2011

Featherweight Threads for Communication

KC Sivaramakrishnan
Purdue University, chandras@cs.purdue.edu

Lukasz Ziarek
Purdue University, lziarek@cs.purdue.edu

Suresh Jagannathan
Purdue University, suresh@cs.purdue.edu

Report Number:

Featherweight Threads for Communication

KC Sivaramakrishnan Lukasz Ziarek Suresh Jagannathan
Purdue University
{chandras, lziarek, suresh}@cs.purdue.edu

Abstract

Message-passing is an attractive thread coordination mechanism because it cleanly delineates points in an execution when threads communicate, and unifies synchronization and communication. To enable greater performance, asynchronous or non-blocking extensions are usually provided, that allow senders and receivers to proceed even if a matching partner is unavailable. However, realizing the potential of these techniques in practice has remained a difficult endeavor due to the associated runtime overheads.

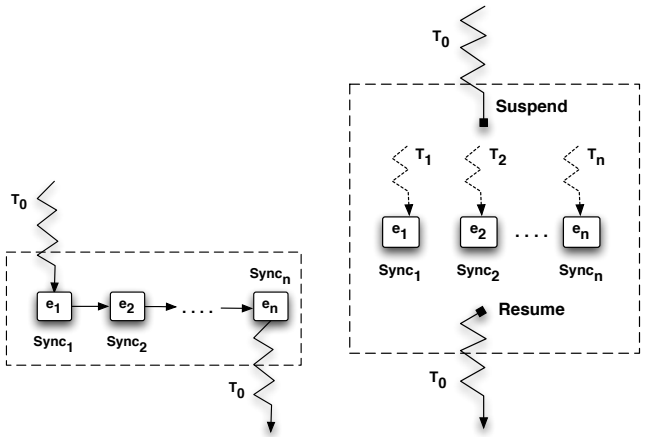
We introduce *parasitic threads*, a novel mechanism for expressing asynchronous computation, aimed at reducing runtime overheads. Parasitic threads are implemented as raw stack frames within the context of its host — a lightweight thread. Parasitic threads are self-scheduled and migrated based on their communication patterns. This avoids scheduling overheads and localizes interaction between parasites.

We describe an implementation of parasitic threads and illustrate their utility in building efficient asynchronous primitives. We present an extensive evaluation of parasitic threads in a large collection of benchmarks, including a full fledged web-server. Evaluation is performed on two garbage collection schemes — a stop-the-world garbage collector and a split-heap parallel garbage collector — and shows that parasitic threads improve the performance in both cases.

1. Introduction

Asynchrony is a well known technique for extracting additional throughput from programs by allowing *conceptually* separate computations to overlap their executions. Typically programmers leverage asynchrony to *mask* high latency computations or to achieve greater *parallelism*. Asynchrony can either be realized through specialized primitives (i.e. asynchronous sends in Erlang or MPI), allowing the primitive action to execute asynchronously with the computation that initiated the action, or in a more general form through threads.

Threads, in all their various forms (native [5], asyncs [6], sparks [18], and others [9, 24]), 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 *conceptually* asynchrony could be leveraged, programmers must often reason about whether the runtime cost of creating a thread, thread scheduling, and any synchronization the thread may perform outweigh the benefit of performing the desired computation asynchronously. It is precisely for this reason that specialized primitives are typically the *de facto* standard for most programming languages.



(a) Synchronous (b) Asynchronous

Figure 1: Implementation of chooseAll

However, we observe that in a language like Concurrent ML (CML) [26], with first class events, as well as in other functional language runtimes [7], asynchrony achieved through threads provides a powerful yet simple mechanism for constructing runtime abstractions. To illustrate, consider the steps involved in the construction of a language abstraction, in CML terms a combinator, `chooseAll`, which conceptually synchronizes between multiple threads, gathering a value from each.

```
val chooseAll : 'a event list -> 'a list event
```

Given a list of events — computations which contain a synchronization mechanism — `chooseAll` returns an abstraction that collects *all* of the results of the individual events, when invoked. Notice that the events from which `chooseAll` was constructed can be executed in any order. An abstraction such as `chooseAll` is typically used to build collective communication operations like scatter, gather, reduce, etc. Consider implementing `chooseAll` by synchronously mapping over the list of events and collecting each of their results in order, one by one. Figure 1a illustrates the behavior of this synchronous encoding. The dotted box represents the execution of `chooseAll`. The thread T_0, executing `chooseAll`, collects the result of the events one by one, and finally resumes execution with the aggregate.

Though this simple construction captures the desired behavior, it obviously suffers from performance bottlenecks. Since each of the events in the list can embody a complex computation, they can take arbitrarily long to execute. For example, if the first event in the list takes the longest time to complete, the rest of the events will stall until its completion. This halts progress on threads that might be waiting for one of the events in the list to complete.

In order to allow for greater concurrency, we might consider spawning lightweight threads to execute each of the events in the list and then aggregate the results of the various computations. The execution of such an abstraction is presented in Figure 1b. Here, the thread T_0 executing `chooseAll` is suspended until all the results are gathered. New lightweight threads T_1 to T_n are created to execute corresponding events. Each lightweight thread will place the result of the event it executes in its corresponding slot in the result list. After all of the threads have completed execution, T_0 is resumed with the result of `chooseAll`.

Even though semantically equivalent to the synchronous implementation, this solution will actually perform *worse* in many scenarios than the synchronous solution due to the overheads of scheduling, synchronization and thread creation costs, even if the threads are implemented as lightweight entities or one/many-shot continuations. The most obvious case where performance degradation occurs, is when each of the events encodes a very *short* computation. In such cases, it takes longer to allocate and schedule the thread than it takes to complete the event. However, there is more at play here, and even when some events are long running, the asynchronous solution incurs additional overheads. We can loosely categorize these overheads 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 in short period of time increases allocation and might subsequently trigger a collection. This problem becomes worse in a parallel setting, when multiple mutators might be allocating in parallel.

In this paper, we explore a novel threading mechanism, called parasitic threads, that reduces costs associated with lightweight threading structures. Parasitic threads allow the expression of *arbitrary* asynchronous computation, while mitigating typical threading costs. Parasitic threads are especially useful to model asynchronous computations that are usually short-lived, but can also be arbitrarily long. Consider once again our `chooseAll` primitive. It takes an arbitrary list of events, so at any given invokation of `chooseAll` we do not know *apriori* if a particular event is short or long lived.

An implementation of `chooseAll` using parasitic threads is described in Section 2.3.2. Such an implementation, abstractly, delays creating threads *unless* a parasite performs a blocking action. In practice, this alleviates scheduling and GC costs. Even when parasites block and resume, they are *reified* into an entity that does not impose synchronization and GC overheads. If the computation they encapsulate is long running, they can be *inflated* to a lightweight thread anytime during execution.

Importantly, in the implementation of `chooseAll` with parasitic threads, if all of the events in the list are available before the execution of `chooseAll`, none of the parasites created for executing individual events will block. In addition to an implementation of `chooseAll`, we show how parasitic threads can be leveraged at the library level to implement a library of specialized asynchronous primitives. We believe parasitic threads are a useful runtime technique to accelerate the performance of functional runtime systems.

This paper makes the following contributions:

- The design and implementation of parasitic threads, a novel threading mechanism that allows for the expression of a logical thread of control using raw stack frames. Parasitic threads can easily be inflated into lightweight threads if necessary.
- A formal semantics governing the behavior of parasitic threads.
- A case study leveraging the expressivity of parasitic threads to implement a collection of asynchronous primitives, and illustrating the performance benefits of parasitic threads through a collection of micro-benchmarks.
- A detailed performance analysis of runtime costs of parasitic threads over a large array of benchmarks, including a full-fledged web server. The performance analysis is performed on two distinct GC schemes to illustrate that the parasitic threads are beneficial irrespective of the underlying GC.

The rest of the paper is organized as follows: in Section 2, we present our base runtime system and the design of parasitic threads. In Section 3, we provide a formal characterization of parasitic threads and show their semantic equivalence to classic threads. We discuss salient implementation details in Section 4. We present a case study illustrating the utility of parasitic threads in the construction of asynchronous primitives 5. We present our experiment details and results in Section 6. Related work and concluding remarks are given in Section 7 and Section 8 respectively.

2. System Design

In this section, we describe the salient details for the design of parasitic threads and relevant characteristic of our runtime system. We envision parasitic threads to be used as a fundamental building block for constructing asynchronous primitives. With this in mind, we introduce an API for programming with parasitic threads, and show how to construct an efficient `chooseAll` primitive introduced earlier.

2.1 Base System

Our runtime system is built on top of lightweight (green) threads, synchronous communication, and leverages a GC. Our threading system multiplexes many lightweight threads on top of a few kernel threads. We leverage one kernel thread for each processor or core for a given system. The kernel threads are also pined to the processor. Hence, the runtime views a kernel thread as a virtual processor. The number of kernel threads is determined statically and is specified by the user. Kernel threads are not created during program execution, instead, all spawn primitives create lightweight threads.

Threads communicate in our system through synchronous message passing primitives based on PCML [25], a parallel definition of CML. Threads may perform sends or receives to pass data between one another. Such primitives block until a matching communication partner is present. Our system supports two different GC schemes; a stop-the-world collector with parallel allocation and a split-heap parallel GC scheme. Necessary details about both GC designs, with respect to parasites and their implementation are given in Section 4.

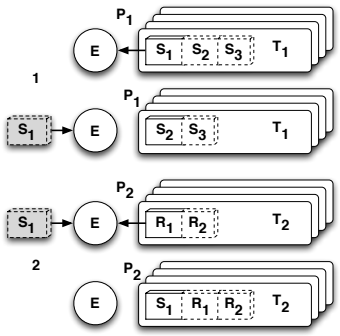


Figure 2: Blocking and unblocking of parasitic threads.

2.2 Parasitic Threads

Our runtime supports two kinds of threads: hosts and parasites. Host threads map directly to lightweight threads in the runtime. Parasitic threads [1] can encapsulate arbitrary computation, just like host threads. However, unlike a regular thread, a parasitic thread executes using the execution context of the host which creates the parasite. 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 becomes *reified* when it performs a blocking action (e.g., a synchronous communication operation, or I/O). Reified parasites are represented as stack objects on the heap. Reified parasites can resume execution once the conditions that 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.

Our system supports synchronous message passing over channels as well as user-defined blocking events. During synchronous communication, the parasite which initiates the synchronous communication blocks and is reified (either sending or receiving on a channel), if a matching communication is not already available. It is subsequently unblocked by the thread that terminates the protocol by performing the opposite communication action (i.e. a matching send or receive). Similarly, a parasite may block on an event (such as I/O). This parasite is available to execute once the I/O event is triggered. Once the conditions that prevent continued execution of the parasite becomes resolved, the parasite enters a suspended state, and can be resumed on the host.

In the following figures, 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 which 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 thread is initially created it contains one such parasitic computation, namely the expression it was given to evaluate when it was spawned.

Figure 2 shows the steps involved in a parasitic communication, or blocking event. 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 which hosted S_1 continues execution by switching to the next parasite S_2. S_1 becomes runnable

[1] An initial characterization of parasitic threads was presented in unpublished form in [27].

```
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
```

Figure 3: Parasitic thread management API.

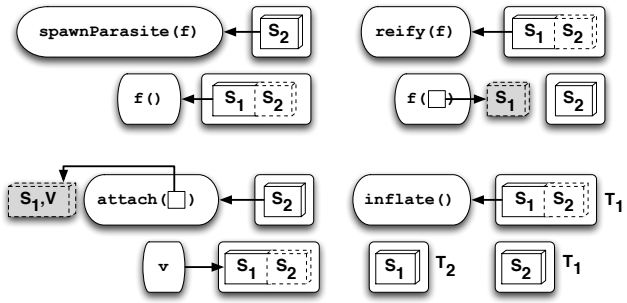


Figure 4: Behavior of parasite API.

when it is unblocked. Part 2 of the figure shows the parasite R_1 on the thread T_2 invoking an unblocking action. This unblocks S_1 and schedules it on top of R_1. Thus, the parasitic threads implicitly migrate to the point of synchronization. This communication driven migration benefits further interaction between S_1 and R_1, if any, by keeping them thread local.

2.3 An API for Parasitic Threads

Parasites are exposed as a library to be utilized to build higher level asynchronous abstractions. Figure 3 shows an interesting subset of the parasite API. The primitives exposed by the parasite API are similar to continuations, but differ from continuations in that the scheduling and migration of parasitic threads is implicitly defined by the control flow abstractions aka the parasite API. Figure 4 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)`. This applies `f` on the reference to the parasite, similar to a call-cc invocation. The function `f` can be used to block the parasite on an event. Once the parasite has been reified, the control immediately switches to the next parasite on the host stack. Reified parasites are represented as stack objects on the heap, reachable from the global resource the parasite is blocked on.

Parasites are also value carrying and can be assigned a value with the help of `prepare`. Such parasites are said to have been *prepared*. When a parasite resumes execution, it is provided with this prepared value. 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, the control returns to the parasite residing below as if a non-tail call has returned.

2.3.1 Inflation

Although parasites are envisioned to be used as short-lived threads, they can encode arbitrarily long computations, as the programming model places no restriction on the closure spawned as a parasite. Executing multiple long running computations on the same stack is unfavorable as it serializes the executions. Parasitic threads provide two options to upgrade the parasite to a full fledged thread:

- Synchronously by invoking `inflate` primitive in the parasite that needs to be inflated.
- Asynchronously inflate a reified parasite by spawning a new host that attaches the reified parasite

If there are potential fast and slow branches in a parasitic computation, on hitting the slow branch, the parasite can be upgraded to a host thread using the inflate primitive. The newly created host is placed on a different processor, if available, for parallel execution. Programmers also have the option of installing timer interrupts to inflate parasites on an interrupt. We observe that the timer interrupt scheme works well in practice, although some unfortunate parasites might be inflated. However, the exact requirements of different programming models built on top of parasites might be different. Hence, we leave it to the programmer to choose the appropriate inflation policy.

2.3.2 Implementing chooseAll

Figure 5 shows how the `chooseAll` primitive introduced in Section 1 is implemented using the parasitic thread API. In order to represent the new event combinator, CML event datatype is extended with one more constructor `CHOOSE_ALL`. From the perspective of the event combinators such as wrap and guard, chooseAll is treated similar to choose. The interesting behavior of chooseAll is during synchronization. In the following code snippet, we define a function `syncOnChooseAll`, which will be invoked to synchronize on a chooseAll event. We assume the existence of an atomic fetch and add instruction — `fetchAndAdd (r, count)` — which atomically increments the reference `r` by `count`, and returns the original value of `r`.

The basic idea of the implementation is to use parasitic threads for synchronizing on each of the constituent events. The reference `c` is used as a condition variable to indicate the completion of synchronization of all events in the list. The array `a` is used as a temporary store to collect the results.

The current parasite is first reified at line 7 and its reference is captured in p. The task of spawning the parasites for synchronization on each event is performed in the `loop` function. After spawning the parasites, at the end of the loop, the current parasite switches to the next parasite on its stack at line 21.

Each of the spawned parasites synchronize on its corresponding event at line 24 and invoke the finish function defined at line 9. Here, the result of synchronization is stored in the array `a`. The parasite also atomically decrements the counter `c`. If it is the last parasite to finish synchronization, it prepares the suspended parasite p with the result and invokes attach. This returns control to the suspended parasite with the result of synchronizing on each of the events. This implementation is safe for execution in a parallel context. Inflation on timer interrupt is turned on so that a long running parasite is automatically inflated to a host and runs in parallel.

```
1  fun syncOnChooseAll(l : 'a event list) =
2  let
3    val c = ref(List.length l)
4    val a = Array.tabulate
5              (List.length l,fn _ => NONE)
6  in
7    reify(fn p =>
8      let
9        fun finish(v,i) =
10         (Array.update(a,i,SOME v);
11          if (fetchAndAdd(c,~1) = 1) then
12            let
13              val res =
14                Array.foldr
15                  (fn(e,l) => valOf(e)::l) [] a
16              val rdy_p = prepare(p,res)
17            in
18              attach(rdy_p)
19            end
20          else ())
21       fun loop ([],_) = ()
22         | loop (e::el,i) =
23             (spawnParasite
24               (fn() => finish(sync e,i))
25             ; loop(el,i+1))
26     in
27       loop (l,0)
28     end)
29 end
```

Figure 5: Implementation of chooseAll using parasitic thread API

3. Formal Semantics

We define our system formally through the use of a formal operational semantics and model host threads in terms of sets of stacks and parasites as stacks. Transitions in our formalism are defined through stack based operations. Our semantics is defined in terms of a core call-by-value functional language with threading and communication primitives. New threads of control, or host threads, are explicitly created through a `spawn` primitive. To model parasitic thread creation we extend this simple language with a primitive `spawnParasite` and `inflate` which inflates a parasite into a host. Therefore, computation in our system is 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.

We formalize the behavior of parasitic threads in terms of an operational semantics expressed using a CEK machine [19]. A typical 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 of the expression. The continuation is modeled as a *stack* of frames.

3.0.3 Language

In the following, we write $\bar{k}$ to denote a set of zero or more elements and $\emptyset$ as the empty set. We write $x : l$ to mean the concatenation of x to a sequence l where . denotes an empty sequence. Evaluation rules are applied up to commutativity of parallel composition ($\|$). Relevant domains and meta-variables used in the semantics are shown in Figure 6.

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 eight unique types of frames: return frames,

argument frames, function frames, receive frames, send frames, send value frames, and receive and send blocked 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 7 `Return Transitions`). The receive and send blocked frames signify that a thread 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 parasitic thread can resume its execution. Argument and function frames enforce left-to-right order of evaluation. order. Similarly, the send and send value frames define the left to right evaluation of the `send` primitive.

Our semantics is defined with respect to a global mapping ($\mathcal{T}$) from thread identifiers (`t`) to thread states (`s`). A thread is a pair composed of a thread identifier and a thread state. A thread state (`s`) is a CEK machine state extended with support for parasitic threads. Therefore, a 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 set of parasitic computations ($\bar{\mathtt{k}}$). A *return state* is simply a collection of parasitic computations ($\bar{\mathtt{k}}$). The *halt state* is reached when all parasitic threads in a given thread have completed. A thread, therefore, is composed of a collection of parasitic threads executing within its stack space. When a thread transitions to a control state, one of the thread's parasites is chosen to be evaluated. A thread switches evaluation between its various parasitic threads non-deterministically when it transitions to a return state.

3.1 CEK Machine Semantics

The rules given in Figure 7 and Figure 8 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 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 which require global coordination, such as the creation of a new channel or thread, or a communication action.

There are six rules which define global transitions given in Figure 8. Rule (`Local Evaluation`) 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 thread and parasite 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 (`Spawn`) rule governs the creation of a new 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`).

Notably there is no rule which deals with communication across a shared channel (`c`) by two threads. Instead we migrate either the sender of the receiver to one of the threads. The two rules (`Migrate Left`) and (`Migrate Right`) capture the behavior of the API calls `reify` and `attach` defined in Section 2.3. Notice that both rules effectively do a reification of a parasite and then attach that parasite to the other thread. Inflation is similar to the rule for spawning a thread, except the stack associated with the parasite which calls inflate is removed from its current host thread and added to a newly created host thread.

There are seven rules which define local control transitions. Because the definition of these rules are standard, we omit their explanation here, with the exception of the last rule (`Parasite`). This 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.

There are eight rules which define local return transitions. These rules, like local control transitions, are mostly standard. We comment on the three rules that involve thread and parasite management. Rule (`Halt Thread`) 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 (`Parasite Halt`) rule. The return value of a thread is thus defined as the last value produced by its last parasite. The lifetime of a thread is bounded by the parasites which inhabit it. Rule (`Local Communication`) is similar to the global communication rule, but is defined as a local transition when both communicating parties reside in the same thread. The transition pops off both blocked frames for the communicating parasitic threads. It also pushes new return frames, the value being sent on to the receiver's stack and the unit value on top of the sender's stack.

4. System Implementation

We have implemented our system in Multi-MLton, a parallel extension of MLton [21], a whole-program, optimizing compiler for Standard ML (SML) [20]. MLton can compile ML programs to both native code as well as C; the results described in this paper are based on code compiled to C and then passed to gcc version 4.4.0. Multi-MLton extends MLton with multi-core support, library primitives for efficient lightweight thread creation and management, as well as ACML [31], an optimized composable asynchronous message passing extension to SML. In this section, we present insights to the implementation of parasitic threads and their implementation within MLton.

4.1 Thread Scheduler

Our implementation is targeted at high performance SMP platforms. The implementation supports m host threads running on top of n processors, where $m \geq n$. Each processor runs a single Posix thread that has been pinned to the corresponding processor. Hence, kernel threads are viewed as virtual processors by the runtime system. Each processor hosts a local scheduler to schedule the host threads. Host threads are spawned in a round-robin fashion on the processors to facilitate load balancing, and are pinned to their respective schedulers. Host threads are preemptively scheduled. Parasitic threads are self scheduled and implicitly migrate to the host stack it communicates with. Bringing the sender and the receiver to the same host stack benefits the cache behavior.

4.2 Host Threads

Each host thread has a contiguous stack, allocated on the MLton heap. The host stacks have a reserved space beyond the top of the stack, such that frames can be pushed by bumping the stack top pointer. This reserved space is key to quickly resuming suspended parasites. Our experimental results show that parasitic stacks are almost always smaller than the free space in the host stack and can thus be copied to the host stack without need to grow the

$$
\begin{array}{rcl}
\mathtt{e} \in \mathit{Exp} & ::= & \mathtt{x} \mid \mathtt{v} \mid \mathtt{e(e)} \mid \mathtt{spawn(e)} \mid \mathtt{parasiteSpawn(e)} \\
& \mid & \mathtt{Chan()} \mid \mathtt{send(e,e)} \mid \mathtt{recv(e)} \mid \mathtt{inflate(e)} \\
\mathtt{v} \in \mathit{Val} & ::= & \mathtt{unit} \mid (\lambda \mathtt{x.e,r}) \mid \mathtt{c} \mid \kappa
\end{array}
$$

$$
\begin{array}{rcrcl}
\kappa & \in & \mathit{Constant} & & \\
\mathtt{c} & \in & \mathit{Channel} & & \\
\mathtt{x} & \in & \mathit{Var} & & \\
\mathtt{t} & \in & \mathit{ThreadID} & & \\
\mathtt{r} & \in & \mathit{Env} & = & \mathit{Var} \to \mathit{Value} \\
\mathtt{k} & \in & \mathit{Cont} & = & \mathit{Frame}* \\
\mathtt{v} & \in & \mathit{Value} & = & \mathit{Unit} + \mathit{Closure} + \mathit{Channel} + \mathit{Constant} \\
(\lambda \mathtt{x.e,r}) & \in & \mathit{Closure} & = & \mathit{LambdaExp} \times \mathit{Env}
\end{array}
\qquad
\begin{array}{rcrcl}
\mathit{ret}\lceil \mathtt{v} \rceil & \in & \mathit{RetFrame} & = & \mathit{Value} \\
\mathit{arg}\lceil \mathtt{e,r} \rceil & \in & \mathit{ArgFrame} & = & \mathit{Exp} \times \mathit{Env} \\
\mathit{fun}\lceil \mathtt{v} \rceil & \in & \mathit{FunFrame} & = & \mathit{Value} \\
\mathit{recv}\lceil \rceil & \in & \mathit{RecvFrame} & = & \mathit{Empty} \\
\mathit{sval}\lceil \mathtt{e,r} \rceil & \in & \mathit{SValFrame} & = & \mathit{Exp} \times \mathit{Env} \\
\mathit{send}\lceil \mathtt{v} \rceil & \in & \mathit{SendFrame} & = & \mathit{Value} \\
\mathit{rblock}\lceil \mathtt{v} \rceil & \in & \mathit{RBlockFrame} & = & \mathit{Value} \\
\mathit{sblock}\lceil \mathtt{v}_1, \mathtt{v}_2 \rceil & \in & \mathit{SBlockFrame} & = & \mathit{Channel} \times \mathit{Value}
\end{array}
$$

$$
\begin{array}{rcrcl}
\mathcal{T} & \in & \mathit{GlobalMap} & = & \mathit{ThreadID} \to \mathit{ThreadState} \\
\mathtt{s} & \in & \mathit{ThreadState} & = & \mathit{ControlState} + \mathit{ReturnState} + \mathit{HaltState} \\
\langle \mathtt{e,r,k,\bar{k}} \rangle & \in & \mathit{ControlState} & = & \mathit{Exp} \times \mathit{Env} \times \mathit{Cont} \times \mathit{Cont}* \\
\langle \mathtt{\bar{k}} \rangle & \in & \mathit{ReturnState} & = & \mathit{Cont}* \\
\mathtt{halt(v)} & \in & \mathit{HaltState} & = & \mathit{Value}
\end{array}
$$

Figure 6: Domains for the CEK machines extended with threads and parasites.

Control Transitions

$$
\begin{array}{lrcl}
\text{(Constant)} & \langle \kappa, \mathtt{r,k,\bar{k}} \rangle & \longrightarrow & \langle \mathit{ret}\lceil \kappa \rceil : \mathtt{k} \| \mathtt{\bar{k}} \rangle \\
\text{(Variable)} & \langle \mathtt{x,r,k,\bar{k}} \rangle & \longrightarrow & \langle \mathit{ret}\lceil \mathtt{r(x)} \rceil : \mathtt{k} \| \mathtt{\bar{k}} \rangle \\
\text{(Closure)} & \langle \lambda \mathtt{x.e,r,k,\bar{k}} \rangle & \longrightarrow & \langle \mathit{ret}\lceil \lambda \mathtt{x.e,r} \rceil : \mathtt{k} \| \mathtt{\bar{k}} \rangle \\
\text{(Application)} & \langle (\mathtt{e_1 e_2}), \mathtt{r,k,\bar{k}} \rangle & \longrightarrow & \langle \mathtt{e_1, r}, \mathit{arg}\lceil \mathtt{e_2, r} \rceil : \mathtt{k}, \mathtt{\bar{k}} \rangle \\
\text{(Send)} & \langle \mathtt{send(e_1, e_2), r, k, \bar{k}} \rangle & \longrightarrow & \langle \mathtt{e_1, r}, \mathit{sval}\lceil \mathtt{e_2, r} \rceil : \mathtt{k}, \mathtt{\bar{k}} \rangle \\
\text{(Receive)} & \langle \mathtt{recv(e), r, k, \bar{k}} \rangle & \longrightarrow & \langle \mathtt{e, r}, \mathit{recv}\lceil \rceil : \mathtt{k}, \mathtt{\bar{k}} \rangle \\
\text{(SpawnParasite)} & \langle \mathtt{parasiteSpawn(e), r, k, \bar{k}} \rangle & \longrightarrow & \langle \mathtt{e, r}, ., (\mathit{ret}\lceil \mathtt{unit} \rceil : \mathtt{k}) \| \mathtt{\bar{k}} \rangle
\end{array}
$$

Return Transitions

$$
\begin{array}{lrcl}
\text{(ThreadHalt)} & \langle \mathit{ret}\lceil \mathtt{v} \rceil : . \| \emptyset \rangle & \longrightarrow & \mathtt{halt(v)} \\
\text{(ParasiteHalt)} & \langle \mathit{ret}\lceil \mathtt{v} \rceil : . \| \mathtt{\bar{k}} \rangle & \longrightarrow & \langle \mathtt{\bar{k}} \rangle \\
\text{(Argument)} & \langle \mathit{ret}\lceil \mathtt{v} \rceil : \mathit{arg}\lceil \mathtt{e,r} \rceil : \mathtt{k} \| \mathtt{\bar{k}} \rangle & \longrightarrow & \langle \mathtt{e, r}, \mathit{fun}\lceil \mathtt{v} \rceil : \mathtt{k}, \mathtt{\bar{k}} \rangle \\
\text{(Function)} & \langle \mathit{ret}\lceil \mathtt{v} \rceil : \mathit{fun}\lceil \lambda \mathtt{x.e,r} \rceil : \mathtt{k} \| \mathtt{\bar{k}} \rangle & \longrightarrow & \langle \mathtt{e, r}[\mathtt{x} \mapsto \mathtt{v}], \mathtt{k}, \mathtt{\bar{k}} \rangle \\
\text{(SendValue)} & \langle \mathit{ret}\lceil \mathtt{c} \rceil : \mathit{sval}\lceil \mathtt{e,r} \rceil : \mathtt{k} \| \mathtt{\bar{k}} \rangle & \longrightarrow & \langle \mathtt{e, r}, \mathit{send}\lceil \mathtt{c} \rceil : \mathtt{k}, \mathtt{\bar{k}} \rangle \\
\text{(SendBlock)} & \langle \mathit{ret}\lceil \mathtt{v} \rceil : \mathit{send}\lceil \mathtt{c} \rceil : \mathtt{k} \| \mathtt{\bar{k}} \rangle & \longrightarrow & \langle \mathit{sblock}\lceil \mathtt{c, v} \rceil : \mathtt{k} \| \mathtt{\bar{k}} \rangle \\
\text{(ReceiveBlock)} & \langle \mathit{ret}\lceil \mathtt{c} \rceil : \mathit{recv}\lceil \rceil : \mathtt{k} \| \mathtt{\bar{k}} \rangle & \longrightarrow & \langle \mathit{rblock}\lceil \mathtt{c} \rceil : \mathtt{k} \| \mathtt{\bar{k}} \rangle \\
\text{(LocalCommunication)} & \langle (\mathit{rblock}\lceil \mathtt{c} \rceil : \mathtt{k}_1) \| (\mathit{sblock}\lceil \mathtt{c, v} \rceil : \mathtt{k}_2) \| \mathtt{\bar{k}} \rangle & \longrightarrow & \langle (\mathit{ret}\lceil \mathtt{v} \rceil : \mathtt{k}_1) \| (\mathit{ret}\lceil \mathtt{unit} \rceil : \mathtt{k}_2) \| \mathtt{\bar{k}} \rangle
\end{array}
$$

Figure 7: Local evaluation defining both control and return transitions.

stack. When the host stack runs out of reserved space, it grows by allocating a new, larger stack object. Host stacks are also re-sized as needed during garbage collection.

4.2.1 Exception Safety in Parasites

Exceptions in MLton are intertwined with the structure of the stack. Exceptions are also stack local and are not allowed to escape 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. We will describe how parasites were made safe for exceptions.

In MLton, every host thread has a pointer to the top handler frame in the currently executing host stack. Every handler frame has a link to the next handler frame, 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 the handler frames are popped, the thread's top-handler pointer is updated to point to the new top handler. How do we ensure that this structure is maintained for parasitic operations?

Just like host threads, exceptions are not allowed to escape parasitic threads and are handled by the default handler. When a parasite is reified and moved out of the stack, the offset of the top handler from the bottom of the parasitic stack is stored in the parasitic meta-data. The handlers installed by the parasite are also captured as a part of the parasitic stack. Control returns to the parasite residing below the reified parasite as a non-tail-call return. Importantly, the default handler of the reified parasite is popped. This resets the thread's top-handler pointer to the next valid handler in the host stack.

When the reified parasite eventually resumes execution on another host stack, the handler offset stored in the parasite meta-data is used to set the thread's top-handler pointer to the top most handler in the attached parasite. This is a constant time operation.

Global Transitions

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

$$\text{(Channel)}\quad \frac{\mathtt{c}\ \textit{fresh}}{\langle \mathcal{T}[\mathtt{t} \mapsto \langle \mathtt{Chan}(), \mathtt{r}, \mathtt{k}, \bar{\mathtt{k}}\rangle]\rangle \longrightarrow \langle \mathcal{T}[\mathtt{t} \mapsto \langle \textit{ret}\lceil \mathtt{c} \rceil : \mathtt{k} \| \bar{\mathtt{k}}\rangle]\rangle}$$

$$\text{(Spawn)}\quad \frac{\mathtt{t}'\ \textit{fresh}}{\langle \mathcal{T}[\mathtt{t} \mapsto \langle \mathtt{spawn}(\mathtt{e}), \mathtt{r}, \mathtt{k}, \bar{\mathtt{k}}\rangle]\rangle \longrightarrow \langle \mathcal{T}[\mathtt{t} \mapsto \langle \textit{ret}\lceil \mathtt{unit} \rceil : \mathtt{k} \| \bar{\mathtt{k}}\rangle, \mathtt{t}' \mapsto \langle \mathtt{e}, \mathtt{r}, ., \emptyset\rangle]\rangle}$$

$$\text{(MigrateLeft)}\quad \frac{\begin{array}{c}\mathtt{s}_1 = \langle \textit{rblock}\lceil \mathtt{c} \rceil : \mathtt{k}_1 \| \bar{\mathtt{k}_1}\rangle, \mathtt{s}_2 = \langle \textit{sblock}\lceil \mathtt{c}, \mathtt{v} \rceil : \mathtt{k}_2 \| \bar{\mathtt{k}_2}\rangle \\ \mathtt{s}_1' = \langle \bar{\mathtt{k}_1}\rangle, \mathtt{s}_2' = \langle \textit{sblock}\lceil \mathtt{c}, \mathtt{v} \rceil : \mathtt{k}_2 \| \textit{rblock}\lceil \mathtt{c} \rceil : \mathtt{k}_1 \| \bar{\mathtt{k}_2}\rangle\end{array}}{\langle \mathcal{T}[\mathtt{t}_1 \mapsto \mathtt{s}_1\mathtt{t}_2 \mapsto \mathtt{s}_2]\rangle \longrightarrow \langle \mathcal{T}[\mathtt{t}_1 \mapsto \mathtt{s}_1'\mathtt{t}_2 \mapsto \mathtt{s}_2']\rangle}$$

$$\text{(MigrateRight)}\quad \frac{\begin{array}{c}\mathtt{s}_1 = \langle \textit{rblock}\lceil \mathtt{c} \rceil : \mathtt{k}_1 \| \bar{\mathtt{k}_1}\rangle, \mathtt{s}_2 = \langle \textit{sblock}\lceil \mathtt{c}, \mathtt{v} \rceil : \mathtt{k}_2 \| \bar{\mathtt{k}_2}\rangle \\ \mathtt{s}_1' = \langle \textit{rblock}\lceil \mathtt{c} \rceil : \mathtt{k}_1 \| \textit{sblock}\lceil \mathtt{c}, \mathtt{v} \rceil : \mathtt{k}_2 \| \bar{\mathtt{k}_1}\rangle, \mathtt{s}_2' = \langle \bar{\mathtt{k}_2}\rangle\end{array}}{\langle \mathcal{T}[\mathtt{t}_1 \mapsto \mathtt{s}_1\mathtt{t}_2 \mapsto \mathtt{s}_2]\rangle \longrightarrow \langle \mathcal{T}[\mathtt{t}_1 \mapsto \mathtt{s}_1'\mathtt{t}_2 \mapsto \mathtt{s}_2']\rangle}$$

$$\text{(Inflate)}\quad \frac{\mathtt{t}'\ \textit{fresh}}{\langle \mathcal{T}[\mathtt{t} \mapsto \langle \mathtt{inflate}(\mathtt{e}), \mathtt{r}, \mathtt{k}, \bar{\mathtt{k}}\rangle]\rangle \longrightarrow \langle \mathcal{T}[\mathtt{t} \mapsto \langle \bar{\mathtt{k}}\rangle, \mathtt{t}' \mapsto \langle \mathtt{e}, \mathtt{r}, \mathtt{k}, \emptyset\rangle]\rangle}$$

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

When parasites are inflated, the top-handler pointer in the newly created host is updated to point to the top-handler in the parasite.

4.3 Interaction with Garbage Collector

Just like host threads in the language runtime, the garbage collector must be aware of parasitic threads and the objects they reference. Parasites either exist as a part of a host stack or reified as a separate heap object. If the parasites are a part of the host, the garbage collector treats the parasites as regular frames while tracing. Reified parasites are represented in the GC as stack objects, and from a GC stand point are indistinguishable from host stacks. Hence, the garbage collection algorithms trace reified parasites as if they were host stacks.

4.3.1 Split-Heap Parallel GC

We provide a short description of the split-heap parallel GC in order to describe interesting parasitic interactions. The split-heap parallel GC is inspired from Doligez's GC for a multi-threaded implementation of ML [8]. Each processor has a local heap, that can have no pointers from other local heaps. This invariant allows collection of local heaps in parallel with other processors. Since ML tends to allocate lot of temporaries, the local heaps act as a *private nursery*, eliminating the need to perform synchronization for local collections. In order to share data between cores, a common shared heap is also present to which transitive closures of objects are copied on demand. Our experimental results show that this implementation improves the performance of our runtime system.

Since parasites migrate to the point of synchronization, we see parasitic closures copied between the local heaps. We observe that parasitic stacks and closures are small enough that inter-heap parasitic communication does not affect the scalability of the system. Parasitic migration also ensures that subsequent communication actions, if any, are local to the processor.

Since the only form of thread migration in our system is through parasitic communication, host stacks are never moved to the shared heap. Due to this invariant, host stacks in the local heap might be reachable from shared heap references, and hence by other processors. Even though the host stacks in a local heap might be reachable, the contents of the stack are never read by other processors.

Pinning host stacks to the local heaps, keeps most of our allocations local. This improves performance since the local garbage can be collected concurrently with respect to operations on the other processors. Stop-the-world collection is only performed in the shared heap. Experimental results show that the split-heap parallel GC in general performs better than the stop-the-world collector.

5. Case Study : Building Asynchronous Primitives

In this section, we will illustrate the utility of parasites in the construction of efficient asynchronous primitives. We also present micro-benchmark measurements for common asynchronous programming patterns, to illustrate the benefits of the parasitic thread design. This section is a summary of the lessons learnt in the implementation of Asynchronous CML (ACML) [31] utilizing parasitic threads. Asynchronous CML (ACML) extends CML with composable asynchronous events and is the core of the Multi-MLton programing model [12]. ACML[2] leverages parasitic threads pervasively, though we only touch on a few of patterns that are applicable to asynchronous primitives in general.

5.1 Unified Scheduler

With two underlying threading systems (hosts and parasites), we need to abstract away the differences in the thread management API. In particular, host threads in Multi-MLton are built on top of one-shot continuations, which is completely separate from the scheduling policy, while the parasitic thread API combines control flow abstractions with parasitic scheduling and migration.

The goal of abstracting the scheduler is to expose both kinds of thread creation mechanisms while unifying the scheduler abstractions, so as to ease the development asynchronous communication primitive. For example, with such a scheduler, the programmer

[2] To download please visit:
http://sss.cs.purdue.edu/projects/multiMLton/mML/Download.html

does not have to separately handle the situations where two parasites communicate and when two hosts communicate, and can treat them uniformly. With this in mind, we have implemented a scheduler that exposes both spawnHost and spawnParasite, and unifies the other scheduler idioms such as ready, switch, etc,. Being able to simply define such a scheduler illustrates that parasitic threads can easily replace host threads, and vice versa. We believe such unified scheduler will be utilized by any library that leverages both host and parasitic threads.

5.2 Lazy thread creation

The most common use case of parasitic threads is to model asynchronous short-lived computation, that might potentially block on a global resource, to be resumed at a later time. For example, ACML defines an asynchronous version of CML sendEvt, the `aSendEvt`, which when synchronized on performs the send asynchronously. Abstractly, a thread is created to perform this potentially blocking action, such that even if the newly created thread blocks, the original thread can continue execution.

Notice that if the asynchronous computation does not block, there is no need to create a separate thread. 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 subactions, all of which might not expose a polling mechanism.

Parasites provide a mechanism to lazily create a thread if needed. We distinguish ourselves from previous work on lazy thread creation in Section 7. For a short-lived computation that does not block, the overhead of parasites is the cost of a non-tail call and overheads of mechanisms in place for parasitic reification and inflation. Whereas, 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. Our empirical measurements show that for a non-blocking, short-lived computation, in the time that takes to create 1 host thread, we can create 46 parasites.

Configuration	Norm. runtime
sendEvt	1
aSendEvt on Host	4.96
aSendEvt on Parasite	1.05

Table 1: Comparison of cost of aSendEvt implemented over hosts and parasites

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 worse case overhead of only 5% over their synchronous counterparts. Not surprisingly, in this micro benchmark, there was no profitable parallelism to extract.

5.3 Inflation

Just like asserting the blocking behavior of asynchronous computations, it is difficult to estimate the running time of an arbitrary asynchronous computation. A long lived asynchronous computation may be run on a different core, if available, for *parallelism*.

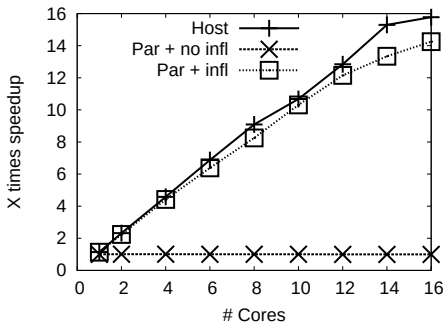


Figure 9: Speedup of parasitic threads with and without inflation compared to host threads

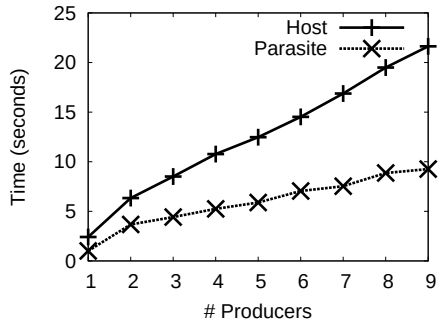


Figure 10: Runtime performance of chooseAll primitive implemented with host and parasitic threads.

Parasites are automatically inflated if they are long lived. As described earlier 2.3.1, parasites can inflate either synchronously or asynchronously on a timer interrupt. For a good runtime behavior, we also assign a *penalty-count* to the underlying host thread, which spawned the long running parasite. A host with penalty-count greater than zero, will spawn the next parasite as a host, and decrement penalty-count. The idea is that such a host will spawn more long running parasites. Instead of spawning such computations as parasites and then inflating them, we eagerly spawn them as hosts. Since parasites and hosts are semantically equivalent, and are treated uniformly by the scheduler, they can be interchanged. We have found that a penalty-count of 10 works well in practice over a wide range of benchmarks.

To provide a better intuition, we implemented a micro-benchmark that spawns a large number of long running computations. Each threads computes a tight-loop. We compare the runtime performance of spawning the computation as a host thread and parasite with and without inflation. Figure 9 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.

5.4 Choice

Communication primitives that abstractly coordinate over multiple channels express a common communication pattern. Examples range from CML's choice primitive, MPI's group communication primitives such as scatter and gather, to our `chooseAll`. In section. 2.3.2, we illustrated how such a primitive, chooseAll, could possibly be modified for increased asynchrony with the help of parasitic threads.

To measure the benefit of using parasites instead of host threads for the implicit threads of chooseAll, we implemented a multiple

producer, single consumer benchmark, where the consumer uses chooseAll to receive the value from the producers. Each producer uses a private channel to send 100,000 messages to the consumer. We varied the number of producers and measured the runtime performance. Results are shown in Figure 10. We observe that parasitic threads perform 2X faster than the host thread implementation on 2 cores.

5.5 Putting it all together

Although we presented only micro benchmarks thus far in this section, we measure the real impact of utilizing parasitic threads, with the help of a full-fledged web-server implemented on top of ACML using parasitic threads. The results of this implementation is presented in the following section.

6. Results

Our experiments were run on a 16-way AMD Opteron 865 server with 8 processors, each containing two symmetric cores, and 32 GB of total memory, with each CPU having its own local memory of 4 GB. Access to non-local memory is mediated by a hyper-transport layer that coordinates memory requests between processors. MLton uses 64-bit pointer arithmetic, SSE2 instructions for floating point calculations, and is IEEE floating point compliant. Multi-MLton is extended from the base MLton version 20100608. Multi-MLton backend generates C files that were compiled with gcc 4.4.0.

6.1 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 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 traveling sales man 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 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 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.

Every benchmark was tested on four different configurations: stop-the-world GC, stop-the-world GC with parasites, shared-heap parallel GC, and shared-heap parallel GC with parasitic threads. 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 versions use synchronous communication, since implementing asynchronous communication over host threads is expensive, as previously shown in Section 5.2. The benchmarks were run with same maximum heap size for each of the configurations.

6.2 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 16 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 even 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 vs. 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,

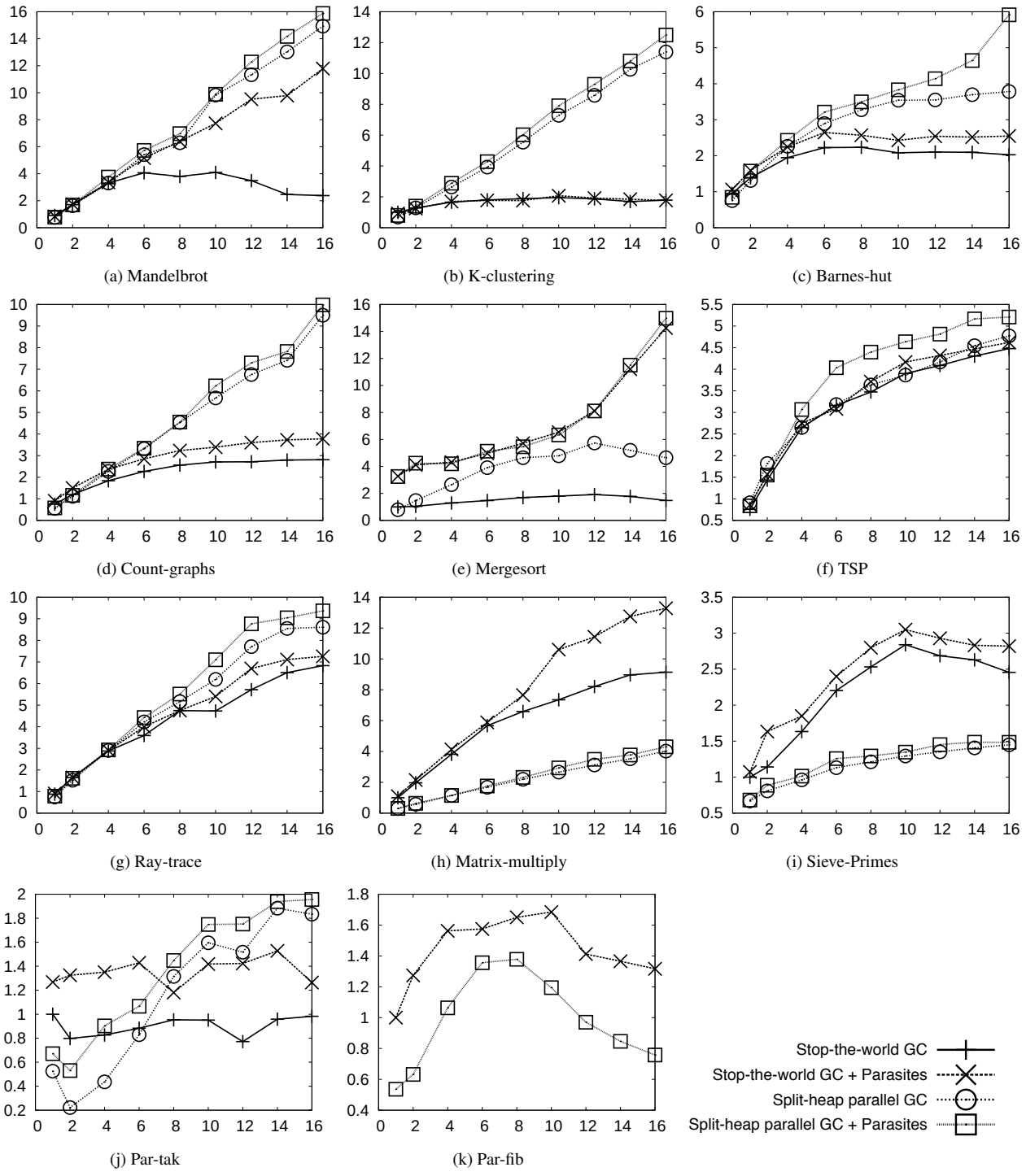


Figure 11: Speedup relative to optimized serial execution. Number of cores is on the x-axis, with relative speedup on the y-axis.

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

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

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 scaled as the number of cores were increased.

6.3 Scalability

Speedup results are presented in Figure 11. Baseline is a version of the programs optimized for serial execution, without the overhead of threading wherever possible. In general, we see that the parasitic versions perform better than their corresponding non-parasitic versions, and shared-heap parallel GC shows better scalability on increasing number of cores. We also noticed that shared-heap parallel GC performs slower than the stop-the-world versions (Sieve-primes and Mat-mult), if the amount of shared data is large. Here, shared-heap parallel GC suffers the cost of copying the transitive closure of the objects from the local heap to the shared heap, when only a small portion of the closure is actually accessed in parallel. Even here, parasitic versions perform better. We believe transitive closure lifting cost can be alleviated by lifting parts of the closure on demand. Parasitic threads also exhibit good scalability in the highly concurrent Par-tak and Par-fib benchmarks. As mentioned earlier, the speedup for the non-parasitic version of Par-fib is not presented as the program did not successfully run to completion.

6.4 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 (PCML) 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. By utilizing asynchronous communication, overall concurrency in the system is increased.

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 16 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 1580 replies per second. By mitigating the overheads of asynchrony, swerve with parasites scaled was able to scale better and offered up to 2230 replies per second.

7. Related Work

There are a number of languages and libraries which support varying kinds of message-passing styles. Systems such as MPI [17, 30] support per-processor static buffers, while CML [26], Erlang [2], F# [29], and MPJ [3] 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. We believe the benefits of parasitic threads can be leveraged regardless of the underlying fundamental message passing primitive used.

There has been much interest in lightweight threading models ranging from using runtime managed thread pools such as those found in C# [28] and the Java util.concurrent package [16], to continuation-based systems found in functional languages such as Scheme [15, 22], CML [26], Haskell [11], and Erlang [2]. 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. Finally, the amount of concurrency is limited by the number of workers in the pool of threads.

Previous work on avoiding thread creation costs have focused on compiler and runtime techniques for different object representations [9, 22] and limiting the computations performed in a thread [14, 15]. The most similar among the previous work to parasitic threads is Lazy Threads [10], which allow encoding arbitrary

computation in a potentially parallel thread. The lazy threads start 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 is 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. Parasite 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 [13, 23] 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 maintain no implicit dependency relationship with the thread that created it. CML [26] 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 [1, 4] are well-known techniques for load balancing multi threaded tree-structured computations, and has been used effectively in languages like Cilk [9] to improve performance. Work sharing is a push model while work stealing is a pull model, and each model has its pros and cons. 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.

8. Conclusions

This paper describes a novel threading mechanism called parasites that enables lightweight asynchronous computation. The key distinguishing feature of parasites is its ability to execute using the resources of a host thread. Parasitic threads are exposed as a threading library for building higher level abstractions, encapsulating demand-driven scheduling and migration. This mechanism enables higher-level programmability without imposing additional scheduling and thread management overheads. Performance evaluation of parasitic threads on two different GC schemes illustrate that parasitic threads are beneficial irrespective of the underlying GC schemes.

References

[1] Kunal Agrawal, Yuxiong He, and Charles E. Leiserson. Adaptive Work Stealing with Parallelism Feedback. In *PPoPP*, pages 112–120, 2007.

[2] Joe Armstrong, Robert Virding, Claes Wikstrom, and Mike Williams. *Concurrent Programming in Erlang*. Prentice-Hall, 2nd edition, 1996.

[3] Mark Baker and Bryan Carpenter. Mpj: A proposed java message passing api and environment for high performance computing. In *IPDPS*, pages 552–559, 2000.

[4] Robert D. Blumofe and Charles E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *J. ACM*, pages 720–748, 1999.

[5] David R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[6] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA*, 2005.

[7] Avik Chaudhuri. A concurrent ml library in concurrent haskell. In *ICFP*, 2009.

[8] Damien Doligez and Xavier Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ml. In *POPL*, 1993.

[9] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *PLDI*, pages 212–223, 1998.

[10] Seth Copen Goldstein, Klaus Erik Schauser, and David E. Culler. Lazy threads: implementing a fast parallel call. *J. Parallel Distrib. Comput.*, 1996.

[11] Tim Harris, Simon Marlow, , and Simon Peyton Jones. Haskell on a Shared-Memory Multiprocessor. In *Haskell Workshop*, pages 49–61, 2005.

[12] Suresh Jagannathan, Armand Navabi, KC Sivaramakrishnan, and Lukasz Ziarek. Design rationale for multi-mlton. In *ML Workshop*, 2010.

[13] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in Dataflow Programming Languages. *ACM Comput. Surv.*, pages 1–34, 2004.

[14] Laxmikant Kale. Charm++. In *Encyclopedia of Parallel Computing (to appear)*. Springer Verlag, 2011.

[15] D. A. Kranz, R. H. Halstead, Jr., and E. Mohr. Mul-T: A High-Performance Parallel Lisp. In *PLDI*, pages 81–90, 1989.

[16] Doug Lea. *Concurrent Programming in Java(TM): Design Principles and Pattern*. Prentice-Hall, 2nd edition, 1999.

[17] Guodong Li, Michael Delisi, Ganesh Gopalakrishnan, and Robert M. Kirby. Formal Specification of the MPI-2.0 Standard in TLA+. In *PPoPP*, pages 283–284, 2008.

[18] Simon Marlow, Simon Peyton Jones, and Satnam Singh. Runtime support for multicore haskell. In *ICFP*, 2009.

[19] M. Matthias Felleisen and Dan Friedman. Control operators, the SECD Machine, and the λ-calculus. In *Formal Description of Programming Concepts III*, pages 193–217, 1986.

[20] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, 1997.

[21] MLton. http://www.mlton.org.

[22] Eric Mohr, David A. Kranz, and Robert H. Halstead, Jr. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. In *LFP*, pages 185–197, 1990.

[23] Rishiyur Nikhil and Arvind. *Implicit Parallel Programming in pH*. Morgan-Kaufmann, 2001.

[24] Chuck Pheatt. Intel threading building blocks. *J. Comput. Small Coll.*, 2008.

[25] John Reppy, Claudio V. Russo, and Yingqi Xiao. Parallel concurrent ml. In *ICFP*, 2009.

[26] John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[27] KC Sivaramakrishnan, Lukasz Ziarek, Raghavendra Prasad, and Suresh Jagannathan. Lightweight asynchrony using parasitic threads. In *DAMP*, 2010.

[28] C# Language Specification. http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf.

[29] Don Syme, Adam Granicz, and Antonio Cisternino. *Expert F#*. Apress, 2007.

[30] Hong Tang and Tao Yang. Optimizing Threaded MPI Execution on SMP Clusters. In *ICS*, pages 381–392, 2001.

[31] Lukasz Ziarek, KC Sivaramakrishnan, and Suresh Jagannathan. Composable asynchronous events. In *PLDI*, 2011.