Places: Adding Message-Passing Parallelism to Racket

Kevin Tew
University of Utah
tewk@cs.utah.edu

James Swaine
Northwestern University
JamesSwaine2010@u.northwestern.edu

Matthew Flatt
University of Utah
mflatt@cs.utah.edu

Robert Bruce Findler
Northwestern University
robby@eecs.northwestern.edu

Peter Dinda
Northwestern University
pdinda@northwestern.edu

Abstract

Places bring new support for message-passing parallelism to Racket. This paper gives an overview of the programming model and how we had to modify our existing, sequential runtime-system to support places. We show that the freedom to design the programming model helped us to make the implementation tractable; specifically, we avoided the conventional pain of adding just the right amount of locking to a big, legacy runtime system. The paper presents an evaluation of the design that includes both a real-world application and standard parallel benchmarks.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.3.4 [*Programming Languages*]: Processors — Run-time environments

General Terms Parallelism, Languages, Design

1. Introduction

The increasing availability of multicore processors on commodity hardware—from cell phones to servers—puts increasing pressure on the design of dynamic languages to support multiprocessing. Support for multiprocessing often mimics the underlying hardware: multiple threads of execution within a shared address space. Unfortunately, the problems with threads of execution in a single address space are well known, non-trivial, and afflict both programmers using a language and the implemetors of the language. Programmers and language implementors alike need better alternatives.

A message-passing architecture, with threads of execution in separate address spaces, is widely recognized as a more scalable design and easier to reason about than shared memory. Besides avoiding the interference problems created by shared memory, the message-passing model encourages programmers to consider the data-placement and communication needs of a program to enable sustained scalability. The design and success of languages like Erlang demonstrate the viability of this model for parallel programming.

DLS'11 October 24, 2011, Portland, Oregon, USA.

Racket's new *place*[1] construct supports message-passing parallelism layered on top of a language that (unlike Erlang) was not originally designed for parallelism. Racket's existing threads and synchronization support for *concurrency* are kept separate from new places support for *parallelism*, except to the degree that message receipt interacts with other concurrent activities within a single place. Message-passing parallelism is not novel in Racket, but our design and experience report for layering places on top of an existing language should be useful to other designers and implementors.

The conventional approach to adding this style of parallelism to a language implementation that has a large, sequential runtime system is to exploit the unix `fork()` primitive, much in the way Python's multiprocessing library works. This approach, however, limits the communication between cooperating tasks to byte streams, making abstraction more difficult and communication less efficient than necessary. We have decided to implement places directly in the runtime system, instead of relying on the operating system. This approach allows the runtime system to maintain more control and also fits our ongoing effort to explore the boundary between the operating system and the programming language (Flatt and Findler 2004; Flatt et al. 1999; Wick and Flatt 2004).

The Racket runtime system begins with a single, initial place. A program can create additional places, send messages to places over channels—including channels as messages, so that any two places can communicate directly. Messages sent between places are normally immutable, preventing the data races that plague shared-memory designs. To allow lower-level communication when appropriate, however, places can share certain mutable data structures, including byte strings, fixnum arrays, and floating-point arrays, all of which contain only atomic values.

As part of Racket's broader approach to parallelism, places fully support our previously reported construct for parallelism, *futures* (Swaine et al. 2010). In particular, each place can spawn and manage its own set of future-executing threads. Places and futures are complementary; places support coarse-grained parallelism without restrictions on the parallel computations, while futures support fine-grained parallelism for sufficiently constrained computations (e.g., no I/O).

The rest of the paper proceeds as follows. Section 2 explains in more detail the design rationale for places. Section 3 briefly outlines the places API. Section 4 demonstrates how message passing, shared memory, and higher-level parallelism constructs can be built on top of place primitives. Section 5 explains the implementation

[1] The choice of the name "place" is inspired by X10's construct. (Charles et al. 2005)

of places within the Racket virtual machine. Section 6 evaluates the performance and scaling of places using the NAS Parallel Benchmarks. Section 7 describes related work.

2. Design Overview

Each *place* is essentially a separate instance of the Racket virtual machine. All code modules are loaded separately in each place, data is (almost always) allocated in a specific place, and garbage collection proceeds (almost always) independently in each place.

Places communicate through *place channels*, which are endpoints for communication channels that are shared among processes in much the way that Unix processes use file descriptors for endpoints of shared pipes. Unlike file descriptors, a place channel supports structured data across the channel, including booleans, numbers, characters, symbols, byte strings, Unicode strings, filesystem paths, pairs, lists, vectors, and "prefab" structures (i.e., structures that are transparent and whose types are universally named). Roughly speaking, only immutable data can be sent across a place channel, which allows the implementation to either copy or share the data representation among places as it sees fit. Place channels themselves can be sent in messages across place channels, so that communication is not limited to the creator of a place and its children places; by sending place channels as messages, a program can construct custom message topologies.

In addition to immutable values and place channels, special mutable byte strings, fixnum vectors, and floating-point vectors can be sent across place channels. For such values, the runtime system is constrained to share the underlying value among places, rather than copy the value as it is sent across a channel. Mutation of the value by one place is visible to other places. By confining shared mutable values to vectors of atomic data, race conditions inherent in sharing cannot create safety problems for the runtime system or complicate garbage collection by allowing arbitrary references from one address space to another. At the same time, shared vectors of atomic data directly support many traditional parallel algorithms, such as a parallel prefix sum on a vector of numbers. Other mutable values could be allowed in place messages with the semantics that they are always copied, but such copying might be confusing, and explicit marshaling seems better to alert a programmer that copying is unavoidable (as opposed to any copying that the runtime system might choose as the best strategy for a given message).

The prohibition against sharing arbitrary mutable values implies that thunks or other procedures cannot be sent from one place to another, since they may close over mutable variables or values. Consequently, when a place is created, its starting code is not specified by a thunk (as is the case for threads) but by a module path plus an exported "main" function. This specification of a starting point is essentially the same as the starting point in Racket itself, except that the "main" function receives a place channel to initiate communication between the new place and its creator. The `place` form simplifies place creation where a procedure would be convenient, but it works by lifting the body of the `place` form to an enclosing module scope at compile time.

Additional place channels can be created and sent to places, allowing the creation of specific constructed capabilities. One common pattern is to have a master place spawn worker places and collect all of the initial place-channels into a list. This list of place channels can then be sent to all the places, which permits all-to-all communication. Place channels are asynchronous, so that the sender of a message need not synchronize with a recipient. Place channels are also two-way as a convenience; otherwise, since a typical communication patterns involve messages in both directions, a program would have to construct two place channels. Finally, place channels are *events* in the sense of Concurrent ML (Reppy 1999; Flatt and Findler 2004). Place channels can be combined with other events to build up complex synchronization patterns, such as fair choice among multiple place channels.

Our current initial implementation of places shares little read-only data among places. Longer term, we would like to automatically share read-only code modules and JIT-generated code across places in much the same way that operating systems share libraries among separate applications. In general, places are designed to allow such sharing optimizations in the language runtime system as much as possible.

3. Places API

The Racket API for places[2] supports place creation, channel messages, shared mutable vectors, and a few administrative functions.

```
(dynamic-place module-path start-proc) → place?
  module-path : module-path?
  start-proc : symbol?
```

Creates a place to run the procedure that is identified by *`module-path`* and *`start-proc`*.[3] The result is a place descriptor value that represents the new parallel task; the place descriptor is returned immediately. The place descriptor is also a place channel to initiate communication between the new place and the creating place.

The module indicated by *`module-path`* must export a function with the name *`start-proc`*. The exported function must accept a single argument, which is a place channel that corresponds to the other end of communication for the place channel that is returned by `dynamic-place`. For example,

```
(dynamic-place "fib.rkt" 'go)
```

starts the module `"fib.rkt"` in a new place, calling the function `go` that is exported by the module.

```
(place id body ...+)
```

The `place` derived form creates a place that evaluates *`body`* expressions with *`id`* bound to a place channel. The *`body`*s close only over *`id`* plus the top-level bindings of the enclosing module, because the *`body`*s are lifted to a function that is exported by the module. The result of `place` is a place descriptor, like the result of `dynamic-place`.

For example, given the definitions

```
(define (fib n) ....)

(define (start-fib-30)
  (place ch (fib 30)))
```

then calling `start-fib-30` creates a place to run a new instantiation of the enclosing module, and the `fib` function (which need not be exported) is called in the new place.

```
(place-channel-put ch v) → void?
  ch : place-channel?
  v : place-message-allowed?
(place-channel-get ch) → place-message-allowed?
  ch : place-channel?
```

The `place-channel-put` function asynchronously sends a message *`v`* on channel *`ch`* and returns immediately. The `place-channel-get` function waits until a message is available from the place channel *`ch`*. See also `sync` below.

[2] This paper describes the API of places for the 5.1.2 release version of Racket at `http://racket-lang.org/download/`.

[3] The `dynamic-` prefix on the function name reflects the similarity of this function to Racket's `dynamic-require` function.

As an example, the following `start-fib` function takes a number *n*, starts `(fib n)` in a new place, and returns a place descriptor to be used as a place channel for receiving the result:

```
(define (fib n) ....)

(define (start-fib n)
  (define p
    (place ch
      (define n (place-channel-get ch))
      (place-channel-put ch (fib n))))
  (place-channel-put p n)
  p)
```

The `start-fib` function could be used to start two computations in parallel and then get both results:

```
(define p1 (start-fib n1))
(define p2 (start-fib n2))
(values (place-channel-get p1)
        (place-channel-get p2))
```

```
(place-channel-put/get ch v)
 → place-message-allowed?
  ch : place-channel?
  v : place-message-allowed?
```

A convenience function to combine a `place-channel-put` with an immediate `place-channel-get`.

```
(place-channel) → place-channel? place-channel?
```

Returns two place channels that are cross-linked through an underlying data channel. Data sent through the first place channel is received through the second place channel and vice versa.

For example, if `buyer` and `seller` places are given channel endpoints, they can communicate directly using the new channel and report only final results through their original channels:

```
(define b (dynamic-place "trade.rkt" 'buyer))
(define s (dynamic-place "trade.rkt" 'seller))

(define-values (b2s s2b) (place-channel))
(place-channel-put b b2s)
(place-channel-put s s2b)
; ... buyer and seller negotiate on their own ...

(values (place-channel-get b)
        (place-channel-get s))
```

```
(sync evt ...+) → any?
  evt : evt?
```

Blocks until at least one of the argument *evt*s is ready, and returns the value of the ready *evt*. A place channel as an event becomes ready when a message is available for the channel, and the corresponding value produced by `sync` is the channel message. Thus, `(sync ch1 ch2)` receives a message from `ch1` or `ch2`—whichever has a message first.

Racket includes other synchronization constructs, such as the `sync/timeout` function to poll an event. Our examples in this paper need only `sync`.

```
(handle-evt evt handle) → handle-evt?
  evt : (and/c evt? (not/c handle-evt?))
  handle : (any/c . -> . any)
```

Creates an event that is in a ready when *evt* is ready, but whose result is determined by applying *handle* to the result of *evt*.

```
(place-wait p) → void?
  p : place?
```

Blocks until *p* terminates.

```
(make-shared-fxvector size [x]) → fxvector?
  size : exact-nonnegative-integer?
  x : fixnum? = 0
(make-shared-flvector size [x]) → flvector?
  size : exact-nonnegative-integer?
  x : flonum? = 0.0
```

Creates a mutable, uniform vector of fixnums or floating-point numbers that can be shared across places. That is, the vector is allowed as a message on a place channel, and mutations of the vector by the sending or receiving place are visible to the other place. The concurrency model for shared data is determined by the underlying processor (e.g., TSO (Sewell et al. July 2010) for x86 processors). Places can use message passing or functions like `place-wait` to synchronize access to a shared vector.

For example,

```
(define (zero! vec)
  (define p
    (place ch
      (define vec (place-channel-get ch))
      (for ([i (fxvector-length vec)])
        (fxvector-set! vec i 0))))
  (place-channel-put p vec)
  (place-wait p))
```

fills a mutable fixnum vector with zeros using a separate place. Waiting until the place is finished ensures that the vector is initialized when `zero!` returns.

```
(processor-count) → exact-positive-integer?
```

Returns the number of parallel computation units (e.g., processors or cores) that are available on the current machine.

4. Design Evaluation

We evaluated the design of places in two main ways. First, we used places for Racket's parallel-build infrastructure, where the implementation uses Erlang-style message handling. Second, we ported the NAS Parallel Benchmark suite to Racket using MPI-like parallelism constructs that are built on top of places. In addition to our main experiments, we present a Mandelbrot example that demonstrates how atomic-value vectors can be shared among places. Together, these examples demonstrate the versatility of places for implementing different patterns of parallelism.

4.1 Parallel Build

The full Racket source repository includes 700k lines of Racket code plus almost 200k of documentation source (which is also code) that is recompiled with every commit to the repository. A full build takes nearly an hour on a uniprocessor, but the build parallelizes well with places, speeding up by 3.2x on 4 cores.

The build is organized as a controller in the main place that spawns workers in their own places. One worker is created for each available processor. The controller keeps track of the files that need to be compiled, while each worker requests a file to compile, applies the `compile` function to the file, and repeats until no more files are available from the controller.

Concretely, the workers are created by `place` in a `for/list` comprehension that is indexed by an integer from `0` to `(processor-count)`:

```
(define ps ;list of place descriptors
  (for/list ([i (processor-count)])
    (place ch
      (let worker ()
        (match (place-channel-put/get ch 'get-job)
          ['done (void)]
          [job
           (compile job)
           (define msg (list 'job-finished job))
           (place-channel-put ch msg)
           (worker)]))))))
```

Each worker runs a `worker` loop that sends a `'get-job` message to the controller via `ch` and then waits for a response. If the response to the `'get-job` request is the symbol `'done`, the controller has no more jobs; the place quits running by returning `(void)` instead of looping. If the controller responds with a job, the worker compiles the job, sends a completion message back to the controller, and loops back to ask for another job.

After spawning workers, the controller waits in a loop for messages to arrive from the workers. Any worker might send a message, and the controller should respond immediately to the first such message. In Concurrent ML style, the loop is implemented by applying `sync` to a list of events, each of which wraps a place channel with a handler function that answers the message and recurs to the message loop. When no jobs are available to answer a worker's request, the worker is removed from the list of active place channels, and the message loop ends when the list is empty.

The message-handling part of the controller matches a given message `m`, handles it, and recurs via `message-loop` (lines 21-33 in figure 1). Specifically, when the controller receives a `'get-job` message, it extracts a job from the job queue. If the job queue has no remaining jobs so that `(get-job job-queue)` returns `#f`, the `'done` message is sent to the worker otherwise, the job from the queue is sent back to the worker. When the controller instead receives a `(list 'job-finished job)` message, it notifies the job queue of completion and resumes waiting for messages.

Figure 1 contains the complete parallel-build example. Racket's actual parallel-build implementation is more complicated to handle error conditions and the fact that compilation of one module may trigger compilation of another module; the controller resolves conflicts for modules that would otherwise be compiled by multiple workers.

```
1  #lang racket
2  (require "job-queue.rkt")
3
4  (define (main)
5    (define ps ;list of place descriptors
6      (for/list ([i (processor-count)])
7        (place ch
8          (let worker ()
9            (place-channel-put ch 'get-job)
10           (match (place-channel-get ch)
11             ['done (void)]
12             [job
13              (compile job)
14              (define msg (list 'job-finished job))
15              (place-channel-put ch msg)
16              (worker)])))))
17
18   (define job-queue (build-job-queue))
19
20   (define (make-message-handler p ps)
21     (define (message-handler m)
22       (match m
23         ['get-job
24          (match (get-job job-queue)
25            [#false
26             (place-channel-put p 'done)
27             (message-loop (remove p ps))]
28            [job
29             (place-channel-put p job)
30             (message-loop ps)])]
31         [(list 'job-finished job)
32          (job-finished job-queue job)
33          (message-loop ps)]))
34     (handle-evt p message-handler))
35
36   (define (message-loop ps)
37     (define (make-event p)
38       (make-message-handler p ps))
39     (unless (null? ps)
40       (apply sync (map make-event ps))))
41
42   (message-loop ps))
```

Figure 1: Parallel Build

4.2 Higher-level Constructs

Repeatedly creating worker modules, spawning places, sending initial parameters, and collecting results quickly becomes tiresome for a Racket programmer. Racket's powerful macro system, however, permits the introduction of new language forms to abstract such code patterns. The Racket version of the NAS parallel benchmarks are built using higher-level constructs: `fork-join`, `CGfor` and `CGpipeline`.

4.2.1 CGfor

The `CGfor` form looks like the standard Racket `for` form, except for an extra *communicator group* expression. The communicator group records a configuration in three parts: the integer identity of the current place, the total number of places in the communicator group, and a vector of place channels for communicating with the other places. The `CGfor` form consults a given communicator group to partition the loop's iteration space based on the number of places in the group, and it executes the loop body only for indices mapped to the current place's identity. For example, if a communication group `cg` specifies 3 places, then `(CGfor cg ([x (in-range 900)]) ...)` iterates a total of 900 times with the first place computing iterations 1–300, the second place iterating 301–600, and the third place iterating 601–900.

The `fork-join` form creates a communicator group and binds it to a given identifier, such as `cg`. The following example demonstrates a parallel loop using `fork-join` and `CGfor`, which are defined in the `"fork-join.rkt"` library:

```
1  #lang racket
2  (require "fork-join.rkt")
3
4  (define (main n)
5    (fork-join (processor-count) cg ([N n])
6      (CGfor cg ([i (in-range N)])
7        (compute-FFT-x i))
8      (CGBarrier cg)
9      (CGfor cg ([i (in-range N)])
10       (compute-FFT-y i))
11     (CGBarrier cg)
12     (CGfor cg ([i (in-range N)])
13       (compute-FFT-z i))))
```

The `fork-join` form on line 5 creates `(processor-count)` places and records the configuration in a communicator group `cg`. The `([N n])` part binds the size `n` from the original place to `N`

```
1 (define-syntax-rule
2   (fork-join NP cg ([params args] ...) body ...)
3     (define ps
4       (for/list ([i (in-range n)])
5         (place ch
6           (define (do-work cg params ...) body ...)
7           (match (place-channel-get ch)
8             [(list-rest id np ps rargs)
9              (define cg (make-CG id np (cons ch ps)))
10             (define r (apply do-work cg rargs))
11             (place-channel-put ch r)])))))
12
13    (for ([i (in-range NP)] [ch ps])
14      (place-channel-put
15       ch
16       (list i NP ps args ...)))
17
18    (for/vector ([i (in-range NP)] [ch ps])
19      (place-channel-get ch)))
```

Figure 2: fork-join

in each place, since the new places cannot access bindings from the original place. The `(CGBarrier cg)` expression blocks until all of the places in the communication group `cg` reach the barrier point.

The complete implementation for `fork-join` is shown in figure 2. First `fork-join` spawns places (line 4), sends a message to each place containing the place's identity and other communication-group parameters, and other arguments specified in the `fork-join` use (line 13). It then waits for each place to report its final result, which is collected into a vector of results (line 18).

Each worker place waits for a message from its controller containing its communicator group settings and initial arguments (lines 7-8). The place builds the local communicator group structure (line 9) and evaluates the `fork-join` body with the received arguments (line 10). Finally, the result of the place worker's computation is sent back across a place channel to the place's controller (line 11).

4.2.2 CGpipeline

In the same way a `CGfor` form supports simple task parallelism, a `CGpipeline` form supports pipeline parallelism. For example, the LU benchmark uses a parallel pipeline to compute lower and upper triangular matrices. As a simpler (and highly contrived) example, the following code uses pipeline parallelism to compute across the rows of a matrix, where a cell's new value is the squared sum of the cell's old value and the value of the cell to its left. Instead of treating each row as a task, each column is a task that depends on the previous column, but rows can be pipelined through the columns in parallel:

```
1 (define v (flvector 0.0 1.0 2.0 3.0 4.0
2                     0.1 1.1 2.1 3.1 4.1
3                     0.2 1.2 2.2 3.2 4.2
4                     0.3 1.3 2.3 3.3 4.3
5                     0.4 1.4 2.4 3.4 4.4))
6
7 (fork-join 5 cg ()
8   (for ([i (in-range 5)])
9     (CGpipeline cg prev-value 0.0
10      (define idx (+ (* i 5) (CG-id cg)))
11      (define (fl-sqr v) (fl* v v))
12      (fl-sqr (fl+ (fl-vector-ref v idx)
13                   prev-value)))))
```

```
(define-syntax-rule
  (CGpipeline cg prev-value init-value body ...)
  (match cg
    [(CG id np pls)
     (define (send-value v)
       (place-channel-put (list-ref pls (add1 id)) v))
     (define prev-value
       (if (= id 0)
           init-value
           (place-channel-get (car pls))))
     (define result (begin body ...))
     (unless (= id (sub1 np)) (send-value result))
     result]))
```

Figure 3: CGpipeline

The pipeline is constructed by wrapping the `CGpipeline` form with a normal `for` loop inside `fork-join`. The `fork-join` form creates five processes, each of which handles five rows in a particular column. The `CGpipeline` form within the `for` loop propagates the value from previous column—in the variable `prev-value`, which is `0.0` for the first column—to compute the current column's value. After a value is produced for a given row, a place can proceed to the next row while its value for the previous row is pipelined to later columns. Like the `CGfor` form, the `CGpipeline` form uses a communicator group to discover a place's identity, the total number of places, and communication channels between places.

Figure 3 shows the implementation of `CGpipeline`. All places except place `0` wait for a value from the previous place, while place `0` uses the specified initial value. After place i finishes executing its body, it sends its result to place i+1, except for the final place, which simply returns its result. Meanwhile, place i continues to the next row, enabling parallelism through different places working on different rows.

4.3 Shared Memory

Certain algorithims benefit from shared-memory communication. Places accommodates a subset of such algorithms through the use of shared vectors. Shared-vector primitives permit a restricted form of shared-memory data structures while preserving the integrity of the language virtual machine. Shared vectors have two integrity-preserving invariants: their sizes are fixed at creation time, and they can only contain atomic values.

In the following example, the `mandelbrot-point` function is a black-box computational kernel. It consumes an (x, y) coordinate and returns a Mandelbrot value at that point. The argument `N` specifies the number lines and columns in the output image.

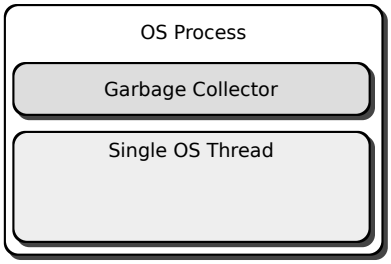
```
1 #lang racket
2 (require "fork-join.rkt"
3          "mandelbrot-point.rkt")
4
5 (define (main N)
6   (define NP (processor-count))
7   (define b (make-shared-bytes (* N N) 0))
8
9   (fork-join NP cg ([N N] [b b])
10    (CGfor cg ([y (in-range N)])
11      (for ([x (in-range N)])
12        (define mp (mandelbrot-point x y N))
13        (byte-2d-array-set! b x y N mp))))
14
15  (for ([y (in-range N)])
16    (write-bytes/newline b y N)))
```

In this implementation, workers communicate `mandelbrot-point` results to the controller through a shared byte vector `b`. Vector `b`'s size is fixed to `(* N N)` bytes, and all `b`'s elements are initialized to 0. The `fork-join` construct spawns the worker places, creates the communicator group `cg`, and sends the line length (`N`) and the shared result vector (`b`) to the workers.

Having received their initial parameters, each place computes its partition of the Mandelbrot image and stores the resulting image fragment into the shared vector (`b`). After all of the worker places finish, the controller prints the shared vector to standard output. The shared-memory implementation speeds up Mandelbrot by 3x on 4 cores.

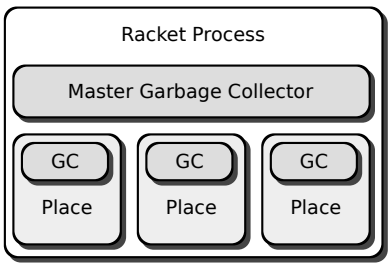
5. Implementing Places

Prior to support for places, Racket's virtual machine used a single garbage collector (GC) and single OS thread:

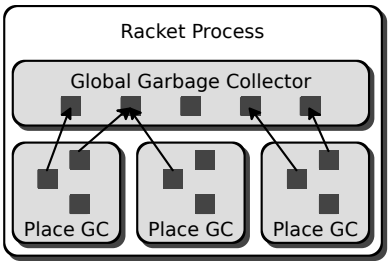


Although Racket has always supported threads, Racket threads support concurrency rather than parallelism; that is, threads in Racket enable organizing a program into concurrent tasks, but threads do not provide a way to exploit multiprocessing hardware to increase a program's performance. Indeed, although threads are preemptive at the Racket level, they are co-routines within the runtime system's implementation.

Racket with places uses OS-scheduled threads within the Racket virtual machine. Each place is essentially an instance of the sequential, pre-places virtual machine. To achieve the best parallel performance, places are as independent and loosely coupled as possible, even to the point of separating memory regions among places to maximize locality within a place. Even better, separate address spaces mean that each place has its own GC that can collect independently from other places.



Each place-local GC allocates and manages almost all of the objects that a place uses. An additional master GC is shared across all places to manage a few global shared objects, such as read-only immortal objects, place channels, and shared vectors of atomic values. Object references from places to the shared master heap are permitted, but references are not permitted in the opposite direction.



Disallowing references from the master space to place-specific spaces maintains isolation between places, and it is the invariant that allows places to garbage collect independently of one another. Only a global collection of the shared master space requires the collective cooperation of all the places, and such collections are rare.

The implementation of places thus consists of several tasks: adding OS schedulable threads to the runtime system, converting global state variables within the runtime system to place-local variables, modifying garbage collection strategies for concurrent-place execution, and implementing channels for communicating between places.

5.1 Threads and Global Variables

The Racket runtime system has been continuously developed for the past decade and a half. Like other mature runtime systems, the Racket implementation includes many global variables. The presence of such global variables in the code base was the largest obstacle to introducing OS-scheduled threads into the runtime system.

Using grep and a simple CIL (Necula et al. 2002) analysis, we conducted an audit of the 719 global variables within the Racket implementation. The audit found 337 variables that fell into the category of read-only singleton objects once they were set (during VM initialization). A few of the variables encountered during the audit, such as `scheme_true`, `scheme_false`, and `scheme_null`, were easy to identify as read-only singleton objects. These were annotated with a `READ_ONLY` tag as documentation and to support further analysis. The auditing of most variables, however, required locating and reviewing all code sites where a particular variable was referenced. About 155 global variables were deemed permissible to share and annotated as `SHARED_OK`. The remaining 227 variables needed to be localized to each place and were tagged as `THREAD_LOCAL_DECL`.

Tool support simplifies the arduous task of annotating and auditing global variables. Tools that simply identify all global variables are remarkably helpful in practice. Finding all the code sites where a global variable is used helps the runtime developer ensure that isolation invariants are preserved in each place that a global variable is referenced.

Testing the global variable audit was relatively easy. We ran the entire Racket test suite in multiple places simultaneously. For almost all global variables that we overlooked or misclassified, parallel execution of the test suite identified the problem.

5.2 Thread-Local Variables

To prevent collisions from concurrent access, many global variables were localized as place-specific variables. We considered moving all global variables into a structure that is threaded through the entire runtime system. Although this restructuring is clean in principle, restructuring the runtime system along those lines would have required extensive modifications to function signatures and code flow. Instead, we decided to use thread-local variables, as supported by the OS, to implement place-local state.

OSes support thread-local variables through library calls, such as `pthread_get_specific()` and `pthread_put_specific()`,

and sometimes through compiler-implemented annotations, such as `__threadlocal` or `__declspec(thread)`. Compiler-implemented thread-local variables tend to be much faster, and they work well for Racket on Linux and most other variants of Unix. Although Windows supports compiler-implemented thread-local variables, Windows XP does not support them within DLLs (as used by Racket); Vista and later Windows versions remedy this problem, but Racket 32-bit builds must work on older versions of Windows. Finally, Mac OS X does not currently support compiler-implemented thread-local variables.

Our initial experiments indicated that using library calls for thread-local variables on Windows and Mac OS X would make the runtime system unacceptably slow. Reducing the cost of thread-local variables on those platforms requires two steps.

First, all place-local variables were first collected into a single table. Each place-local variable, such as `toplevels_ht`, has an entry in the table with an underscore suffix:

```
struct Thread_Locals {
  struct Scheme_Hash_Table *toplevels_ht_;
  ....
};

inline struct Thread_Locals *GET_TLV() { ... }

#define toplevels_ht (GET_TLV()->toplevels_ht_)
```

A preprocessor definition for each variable avoids the need to change uses in the rest of the source. Collecting all thread-local variables into a table supports threading a pointer to the table through the most performance-sensitive parts of the runtime system, notably the GC. Along similar lines, JIT-generated code keeps a pointer to the thread-local table in a register or in a local variable.

Second, for uses of thread-local variables outside the GC or JIT-generated code, we implement `GET_TLV()` in a way that is faster than calling `pthread_get_specific()`. In 32-bit Windows, a host executable (i.e., the one that links to the Racket DLL) provides a single thread-local pointer to hold the table of thread-local variables; inline assembly in `GET_TLV()` imitates compiler-supported access to the executable-hosted variable. For Mac OS X, `GET_TLV()` contains an inline-assembly version of `pthread_get_specific()` that accesses the table of thread-local variables.

5.3 Garbage Collection

At startup, a Racket process creates an initial GC instance and designates it the master GC. Read-only global variables and shared global tables such as a symbol table, resolved-module path table, and the type table are allocated from the master GC. After the prerequisite shared structures are instantiated, the initial thread disconnects from the master GC, spawns its own GC instance, and becomes the first place. After the bootstrapping phase of the Racket process, the master GC does little besides allocating communication channels and shared atomic-value containers.

Places collect garbage in one of two modes: independently, when collecting only the local heap, or cooperatively as part of a global collection that includes the master GC. Place-local GCs collect their local heap without any synchronization; a place collector traverses the heap and marks objects it allocated as live, and all other encountered objects, including objects allocated by the master GC, are irrelevant and ignored.

When the master GC needs to perform a collection, all places must pause and cooperate with the master GC. Fortunately, most allocation from the master GC occurs during the initialization of a program. Thus, the master GC normally reaches a steady state at the beginning of some parallel program, allowing places to run in parallel without interruption in common situations.

To initiate a global collection, the master GC sends a signal to all places asking them to pause mutation and cooperatively collect. Each place then performs a local collection in parallel with one another. During cooperative collection, a place GC marks as live not only traversed objects it allocated but also objects that were allocated by the master GC; races to set mark bits on master-GC objects are harmless. Master-GC objects that are referenced only by place-local storage are thus correctly preserved.

After all place-specific collections have finished, each place waits until the master GC marks and collects. Although place-specific collection can move objects to avoid fragmentation, the master GC never moves objects as it collects; master-GC allocation is rare and coarse-grained enough that compaction is not needed. Each place can therefore resume its normal work as soon as the master-GC collection is complete.

5.4 Place Channels

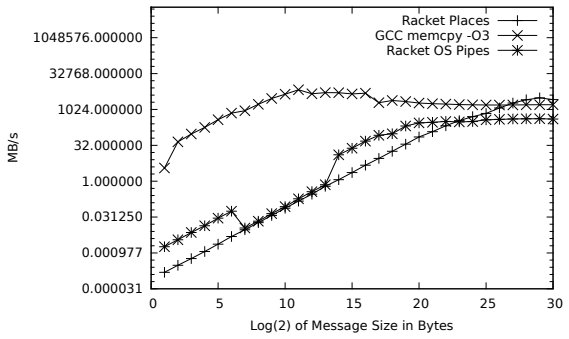
To maintain the invariant that allows the place-specific GCs to work independently, sending a message over a place channel copies data from the originating place to the destination place.

Place channels implement efficient, one-copy message passing by orphaning memory pages from the source place and adopting those memory pages into the destination place. A place channel begins this process by asking its local allocator for a new orphan allocator. The orphan allocator groups all its allocations onto a new set of orphaned memory pages. Orphaned pages are memory blocks that are not owned by any GC. The place channel then proceeds to copy the entire message using the orphan allocator. After the copy is completed, the new orphaned message only contains references to objects within itself and shared objects owned by the master GC. The originating place sends this new message and its associated orphaned memory pages to the destination place.

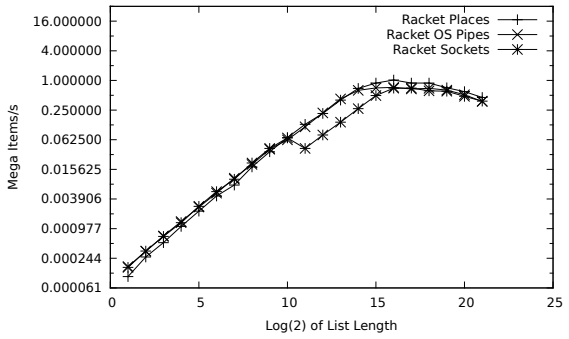
A place channel, receiving a message, adopts the message's orphaned memory pages into its own nursery generation and returns the received message to the user program. Message contents that survive the nursery generation will relocate to memory more localized to the receiving place as the objects are promoted from the nursery to the mature object generation. This orphan-adoption process allows for single copy asynchronous message passing without needing to coordinate during message allocation.

Messages less than 1024 bytes in length are handled in a slightly different manner. These short messages are allocated onto an orphan page and sent to the destination place exactly as described above. At the short message's destination, instead of adopting the messages orphaned pages, the destination place copies the message from the orphan page into its local allocator. By immediately copying short messages into the destination place allocator, the orphaned page can be returned to the system immediately for use by subsequent place-channel messages.

The graphs in figure 4 summarize the performance of place-channel communication. The first graph compares `memcpy()` in C, place channels in Racket, and pipes in Racket on a byte-string message. The results, which are plotted on a log scale, show that place channels can be much slower than raw `memcpy()` for small messages, where the cost of memory-page management limits place-channel throughput. Messages closer to a page size produce similar throughput with all techniques. The second graph shows place-channel, pipe, and socket performance when the message is a list, where Racket's `write` and `read` are used to serialize lists for pipes and sockets. The graph shows that place-channel communication remains similar to pipe and socket communication for structured data. Together, the results show that our communication strategy does not make communication particularly cheap, but it is competitive with services that have been optimized by OS implementors.



Byte-String Throughput



Structured-Data Throughput

Figure 4: Place-Channel Performance

5.5 OS Page-Table Locks

Compilation of Racket's standard library was one of our early tests of performance with places. After eliminating all apparent synchronization points as possible points of contention, we found that using separate processes for the build scaled better than using places within a single Racket process. On closer inspection of the system calls being made in each case, we saw that the build used many `mprotect()` calls that took a long time to complete.

The Racket generational garbage collector uses OS-implemented memory protection to implement write barriers. Each garbage collection uses `mprotect()` to clear and set read-only permissions on memory pages. After consulting the Linux source code, we realized that `mprotect()` acquires a lock on the process's page table. When two or more places garbage collect at the same time, contention for the process's page table lock greatly increased the time for `mprotect()` calls to complete. To avoid this problem, we implemented an extra layer for the Racket allocator to produce larger blocks of contiguous mature objects; issuing a single `mprotect()` call for the contiguous block reduces the overall number of `mprotect()` calls by an order of magnitude, which eliminates the bottleneck.

The more general lesson is that OSes are designed to support separate processes with minimal interference, but some corners of an OS rely on relatively heavy locks within a single process. Fortunately, we do not encounter these corners often—for example, concurrent filesystem access seems to perform as well with places as with separate processes—but the possibility is an extra concern for the implementation.

5.6 Overall: Harder than it Sounds, Easier than Locks

The conversion of Racket to support places took approximately two graduate-student years, which is at least four times longer than we originally expected. At the same time, the implementation of places has proven more reliable than we expected; when we eventually flipped the default configuration of Racket from no-places (and a parallel library build based on OS processes) to places (and using them for building libraries), our automatic test builds continued exactly as before—with the same success rate and performance. Further deployments uncovered memory leaks, but those were quickly corrected.

Our experience with places contrasts sharply with our previous experience introducing concurrency into the runtime system, where months of additional testing and use were required to uncover many race conditions that escaped detection by the test suite. We attribute this difference primarily to the small amount of sharing among places, and therefore the small number of locking locations and potential races in the code.

While implementing places, we made many mistakes where data from one place was incorrectly shared with another place, either due to incorrect conversion of global variables in the runtime system or an incorrect implementation of message passing. Crashes from such bugs were highly reproducible, however, because a bad reference in a place tends to stick around for a long time, so it is detected by an eventual garbage collection. Bugs due to incorrect synchronization, in contrast, hide easily because they depend on relatively unlikely coincidences of timing that are exacerbated by weak memory models.

In adding places to Racket, we did not find as easy a path to parallelism as we had hoped. We did, however, find a preferable alternative to shared memory and locks.

6. Performance Evaluation

We evaluated the performance of places by running the NASA Advanced Supercomputing (NAS) Parallel Benchmarks (Bailey et al. Aug. 1991).[4] These benchmarks represent simplified kernels from computation fluid-dynamics problems. This section presents results for Racket,[5] Java, and Fortran/C versions of the NAS benchmarks.

We use two high-end workstations that might be typical of a scientist's desktop machine. *Penghu* is a dual socket, quad-core per processor, Intel Xeon machine running Mac OS X. *Drdr* is a dual socket, hex-core per processor, AMD machine running Linux.

The NAS Parallel Benchmarks consists of seven benchmarks. Integer Sort (IS) is a simple histogram integer sort. Fourier Transform (FT) is a 3-D fast Fourier transform. FT computes three 1-D FFTs, one for each dimension. Conjugate Gradient (CG) approximates the smallest eigenvalue of a sparse unstructured matrix, which tests the efficiency of indirect memory access. MultiGrid (MG) solves a 3-D scalar Poisson equation and exercises memory transfers. Scalar Pentadiagonal (SP) is a 3-D Navier-Stokes solver using Beam-Warming approximate factorization. Block Tridiagonal (BT) is a Navier-Stokes solver using Alternating Direction Implicit approximate factorization. Lower and Upper (LU) is a Navier-Stokes solver using the symmetric successive over-relaxation method.

Each NAS benchmark consists of a range of problem size classes, from smallest to largest they are S, W, A, B, and C. We ran the A size class on the shorter IS, FT, CG, MG benchmarks. On the longer benchmarks, SP, BT, and LU, we ran the W size class.

Each benchmark is represented by a row of graphs in Figure 6 and Figure 7. The raw-performance graphs for each of the two benchmark machines comes first, followed by the speedup graphs. The raw-performance graph plots the number of threads versus the time to complete the benchmark with the left-most point (labelled

[4] `http://www.nas.nasa.gov/Resources/Software/npb.html`

[5] The Racket version of the NAS Parallel Benchmarks is available at `https://github.com/tewk/racketNAS`.

"S") indicating the time for running the sequential benchmark without creating any places. The speedup graphs plot the number of threads versus the benchmark runtime divided by the benchmark time for one parallel thread. The gray line in the speed up graphs indicates perfect linear speedup.

In terms of raw performance, the Fortran/C implementation is the clear winner. Java comes in second in most benchmarks. Racket is third in most benchmarks, although it handily wins over Java in the SP and LU benchmarks.

More importantly, the Racket results demonstrate that our places implementation generally scales as well as the Java and Fortran/C versions do. In many of the benchmarks, running the Racket code with one parallel place takes only slightly longer than running the sequential code. The small difference in run times between sequential and one-place parallel versions suggests that the runtime cost of places for parallelization is practical.

The IS C result for Penghu (Mac OS X) machine is uncharacteristically slower than the Java and Racket run times. The IS benchmark on the Drdr (Linux) machine is much faster. The NPB implementors wrote all the reference benchmarks in Fortran, except for IS. The NPB developers wrote the IS benchmark in C, using OpenMP's `threadprivate` directive. GCC versions prior to 4.6 refused to compile the IS benchmark under Mac OS X, emitting an error that `__threadlocal` was not supported. However, the prerelease GCC 4.6 successfully compiles and runs the IS benchmark. We believe that GCC 4.6 calls the `pthread_get_specific()` API function to implement OpenMP thread private variables, which increases the runtime of the IS implementation on Mac OS X.

The 3x difference in FT performance between Racket and Java is most likely due to Racket's lack of instruction-level scheduling and optimization. The negative scaling seen in the CG benchmark on Drdr for processor counts 7-12 is likely a chip locality issue when the computation requires both processor sockets. Unlike all the other benchmark kernels, the CG benchmark operates on a sparse matrix. The extra indirection in the sparse matrix representation reduces the effectiveness of memory caches and tests random memory accesses.

The MG benchmark stresses a machine's memory subsystem in a different manner. During its computation, MG copies data back and forth between coarse and fine representations of its grid. On Mac OS X, we had to increase the Java maximum heap size from 128MB to 600MB for the MG benchmark to finish successfully. Java's maximum heap size on Linux appears to default to approximately 1/4th of the total system memory, which was sufficient for the MG benchmark to finish on our Linux test platform.

The 4x difference in runtimes between Java and Racket in SP and LU is most likely due to poor common sub-expression elimination. While porting the Java benchmarks to Racket, we manually eliminated hundreds of common sub-expressions by introducing local variables. The reference implementation's Fortran code has the same duplicated sub-expressions as the Java version. In contrast to Java, the Fortran compiler appears to have a very effective sub-expression elimination optimization pass.

	Penghu	**Drdr**
OS	OS X 10.6.2	Ubuntu 10.4
Arch	x86_64	x86_64
Processor Type	Xeon	Opteron 2427
Processors	2	2
Total Cores	8	12
Clock Speed	2.8 GHz	2.2 GHz
L2 Cache	12MB	3MB
Memory	8 GB	16 GB
Bus Speed	1.6 GHz	1 GHz
Racket	v5.1.1.6	v5.1.1.6
gfortran	4.6.0 2010/7	4.4.3
Java	1.6.0_20	OpenJDK 1.6.0_18

Figure 5: Benchmark Machines

7. Related Work

Racket's futures (Swaine et al. 2010), like places, provide a way to add parallelism to a legacy runtime system. Futures are generally easier to implement than places, but the programming model is also more constrained. Specifically, a place can run arbitrary Racket code, but a future can only run code that is already in the "fast path" of the runtime system's implementation. There are, however, a few situations where futures are less constrained, namely when operating on shared, mutable tree data structures. Some tasks (including many of the benchmarks in Section 6), are well-supported by both futures and places and, in those cases, the performance is almost identical. We expect to develop new constructs for parallelism in Racket that internally combine futures and places to get the advantages of each.

Concurrent Caml Light (Doligez and Leroy 1993) relies on a compile-time distinction between mutable and immutable objects to enable thread-local collection. Concurrent Caml Light gives its threads their own nurseries, but the threads all share a global heap. Concurrent Caml Light is more restrictive than Racket places. In Concurrent Caml Light, only immutable objects can be allocated from thread-local nurseries; mutable objects must be allocated directly from the shared heap. Concurrent Caml Light presumes allocation of mutable objects is infrequent and mutable objects have longer life spans. Racket's garbage collector performs the same regardless of mutable object allocation frequency or life span.

Erlang (Sagonas and Wilhelmsson Oct. 2006) is a functional language without destructive update. The Erlang implementation uses a memory management system similar to Racket's master and place-local GCs. All Erlang message contents must be allocated from the shared heap; this constraint allows O(1) message passing, assuming message contents are correctly allocated from the shared heap, and not from the Erlang process's local nursery. The Erlang implementation employs static analysis to try to determine which allocations will eventually flow to a message send and therefore should be allocated in the shared heap. Since messages are always allocated to the shared heap, Erlang must collect the share heap more often then Racket, which always allocates messages into the destination place's local heap. Erlang's typical programming model has many more processes than CPU cores and extensive message exchange, while places are designed to be used one place per CPU core and with less message-passing traffic.

Haskell (Marlow et al. 2008; Marlow et al. 2009) is a pure functional language with support for concurrency. Currently, Haskell garbage collection is global; all threads must synchronize in order to garbage collect. The Haskell implementors plan to develop local collection on private heaps, exploiting the predominance of immutable objects similarly to Concurrent Caml Light's implementation. In contrast to pure functional languages, Racket programs often include mutable objects, so isolation of local heaps, not inherent immutability, enables a place in Racket to independently garbage-collect a private heap.

Manticore (Fluet et al. 2008) is designed for parallelism from the start. Like Erlang and Haskell, Manticore has no mutable datatypes. In contrast, places add parallelism to an existing language with mutable datatypes. As the implementation of places matures, we hope to add multi-level parallelism similar to Manticore.

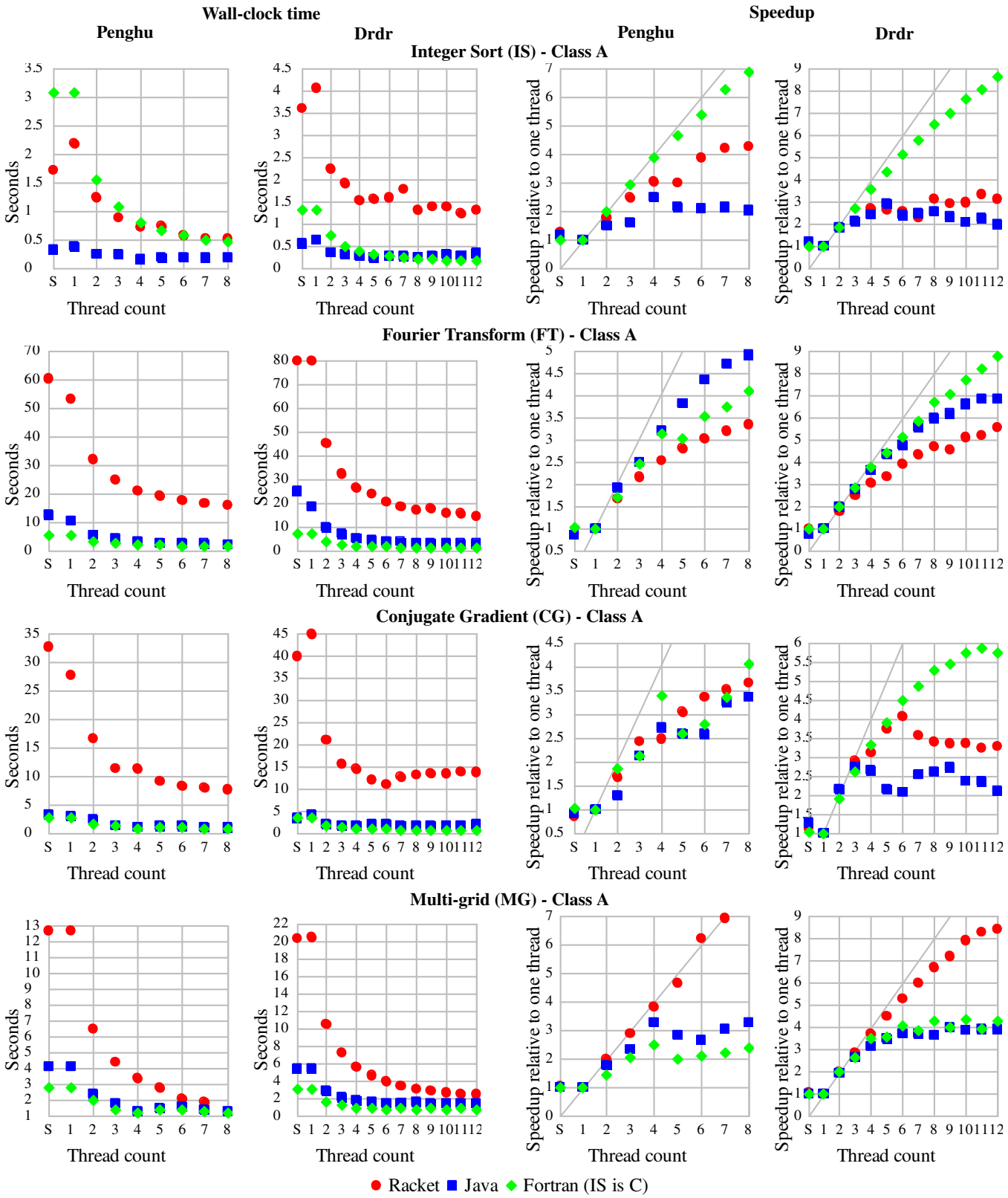


Figure 6: IS, FT, CG, and MG results

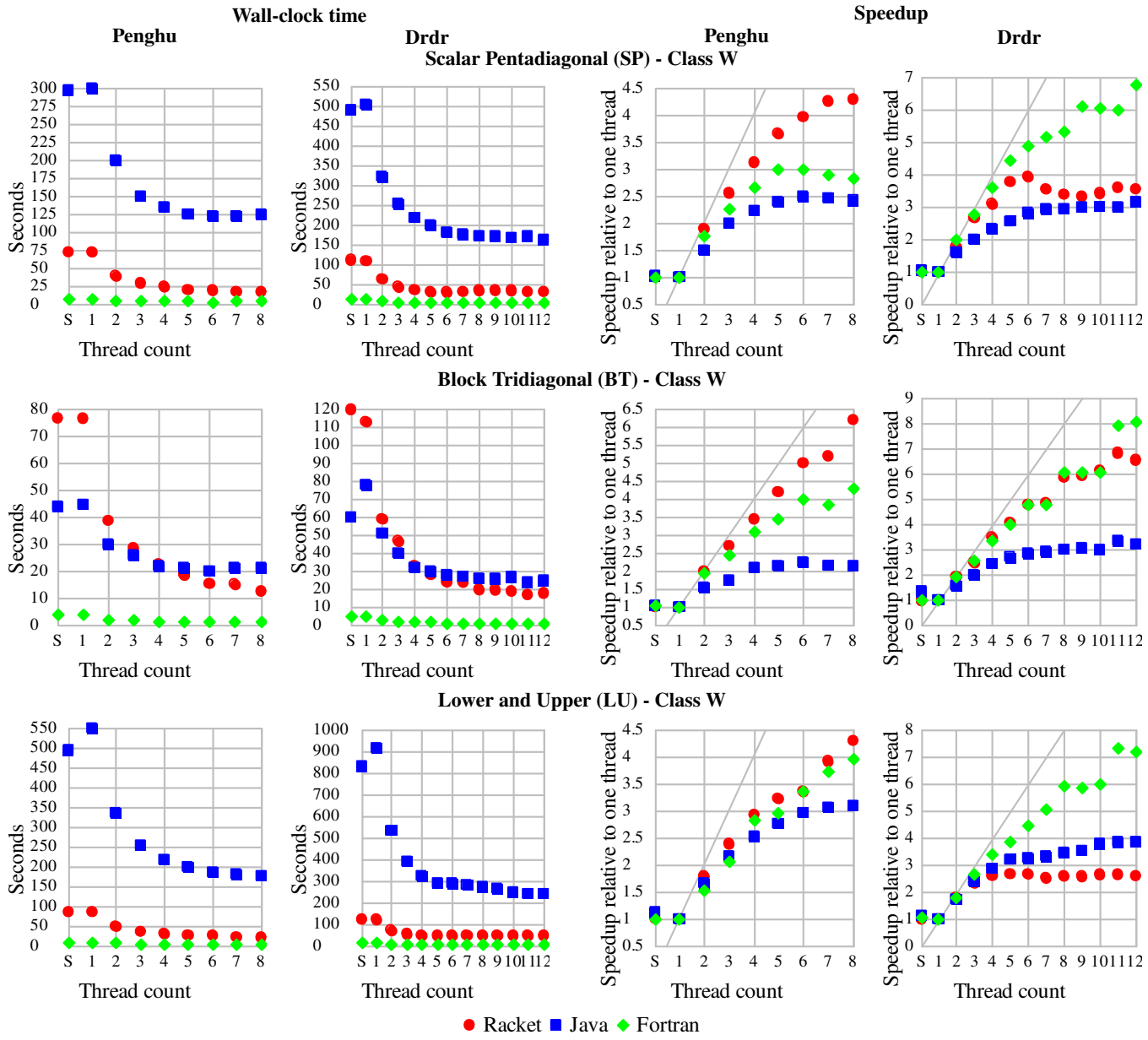


Figure 7: SP, BT, and LU results

Matlab provides programmers with several parallelism strategies. First, compute intensive functions, such as BLAS matrix operations, are implemented using multi-threaded libraries. Simple Matlab loops can be automatically parallelized by replacing `for` with `parfor`. Matlab's automatic parallelization can handle reductions such as min, max and sum, but it does not parallelize loop dependence. Matlab also provides task execution on remote Matlab instances and MPI functionality. Rather than adding parallelism through libraries and extensions, places integrate parallelism into the core of the Racket runtime.

Python's multiprocessing library (Python Software Foundation 2011) provides parallelism by forking new processes, each of which has a copy of the parent's state at the time of the fork. In contrast, a Racket place is conceptually a pristine instance of the virtual machine, where the only state a place receives from its creator is its starting module and a communication channel. More generally, however, Python's multiprocessing library and Racket's places both add parallelism to a dynamic language without retrofitting the language with threads and locks.

Communication between Python processes occurs primarily through OS pipes. The multiprocessing library includes a shared-queue implementation, which is implemented by using a worker thread to send messages over pipes to the recipient process. Any "picklable" (serializable) python object can be sent through a multiprocessing pipe or queue. Python's multiprocessing library also provides shared-memory regions implemented via `mmap()`. Python's pipes, queues and shared-memory regions must be allocated prior to forking children, which need to use them. Racket's approach offers more flexibility in communication; channels and shared-memory vectors can be created and sent over channels to

already-created places; and channels can communicate immutable data without the need for serialization.

Python and Ruby implementors, like Racket implementors, have tried and abandoned attempts to support OS-scheduled threads with shared data (Beazley 2010; Python Software Foundation 2008; Schuster July 31, 2009). All of these languages were implemented on the assumption of a single OS thread—which was a sensible choice for simplicity and performance throughout the 1990s and early 2000s—and adding all of the locks needed to support OS-thread concurrency seems prohibitively difficult. A design like places could be the right approach for those languages, too.

X10 (Charles et al. 2005) is a partitioned global address space (PGAS) language whose sequential language is largely taken from Java. Although our use of the term "place" is inspired by X10, places are more static in X10, in that the number of places within an X10 program is fixed at startup. Like Racket places, objects that exist at an X10 place are normally manipulated only by tasks within the place. X10 includes an `at` construct that allows access to an object in one place from another place, so `at` is effectively the communication construct for places in X10. Racket's message-passing communication is more primitive, but also more directly directly exposes the cost of cross-place communication. We could implement something like X10's cross-place references and `at` on top of Racket's message-passing layer.

8. Conclusion

Places in Racket demonstrate how adding a message-passing layer to an existing runtime system can provide effective support for parallelism with a reasonable implementation effort. Our benchmark results demonstrate good scaling on traditional parallel tasks, and the use of places for parallel library compilation demonstrates that the implementation holds up in real-world use. We are currently developing new tools based on places, including background parsing and compilation of programs within DrRacket.

Although places are primarily designed for message-passing parallelism, shared mutable vectors of bytes, fixnums, or floating-point numbers are also supported; careful programmers may have good reasons to use these structures. Crucially, shared vectors of atomic data create few problems for the language implementation, so they are easily accommodated by the places API. Meanwhile, the Racket implementation is free to implement message-passing of immutable objects through sharing, if the trade-off in implementation complexity versus performance favors that direction, since sharing of immutable data is safe.

We are particularly convinced that places are a better model than the conventional "add threads; add locks until it stops crashing; remove locks until it scales better; repeat" approach to programming-language concurrency. Simply running the Racket test suite in multiple places uncovered the vast majority of bugs in our implementation. The same has not been true of our attempts to support concurrency with shared memory (e.g., with futures). Indeed, there seems to be no comparably simple way to find race conditions with threads and locks; many tools have been designed to help programmers find concurrency bugs, and many—from Eraser (Savage et al. 1997) to GAMBIT (Coons et al. 2010)—but they suffer from problems with false positives, restrictions on supported code, problems scaling to large systems, or requiring assertions or other manual annotations. In contrast, bugs in the implementation of places were easy to find because they create permanently broken references (that are detected by the garbage collector) rather than fleeting timing effects.

Acknowledgments Thanks to Jay McCarthy for access to the 12-core machine we used to run our experiments and the anonymous DLS reviewers for their constructive suggestions. This work was supported by the NSF.

Bibliography

David Bailey, John Barton, Thomas Lasinski, and Horst Simon. The NAS Parallel Benchmarks. NAS Technical Report RNR-91-002, Aug. 1991.

David Beazley. Understanding the Python GIL. PyCon 2010, 2010.

Phillippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarker. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *Proc. ACM Intl. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, 2005.

Katherine E. Coons, Sebastian Burckhardt, and Madanlal Musuvathi. GAMBIT: Effective Unit Testing for Concurrency Libraries. In *Proc. ACM Symp. Principles and Practice of Parallel Programming*, 2010.

Damien Doligez and Xavier Leroy. A Concurrent, Generational Garbage Collector for a Multithreaded Implementation of ML. In *Proc. ACM Symp. Principles of Programming Languages*, 1993.

Matthew Flatt and Robert Bruce Findler. Kill-Safe Synchronization Abstractions. In *Proc. ACM Conf. Programming Language Design and Implementation*, 2004.

Matthew Flatt, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen. Programming Languages as Operating Systems (or, Revenge of the Son of the Lisp Machine). In *Proc. ACM Intl. Conf. Functional Programming*, 1999.

Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Implicitly-threaded parallelism in Manticore. In *Proc. ACM Intl. Conf. Functional Programming*, 2008.

Simon Marlow, Tim Harris, Roshan P. James, and Simon Peyton Jones. Parallel Generational-copying Garbage Collection with a Block-structured Heap. In *Proc. Intl. Symp. on Memory Management*, 2008.

Simon Marlow, Simon Peyton Jones, and Satnam Singh. Runtime Support for Multicore haskell. In *Proc. ACM Intl. Conf. Functional Programming*, 2009.

George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proc. Intl. Conf. Compiler Construction*, pp. 213–228, 2002.

Python Software Foundation. Python design note on threads. http://www.python.org/doc/faq/library/#can-t-we-get-rid-of-the-global-interpreter-lock, 2008.

Python Software Foundation. multiprocessing — Process-based "threading" interface. http://docs.python.org/release/2.6.6/library/multiprocessing.html#module-multiprocessing, 2011.

John H. Reppy. Concurrent Programming in ML. Cambridge University Press, 1999.

Konstantinos Sagonas and Jesper Wilhelmsson. Efficient Memory Management for Concurrent Programs that use Message Passing. *Science of Computer Programming* 62(2), pp. 98–121, Oct. 2006.

Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multi-threaded Programs. *Transactions on Computer Systems* 15(4), pp. 391–411, 1997.

Werner Schuster. Future of the Threading and Garbage Collection in Ruby - Interview with Koichi SasadaThread State and the Global Interpreter Lock. http://www.infoq.com/news/2009/07/future-ruby-gc-gvl-gil, July 31, 2009.

Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. *Communications of the ACM* 53(7), pp. 89–97, July 2010.

James Swaine, Kevin Tew, Peter Dinda, Robert Bruce Findler, and Matthew Flatt. Back to the futures: Incremental Parallelization of Existing Sequential Runtime Systems. In *Proc. ACM Intl. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, 2010.

Adam Wick and Matthew Flatt. Memory Accounting without Partitions. In *Proc. Intl. Symp. on Memory Management*, 2004.