Back to the Futures: Incremental Parallelization of Existing Sequential Runtime Systems

James Swaine
Northwestern University
JamesSwaine2010@u.northwestern.edu

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

Peter Dinda
Northwestern University
pdinda@northwestern.edu

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

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

Abstract

Many language implementations, particularly for high-level and scripting languages, are based on carefully honed runtime systems that have an internally sequential execution model. Adding support for parallelism in the usual form—as threads that run arbitrary code in parallel—would require a major revision or even a rewrite to add safe and efficient locking and communication. We describe an alternative approach to incremental parallelization of runtime systems. This approach can be applied inexpensively to many sequential runtime systems, and we demonstrate its effectiveness in the Racket runtime system and Parrot virtual machine. Our evaluation assesses both the performance benefits and the developer effort needed to implement our approach. We find that incremental parallelization can provide useful, scalable parallelism on commodity multicore processors at a fraction of the effort required to implement conventional parallel threads.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors — Run-time environments

General Terms Languages, Performance

Keywords Racket, functional programming, parallel programming, runtime systems

1. Runtime Systems from a Sequential Era

Many modern high-level or scripting languages are implemented around an interpretive runtime system, often with a JIT compiler. Examples include the Racket [11] (formerly PLT Scheme) runtime system, the Parrot virtual machine, and the virtual machines underlying Perl, Python, Ruby, and other productivity-oriented languages. These runtime systems are often the result of many man-years of effort, and they have been carefully tuned for capability, functionality, correctness, and performance.

For the most part, such runtime systems have *not* been designed to support parallelism on multiple processors. Even when a language supports constructs for concurrency, they are typically implemented through co-routines or OS-level threads that are constrained to execute one at a time. This limitation is becoming a serious issue, as it is clear that exploiting parallelism is essential to harnessing performance in future processor generations. Whether computer architects envision the future as involving homogeneous or heterogeneous multicores, and with whatever form of memory coherence or consistency model, the common theme is that the future is parallel and that language implementations must adapt. The essential problem is making the language implementation safe for low-level parallelism, i.e., ensuring that even when two threads are modifying internal data structures at the same time, the runtime system behaves correctly.

One approach to enabling parallelism would be to allow existing concurrency constructs to run in parallel, and to rewrite or revise the runtime system to carefully employ locking or explicit communication. Our experience with that approach, as well as the persistence of the global interpreter lock in implementations for Python and Ruby, suggests that such a conversion is extremely difficult to perform correctly. Based on the even longer history of experience in parallel systems, we would also expect the result to scale poorly as more and more processors become available. The alternative of simply throwing out the current runtime and re-designing and implementing it around a carefully designed concurrency model is no better, as it would require us to discard

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

OOPSLA/SPLASH'10, October 17–21, 2010, Reno/Tahoe, Nevada, USA.

Copyright © 2010 ACM 978-1-4503-0203-6/10/10...$10.00

years or decades of effort in building an effective system, and this approach also risks losing much of the language's momentum as the developers are engaged in tasks with little visible improvement for a long period.

In this paper, we report on our experience with a new technique for parallelizing runtime systems, called *slow-path barricading*. Our technique is based on the observation that the core of many programs—and particularly the part that runs fast sequentially and could benefit most from parallelism—involves relatively few side effects with respect to the language implementation's internal state. Thus, instead of wholesale conversion of the runtime system to support arbitrary concurrency, we can add language constructs that focus and restrict concurrency where the implementation can easily support it.

Specifically, we partition the set of possible operations in a language implementation into *safe* (for parallelism) and *unsafe* categories. We then give the programmer a mechanism to start a parallel task; as long as the task sticks to safe operations, it stays in the so-called *fast path* of the implementation and thus is safe for parallelism. As soon as the computation hits a barricade, the runtime system suspends the computation until the operation can be handled in the more general, purely sequential part of the runtime system.

Although the programming model allows only a subset of language operations to be executed in parallel, this subset roughly corresponds to the set of operations that the programmer already knows (or should know) to be fast in sequential code. Thus, a programmer who is reasonably capable of writing a fast programs in the language already possesses the knowledge to write a program that avoids unsafe operations—and one that therefore exhibits good scaling for parallelism. Furthermore, this approach enables clear feedback to the programmer about when and how a program uses unsafe operations.

We continue our presentation by discussing parallel computation in Racket and introducing our variant of *futures* (Section 2), which we use as the programmer's base mechanism for spawning a parallel task. We then explain in general terms our approach to incremental parallelization (Section 3). The core of our work is the implementation of futures in the Racket runtime system (Section 4) and Parrot virtual machine (Section 5). Our Racket implementation is more mature and is included in the current release. We evaluate the implementations both in terms of developer effort to create them, and in terms of their raw performance and scaling for simple parallel benchmarks, including some of the NAS parallel benchmarks (Section 6).

The contributions of this work are the slow-path barricading technique itself, as well as an evaluation of both how difficult it is to add to Racket and Parrot and the performance and scalability of the resulting runtime systems. Our evaluation suggests that the developer effort needed to use the approach is modest and that the resulting implementations have reasonable raw performance and scaling properties.

2. Parallelism through Futures

Racket provides `future` to start a parallel computation and `touch` to receive its result:

$$\texttt{future} : (\rightarrow\ \alpha) \rightarrow \alpha\text{-}\textit{future}$$
$$\texttt{touch} : \alpha\text{-}\textit{future} \rightarrow \alpha$$

The `future` function takes a thunk (i.e., a procedure with no arguments) and may start evaluating it in parallel to the rest of the computation. The `future` function returns a future descriptor to be supplied to `touch`. The `touch` function waits for the thunk to complete and returns the value that the thunk produced. If `touch` is applied to the same future descriptor multiple times, then it returns the same result each time, as computed just once by the future's thunk.

For example, in the function

```
(define (f x y)
  (* (+ x y) (- x y)))
```

the expressions `(+ x y)` and `(- x y)` are independent. They could be computed in parallel using `future` and `touch` as follows:

```
(define (f x y)
  (let ([s (future (lambda () (+ x y)))]
        [d (future (lambda () (- x y)))])
    (* (touch s) (touch d))))
```

The main computation can proceed in parallel to a future, so that the variant

```
(define (f x y)
  (let ([d (future (lambda () (- x y)))])
    (* (+ x y) (touch d))))
```

indicates as much parallelism as the version that uses two futures, since the addition `(+ x y)` can proceed in parallel to the future. In contrast, the variant

```
(define (f x y)
  (let ([s (future (lambda () (+ x y)))])
    (* (touch s) (- x y))))
```

includes no parallelism, because Racket evaluates expressions from left to right; `(- x y)` is evaluated only after the `(touch s)` expression.

A future's thunk is not necessarily evaluated in parallel to other expressions. In particular, if the thunk's computation relies in some way on the evaluation context, then the computation is suspended until a `touch`, at which point the computation continues in the context of the `touch`. For example, if a future thunk raises an exception, the exception is raised at the point of the `touch`. (If an exception-raising

```
(define MAX-ITERS 50)
(define MAX-DIST 2.0)
(define N 1024)

(define (mandelbrot-point x y)
  (define c (+ (- (/ (* 2.0 x) N) 1.5)
               (* +i (- (/ (* 2.0 y) N) 1.0))))
  (let loop ((i 0) (z 0.0+0.0i))
    (cond
     [(> i MAX-ITERS) (char->integer #\*)]
     [(> (magnitude z) MAX-DIST)
      (char->integer #\space)]
     [else (loop (add1 i) (+ (* z z) c))])))

(for ([y (in-range N)])
  (for ([x (in-range N)])
    (write-byte (mandelbrot-point x y)))
  (newline))
```

Figure 1. Sequential Mandelbrot plotting

future is `touch`ed a second time, the second attempt raises a fresh exception to report that no value is available.)

A future's thunk can perform side effects that are visible to other computations. For example, after

```
(define x 0)
(define (inc!) (set! x (+ x 1)))
(let ([f1 (future inc!)]
      [f2 (future inc!)])
  (touch f1)
  (touch f2))
```

the possible values of `x` include 0, 1, and 2. The `future` and `touch` operations are intended for use with thunks that perform independent computations, though possibly storing results in variables, arrays or other data structures using side effects.

The `f` example above has no *useful* parallelism, because the work of adding or subtracting numbers is far simpler than the work of creating a parallel task and communicating the result. For an example with potentially useful parallelism, Figure 1 shows a simple Mandelbrot-set rendering program, a classic embarrassingly parallel computation.

In an ideal language implementation, the Mandelbrot computation could be parallelized through a future for each point. Figure 2 shows such an implementation, where `for/list` is a list-comprehension form that is used to create a list of list of futures, and then each future is `touch`ed in order. This approach does not improve performance, however. Although a call to `mandelbrot-point` involves much more computation than `(+ x y)`, it is still not quite as much as the computation required for `future` plus `touch`.

Figure 3 shows a per-line parallelization of the Mandelbrot computation. Each line is rendered independently to a buffer, and then the buffered lines are written in order. This approach is typical for a system that supports parallelism, and it is a practical approach for the Mandelbrot program in Racket.

```
(define fss
  (for/list ([y (in-range N)])
    (for/list ([x (in-range N)])
      (future
        (lambda () (mandelbrot-point x y))))))
(for ([fs (in-list fss)])
  (for ([f (in-list fs)])
    (write-byte (touch f)))
  (newline)))
```

Figure 2. Naive Mandelbrot parallelization

```
(define fs
  (for/list ([y (in-range N)])
    (let ([bstr (make-bytes N)])
      (future
        (lambda ()
          (for ([x (in-range N)])
            (bytes-set! bstr x (mandelbrot-point x
                 y)))
          bstr)))))
(for ([f (in-list fs)])
  (write-bytes (touch f))
  (newline)))
```

Figure 3. Per-line Mandelbrot parallelization

Perhaps surprisingly, then, the per-line refactoring for Mandelbrot rendering runs much slower than the sequential version. The problem at this point is not the decomposition approach or inherent limits in parallel communication. Instead, the problem is due to the key compromise between the implementation of Racket and the needs of programmers with respect to parallelization. Specifically, the problem is that complex-number arithmetic is currently treated as a "slow" operation in Racket, and the implementation makes no attempt to parallelize slow operations, since they may manipulate shared state in the runtime system. Programmers must learn to avoid slow operations within parallel tasks—at least until incremental improvements to the implementation allow the operation to run in parallel.

A programmer can discover the slow operation in this case by enabling debugging profiling, which causes `future` and `touch` to produce output similar to

```
future: 0 waiting for runtime at 126.741: *
future: 0 waiting for runtime at 126.785: /
```

The first line of this log indicates that a future computation was suspended because the `*` operation could not be exe-

```
(define (mandelbrot-point x y)
  (define ci
    (fl- (fl/ (fl* 2.0 (->fl y))(->fl N)) 1.0))
  (define cr
    (fl- (fl/ (fl* 2.0 (->fl x))(->fl N)) 1.5))
  (let loop ((i 0) (zr 0.0) (zi 0.0))
    (if (> i MAX-ITERS)
        (char->integer #\*)
        (let ((zrq (fl* zr zr))
              (ziq (fl* zi zi)))
          (if (fl> (fl+ zrq ziq)
                   (expt MAX-DIST 2))
              (char->integer #\space)
              (loop (add1 i)
                    (fl+ (fl- zrq ziq) cr)
                    (fl+ (fl* 2.0 (fl* zr zi))
                         ci)))))))
```

Figure 4. Mandelbrot core with flonum-specific operations

cuted in parallel. A programmer would have to consult the documentation to determine that `*` is treated as a slow operation when it is applied to complex numbers.

Another way in which an operation can be slow in Racket is to require too much allocation. Debugging-log output of the form

```
future: 0 waiting for runtime at 126.032:
  [acquire_gc_page]
```

indicates that a future computation had to synchronize with the main computation to allocate memory. Again, the problem is a result of an implementation compromise, because Racket's memory allocator is basically sequential, although moderate amounts of allocation can be performed in parallel.

Figure 4 shows a version of `mandelbrot-point` for which per-line parallelism offers the expected performance improvement. It avoids complex numbers, and it also uses flonum-specific arithmetic (i.e., operations that consume and produce only floating-point numbers). Flonum-specific operations act as a hint to help the compiler "unbox" intermediate flonum results—keeping them in registers or allocating them on a future-local stack, which avoids heap allocation. (In a language with a static type system, types provide the same hint.) As a result, in sequential mode, this version runs about 30 times as fast as the original version; a programmer who needs performance will always prefer it, whether using futures or not. Meanwhile, for much the same reason that it can run fast sequentially, this version also provides a speed-up when run in parallel.

All else being equal, obtaining performance through parallelism is no easier in our design for futures than in other systems for parallel programming. The programmer must still understand the relative cost of computation and communication, and the language's facilities for sequential performance should be fully deployed before attempting parallelization. All else is *not* equal, however; converting the initial Racket program to one that performs well is far simpler than, say, porting the program to C. For more sophisticated programs, where development in Racket buys productivity from the start, futures provide a transition path to parallelism that keep those productivity benefits intact. Most importantly, our approach to implementing futures makes these benefits available at a tractable cost for the implementer of the programming language.

3. Implementing Futures

Since `future` does not promise to run a given thunk in parallel, a correct implementation of `future` and `touch` is easily added to any language implementation; the result of `future` can simply encapsulate the given thunk, and `touch` can call the thunk if no previous `touch` has called it. Of course, the trivial implementation offers no parallelism. At the opposite extreme, in an ideal language implementation, `future` would immediately fork a parallel task to execute the given thunk—giving a programmer maximal parallelism, but placing a large burden on the language implementation to run arbitrary code concurrently.

The `future` and `touch` constructs are designed to accommodate points in between these two extremes. The key is to specify when computations can proceed in parallel in a way that is (1) simple enough for programmers to reason about and rely on, and (2) flexible enough to accommodate implementation limitations. In particular, we are interested in starting with an implementation that was designed to support only sequential computation, and we would like to gradually improve its support for parallelism.

To add futures to a given language implementation, we partition the language's set of operations among three categories:

- A *safe* operation can proceed in parallel to any other computation without synchronization. For example, arithmetic operations are often safe. An ideal implementation categorizes nearly all operations as safe.
- An *unsafe* operation cannot be performed in parallel, either because it might break guarantees normally provided by the language, such as type safety, or because it depends on the evaluation context. Its execution must be deferred until a `touch` operation. Raising an exception, for example, is typically an unsafe operation. The simplest, most conservative implementation of futures categorizes all operations as unsafe, thus deferring all computation to `touch`.
- A *synchronized* operation cannot, in general, run in parallel to other tasks, but by synchronizing with other tasks, the operation can complete without requiring a `touch`. It thus allows later safe operations to proceed in parallel.

Operations that allocate memory, for example, are often synchronized.

In a language like Racket, the key to a useful categorization is to detect and classify operations dynamically and at the level of an operator plus its arguments, as opposed to the operator alone. For example, addition might be safe when the arguments are two small integers whose sum is another small integer, since small integers are represented in Racket as immediates that require no allocation. Adding an integer to a string is unsafe, because it signals an error and the corresponding exception handling depends on the context of the `touch`. Adding two floating-point numbers, meanwhile, is a synchronized operation if space must be allocated to box the result; the allocation will surely succeed, but it may require a lock in the allocator or a pause for garbage collection.

This partitioning strategy works in practice because it builds on an implicit agreement that exists already between a programmer and a language implementation. Programmers expect certain operations to be fast, while others are understood to be slow. For example, programmers expect small-integer arithmetic and array accesses to be fast, while arbitrary-precision arithmetic or dictionary extension are relatively slow. From one perspective, implementations often satisfy such expectations though "fast paths" in the interpreter loop or compiled code for operations that are expected to be fast, while other operations can be handled through a slower, more generic implementation. From another perspective, programmers learn from experimentation that certain operations are fast, and those operations turn out to be fast because their underlying implementations in the runtime have been tuned to follow special, highly optimized paths.

The key insight behind our work is that fast paths in a language implementation tend to be safe to run in parallel and that it is not difficult to barricade slow paths, preventing them from running in parallel. An implementation's existing internal partitioning into fast and slow paths therefore provides a natural first cut for distinguishing safe and unsafe operations. Our implementation strategy is to set up a channel from `future` to the language implementation's fast path to execute a future in parallel. If the future's code path departs from the fast path, then the departing operation is considered unsafe, and the computation is suspended until it can be completed by `touch`.

The details of applying our technique depend on the language implementation. Based on our experience converting two implementations and our knowledge of other implementations, we expect certain details to be common among many implementations. For example, access to parallelism normally builds on a POSIX-like thread API. Introducing new threads of execution in a language implementation may require that static variables within the implementation are converted to thread-local variables. The memory manager may need adjustment to work with multiple threads; as a first cut, all allocation can be treated as an unsafe slow path. To

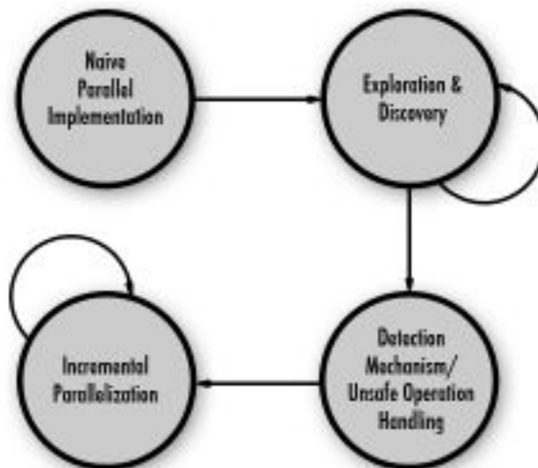


Figure 5. Incremental parallelization methodology

support garbage collection and similar global operations, the language implementation's fast path needs hooks where the computation can be paused or even shifted to the slow path.

Figure 5 illustrates our general methodology. We start by adding low-level support for parallelism, then experimenting with the paths in the implementation that are affected. Based on that exploration, we derive a partitioning of the language's operations into safe and unsafe. Having partitioned the operations, we must implement a trapping mechanism capable of suspending a parallel task before it executes an unsafe operation. Finally, we refine our partitioning of the operations (perhaps designating and implementing some operations as synchronized) as guided by the needs of applications.

In the following two sections, we report in detail on two experiments adding futures to an existing language. Our first and more extensive experiment is in adding futures to Racket. We also added a simple version of futures to the Parrot virtual machine in order to test whether the lessons from Racket generalized.

4. Adding Futures to Racket

The Racket runtime system is implemented by roughly 100k lines of C code. It includes a garbage collector, macro expander, bytecode compiler, bytecode interpreter, just-in-time (JIT) compiler, and core libraries. The core libraries include support for threads that run concurrently at the Racket level, but internally threads are implemented as co-routines (i.e., they are "user threads").

Execution of a program in the virtual machine uses a stack to manage the current continuation and local bindings. Other execution state, such as exception handlers and dynamic bindings, are stored in global variables within the virtual-machine implementation. Global data also includes the symbol table, caches for macro expansion, a registry of JIT-generated code, and the garbage collector's meta-data. In

addition, some primitive objects, such as those representing I/O streams, have complex internal state that must be managed carefully when the object is shared among concurrent computations.

The virtual machine's global and shared-object states present the main obstacles to parallelism for Racket programs. An early attempt to implement threads as OS-level threads—which would provide access to multiple processors and cores as managed by the operating system—failed due to the difficulty of installing and correctly managing locks within the interpreter loop and core libraries. Since that early attempt, the implementation of Racket has grown even more complex.

A related challenge is that Racket offers first-class continuations, which allow the current execution state to be captured and later restored, perhaps in a different thread of execution. The tangling of the C stack with execution state means that moving a continuation from one OS-level thread to another would require extensive changes to representation of control in the virtual machine.

The design of futures side-steps the latter problem by designating operations that inspect or capture the current execution state as usafe; thus, they must wait until a `touch`. Meanwhile, the notions of unsafe and synchronized operations correspond to using a single "big lock" to protect other global state in the virtual machine.

The following sections provide more details on how we adjusted the implementation of execution state and categorized operations in Racket while limiting the changes that were needed overall.

4.1 Compilation, Execution, and Safety Categorization

Execution of a Racket program uses two phases of compilation. First, a bytecode compiler performs the usual optimizations for functional languages, including constant and variable propagation, constant folding, inlining, loop unrolling (a special case of inlining in Racket), closure conversion, and unboxing of intermediate floating-point results. The bytecode compiler is typically used ahead of time for large programs, but it is fast enough for interactive use. Second, when a function in bytecode form is called, a JIT compiler converts the function into machine code. The JIT creates inline code for simple operations, including type tests, arithmetic on small integers or flonums, allocations of cons cells, array accesses and updates, and structure-field operations. When the JIT compiler is disabled, bytecode is interpreted directly.

The first step in supporting useful parallelism within Racket was to make the execution-state variables thread-local at the level of OS threads, so that futures can be executed speculatively in new OS-level threads. To simplify this problem, we confined our attention to the execution state that is used by JIT-generated code. Consequently, our first cut at categorizing operations was to define as safe any operation that is implemented directly in JIT-generated code (e.g. any operation that can be translated by the JIT compiler into machine instructions which do not include function calls back into runtime code), and any other operation was unsafe. This first-cut strategy offered a convenient starting point for the incremental process, in which we modify performance-critical unsafe operations to make them future-safe.

```
typedef (*primitive)(int argc,
                     Scheme_Object **argv);

Scheme_Object *handler(int argc,
                       Scheme_Object **argv,
                       primitive func) {
  Scheme_Object *retval;
  if (pthread_self() != g_runtime_thread_id) {
    /* Wait for the runtime thread */
    retval = do_runtimecall(func, argc, argv);
    return retval;
  } else {
    /* Do the work directly */
    retval = func(argc, argv);
    return retval;
  }
}
```

Figure 6. Typical primitive trap handler

When a future is created in Racket, the corresponding thunk is JIT-compiled and then added to a queue of ready futures. The queue is served by a set of OS-level future threads, each of which begins execution of the JIT-generated code. At points where execution would exit JIT-generated code, a future thread suspends to wait on the result of an unsafe operation. The operation is eventually performed by the original runtime thread when it executes a `touch` for the future. In our current implementation, a future thread remains blocked as long as it waits for the runtime thread to complete an unsafe operation.

Since the JIT compiler was designed to work for a non-parallelized runtime system, the code that it generates uses several global variables to manage execution state. In some cases, state is kept primarily in a register and occasionally synchronized with a global variable. To allow multiple JIT-generated code blocks to execute in parallel, we changed the relevant global variables to be thread-local variables in the C source of the Racket implementation.[1] We then adjusted the JIT compiler to access a global variable through an thread-specific indirection. The thread-specific indirection is supplied on entry to JIT-generated code.

4.2 Handling Unsafe Operations

When JIT-generated code invokes an operation that is not implemented inline, it invokes one of a handful of C func-

[1] On some platforms, we could simply annotate the variable declaration in C with `__thread`. On other platforms, we use pre-processor macros and inline assembly to achieve similar results.

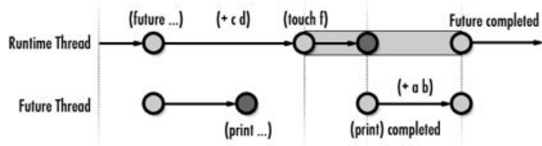


Figure 7. Timeline for a future with an unsafe operation

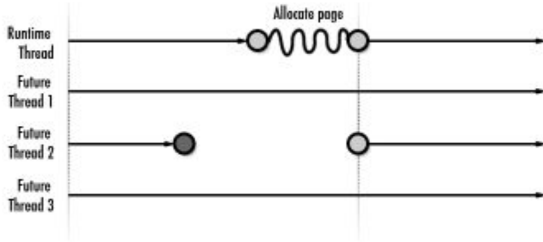


Figure 8. Timeline for a synchronized operation

tions that call back into the general interpreter loop. When a future thread takes this path out of JIT-generated code, the call is redirected to send the call back to the runtime thread and wait for a response. Figure 6 illustrates the general form of such functions. Each first checks whether it is already executing on the runtime thread. If so, it performs the external call as usual. If not, the work is sent back to the runtime thread via `do_runtimecall`.

The runtime thread does not execute the indirect call until `touch` is called on the corresponding future. Figure 7 illustrates the way that an unsafe operation suspends a future thread until its value can be computed by the runtime thread in response to a `touch`. Note that the `touch` function itself is considered unsafe, so if `touch` is called in a future thread, then it is sent back to the runtime thread. Thus, the `touch` function need only work in the runtime thread.

4.3 Synchronized Operations

Like unsafe operations, synchronized operations always run on the runtime thread. Unlike unsafe operations, however, the runtime thread can perform a synchronized operation on a future thread's behalf at any time, instead of forcing the future thread to wait until `touch` is called.

As part of its normal scheduling work to run non-parallel threads, the runtime system checks whether any future thread is waiting on a synchronized operation. If so, it immediately performs the synchronized operation and returns the result; all synchronized operations are short enough to be performed by the scheduler without interfering with thread scheduling.

Currently, the only synchronized operations are allocation and JIT compilation of a procedure that has not been called before. More precisely, allocation of small objects usually can be performed in parallel (as described in the next section), but allocation of large objects or allocation

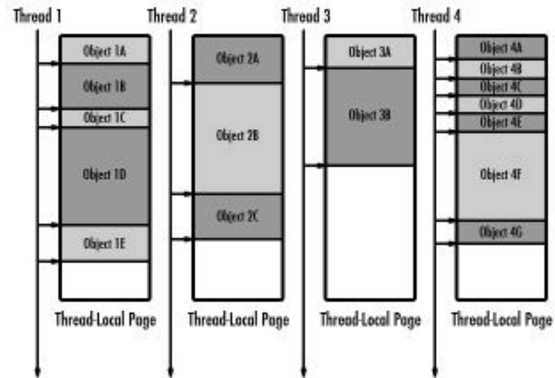


Figure 9. Per-future allocation

of a fresh page for small objects requires cooperation and synchronized with the memory manager. Figure 8 illustrates the synchronized allocation a new page with the help of the runtime thread.

4.4 Memory Management

Racket uses a custom garbage collector that, like the rest of the system, was written for sequential computation. Specifically, allocation updates some global state and collection stops the world. As in many runtime systems, the virtual machine and its garbage collector cooperate in many small ways that make dropping in a third-party concurrent garbage collector prohibitively difficult. Similarly, converting the garbage collector to support general concurrency would be difficult. Fortunately, adapting the collector to support a small amount of concurrency is relatively easy.

The garbage collector uses a nursery for new, small objects, and then compacting collection for older objects. The nursery enables inline allocation in JIT-generated code by bumping a pointer. That is, a memory allocation request takes one of the following paths:

- *Fast Path* — the current nursery page has enough space to accommodate the current request. In this case, a page pointer is incremented by the size of the object being allocated, and the original page pointer is returned to the caller. This path is executed purely in JIT-generated code.
- *Slow Path* — the current page being used by the allocator does not have enough space to accommodate the current request. In this case, a new page must either be fetched from either the virtual machine's own internal page cache, or must be requested from the operating system. If the entire heap space has been exhausted, a garbage collection is triggered.

The nursery itself is implemented as a collection of pages, so adding additional thread-specific pages was straightforward. As long as it is working on its own page, a future

thread can safely execute the inline-allocation code generated by the JIT compiler. Figure 9 illustrates the use of future-specific nurseries.

Acquiring a fresh nursery page, in contrast, requires synchronization with the runtime thread, as described in the previous section. The size of the nursery page adapts to the amount of allocation that is performed by the future requesting the page.

Garbage collection still requires stopping the world, which includes all future threads. The JIT compiler generates code that includes safe points to swap Racket-level threads. When JIT-generated code is running in a future, it never needs to stop for other Racket threads, but the same safe points can be repurposed as garbage-collection safe points. That is, the inlined check for whether the computation should swap threads is instead used as a check for whether the future thread should pause for garbage collection. Meanwhile, garbage collection in the runtime thread must not be allowed unless all future threads are blocked at a safe point.

Besides providing support for thread-specific nursery pages, the garbage collector required minor adjustments to support multiple active execution contexts to be treated as roots. Roughly, the implementation uses fake Racket threads that point to the execution state of a computation in a future thread.

4.5 Implementing touch

To tie all of the preceding pieces together, the implementation of `touch` is as follows:

- If the future has produced a result already, return it.
- If a previous `touch` of the future aborted (e.g., because the future computation raised an exception), then raise an exception.
- If the future has not started running in a future thread, remove it from the queue of ready futures and run it directly, recording the result (or the fact that it aborts, if it does so).
- If the future is running in a future thread, wait until it either completes or encounters an unsafe operation:
  - If the future thread has encountered an unsafe operation, perform the unsafe operation, return the result, and wait again. If performing the unsafe operation results in an exception or other control escape, tell the future thread to abort and record the abort for the future.
  - If the future completes in a future thread, record and return the result.

In addition, the scheduler loop must poll future threads to see if any are waiting on synchronized operations; if so, the operation can be performed and the result immediately returned to the future thread. By definition, a synchronized operation cannot raise an exception.

5. Adding Futures to Parrot

Parrot is a register-based virtual machine with heap-allocated continuation frames. Compilers target Parrot by emitting programs in the Parrot intermediate language, which is a low-level, imperative assembly-like programming language, but with a few higher-level features, including garbage collection, subroutines, dynamic container types, and a extensible calling convention.

Three key characteristics made adding futures to the Parrot VM machine relatively easy:

1. an existing abstraction and wrapping of OS-level threads;
2. a concurrency or green thread implementation that abstracts and encapsulates thread of execution state; and
3. a pluggable *runloop* (i.e., interpreter loop) construct that allows switching between different interpreter cores.

With such groundwork in place, the following enhancements to the Parrot C implementation were needed:

- refactoring of representation of threads to allow it to be reused for futures,
- allowing an unfinished future computation to be completed by the main interpreter after a OS-level join, and
- creating a new runloop that executes only safe (for parallelism) operations and reverts back to the main thread for unsafe operations.

In Parrot, spawning a `future` consists of creating a new interpreter and specifying what data, if any, to share between the parent and child OS-level threads. Parrot futures have their own execution stack, but they share the same heap and bytecodes. To implement `touch`, Parrot waits for the future thread to return and then checks to see if the future returned a value, in which case the value is returned. If the future encountered an unsafe instruction, the future thread returns a computation, which is completed in the caller.

Parrot's runloop is the core of the interpreter, where bytecodes are fetched and executed. Parrot has several different runloops that provide debugging, execution tracing, profiling, and experimental dispatching. Parallel futures adds a future runloop that checks each bytecode just before it is executed to see if it is safe to execute or if the future needs to be suspended and executed sequentially in the main interpreter. This runtime safety checking includes argument type checking and bounds checking on container data structures.

Parrot is a highly dynamic virtual machine. Many bytecodes are actually virtual function calls on objects that can be defined by the users. For example, Parrot has typical array get and set opcodes, but the opcode are translated into virtual method calls on a wide variety of container object types. Some container object types are a fixed size at construction

Task	Person hours expert	non-expert
General Steps		
Naive implementation	6	40
Exploration and discovery	-	480
Unsafe-operation handling	6	16
Blocked-future logging	1	-
Total General:	13	536
Implementation-specific Steps		
Thread-local variables	8	-
Future-local allocation	8	-
Garbage-collection sync	6	-
Thread-local performance	6	-
Total Specific:	28	-
Overall Total:	41	536

Figure 10. Racket implementation effort by task. Adding a release-quality futures implementation to Racket using our approach required only one week of expert time, and one academic quarter of non-expert time.

time; others grow dynamically. This indirection inherent in the bytecode makes compile-time safety checking difficult. To compensate, the future runloop does checks at run time that an operand container object is of the fixed-size variety (and thus safe to run in parallel), not a dynamically growing variant (which would not be safe to run in parallel).

For our Parrot experiment, we stopped short of implementing any operations as synchronized or implementing future-local allocation, and the Parrot future runloop treats most opcodes as unsafe operations. Arithmetic, jump, and fixed-container access and update are the only operations designated as safe—enough to run a parallel benchmark and check that it scales with multiple processors. Our experience working with the Parrot VM and Racket VM suggest that the work required to add those features to the Parrot VM would be very similar to the work we did to add them to the Racket VM and would achieve similar improvements in performance.

6. Evaluation

Our evaluation supports the hypothesis that our approach to incremental parallelization of existing, sequential runtime systems is (a) developer-efficient, and (b) leads to reasonable performance gains for parallelized programs. Our evaluation of the first claim is based on measurements of the overall amount of developer time (person-hours) invested in Racket and the Parrot VM implementation work. Our evaluation of the second claim is based on measuring both raw performance and scaling of several benchmarks, focusing primarily on the Racket implementation.

Task	Person hours expert
General Steps	
Naive implementation	8
Exploration and discovery	24
Unsafe-operation handling	8
Total General:	40
Implementation-specific Steps	
Wrapping OS-level threads	8
Future runloop	4
Total Specific:	12
Overall Total:	52

Figure 11. Parrot implementation effort by task. Adding a proof-of-concept implementation of futures to Parrot using our approach required only about a week of expert time.

6.1 Developer Effort is Low

Because our approach was developed in the context of the Racket runtime system, our parallel Racket implementation is the most mature. Given that, one might expect to see substantial development effort; however, this is not the case. Figure 10 lists the overall development costs (in person-hours) required to apply our approach to Racket. Costs are partitioned into two categories: general and implementation-specific. The general category reflects the first three steps described in Figure 5; the implementation-specific category reflects the incremental parallelization step, which includes work that was necessary for Racket, but may not apply in other runtime adaptation work.

The columns in Figure 10 show the efforts of two developers, the expert being the designer and primary implementer of the Racket runtime and the non-expert being a first-year graduate student (working in a different city). Notice that with roughly one week of expert time, and one academic quarter of non-expert time, it was possible to apply our approach to a widely used,[2] mature[3] runtime system and achieve reasonable performance results. Furthermore, the effort produced a parallel futures implementation that is now a part of the main-line release of Racket.

Having gained experience with our approach, we also applied it to the Parrot VM to ensure our experience is not Racket-specific. Figure 11 lists the development time we required to add a first-cut futures implementation. The Parrot effort was undertaken by a third-year Ph.D. student who, while not a core developer of Parrot, is intimately familiar with its internals. He also was familiar with the Racket effort. The upshot of Figure 11 is that adding a proof-of-concept implementation of futures to Parrot using our approach required only about a week of expert time.

[2] The Racket distribution is downloaded more than 300 times per day.

[3] The runtime system has been in continuous development since 1995.

	Penghu	Cosmos
OS	OS X 10.6.2	CentOS 5.4
Processor Type	Intel Xeon	AMD Opteron 8350
Processors	2	4
Total Cores	8	16
Clock Speed	2.8 GHz	2.0 GHz
L2 Cache	12 MB	4x512 KB
Memory	8 GB	16 GB
Bus Speed	1.6 GHz	1 GHz
Racket	5.0 32-bit mode	5.0 64-bit mode
GCC	4.2.1 (Apple)	4.1.2 (Red Hat)
Java	Java SE 1.6	OpenJDK 1.6

Figure 12. Machine configurations used for benchmarks

That the same approach has been applied successfully and efficiently to two very different runtime systems suggests that it is quite general.

6.2 Testbed, Metrics, and Benchmarks

We evaluated the performance of our futures implementations using two different machines and five benchmarks. We use the commonplace parallel systems performance metrics of strong scalability and raw performance, and we also compare our results against the same algorithms implemented in other languages.

Machines

We conducted performance evaluations on both a high-end desktop workstation with two quad-core processors (8 cores), and a mid-range server machine with four quad-core processors (16 cores). The detailed configuration of these machines is given in Figure 12. During the execution of a given benchmark, no other significant load was placed on the machine. We verified that separate threads of execution used by the runtime system were in fact mapped to separate processing cores.

Metrics

The number of threads used by each of our benchmarks is a runtime parameter. We measured the wall-clock execution time of each benchmark as a function of this parameter, and we present both the raw numbers and a speedup curve. The speedup curve shows the wall-clock time of the parallel implementation using a single thread divided by the wall-clock time of the parallel implementation using the indicated number of threads. The problem size remains constant as we increase the number of threads; thus the speedup curve measures "strong scaling."

For several benchmarks, we also measured the wall-clock time of sequential implementations in various languages, including optimized C and Java.

Program	Implementation
Microbenchmarks (self-developed)	
MV-Sparse	Racket
Mergesort	Racket
Signal Convolution	Racket, Parrot
NAS Parallel Benchmarks [4]	
Integer Sort	Racket, Java
Fourier Transform	Racket, Java

Figure 13. Benchmarks, sources, and parallel implementations. Sequential implementations in C, Java, Racket, and Parrot are also used for comparison.

Benchmarks

Figure 13 lists the benchmarks we used to evaluate the performance of parallel futures in the Racket and Parrot VM implementations. Several of the evaluation programs are not drawn from a particular benchmark suite; rather, they are common implementations of well-known algorithms. Note that not all programs have a Parrot VM implementation.

Signal convolution is a signal-processing algorithm used to determine the output signal of a system given its *impulse response* or *kernel*, which defines how the system will respond given an impulse function applied to the input signal. For any input signal x, we can compute each value in the corresponding output signal y using the following equation:

$$y^n = \sum_{-k}^{k} x^k \cdot h^{n-k}$$

where k is time and h is the impulse response/kernel. Our implementation computes an output signal given a one-dimensional input signal and kernel, both of which are made up of floating-point values.

We have implemented signal convolution in sequential Racket, Racket with futures, sequential Parrot, Parrot with futures, and sequential C.

Mergesort sorts a vector of floating-point numbers. We consider two variants of the mergesort algorithm, one that is readily parallelizable, and one that is not, but runs significantly faster than the parallel version on one processor [16]. We implemented both of the algorithms in sequential Racket, Racket with futures, and sequential C.

MV-Sparse does sparse matrix-vector multiplication using the compressed row format to store the matrix. For those unfamiliar with compressed row format, the essential idea is to flatten the matrix into a single 1D array and combine it with parallel 1D arrays indicating both where rows begin and what the column indices are. We implemented MV-Sparse in sequential Racket, Racket with futures, and sequential C. Our Racket version employs the higher-level nested data parallel primitive called a `gather`, which we have implemented using futures.

The NAS Parallel Benchmarks [4] are a suite of benchmarks derived from computational fluid dynamics applications of interest to NASA. They are widely used in the parallel systems community as application benchmarks. We consider two of them here.

NAS Integer Sort (IS) sorts an array of integer keys where the range of key values is known at compile-time. Sorting is performed using the histogram-sort variant of the bucket sort algorithm. We implemented NAS IS in sequential Racket and Racket with futures. We compare against the publicly available sequential and parallel Java reference implementations [13].

NAS Fourier Transform (FT) is the computational kernel of a 3-dimensional Fast Fourier Transform. Each iteration performs three sets of one-dimensional FFT's (one per dimension). We implemented NAS FT in sequential Racket and Racket with futures. We again compare against the publicly available sequential and parallel Java reference implementations.

6.3 Performance is Reasonable

Using futures implemented via our approach to incrementally parallelizing existing sequential runtime systems, it is possible to achieve both reasonable raw performance and good scaling for the benchmarks we tested.

Racket

Figure 14 shows running time and speedup curves for the Racket implementations of the three microbenchmarks listed in Figure 13. The results confirm that using futures, implemented using our developer-efficient incremental parallelization approach, it is feasible to achieve reasonable parallel performance on commodity desktops and servers, both in terms of raw performance and speedup.

Though the Racket implementations are slower than the optimized C versions in the sequential case, all three parallel Racket versions are able to yield better performance than sequential C after employing relatively small numbers of processors (2 for both convolution and MV-sparse, and 6 for mergesort). The parallel convolution implementation exhibits good scaling through the maximum number of processors available on both machine configurations, owing to the tight nested-loop structure of the algorithm, which involves only floating-point computations. Here the parallel convolution is able to avoid slow-path exits by using Racket's floating point-specific primitives, as discussed in Section 2. The Racket benchmarks also use unsafe versions of the arithmetic and array indexing operations.

Figure 15 shows running time and speedup curves for the Racket implementations for the NAS IS and FT benchmarks. We compare performance with both sequential and parallel Java implementations.

While sequential Racket implementations for these benchmarks are considerably slower than the Java implementations, the parallel implementations scale better. However, this scaling does not allow us to catch up with the parallel Java implementation in absolute terms. We suspect that, especially in the case of the IS benchmark, this is due to the majority of work being performed in the parallel portion of the benchmark being array accesses (e.g. `vector-ref` and `vector-set!` in Racket), operations, which are more heavily optimized in the Java runtime system. The Racket with futures implementation of NAS FT, however, is able to outperform sequential Java after 3 processors (on the Cosmos machine configuration), and generally exhibits similar scaling characteristics to the reference parallel Java implementation.

As with the self-developed benchmarks (Signal Convolution, Mergesort, and MV-Sparse), the results demonstrate that our approach to incremental parallelization of sequential runtimes can lead to reasonable parallel performance.

Parrot

We tested our prototype implementation of Parrot with futures using only the convolution benchmark. The results, as can be seen in Figure 16, are comparable to those we saw with Racket with futures in terms of speedup (compare to Figure 14(a)-(d)). This is supportive of our claim that our approach leads to reasonable parallel performance.

It is important to point out that the raw performance is not comparable, however. This is the result of our implementation being preliminary (contrast Figure 10 and Figure 11). More specifically, our current implementation is based on a version of Parrot in which the JIT is in a state of flux. Our benchmark results reflect *interpreted* performance without the JIT, and thus should be taken with a grain of salt. Our current implementation of futures in Parrot only supports operations on unboxed floating-point numbers.

Caveats notwithstanding, our results for the Parrot with futures proof-of-concept implementation suggest that our approach can be generalized to other high level language implementations.

7. Related Work

Our work builds on the ideas of futures from MultiLisp [14], a parallel dialect of Scheme. Parallelism in MultiLisp is also expressed via `future`. However, MultiLisp does not require an explicit `touch` on the part of the programmer. Instead, touches are implicitly performed whenever the value of a future is needed. Also unlike our work, futures in Multilisp always execute in parallel, whereas ours only execute in parallel when it is safe (based on the constraints of the runtime system).

Many language communities face the problem of retrofitting their implementations to support parallelism. The typical approach is to allow arbitrary threads of computation to run in parallel, and to adapt the runtime system as necessary. Some succeed in the transition through substantial reimplementation efforts; threads in the original Java 1.1 vir-

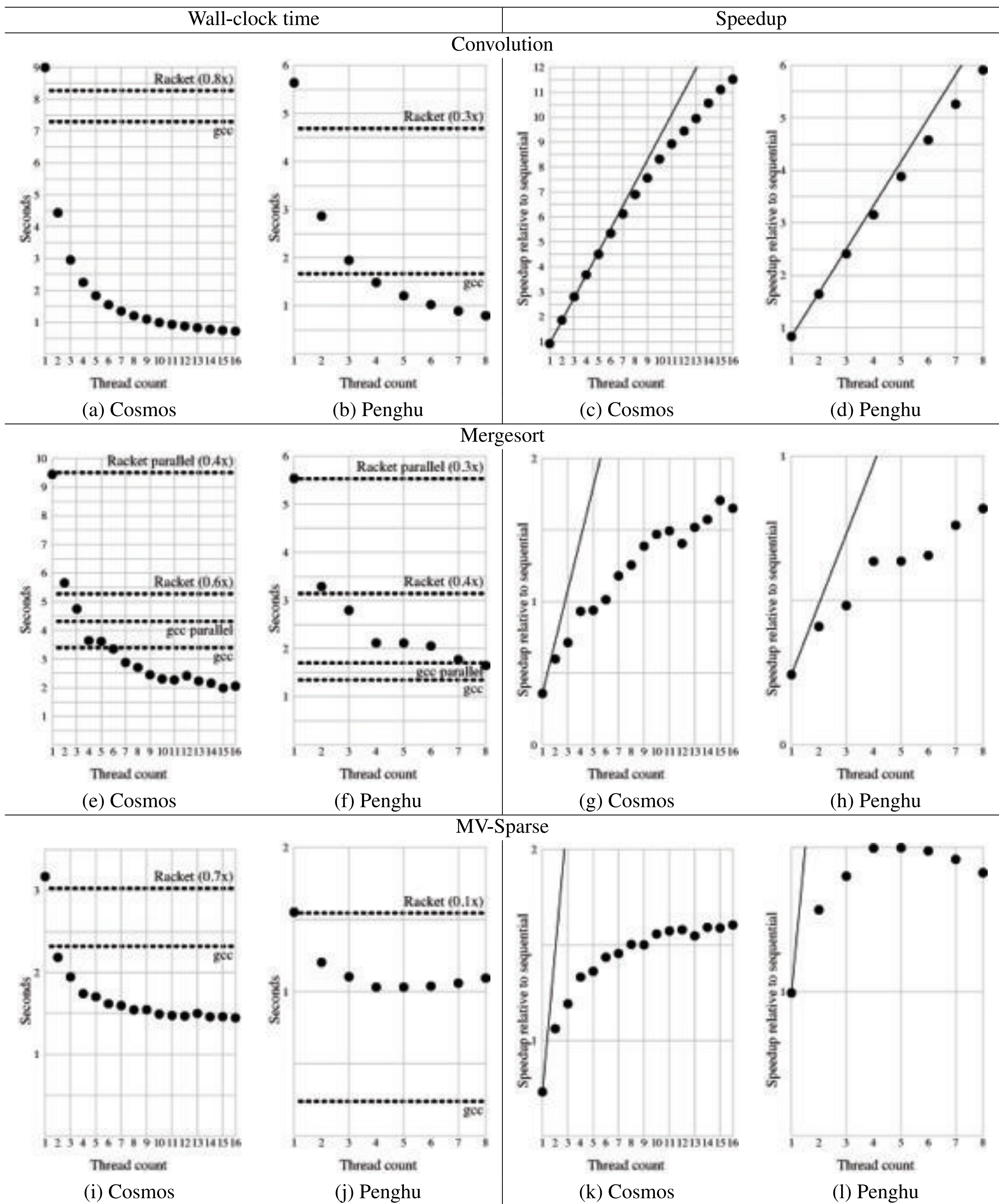


Figure 14. Raw performance and speedup of microbenchmarks using Racket with futures: lower dotted line is sequential, optimized C, upper dotted line is sequential Racket, dots indicate Racket with futures. Numbers next to "Racket" labels give the ratio of speed of the C version to the speed of the Racket version. The solid lines in speedup graphs indicate ideal speedup.

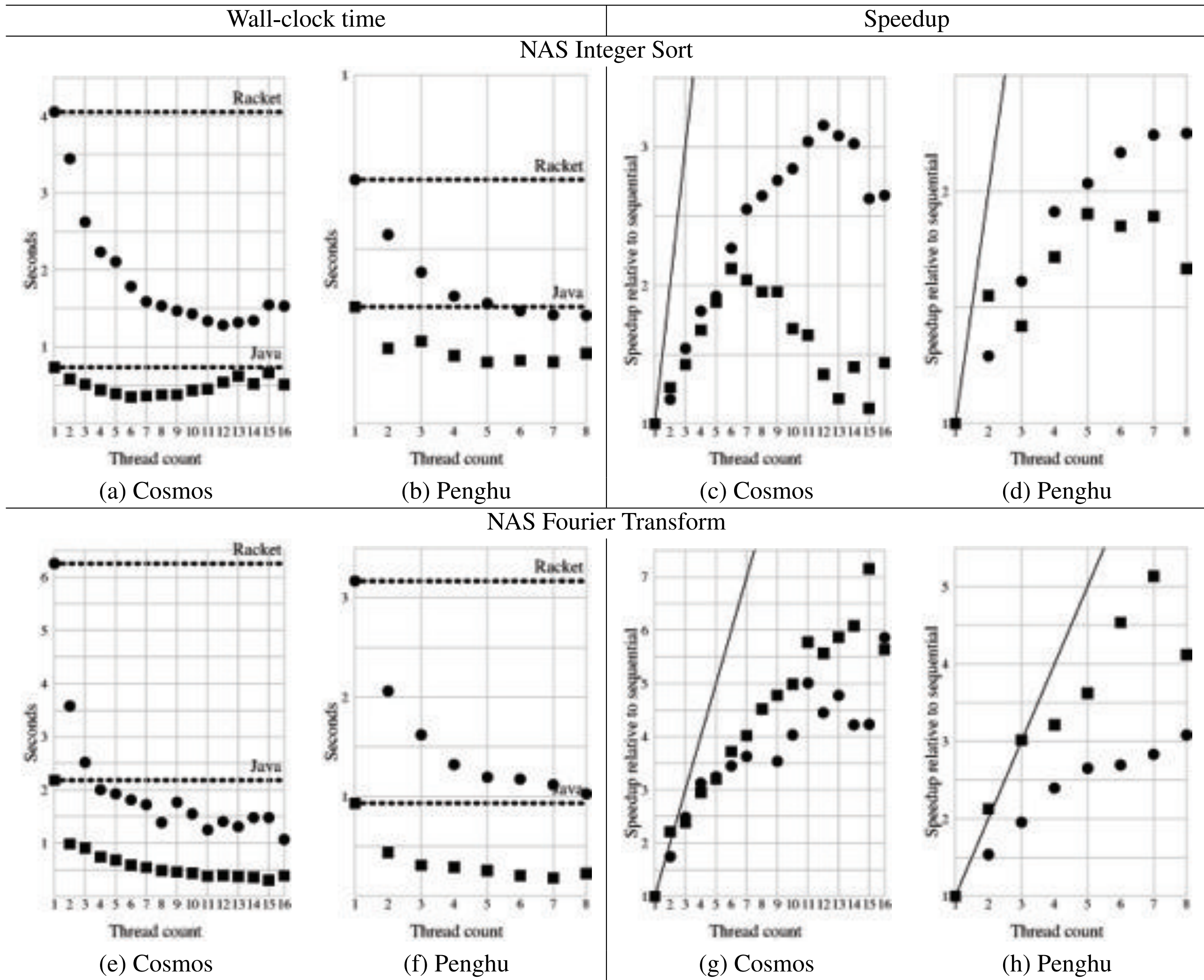
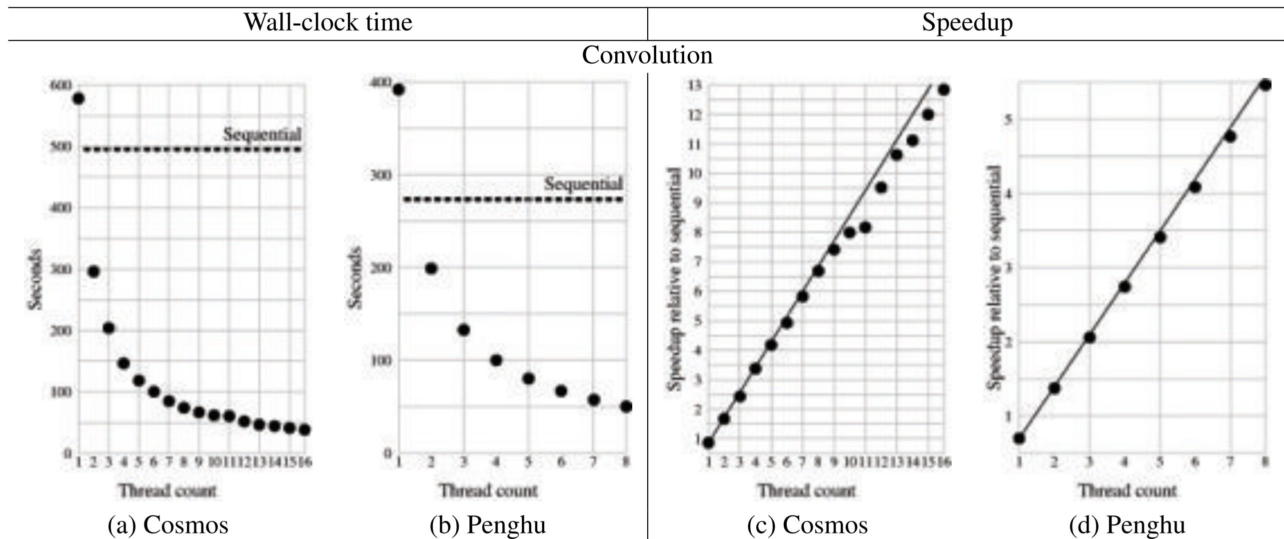


Figure 15. Raw performance and speedup for NAS parallel benchmarks in Racket with futures: upper dotted line is sequential Racket, lower dotted line indicates sequential Java, dots indicate Racket with futures, squares indicate parallel Java. The solid lines in speedup graphs indicate ideal speedup.

tual machine where implemented as user threads, but many re-implementations of Java now support threads that use hardware concurrency. Others succeed in the transition with the help of their language designs; Erlang and Haskell are prime examples of this category, where the purely functional nature of the language (and much of the language implementation) made a transition to support for parallelism easier, through it required substantial effort [3, 15]. Finally, many continue to struggle with the transition; our own attempts to map Racket-level threads to OS-level threads failed due to the complexity of the runtime system, and frequent attempts to rid the main Python and Ruby implementations of the global interpreter lock (GIL) have generally failed [1, 17]. An attempt to support OS-level threads in OCaml has so far produced an experimental system [7].

Our approach of introducing parallelism through constrained futures is somewhat similar to letting external C code run outside the GIL (and therefore concurrently) in Python or Ruby. Instead of pushing parallelism to foreign libraries, however, our approach draws parallelism into a subset of the language. Our approach is also similar to adding special-purpose parallel operators to a language, as in data-parallel operations for Data Parallel Haskell [8]; instead of constraining the set of parallel operations a priori, however, our approach allows us to gradually widen the parallelism available through existing constructs.

In adding support for parallelism to Racket, we hope to move toward the kind of support for parallelism that is provided by languages like NESL [5], X10 [10], Chapel [9], Fortress [2], Manticore [12], and Cilk [6], which were all designed to parallelism from the start. Adding parallelism



(a) Cosmos (b) Penghu (c) Cosmos (d) Penghu

Figure 16. Raw performance and speedup of microbenchmarks using Parrot with futures: dotted line is sequential Parrot, dots indicate Parrot with futures. The solid lines in speedup graphs indicate ideal speedup.

to a sequential run-time system is a different problem than designing a parallel language from scratch, but we take inspiration from designs that largely avoid viewing parallelism as concurrent threads of arbitrary computation.

8. Conclusion

We have proposed and described *slow-path barricading*, a new approach for incrementally parallelizing sequential runtime systems, and we have implemented and benchmarked it in two real-world runtime systems. In both cases, slow-path barricading allowed programs to safely take advantage of all the processor cores available in commodity multicore machines. These efforts required modest amounts of developer effort, and the scalability of parallel benchmarks that we wrote using the futures implementations was surprisingly good. Using our approach, implementers can quickly produce a mature parallel adaptation of an existing sequential runtime system.

The key insight behind our approach is that in existing sequential runtime systems, fast-path operations are generally already safe to run in parallel, while slow-path ones may not be. Furthermore, we can use this decomposition to instrument the codebase to find and improve problematic fast-paths. Finally, the application developer interested in parallelizing a part of a program is already likely using fast-path constructs for maximum sequential performance, or should be. In essence, we leverage both the runtime system implementers' extensive efforts to optimize fast-path operations and the application developers' use of these fast-path operations in optimized sequential code.

In this paper, we focus on adding the low-level parallel construct of futures to existing runtime systems. While it is clearly feasible to write to write parallel programs directly with futures (much like one can write them in a lower-level language with threads), we envision futures as only the starting point. We have already used futures in Racket to construct a prototype library of higher-level parallel constructs, specifically nested data-parallel constructs. We are currently working on expanding that library and determining which higher-level constructs need to be implemented in the runtime system for performance reasons. Efforts are also in place to improve the futures implementation in Racket, particularly to enhance support for nested futures and scheduling/mapping of futures. Another effort involves the design and implementation of parallel constructs for NUMA architectures and distributed memory machines.

We believe that our approach could be applied to a wide range of runtime systems, and obvious possibilities are scripting languages such as Perl, Python, and Ruby, as well as JavaScript/ECMAScript.

Acknowledgments

We would like to thank Steven P. Tarzia for generously granting us access to Cosmos, allowing us to expand our measurements to 16 cores. Thanks also to Nikos Hardavellas for discussions on the work in general and on multi-core architectures specifically.

References

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

[2] ALLEN, E., CHASE, D., HALLETT, J., LUCHANGCO, V., MAESSEN, J., AND STEELE, G. The Fortress language specification. `http://research.sun.com/projects/plrg/fortress.pdf`, 2005.

[3] ARMSTRONG, J. A history of Erlang. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages* (2007), pp. 6–1–6–26.

[4] BAILEY, D., BARTON, J., LASINSKI, T., AND SIMON, H. The NAS parallel benchmarks. Tech. Rep. RNR-91-002, NASA Ames Research Center, August 1991.

[5] BLELLOCH, G. Implementation of a portable nested data-parallel language. *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (1993), 102–111.

[6] BLUMOFE, R., JOERG, C., KUSZMAUL, B., LEISERSON, C., RANDALL, K., AND ZHOU, Y. Cilk: An efficient multithreaded runtime system. *ACM SIGPLAN Notices* (1995), 207–216.

[7] BOURGOIN, M., JONQUET, A., CHAILLOUX, E., CANOU, B., AND WANG, P. OCaml4Multicore, 2007. `http://www.algo-prog.info/ocmc/web/`.

[8] CHAKRAVARTY, M. T., LESHCHINSKIY, R., JONES, S. P., KELLER, G., AND MARLOW, S. Data Parallel Haskell: A status report. *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming* (2007), 10–18.

[9] CHAMBERLAIN, B. L., CALLAHAN, D., AND ZIMA, H. P. Parallel programming and the Chapel language. *International Journal of High Performance Computing Applications* (August 2007), 291–312.

[10] CHARLES, P., GROTHOFF, C., SARASWAT, V., DONAWA, C., KIELSTRA, A., EBCIOGLU, K., VON PRAUN, C., AND SARKER, V. X10: An object-oriented approach to non-uniform cluster computing. *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (2005), 519–538.

[11] FLATT, M., AND PLT. Reference: Racket. `http://www.racket-lang.org/tr1/`.

[12] FLUET, M., RAINEY, M., REPPY, J., SHAW, A., AND XIAO, Y. Manticore: A heterogeneous parallel language. *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming* (2007), 37–44.

[13] FRUMKIN, M., SCHULTZ, M., JIN, H., AND YAN, J. Implementation of NAS parallel benchmarks in Java. Tech. Rep. NAS-02-009, Ames Research Center, Moffett Field, CA, USA, 2002.

[14] HALSTEAD, JR., R. H. MULTILISP: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (October 1985), 501–538.

[15] JONES, S., GORDON, A., AND FINNE, S. Concurrent Haskell. *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)* (January 1996), 295–308.

[16] JUSZCZAK, C. Fast mergesort implementation based on half-copying merge algorithm, 2007. `http://kicia.ift.uni.wroc.pl/algorytmy/mergesortpaper.pdf`.

[17] Thread state and the global interpreter lock. `http://docs.python.org/c-api/init.html#thread-state-and-the-global-interpreter-lock`, March 2010.