within a set of graphs.
- **GameOfLife**: Conway's Game of Life simulator
- **Kclustering**: a k-means clustering algorithm, where each stage is spawned as a server.
- **Mandelbrot**: a Mandelbrot set generator.
- **Nucleic**: Pseudoknot (Hartel *et al.*, 1996) benchmark applied on multiple inputs.
- **Raytrace**: a ray-tracing algorithm to render a scene.

Parameters are appropriately scaled for different architectures to ensure sufficient work for each of the cores. The benchmarks running on AMD and SCC were given the same input size. Hence, we see that the benchmarks allocate the same amount of memory during their lifetime. But, we increase the number of threads on the SCC when compared to AMD since there is more hardware parallelism available. For Azul, we scale both the

input size and the number of threads, and as a result we see a large increase in bytes allocated when compared to the other platforms. Out of the total bytes allocated during the program execution, on average 5.4% is allocated in the shared heap. Thus, most of the objects allocated are collected locally, without the need for stalling all of the mutators.

We observe that the allocation rate is highly architecture dependent, and is the slowest on the SCC. Allocation rate is particularly dependent on memory bandwidth, processor speed and cache behavior. On the SCC, not only is the processor slow (533MHz) but also the serial memory bandwidth for our experimental setup is only around 70 MB/s.

5.10 Performance

Next, we analyze the performance of the new local collector design. In order to establish a baseline for the results presented, we have ported our runtime system to utilize the Boehm-Demers-Weiser (BDW) conservative garbage collector (Boehm, 2012). We briefly describe the port of our runtime system utilizing BDW GC.

Although BDW GC is conservative, it can utilize tracing information when provided. MULTIMLTON generates tracing information for all objects, including the stack. However, in the case of BDW GC under MULTIMLTON, we utilize the tracing information for all object allocations except the stack. This is because stack objects in our runtime system represent all of the reserved space for a stack, while only a part of the stack is actually used which can grow and shrink as frames are pushed and popped. Since the BDW GC does not allow tracing information of objects to be changed after allocation, we only scan the stack objects conservatively. BDW uses a mark-sweep algorithm, and we enable parallel marking and thread-local allocations.

Figure 24(a) illustrates space-time trade-offs critical for any garbage collector evaluation. STW GC is the baseline stop-the-world collector described in Section 4.1, while RB+ and RB- are local collectors. RB+ is a local collector with read barriers while RB- is our new local collector design without read barriers, exploiting procrastination and cleanliness. We compare the normalized running times of our benchmarks under different garbage collection schemes as we decrease the heap size. For each run of the experiment, we decrease the maximum heap size allowed and report the maximum size of the heap utilized. Thus, we leave it to the collectors to figure out the optimal heap size, within the allowed space. This is essential for the local collectors, since the allocation pattern of each core is usually very different and depends on the structure of the program.

The results presented here were collected on 16 cores. As we decrease overall heap sizes, we see programs under all of the different GC schemes taking longer to run. But RB- exhibits better performance characteristics than its counterparts. We observe that the minimum heap size under which the local collectors would run is greater than the STW and BDW GCs. In the local collectors, since the heap is split across all of the cores, there is more fragmentation. Also, under the current scheme, each local collector is greedy and will try to utilize as much heap as it can in order to reduce the running time (by choosing semi-space collection over mark-compact), without taking into account the heap requirements of other local collectors. Currently, when one of the local cores runs out of memory, we terminate the program. Since we are interested in throughput on scalable architectures where memory is not a bottleneck, we have not optimized the collectors for memory

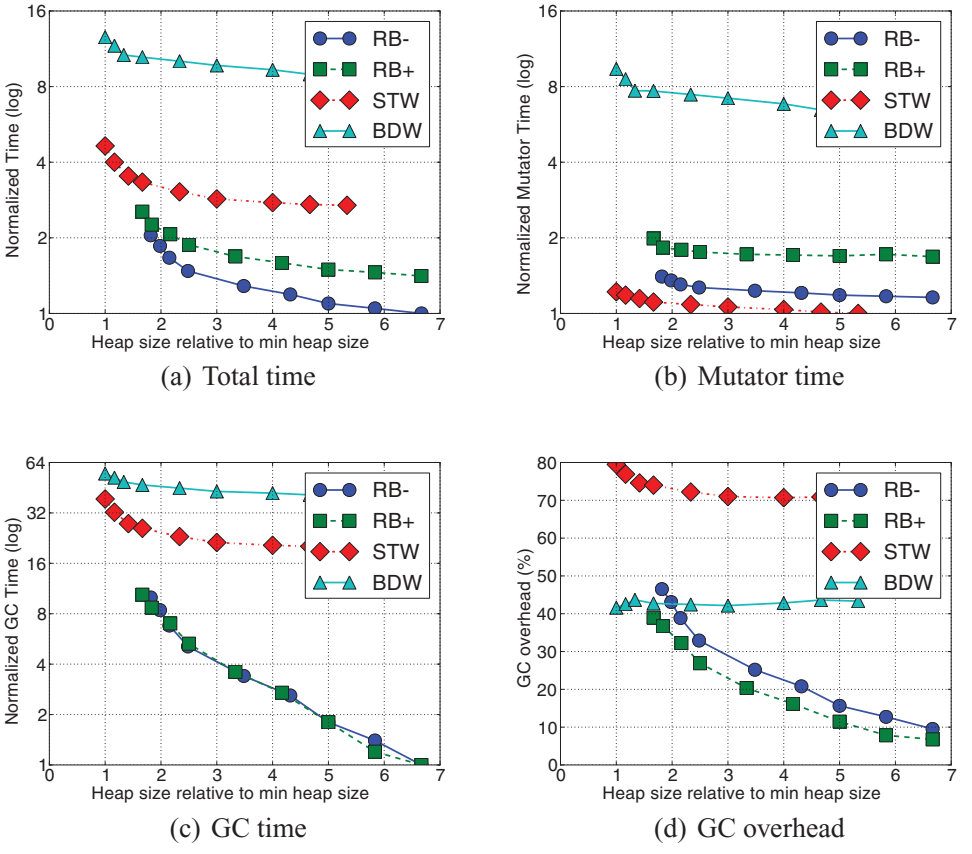


Fig. 24. (Colour online) Performance comparison of Stop-the-world (STW), Boehm-Demers-Weiser conservative garbage collector (BDW), local collector with read barriers (RB+), and local collector without read barriers (RB-): Geometric mean for 8 benchmarks running on AMD64 with 16 cores.

utilization. We believe we can modify our collector for memory constrained environments by allowing local heaps to shrink on demand and switch from semi-space to compacting collection, if other local heaps run out of memory.

The STW and BDW GCs are much slower than the two local collectors. In order to study the reason behind this slowdown, we separate the mutator time (Figure 24(b)) and garbage collection time (Figure 24(c)). We see that STW GC is actually faster than the local collectors in terms of mutator time, since it does not pay the overhead of executing read or write barriers. But, since every collection requires stopping all the mutators and a single collector performs the collection, it executes sequentially during a GC. Figure 24(d) shows that roughly 70% of the execution total time for our benchmarks under STW is spent performing GCs, negatively impacting scalability.

Interestingly, we see that programs running under the BDW GC are much slower when compared to other GCs. This is mainly due to allocation costs. Although we enabled thread-local allocations, on 16 cores, approximately 40% of the time was spent on object allocation. While the cost of object allocation for our other collectors only involves bumping the frontier, allocation in BDW GC is significantly more costly, involving scanning through a

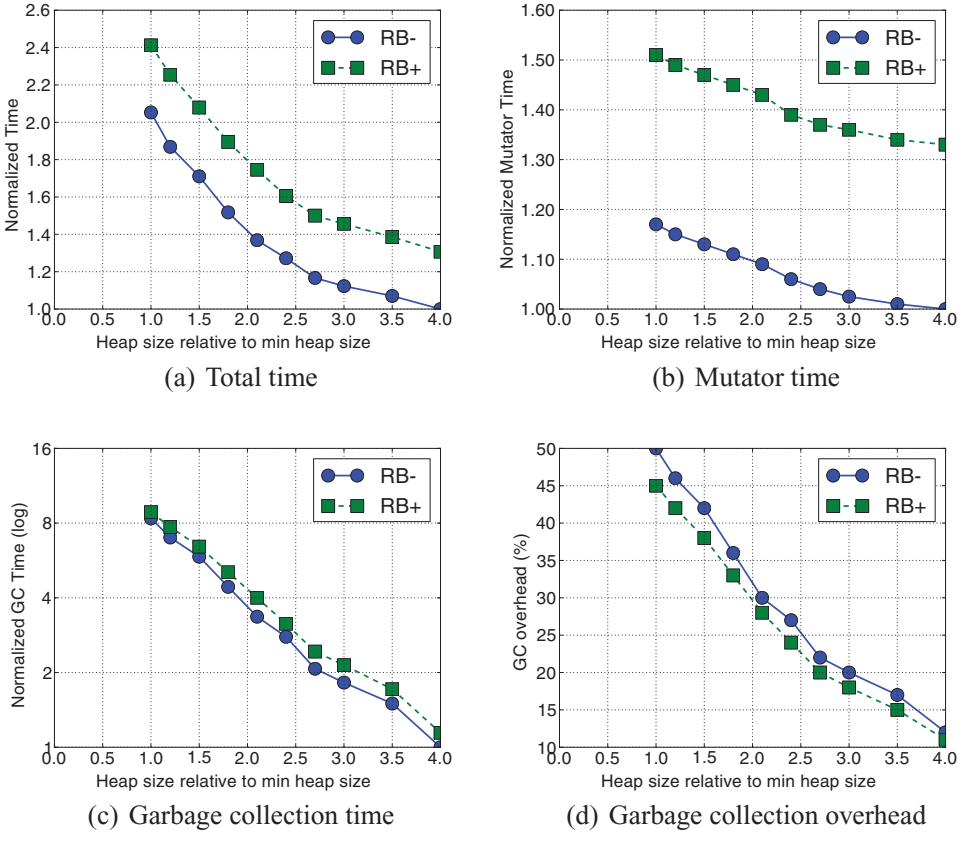


Fig. 25. (Colour online) Performance comparison of local collector with read barriers (RB+) and local collector without read barriers (RB-): Geometric mean for 8 benchmarks running on Azul with 846 cores.

free list, incurring substantial overhead. Moreover, BDW GC is tuned for languages like C/C++ and Java, where the object lifetimes are longer and allocation rate is lower when compared to functional programming languages.

In Figure 24(a), at 3X the minimum heap size, RB+, STW and BDW GCs are 32%, 106% and 584% slower than the RB- GC. We observe that there is very little difference between RB+ and RB- in terms of GC time but the mutator time for RB+ is consistently higher than RB- due to read barrier costs. The difference in mutator times is consistent since the increased number of GCs incurred as a result does not adversely affect it. This also explains why the total running time of RB- approaches RB+ as the heap size is decreased in Figure 24(a). With decreasing heap size, the programs spend a larger portion of the time performing GCs, while the mutator time remains consistent. Hence, there is diminishing returns from using RB- as heap size decreases.

Next, we analyze the performance on Azul (see Figure 25). We only consider performance of our local collectors since our AMD results show that the other collectors (STW and BDW) simply do not have favorable scalability characteristics. At 3X the minimum heap size, RB- is 30% faster than RB+.

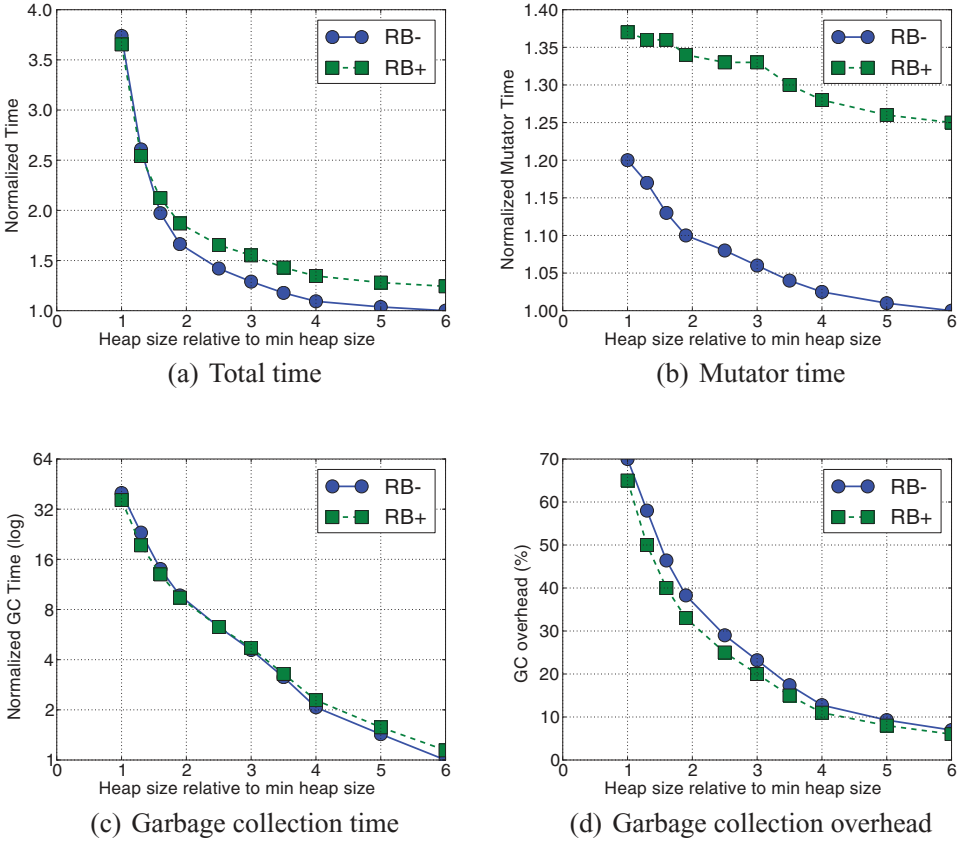


Fig. 26. (Colour online) Performance comparison of local collector with read barriers (RB+) and local collector without read barriers (RB-): Geometric mean for 8 benchmarks running on SCC with 48 cores.

SCC performance results are presented in Figure 26. At 3X the minimum heap size, RB- is 20% faster than RB+. From the total time graphs, we can see that the programs tend to run much slower as we decrease the heap sizes on SCC. Compared to the fastest running times, the slowest running time for RB- is 2.01X, 2.05X, and 3.74X slower on AMD, Azul, and SCC respectively. This is due to the increased number of shared heap collections, which are more expensive than other architectures as a result of the absence of caching. This is noticeable by a more rapid increase in garbage collection overhead percentages (Figure 26(d)).

5.11 Impact of cleanliness

Cleanliness information allows the runtime system to avoid preempting threads on a write barrier when the source of an exporting write is clean. In order to study the impact of cleanliness, we removed the reference counting code and cleanliness check from the write barrier; thus, every exporting write results in a thread preemption and stall. The results presented here were taken on the AMD machine with programs running on 16 cores with the benchmark configurations given in Figure 23. The results are similar on SCC and Azul.

Benchmark	AllPairs	BarnesHut	CountGraphs	GameOfLife	Kclustering	Mandelbrot	Nucleic	Raytrace
RB-	1831	46532	154	38621	25812	132	156	3523
RB- MU-	1831	4092312	192	735543	50323	209	433092	3743
RB- CL-	124232	67156821	50178	5867423	27023911	25491	912349	61198

Fig. 27. Number of preemptions on write barrier.

Benchmark	AllPairs	BarnesHut	CountGraphs	GameOfLife	Kclustering	Mandelbrot	Nucleic	Raytrace
RB-	0.08	0.17	0	3.54	0	1.43	0	1.72
RB- MU-	0.08	19.2	0.03	9.47	0.02	2.86	9.37	1.72
RB- CL-	38.55	100	0.18	99.75	21.64	86.22	19.3	24.86

Fig. 28. Forced GCs as a percentage of the total number of major GCs.

Figure 27 shows the number of preemptions on write barrier for different local collector configurations. RB- row represents the local collector designs with all of the features enabled; RB- MU- row shows a cleanliness optimization that does not take an object's mutability into consideration in determining cleanliness (using only recorded reference counts instead), and row RB- CL- row represents preemptions incurred when the collector does not use any cleanliness information at all. Without cleanliness, on average, the programs perform substantially more preemptions when encountering a write barrier.

Recall that if all of the threads belonging to a core get preempted on a write barrier, a local major GC is *forced*, which lifts all of the sources of exporting writes, fixes the references to forwarding pointers and unblocks the stalled threads. Hence, an increase in the number of preemptions leads to an increase in the number of local collections.

Figure 28 shows the percentage of local major GCs that were forced compared to the total number of local major GCs. Row RB- CL- shows the percentage of forced GCs if cleanliness information is not used. On average, 49% of local major collection performed is due to forced GCs if cleanliness information is not used, whereas it is less than 1% otherwise. On benchmarks like `BarnesHut`, `GameOfLife` and `Mandelbrot`, where all of the threads tend to operate on a shared global data structure, there are a large number of exporting writes. On such benchmarks almost all local GCs are forced in the absence of cleanliness. This adversely affects the running time of programs.

Figure 29 shows the running time of programs without using cleanliness. On average, programs tend to run 28.2% slower if cleanliness information is ignored. The results show that cleanliness analysis therefore plays a significant role in our GC design.

5.12 Impact of immutability

If the source of an exporting write is immutable, we can make a copy of the object in the shared heap and assign a reference to the new shared heap object to the target. Hence, we

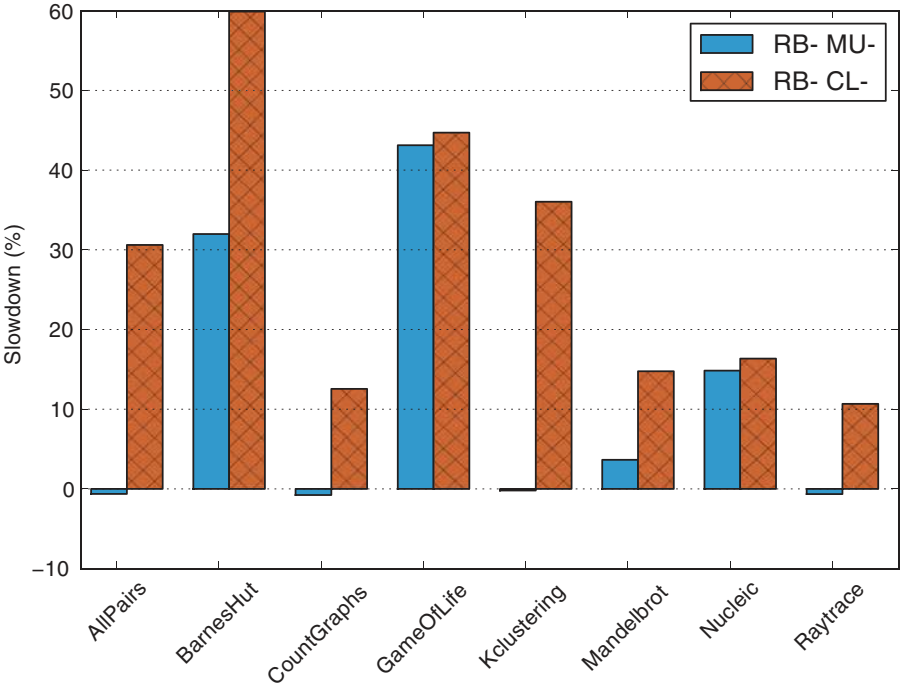


Fig. 29. (Colour online) Impact of utilizing object mutability information and cleanliness analysis on the performance of RB- GC.

can ignore the reference count of such objects. Not all languages may have the ability to distinguish between mutable and immutable objects in the compiler or in the runtime system. Hence, we study the impact of our local collector design with mutability information in mind. To do this, we ignore the test for mutability in the cleanliness check (Line 4 in Figure 11) and modify the object lifting code in Figure 14 to treat all objects as mutable.

RB- MU- row in Figures 27 and 28 show the number of write barrier preemptions and the percentage of forced GCs, respectively, if all objects were treated as mutable. For some programs such as `AllPairs`, `CountGraphs`, or `Kclustering`, object mutability does not play a significant factor. For benchmarks where it does, distinguishing between mutable and immutable objects helps avoid inducing preemptions on a write barrier since a copy of the immutable object can be created in the shared heap without the need to repair existing references to the local heap copy.

Figure 29 shows the performance impact of taking object mutability into account. While ignoring object mutability information, `BarnesHut`, `GameOfLife` and `Nucleic` are slower due to the increased number of forced GCs. Interestingly, `AllPairs`, `CountGraphs`, `Kclustering` and `Raytrace` are marginally faster if the mutability information is ignored. This is due to not having to manipulate the `imSet` (Line 14 in Figure 14), and walking immutable objects after the objects are lifted (Lines 25-27 in Figure 14). On average, we see a 11.4% performance loss if mutability information is not utilized for cleanliness.

Benchmark	AllPairs	Barneshut	Countgraphs	GameOfLife	Kclustering	Mandelbrot	Nucleic	Raytrace
% LM clean	5.3	13.4	8.6	23.2	17.6	4.5	13.3	8.2
Avg. session size (bytes)	2908	1580	3612	1344	2318	8723	1264	1123

Fig. 30. Impact of heap session: % LM clean represents the fraction of instances when a clean object closure has at least one object with `LOCAL_MANY` references.

5.13 Impact of heap session

In order to assess the effectiveness of using heap sessions, we measured the percentage of instances where the source of an exporting write is clean with at least one of the objects in the closure has a `LOCAL_MANY` reference. During such instances, we walk the current heap session to fix any references to forwarded objects. Without using heap sessions, we would have preempted the thread in the write barrier, reducing available concurrency. The results were obtained on the AMD with programs running on 16 cores with the configuration given in Figure 23. The results are presented in Figure 30.

The first row shows the percentage of instances when an object closure is clean and has at least one object with `LOCAL_MANY` references. On average, we see that 12% of clean closures have at least one object with `LOCAL_MANY` references. We also measured the average size of heap sessions when the session is traced as a part of lifting an object closure to the shared heap (Lines 29-31 in Figure 14). The average size of a heap session when it is traced is 2859 bytes, which is less than a page size. These results show that utilizing heap sessions significantly contributes to objects being tagged as clean, and heap sessions are small enough to not introduce significant overheads during tracing.

6 Related work

In this section we discuss related work, focusing on the most closely related systems to MULTIMLTON. We split our discussion into threading mechanisms and garbage collection strategies.

6.1 Lightweight threads

There are a number of languages and libraries that support varying kinds of message-passing styles. Systems such as MPI (Li *et al.*, 2008; Tang & Yang, 2001) support per-processor static buffers, while CML (Reppy, 2007), Erlang (Armstrong *et al.*, 1996), F# (Syme *et al.*, 2007), and MPJ (Baker & Carpenter, 2000) allow for dynamic channel creation. Although MPI supports two distinct styles of communication, both asynchronous and synchronous, not all languages provide primitive support for both. For instance, Erlang's fundamental message passing primitives are asynchronous, while CML's primitives are synchronous. We described how parasites can be utilized to improve the performance of ACML, which supports both synchronous and asynchronous communication.

There has been much interest in lightweight threading models ranging from using runtime managed thread pools such as those found in C# (C# Language Specification, 2014) and the java.util.concurrent package (Lea, 1999), to continuation-based systems found in functional languages such as Scheme (Mohr *et al.*, 1990; Kranz *et al.*, 1989), CML (Reppy, 2007), Haskell (Sivaramakrishnan *et al.*, 2013; Harris *et al.*, 2005), and Erlang (Armstrong *et al.*, 1996). Parasitic threads can be viewed as a form of lightweight threading, though there are a number of key differences. In addition to describing a non-local control flow abstraction, scheduling and migration are an integral part of parasitic threads. This makes parasites ideal for short-lived computation, by avoiding the scheduler overheads and localizing the thread interactions.

Worker pool models for executing asynchronous tasks have been explored in languages like F#, for latency masking in I/O operations. However, this model is not effective for implementing tightly interacting asynchronous tasks. In the worker pool model, even a short-lived asynchronous computation is not started immediately, but only when a worker is free, and hence delaying transitively dependent tasks. This model also suffers from contention on the task queues in a parallel setting.

Previous work on avoiding thread creation costs have focused on compiler and runtime techniques for different object representations (Mohr *et al.*, 1990; Frigo *et al.*, 1998) and limiting the computations performed in a thread (Kale & Krishnan, 1993; Kranz *et al.*, 1989). The most similar among the previous work to parasitic threads is Lazy Threads (Goldstein *et al.*, 1996), which allow encoding arbitrary computation in a potentially parallel thread. A lazy thread starts off as regular function calls, just like parasites. The key difference of this work to parasites is that when a lazy thread blocks, it is not reified, and is left in the host stack, right on top of its parent. Any subsequent function calls made by the parent are executed in a new stack, and the system thus has spaghetti stacks.

Lazy threads assume that blocking is rare, whereas the common case in a parasitic thread is blocking on a synchronous communication. Leaving the parasites on the host would be detrimental to the performance since even if one parasite blocks, all subsequent sequential function calls and parasite creation on the host will have to be run on a separate thread. Parasites also support implicit migration to the point of synchronization, and helps to localize interactions, which is not possible if the parasites are pinned to the hosts.

Parasites also share some similarity to dataflow (Johnston *et al.*, 2004; Nikhil & Arvind, 2001) languages, insofar as they represent asynchronous computations that block on dataflow constraints; in our case, these constraints are manifest via synchronous message-passing operations. Unlike classic dataflow systems, however, parasites are not structured as nodes in a dependency graph, and a parasite maintains no implicit dependency relationship with the thread that created it. CML (Reppy, 2007) supports buffered channels with mailboxes. While parasites can be used to encode asynchronous sends, unlike buffered channels, they provide a general solution to encode arbitrary asynchronous computation.

Work sharing and work stealing (Blumofe & Leiserson, 1999; Agrawal *et al.*, 2007) are well-known techniques for load balancing multi threaded tree-structured computations, and have been used effectively in languages like Cilk (Frigo *et al.*, 1998) to improve performance. In our system, host threads are scheduled by work sharing, by eagerly spawning in a round-robin fashion, while parasites are implicitly stolen during channel communication.

6.2 Garbage Collection

Over the years, several local collector designs (Steele, 1975; Doligez & Leroy, 1993; Steensgaard, 2000; Anderson, 2010) have been proposed for multithreaded programs. Recently, variations of local collector design have been adopted for multithreaded, functional language runtimes like GHC (Marlow & Peyton Jones, 2011) and Manticore (Auhagen *et al.*, 2011). Doligez et al. (Doligez & Leroy, 1993) proposed a local collector design for ML with threads where all mutable objects are allocated directly on the shared heap, and immutable objects are allocated in the local heap. Similar to our technique, whenever local objects are shared between cores, a copy of the immutable object is made in the shared heap. Although this design avoids the need for read and write barriers, allocating all mutable objects, irrespective of their sharing characteristics can lead to poor performance due to increased number of shared collections, and memory access overhead due to NUMA effects and uncached shared memory as in the case of SCC. It is for this reason we do not treat the shared memory as the oldest generation for our local generation collector unlike other designs (Doligez & Leroy, 1993; Marlow & Peyton Jones, 2011).

Several designs utilize static analysis to determine objects that might potentially escape to other threads (Jones & King, 2005; Steensgaard, 2000). Objects that do not escape are allocated locally, while all others are allocated in the shared heap. The usefulness of such techniques depends greatly on the precision of the analysis, as objects that might potentially be shared are allocated on the shared heap. This is undesirable for architectures like the SCC where shared memory accesses are very expensive compared to local accesses. Compared to these techniques, our design only exports objects that are definitely shared between two or more cores. Our technique is also agnostic to the source language, does not require static analysis, and hence can be implemented as a lightweight runtime technique.

Anderson (Anderson, 2010) describes a local collector design (TGC) that triggers a local garbage collection on every exporting write of a mutable object, while immutable objects, that do not have any pointers, are copied to the shared heap. This scheme is a limited form of our cleanliness analysis. In our system, object cleanliness neither solely relies on mutability information, nor is it restricted to objects without pointer fields. Moreover, TGC does not exploit delaying exporting writes to avoid local collections. However, the paper proposes several interesting optimizations that are applicable to our system. In order to avoid frequent mutator pauses on exporting writes, TGC's local collection runs concurrently with the mutator. Though running compaction phase concurrently with the mutator would require read barriers, we can enable concurrent marking to minimize pause times. TGC also proposes watermarking scheme for minimizing stack scanning, which can be utilized in our system to reduce the stack scanning overheads during context switches and exporting writes of clean objects.

Marlow *et al*. (Marlow & Peyton Jones, 2011) propose exporting only part of the transitive closure to the shared heap, with the idea of minimizing the objects that are globalized. The rest of the closure is exported essentially on demand during the next access from another core. This design mandates the need for a read barrier to test whether the object being accessed resides in the local heap of another core. However, since the target language is Haskell, there is an implicit read barrier on every load, to check whether the thunk has

already been evaluated to a value. Since our goal is to eliminate read barriers, we choose to export the transitive closure on an exporting write.

7 Conclusion

In this paper we introduced ACML, an extension of CML that allows for the creation of composable asynchronous and heterogeneous events and protocols, in the context of MULTIMLTON. To support efficient ACML programs we introduce the design and implementation of *parasites*, a runtime thread management mechanism intended to reduce the cost of supporting asynchrony. Further, we describe a new garbage collector design tuned for ACML that obviates the need for read barriers by stalling writes and a new cleanliness property for identifying which objects can be moved from a local heap to the global heap safely without establishing forwarding pointers.

Acknowledgments

We sincerely thank Michael Sperber and the anonymous reviewers for their requests for clarification, constructive criticism, and numerous detailed suggestions that have helped improve our presentation. This work is supported by the National Science Foundation under grants CCF-1216613, CCF-0811631, and CNS-0958465, and by gifts from Intel and Samsung Corporation.

References

Agrawal, K., He, Y., & Leiserson, C. E. (2007) Adaptive work stealing with parallelism feedback. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. (PPoPP '07)*. New York, NY, USA: ACM, pp. 112–120.

Anderson, T. A. (2010) Optimizations in a private nursery-based garbage collector. In *Proceedings of the 2010 International Symposium on Memory Management. (ISMM '10)*. New York, NY, USA: ACM, pp. 21–30.

Appel, A. W. (1989) Simple generational garbage collection and fast allocation. *Softw. Pract. Exp.* **19**(February), 171–183.

Armstrong, J., Virding, R., Wikstrom, C., & Williams, M. (1996) *Concurrent Programming in Erlang*, 2nd ed. Prentice-Hall.

Auhagen, S., Bergstrom, L., Fluet, M., & Reppy, J. (2011) Garbage collection for multicore NUMA machines. In *Proceedings of the 2011 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*. MSPC '11. New York, NY, USA: ACM, pp. 51–57.

Bacon, D. F., Cheng, P., & Rajan, V. T. (2003) A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '03)*. New York, NY, USA: ACM, pp. 285–298.

Baker, Jr. and Henry G. (1978) List processing in real time on a serial computer. *Commun. ACM*, **21**(April), 280–294.

Baker, M., & Carpenter, B. (2000) MPJ: A proposed java message passing API and environment for high performance computing. In *Parallel and Distributed Processing*, Rolim, J. (ed), Lecture Notes in Computer Science, vol. 1800. Berlin, Heidelberg: Springer, pp. 552–559.

Biagioni, E., Cline, K., Lee, P., Okasaki, C., & Stone, C. (1998) Safe-for-space threads in standard ML. *Higher Order Symbo. Comput.* **11**(2), 209–225.

Blackburn, S. M., & Hosking, A. L. (2004) Barriers: Friend or foe? In *Proceedings of the 4th International Symposium on Memory Management (ISMM '04)*. New York, NY, USA: ACM, pp. 143–151.

Blumofe, R. D. & Leiserson, C. E. (1999) Scheduling multithreaded computations by work stealing. *J. ACM*, **46**(5), 720–748.

Boehm, H. (2012) *A Garbage Collector for C and C++*. Available at: http://www.hpl.hp.com/personal/Hans_Boehm/gc.

Brooks, R. A. (1984) Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming (LFP '84)*. New York, NY, USA: ACM, pp. 256–262.

Bruggeman, C., Waddell, O., & Dybvig, R. K. (1996) Representing control in the presence of one-shot continuations. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation (PLDI '96)*. New York, NY, USA: ACM, pp. 99–107.

C# Language Specification. (2014)

Available at: http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf.

Chaudhuri, A. (2009) A concurrent ML library in concurrent haskell. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09)*. New York, NY, USA: ACM, pp. 269–280.

Doligez, D. & Leroy, X. (1993) A concurrent, generational garbage collector for a multithreaded implementation of ML. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '93)*. New York, NY, USA: ACM, pp. 113–123.

Feeley, M. & Miller, J. S. (1990) A parallel virtual machine for efficient scheme compilation. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming (LFP '90)*. New York, NY, USA: ACM, pp. 119–130.

Felleisen, M. & Friedman, D. (1986) Control operators, the SECD Machine, and the λ-calculus. In *Formal Description of Programming Concepts III*, pp. 193–217.

Frigo, M., Leiserson, C. E., & Randall, K. H. (1998) The Implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*. PLDI '98. New York, NY, USA: ACM, pp. 212–223.

GHC. (2014) *Glasgow Haskell Compiler*. Available at: http://www.haskell.org/ghc.

Goldman, R. & Gabriel, R. P. (1988). Qlisp: Experience and new directions. In *Proceedings of the ACM/SIGPLAN Conference on Parallel Programming: Experience with Applications, Languages and Systems (PPEALS '88)*. New York, NY, USA: ACM, pp. 111–123.

Goldstein, S. C., Schauser, K. E., & Culler, D. E. (1996) Lazy threads: Implementing a fast parallel call. *J. Parallel and Distrib. Comput. - Special Issue on Multithreading for Multiprocessors*, **37**(1), 5–20.

Harris, T., Marlow, S., & Jones, S. P. (2005) Haskell on a shared-memory multiprocessor. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell. (Haskell '05)*. New York, NY, USA: ACM, pp. 49–61.

Hartel, P. H., Feeley, M., Alt, M., & Augustsson, L. (1996) Benchmarking implementations of functional languages with "Pseudoknot", a float-intensive benchmark. *J. Funct. Program.* **6**(4), 621–655. Available at: http://doc.utwente.nl/55704/.

Hot-Split. (2013) *Contiguous Stacks in Go*. Available at: http://golang.org/s/contigstacks.

Intel. (2012) *SCC Platform Overview*. Available at: http://communities.intel.com/docs/DOC-5512.

Johnston, W. M., Hanna, J. R. Paul, & Millar, R. J. (2004) Advances in dataflow programming languages. *ACM Comput. Surv.* **36**(1), 1–34.

Jones, R. & King, A. C. (2005) A fast analysis for thread-local garbage collection with dynamic class loading. In *Proceedings of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation*. Washington, DC, USA: IEEE Computer Society, pp. 129–138.

Kale, L. V. & Krishnan, S. (1993) CHARM++: A portable concurrent object oriented system based on C++. In *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '93)*. New York, NY, USA: ACM, pp. 91–108.

Kranz, D. A., Halstead, Jr., R. H., & Mohr, E. (1989). Mul-T: A high-performance Parallel Lisp. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation (PLDI '89)*. New York, NY, USA: ACM, pp. 81–90.

Lea, Doug. (1999) *Concurrent Programming in Java. Second Edition: Design Principles and Patterns.* 2nd edn. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Li, G., Delisi, M., Gopalakrishnan, G., & Kirby, R. M. (2008) Formal specification of the MPI-2.0 standard in TLA+. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08)*. New York, NY, USA: ACM, pp. 283–284.

Marlow, S. & Peyton Jones, S. (2011) Multicore garbage collection with local heaps. In *Proceedings of the 2011 International Symposium on Memory Management (ISMM '11)*. New York, NY, USA: ACM, pp. 21–32.

McKay, D. P. & Shapiro, S. C. (1980) MULTI - a LISP based multiprocessing system. In *Proceedings of the 1980 ACM Conference on LISP and Functional Programming (LFP '80)*. New York, NY, USA: ACM, pp. 29–37.

Miller, J. S. (1988) Implementing a scheme-based Parallel processing system. *Int. J. Parallel Program.* **17**(5), 367–402.

MLton. (2012) *The MLton Compiler and Runtime System*. Available at: http://www.mlton.org.

Mohr, E., Kranz, D. A., & Halstead, Jr., R. H. (1990). Lazy task creation: A technique for increasing the granularity of Parallel programs. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming. (LFP '90)*. New York, NY, USA: ACM, pp. 185–197.

Nikhil, R. & Arvind. (2001) *Implicit Parallel Programming in pH*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Raymond, D. J. (2000). SISAL: A safe and efficient language for numerical calculations. *Linux J.* **2000**(80es).

Reppy, J. H. (2007) *Concurrent Programming in ML*. Cambridge, UK: Cambridge University Press.

Reppy, J., Russo, C. V., & Xiao, Y. (2009) Parallel concurrent ML. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*. ICFP '09. New York, NY, USA: ACM, pp. 257–268.

Sansom, P. M. (1991) Dual-mode garbage collection. In *Proceedings of the Workshop on the Parallel Implementation of Functional Languages,* pp. 283–310.

Sivaramakrishnan, K. C., Ziarek, L., Prasad, R., & Jagannathan, S. (2010) Lightweight asynchrony using parasitic threads. In *Proceedings of the 5th ACM SIGPLAN workshop on Declarative Aspects of Multicore Programming (DAMP '10)*. New York, NY, USA: ACM, pp. 63–72.

Sivaramakrishnan, K. C., Ziarek, L., & Jagannathan, S. (2012) Eliminating read barriers through procrastination and cleanliness. In *Proceedings of the 2012 International Symposium on Memory Management (ISMM '12)*. New York, NY, USA: ACM, pp. 49–60.

Sivaramakrishnan, K. C., Harris, T., Marlow, S., & Peyton Jones, S. (2013) *Composable Schedular Activations for Haskell*. Tech. rept. Microsoft Research, Cambridge.

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

Steele, Jr., G. L. (1975) Multiprocessing compactifying garbage collection. *Commun. ACM* **18**(9), 495–508.

Steensgaard, B. (2000) Thread-specific heaps for multi-threaded programs. In *Proceedings of the 2000 International Symposium on Memory Management (ISMM '00)*. New York, NY, USA: ACM, pp. 18–24.

Svensson, H., Fredlund, L.-A., & Benac Earle, C. (2010) A unified semantics for future erlang. In *Proceedings of the 9th ACM SIGPLAN Workshop on Erlang (Erlang '10)*. New York, NY, USA: ACM, pp. 23–32.

Syme, D., Granicz, A., & Cisternino, A. (2007) *Expert F#*. Apress.

Tang, H. & Yang, T. (2001) Optimizing threaded MPI execution on SMP clusters. In *Proceedings of the 15th International Conference on Supercomputing*. ICS '01. New York, NY, USA: ACM, pp. 381–392.

Wand, M. (1980) Continuation-based multiprocessing. In *Proceedings of the 1980 ACM Conference on LISP and Functional Programming (LFP '80)*. New York, NY, USA: ACM, pp. 19–28.

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