# Programming in Manticore,
# a Heterogenous Parallel Functional Language

Matthew Fluet
Toyota Technological Institute at Chicago
fluet@tti-c.org

Lars Bergstrom  Nic Ford  Mike Rainey
John Reppy  Adam Shaw  Yingqi Xiao
University of Chicago
{larsberg,nford,mrainey,jhr,adamshaw,xiaoyq}@cs.uchicago.edu

May 2009

**Abstract**

The Manticore project is an effort to design and implement a new functional language for parallel programming. Unlike many earlier parallel languages, Manticore is a *heterogeneous* language that supports parallelism at multiple levels. Specifically, the Manticore language combines Concurrent ML-style explicit concurrency with fine-grain, implicitly threaded, parallel constructs. These lectures will introduce the Manticore language and explore a variety of programs written to take advantage of heterogeneous parallelism.

At the explicit-concurrency level, Manticore supports the creation distinct threads of control and the coordination of threads through first-class synchronous-message passing. Message-passing synchronization, in contrast to shared-memory synchronization, fits naturally with the functional-programming paradigm.

At the implicit-parallelism level, Manticore supports a diverse collection of parallel constructs for different granularities of work. Many of these constructs are inspired by common functional-programming idioms.

In addition to describing the basic mechanisms, we will present a number of useful programming techniques that are enabled by these mechanisms. Finally, we will briefly discuss some of the implementation techniques used to execute Manticore programs on commodity multicore computers.

## 1 Introduction

Future improvements in microprocessor performance will largely come from increasing the *computational width* of processors, rather than increasing the clock frequency [57]. This trend is exhibited by multiple levels of hardware parallelism: single-instruction, multiple-data (SIMD) instructions; simultaneous-multithreading executions; multicore processors; multiprocessor systems. As a result, parallel computing is becoming widely available on commodity hardware. While these new designs solve the computer architect's problem of how to use an increasing number of transitors in a given power envelope, they create a problem for programmers and language implementors. Ideal applications for this hardware, such as multimedia processing, computer games, and small-scale simulations, can themselves exhibit exhibit parallelism at multiple levels with different granularities, which means that a homogeneous language design will not take full advantage of the available hardware resources. For example, a language that provides data parallelism but not explicit concurrency will be inconvenient for the development of the networking and GUI components of a program. Similarly, a language that provides concurrency but not data parallelism will be ill-suited for the components of a program that demand fine-grain SIMD parallelism, such as image processing and particle systems.

Our thesis is that parallel programming languages must provide mechanisms for multiple levels of parallelism, both because applications exhibit parallelism at multiple levels and because hardware requires parallelism at multiple levels to maximize performance. For example, consider a networked flight simulator (Figure 1). This application might use SIMD instructions for physics simulation; data-parallel computations for particle systems [65] to model natural phenomena (*e.g.*, rain, fog, and clouds); light-weight parallel executions for preloading terrain and computing level-of-detail refinements; speculative search for artificial intelligence; concurrent threads for user interface and network components. Programming such an application will be challenging without language support for parallelism at multiple levels.

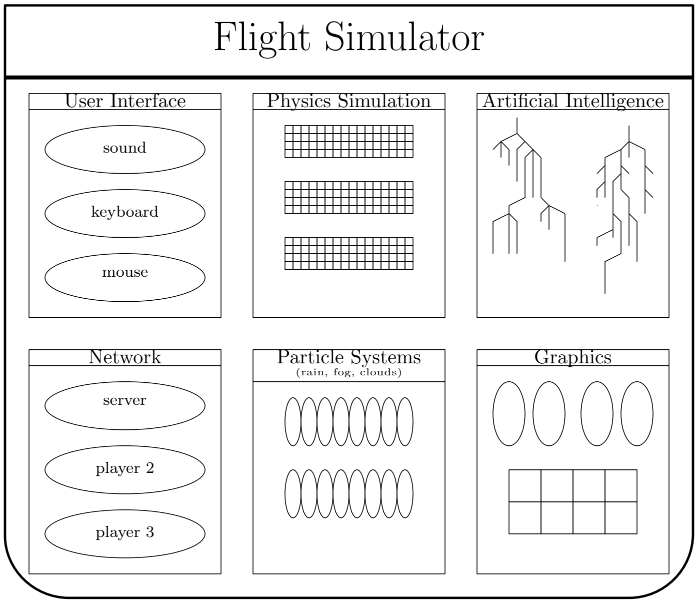

Figure 1: An application with multiple levels of parallelism

Traditional imperative and object-oriented languages are poor choices for parallel applications. While they may support, or be extended with, concurrency constructs, their reliance on mutation of state as their core mechanism makes both writing correct programs and compiling efficient executables difficult. Existing parallel languages are also not a solution, since they have mostly been targeted at the narrow domain of high-performance scientific computing and large-scale parallelism. We need languages that can be used to write traditional commodity applications while exploiting the performance of tomorrow's multicore hardware.

The Manticore project at the University of Chicago and the Toyota Technological Institute at Chicago is an ambitious effort to lay the foundation for programming the commodity processors of the future, addressing the issues of *language design and implementation for multicore processors* [33, 30]. As described above, our emphasis is on applications that might run on commodity multicore processors — applications that can exhibit parallelism at multiple levels with different granularities. To meet the demands of such applications, we propose a *heterogeneous* parallel language: a language that combines support for parallel computation at different levels into a common linguistic and execution framework.

We envision a high-level parallel programming language targeted at what we expect to be a typical commodity microprocessor in 2012. While predicting the future is always fraught with danger, we expect that these processors will have 8 or more general-purpose cores (*e.g.*, x86-64 processors) with SIMD instructions and 2–4 hardware thread contexts [57]. It is quite likely that these processors will also have special-purpose vector units, similar to those of the IBM Cell processor [45]. Furthermore, since it is unlikely that shared caches will scale to large numbers of cores, we expect a non-uniform or distributed-memory architecture inside the processor.

The problem posed by such processors is how to effectively exploit the different forms of parallelism provided by the hardware. We believe that mechanisms that are well integrated into a programming language are the best hope for achieving parallelism across a wide range of applications, which is why we are focusing on language design and implementation.

In the Manticore project, we are designing and implementing a parallel programming language that supports a range of parallel programming mechanisms. These include explicit threading with message passing to support both concurrent systems programming and coarse-grain parallelism, and nested-data parallelism mechanisms to support fine-grain computations.

The Manticore language is rooted in the family of statically-typed strict functional languages such as OCaml and SML. We make this choice because functional languages emphasize a value-oriented and mutation-free programming model, which avoids entanglements between separate concurrent computations [41, 70, 46, 56]. We choose a strict language, rather than a lazy or lenient one, because we believe that strict languages are easier to implement efficiently and accessible to a larger community of potential users. On top of the sequential base language, Manticore provides the programmer with mechanisms for explicit concurrency and coarse-grain parallelism and mechanisms for fine-grain parallelism.

Manticore's concurrency mechanisms are based on Concurrent ML (CML) [71], which provides support for threads and synchronous message passing. Manticore's support for fine-grain parallelism is influenced by previous work on nested data-parallel languages, such as NESL [7, 6, 8] and Nepal [15, 16, 49].

In addition to language design, we are exploring a unified runtime framework, capable of handling the disparate demands of the various heterogeneous parallelism mechanisms exposed by a high-level language design and capable of supporting a diverse mix of scheduling policies. It is our belief that this runtime framework will provide a foundation for rapidly experimenting with both existing parallelism mechanisms and additional mechanisms not yet incorporated into high-level language designs for heterogeneous parallelism.

Our runtime framework consists of a composition of runtime-system and compiler features. It supports a small core of primitive scheduling mechanisms, such as virtual processors, preemption, and computation migration. Our design favors minimal, light-weight representations for computational tasks, borrowing from past work on *continuations*. On top of this substrate, a language implementor can build a wide range of parallelism mechanisms with complex scheduling policies. By following a few simple rules, these schedulers can be implemented in a modular and nestable way.

These lecture notes will introduce the Manticore language and selected programming techniques. Section 2 gives a brief overview of the Manticore language, setting the stage for more detailed treatment of specific language features. Section 3 describes the explicit-concurrency level of the Manticore language. Section 4 describes the implicit-parallelism level of the Manticore language. Section 5 briefly describes some of the implementation techniques that are used to efficiently execute Manticore programs.

# 2 Overview of the Manticore Language

Parallelism mechanisms can be roughly grouped into three catagories:

- *implicit parallelism*, where the compiler and runtime system are exclusively responsible for partitioning the computation into parallel threads. Examples of this approach include Id [55], pH [56], and Sisal [37].
- *implicit threading*, where the programmer provides annotations (or hints) to the compiler as to which parts of the program are profitable for parallel evaluation, but mapping onto parallel threads is left to the compiler and runtime system. Examples of this approach include Nesl [6] and Nepal [16].
- *explicit threading*, where the programmer explicitly creates parallel threads. Examples of this approach include CML [71] and Erlang [3].

These different design points represent different trade-offs between programmer effort and programmer control. Automatic techniques for parallelization have proven effective for dense regular parallel computations (*e.g.*, dense matrix algorithms), but have been less successful for irregular problems. Manticore provides both implicit threading and explicit threading mechanisms. The former supports fine-grained parallel computation, while the latter supports coarse-grained parallel tasks and explicit concurrent programming. These parallelism mechanisms are built on top of a sequential functional language. In the sequel, we briefly discuss each of these in turn, starting with the sequential base language.

## 2.1 Sequential Programming

Manticore's sequential core language is based on the Standard ML (SML) language. The main differences are that Manticore does not have mutable data (*i.e.*, reference cells and arrays) and, in the present language implementation, Manticore has a simplified module system (omitting functors and sophisticated type sharing). Manticore does, however, have the functional elements of SML (datatypes, polymorphism, type inference, and higher-order functions) as well as exceptions. The inclusion of exceptions has interesting implications for the implicitly threaded mechanisms, but we believe that some form of exception mechanism is necessary for systems programming. As many researchers have observed, using a mutation-free computation language greatly simplifies the implementation and use of parallel features [41, 70, 46, 56, 21]. In essence, mutation-free functional programming reduces interference and data dependencies.

As the syntax and semantics of the sequential core language are largely orthogonal to (but potentially synergystic with) the parallel language mechanisms, we have resisted tinkering with the sequential SML core. The Manticore Basis, however, differs significantly from the SML Basis Library [35]. For example, we have a fixed set of numeric types — `int`, `long`, `integer`, `float`, and `double` — instead of SML's families of numeric modules.

## 2.2 Explicitly-Threaded Parallelism

The explicit concurrent programming mechanisms presented in Manticore serve two purposes: they support concurrent programming, which is an important feature for systems programming [42], and they support explicit parallel programming. Like CML, Manticore supports threads that are explicitly created using the **`spawn`** primitive. Threads do not share mutable state (as there is no mutable state in the sequential core language); rather they use synchronous message passing over typed channels to communicate and synchronize. Additionally, we use CML communication mechanisms to represent the interface to imperative features such as input/output. Section 3 explores this programming paradigm in more detail.

The main intellectual contribution of CML's design is an abstraction mechanism, called *first-class synchronous operations*, for building synchronization and communication abstractions. This mechanism allows programmers to encapsulate complicated communication and synchronization protocols as first-class abstractions, called *event values*, which encourages a modular style of programming where the actual underlying channels used to communicate with a given thread are hidden behind data and type abstraction. Events can range from simple message-passing operations to client-server protocols to protocols in a distributed system.

CML has been used successfully in a number of systems, including a multithreaded GUI toolkit [34], a distributed tuple-space implementation [71], a system for implementing partitioned applications in a distributed setting [81], and a higher-level library for software checkpointing [82]. CML-style primitives have also been added to a number of other languages, including HASKELL [72], JAVA [22], OCAML [47], and SCHEME [29]. We believe that this history demonstrates the effectiveness of CML's approach to concurrency.

## 2.3 Implicitly-Threaded Parallelism

Manticore provides implicitly-threaded parallel versions of a number of sequential forms. These constructs can be viewed as *hints* to the compiler about which computations are good candidates for parallel execution; the semantics of (most of) these constructs is sequential and the compiler and/or runtime system may choose to execute them in a single thread.[1]

Having a sequential semantics is useful in two ways: it provides the programmer with a deterministic programming model and it formalizes the expected behavior of the compiler. Specifically, the compiler must verify that the individual sub-computations in a parallel computation do not send or receive messages before executing the computation in parallel. Furthermore, if a sub-computation raises an exception, the runtime code must delay delivery of that exception until it has verified that all sequentially prior computations have terminated. Both of these restrictions require program analysis to implement efficiently.

Section 4 explores this programming paradigm in more detail. Here, we briefly introduce the implicitly parallel mechanisms:

**Parallel arrays** Support for parallel computations over arrays and matrices is common in parallel languages. In Manticore, we support such computations using the nested parallel array mechanism inspired by NESL [7, 6, 8] and developed further by Nepal [15, 16, 49].

The key operations involving parallel arrays are *parallel comprehensions*, which allow the consise expressions of parallel loops that consume arrays and return a new array, and *parallel reductions*, which allow the concise expression of parallel loops that consume arrays and return scalars.

**Parallel tuples** The parallel tuple expression form provides a hint to the compiler that the elements of the tuple may be evaluated in parallel. The basic form is

```
(| e1, ..., en |)
```

which describes a fork-join evaluation of the expressions $e_i$ in parallel. The result is a normal tuple value.

**Parallel bindings** Parallel arrays and tuples provide a fork-join pattern of computation, but in some cases more flexible scheduling is desirable. In particular, we may wish to execute some computations speculatively. Manticore provides a parallel binding form

```
pval pat = exp
```

that launches the evaluation of the expression $e$ as a parallel thread. The sequential semantics of a parallel binding are similar to lazy evaluation: the binding is only evaluated (and only evaluated once) when one of its bound variables is demanded. In the parallel implementation, we use eager evaluation for parallel bindings, but such computations are canceled when the main thread of control reaches a point where their result is guaranteed never to be demanded.

**Parallel cases** The parallel case expression form is a nondeterministic counterpart to SML's sequential case form. In a parallel case expression, the discriminants are evaluated in parallel and the match rules may include wildcard patterns that match even if their corresponding discriminants have not yet been fully evaluated. Thus, a parallel case expression nondeterministically takes any match rule that matches after sufficient discriminants have been evaluated.

Unlike the other implicitly-threaded mechanisms, parallel case is nondeterministic. We can still give a sequential semantics, but it requires including a source of non-determinism, such as McCarthy's **amb** [51], in the sequential language.

[1] Shaw's Master's paper [75] provides a rigorous account of the semantics of a subset of these mechanisms.

### 2.4 Future Directions

This section describes a first-cut design meant to give us a base for exploring multi-level parallel programming. Based on experience with this design, we plan to explore a number of different evolutionary paths for the language. First, we plan to explore other parallelism mechanisms, such as the use of futures with work stealing [54, 13, 9]. Such medium-grain parallelism would nicely complement the fine-grain parallelism (via parallel arrays) and the coarse-grain parallelism (via concurrent threads) present in Manticore. Second, there has been significant research on advanced type systems for tracking effects, which we may use to introduce imperative features into Manticore. As an alternative to traditional imperative variables, we will also examine synchronous memory (*i.e.*, I-variables and M-variables *à la* Id [55]) and *software transactional memory* (STM) [74].

# 3 Explicit Concurrency in Manticore

## 3.1 Introduction

The explicit-concurrency mechanisms of Manticore are based on Concurrent ML (CML) [71]. CML extends SML with synchronous message passing over typed channels and a powerful abstraction mechanism, called *first-class synchronous operations*, for building synchronization and communication abstractions. This mechanism allows programmers to encapsulate complicated communication and synchronization protocols as first-class abstractions, which encourages a modular style of programming where the actual underlying channels used to communicate with a given thread are hidden behind data and type abstraction. CML has been used successfully in a number of systems, including a multithreaded GUI toolkit [34], a distributed tuple-space implementation [71], and a system for implementing partitioned applications in a distributed setting [81]. The design of CML has inspired many implementations of CML-style concurrency primitives in other languages. These include other implementations of SML [53], other dialects of ML [47], other functional languages, such as HASKELL [72], SCHEME [29], and other high-level languages, such as JAVA [22].

Concurrent ML, as its name implies, emphasizes *concurrent* programming — programs consisting of multiple independent flows of sequential control, called processes. The execution of a concurrent program can be viewed as an interleaving of the sequential executions of its constituent processes. Although concurrent programming can be motivated by a desire to improve performance by exploiting multiprocessors, concurrency is a useful programming paradigm for certain application domains. For example, interactive systems (*e.g.*, graphical-user interfaces) have a naturally concurrent structure; similarly, distributed systems can often be viewed as concurrent programs.

As noted above, Manticore adopts Standard ML as its sequential core language, providing first-class functions, datatypes and pattern matching, exception handling, strong static typing, parametric polymorphicsm, *etc.* The explicit-concurrency mechanisms add the following features:

- dynamic creation of threads and typed channels.
- rendezvous communication via synchronous message passing.
- first-class synchronous operations, called events.
- automatic reclamation of threads and channels.
- pre-emptive scheduling of explicitly concurrent threads.
- efficient implementation — both on uniprocessors and multiprocessors.

## 3.2 Basic Concurrency Primitives

This section discusses the basic concurrency primitives provided by Manticore, including process creation and simple message passing via typed channels. Both processes and channels are created dynamically.

### 3.2.1 Threads

Processes (independent flows of sequential control) in Manticore are called *threads*. This choice of terminology emphasizes the fact that threads are lightweight and to distinguish them from other forms of process abstraction used in the Manticore runtime model (see Section 5). When a Manticore program begins executing, it consists of a single thread; this initial thread may create additional threads using the **`spawn`** `e` expression form. In the expression **`spawn`** `e`, the expression `e` is of type `unit` and the expression **`spawn`** `e` is of type `tid` (the type of a thread identifier). When the expression **`spawn`** `e` is evaluated, a new thread is created to evaluate the expression `e`. The newly created thread is called the *child* and its creator is called the *parent*.

The child thread will execute until the evalution of its initial expression is complete, at which time it terminates. In Manticore, the parent-child relationships between threads have no effect on the semantics of the program. For example, the termination of a parent thread does not affect child threads; each child thread is an independent flow of sequential control. Note that this means that the initial Manticore thread may terminate while other (children, grand-children, *etc.*) threads continue to execute. The whole program does not terminate until *all* threads have terminated or are blocked.

A thread may terminate in one of three ways. First, a thread may complete the evaluation of its initial expression. Second, a thread may explicitly terminate itself by calling the `exit` function, which has the signature:

```
val exit : unit -> 'a
```

Like a **`raise`** `e` expression, the result type of `exit` is `'a` since it never returns. Third, a thread may raise an uncaught exception.[2] Note that such an exception is local to the thread in which it is raised; it is not propagated to its parent thread.

Because the number of threads in a Manticore program is unbounded and the number of (physical) processors is finite, the processors are multiplexed among the Manticore threads.[3] This is handled automatically by the Manticore runtime system, using periodic timer interrupts to provide preemptive scheduling of Manticore threads. Thus, the programmer is not required to ensure that each thread yields the processor at regular intervals (as is required by the so-called coroutine implementations of concurrency). This preemptive scheduling is important to support program modularity, because sequential code does not need to be modified to support explicit scheduling code. On the other hand, it places additional burden on the runtime system to efficiently manage the disparate demands of computation-bound and interactive threads.

In the concurrent-programming style promoted by Concurrent ML, threads are used very liberally. This style is supported by the fact that threads are extremely cheap to create and impose very little space overhead. Furthermore, the storage used to represent threads can be reclaimed by the garbage collector.

### 3.2.2 Channels

In order for multiple independent flows of sequential control to be useful, there must be some mechanism for communication and synchronization between the threads. In Manticore, the most important such mechanism is synchronous message passing on typed channels. The type constuctor `'a chan` is used to generate the types of channels; a channel for communicating values of type `t` has the type `t chan`. There are two operations for channel communication, which have the signatures:

```
val recv : 'a chan -> 'a
val send : 'a chan * 'a -> unit
```

Message passing is synchronous, which means that both the sender and the receiver must be ready to communicate before either can proceed. When a thread executes a `recv` or `send` on a channel, we say that the thread is *offering* communication. The thread will block until some other thread offers a *matching* communication: the complementary operation on the same channel. When two threads offer matching communications, the massage is passed from the sender to the receiver and both threads continue execution. Thus, message passing involves both communication of data and synchronization.

Note that channels are first-class values, created by the *channel* function, which has the signature:

```
val channel : unit -> 'a chan
```

Channels can be viewed as labels for *rendezvous* points — they do not name the sender or receiver, and they do not specify a direction of communication. Over the course of its lifetime, a channel may pass multiple values between multiple different threads. At any given time, there may be multiple threads offering to `recv` or `send` on the same channel. The nature of synchronous message passing ensures that each `recv` is matched with exactly one `send`.

### 3.2.3 Examples

**Updatable storage cells** Although mutable state makes concurrent programming difficulty, it is relatively easy to give an implementation of updatable storage cells on top of threads and channels. (Furthermore, updatable storage cells are a natural first example, since the desired behavior is well-known, placing the focus on the use of threads and channels.)

We define the following abstract interface to storage cells:

[2] In a sense, this is equivalent to the first manner in which a thread may terminate: the thread has completed the evaluation of its initial expression to an uncaught exception.

[3] The processors are further multiplexed to support the implicitly-threaded parallelism described in Section 4.

```
signature CELL =
  sig
    type 'a cell
    val cell : 'a -> 'a cell
    val get  : 'a cell -> 'a
    val put  : 'a cell * 'a -> unit
  end
```

The operation `cell` creates a new cell initialized to the given value; the operations `get` and `put` are used to read and write a cell's value. Our approach is to represent the state of a cell by a thread, which we call the *server*, and to represent the `'a cell` type as a channel for communicating with the server. The complete implementation is as follows:

```
structure Cell : CELL =
  struct
    datatype 'a req = GET of 'a chan | PUT of 'a
    datatype 'a cell = CELL of 'a req chan

    fun get (CELL reqCh) =
      let
        val replyCh = channel ()
      in
        send (reqCh, GET replyCh) ;
        recv replyCh
      end

    fun put (CELL reqCh, y) =
      send (reqCh, PUT y)

    fun cell z =
      let
        val reqCh = channel ()
        fun loop x =
          case recv reqCh of
             GET replyCh => (send (replyCh, x) ; loop x)
           | PUT y = loop y
        val _ = spawn (loop z)
      in
        CELL reqCh
      end
  end
```

The datatype `'a req` defines the type of requests: either a `GET` to read the cell's value or a `PUT` to write the cell's value. The implementations of the `get` and `put` operations is straightforward. Each operation requires sending the appropriate request message to the server on the request channel. In the case of the `get` operation, the client first creates a reply channel, then the client sends the `GET` message (carrying the reply channel) to the server, and finally the client receives the cell's value on the reply channel. In the case of the `put` operation, the client sends the `PUT` message (carrying the new value for the cell).

The implementation of the `cell` operation is slightly more complicated. The `cell` operation creates a new cell, which involves allocating the request channel and spawning a new server thread to handle requests. Servers are typically implemented as infinite loops, with each iteration corresponding to a single client request. Since we are programming in a functional programming language, we use a tail recursive function to implement the loop.

The implementation of the `CELL` abstraction is a prototypical example of the *client-server* style of concurrent programming.

**Sieve of Eratosthenes** Another important style of concurrent programming is the *dataflow network* style. In this style, computations are structured as networks of processes, where the data from one process flows to other processes in the network. Many of the processes in a dataflow network can be implemented as an infinitely looping thread that carries some local state from one iteration to the next. The `forever` function is useful for constructing such threads:

```
fun forever (init : 'a) (f: 'a -> 'a) : unit =
  let
    fun loop s = loop (f s)
    val _ = spawn (loop init)
  in
    ()
  end
```

The forever function takes two arguments, an initial state (of type 'a) and a function from states to states (of type 'a -> 'a), and it spawns a thread that repeatedly iterates the function.

A classic application of dataflow networks is for *stream processing*. A stream can be viewed as a possibly infinite sequence of values. For example, the succs function takes an initial integer and returns a channel on which a client may receive the stream of successors of the initial integer:

```
fun succs (i : int) : int chan =
  let
    val succsCh = channel ()
    fun succsFn i = (send (succsCh, i) ; i + 1)
    val () = forever i succsFn
  in
     succsCh
  end
```

Each application of the succs function creates a new instance of the stream of numbers; values are consumed from the stream by applying recv to the result channel.

This style of stream processing is similar to the notion of *lazy streams* as used in idiomatic functional programming. A principal difference is that these streams are stateful: once a value is read from a stream it cannot be read again.

A traditional example of stream programming is computing prime numbers using the *Sieve of Eratosthenes*. We start with the stream of integers beginning with 2 (the first prime number). To compute the primes, we filter out multiples of 2, which gives a stream beginning with 3 (the second prime number). We then filter out multiples of 3, yielding a stream beginning with 5 (the third prime number). At each step, we take the head of the stream (which is the next prime) and construct a new stream by filtering out multiples of the prime.

The filtering of a stream is provided by the following function, which takes a prime number p and an input stream inCh and returns a new stream outCh with multiples of p removed:

```
fun filter (p, inCh : int chan) : int chan =
  let
    val outCh = channel ()
    fun filterFn () =
      let
        val i = recv inCh
      in
        if (i mod p) <> 0 then send (outCh, i) else ()
      end
    val () = forever () filterFn
  in
    outCh
  end
```

The stream of primes is created by the following function:

```
fun primes () : int chan =
  let
    val primesCh = channel ()
    fun primesFn ch =
      let
        val p = recv ch
      in
        send (primesCh, p) ;
        filter (p, ch)
      end
    val _ = forever (succs 2) primesFn
  in
    primesCh
  end
```

The `primes` function creates a network of threads consisting of a `succFn` thread, which produces the stream of integers starting at `2`, a chain of `filterFn` threads, which filter out multiples of primes that have been sent, and a `primeFn` thread, which sends primes on the `primesCh` channel and spawns new `filterFn` threads.

We can use the `primes` function (and associated dataflow network) to compute a list of the first `n` prime numbers:

```
fun firstPrimes n =
  let
    val primesCh = Primes.primes ()
    fun loop (i, acc) =
      if i = 0
        then rev acc
        else loop (i - 1, (recv primesCh)::acc)
  in
    loop (n, [])
  end
```

**Fibonacci series** Another classic example of dataflow programming is the *Fibonacci series*, defined by the recurrence:

$$\begin{aligned} \mathit{fib}_1 &= 1 \\ \mathit{fib}_2 &= 1 \\ \mathit{fib}_{i+2} &= \mathit{fib}_{i+1} + \mathit{fib}_i \end{aligned}$$

Before implementing a Manticore program that generates the stream of Fibonnaci numbers, it is worthwile to consider the structure of the process network. Each element in the series is computed from the two previous elements. Thus, when the value $\mathit{fib}_i$ is computed, it needs to be fed back into the network for the computation of $\mathit{fib}_{i+1}$ and $\mathit{fib}_{i+2}$. Such a process network requires nodes that copy an input stream to two output streams and a node that provides a one-element delay in a stream. We also require a node that adds the values received from two streams. Figure 2 gives a pictorial representation of the process network for generating the Fibonacci series.

As show in Figure 2, we can implement the Fibonacci stream using three general-purpose dataflow combinators: *add*, *copy*, and *delay*:

```
fun addStrms (inCh1, inCh2, outCh) =
  forever () (fn () =>
    send (outCh, (recv inCh1) + (recv inCh2)))

fun copyStrm (inCh, outCh1, outCh2) =
   forever () (fn () =>
     let val x = recv inCh
     in  send (outCh1, x) ; send (outCh2, x)
     end)

fun delayStrm first (inCh, outCh) =
   forever first (fn x => (send (outCh, x) ; recv inCh))
```

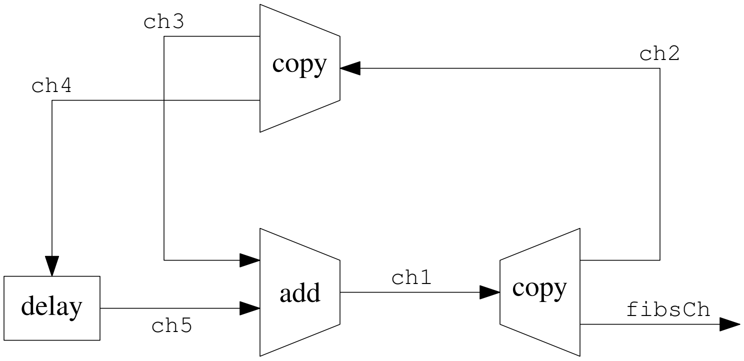

Figure 2: The Fibonacci stream process network

Unlike the Sieve of Eratosthenes example, the process creation functions do not create their own channels. Because we need to construct a cyclic process network, we will pre-allocate the channels and supply them as arguments to the process creation functions. Note that the `delayStrm` combinator is similar to the `copyStrm` combinator, since it copies the elements from the `inCh` to the `outCh`; the difference is that the `delayStrm` first sends an initial value on the `outCh`, before copying the elements from the `inCh` to the `outCh`.

The implementation of the Fibonacci stream simply constructs the process network described in Figure 2:

```
fun fibs () : int chan =
  let
     val fibsCh = channel ()
     val ch1 = channel ()
     val ch2 = channel ()
     val ch3 = channel ()
     val ch4 = channel ()
     val ch5 = channel ()
  in
     copyStrm (ch1, ch2, fibsCh) ;
     copyStrm (ch2, ch3, ch4) ;
     delayStrm 0 (ch4, ch5) ;
     addStrms (ch3, ch5, ch1) ;
     send (ch1, 1) ;
     fibsCh
  end
```

As noted above, the channels for the network are allocated first and then the process nodes are created. A minor subtlety of the implementation is initializing the network: the delay node is initialized with the value `0` (one may think of this as $fib_0 = 0$) and the value `1` ($fib_1$) is sent on the channel `ch1`. This value is fed back into the network (via `ch2` and `ch3`) to be added to `0` to produce the value `1` ($fib_2$).

## 3.3 First-class Synchronous Operations

### 3.3.1 Basic First-class Synchronous Operations

Thus far, we have seen simple synchronous message passing examples using `recv` and `send`. While these operations permit the implementation of interesting concurrent programs, there are limits to the kinds of concurrent programs that can be expressed with just these operations.

A key programming mechanism in message-passing concurrency programming is *selective communication*. The basic idea is to allow a thread to block on a nondeterministic choice of several blocking communications — the first communication that becomes *enabled* is chosen. If two or more communications are simultaneously enabled, then one is chosen nondeterministically. For example, note that there is a subtlety in the implementation of the Fibonacci network. The correctness of the implementation depends on the order in which the `copyStrm` combinator sends messages on its two output channels and on the order in which the `addStrms` combinator receives messages on its two input channels. If one were to reverse the order of the sends on `ch3` and `ch4`, then the network would deadlock: the `addStrms` node would be attempting to receive a value on ch3, while the copyStrm node would be attempting to send a value on ch4, and the delayStrm node would be attempting to send a value on ch5.

Although we were able to carefully construct the Fibonacci network to avoid this problem, a more robust solution is to eliminate the dependence on the order of the blocking operations. For example, the addStrm combinator should block on reading a value from either inCh1 or inCh2; if it receives a value on inCh1 first, then it must block to receive a value on inCh2 (and vice versa). Similarly, the copyStrm combinator should block on sending a value to either outCh1 or outCh2; if it sends a value on outCh2 first, then it must block to send a value on outCh2 (and vice versa). Most concurrent languages with message passing provide a mechanism for selecting from a choice of several blocking communications.

However, there is a fundamental conflict between the desire for abstraction and the need for selective communication. In most concurrent languages with message passing, in order to formulate the selection from a choice of several blocking communications, one must explicitly list the blocking communications (*i.e.*, the individual recvs and sends with their arguments). This makes it difficult to construct abstract synchronous operations, because the constituent recvs and sends must be revealed (breaking the abstraction) in order for the synchronous operation to be used in selective communication.

Concurrent ML solves this problem by introducing *first-class synchronous operations*. The basic idea is to decouple the description of a synchronous operation (*e.g.*, "send the message $m$ on the channel $c$") from the actual act of synchronizing on the operation. To do so, we introduce a new kind of abstract value, called an *event*, which represents a potential synchronous operation. (This is analogous to the way in which a function value represents a potential computation.) The type constructor 'a event is used to generate the types of abstract synchronous operations; the type t event is the type of a synchronous operation that returns a value of type t when it is synchronized upon.

The basic event operations have the following signature:

```
val sync : 'a event -> 'a

val recvEvt : 'a chan -> 'a event
val sendEvt : 'a chan * 'a -> unit event

val choose : 'a event * 'a event -> 'a event
val wrap   : 'a event * ('a -> 'b) -> 'b event
val guard  : (unit -> 'a event) -> 'a event
```

The sync operator forces synchronization on an event value. (This is analagous to the way in which a function application forces the potential computation represented by a function value.)

The recvEvt and sendEvt operators represent channel communication. The recvEvt and sendEvt operators are called *base-event constructors*, because they create event values that describe a single primitive synchronous operation. We can define recv and send as follows:

```
val recv = fn ch => sync (recvEvt ch)
val send = fn (ch, x) => sync (sendEvt (ch, x))
```

Later, we will see a small number of other base-event constructors.

The power of first-class synchronous operations comes from *event combinators*, which can be used to build more complicated event values from the base-event values. The choose combinator provides a generalized selective communication mechanism; the wrap combinator augments an event with a post-synchronization action (called the wrapper function); the guard combinator creates and event from a pre-synchronous action (called the guard function). Note that it is important that both the wrapper function and the guard function may spawn threads and may perform synchronizations.

It is worth considering (informally) the semantics of event synchronization. An event value can be viewed as a tree, where the leaves correspond to the base events (*e.g.*, recvEvt and sendEvt) and applications of guard, and the internal nodes correspond to applications of choose, wrap. For example, consider the event value constructed by:

```
val g2 = fn () => (spawn e2 ; wrap (bev2, w2))
val g3 = fn () => (spawn e3 ; bev2)
val ev = choose (
           wrap (bev1, w1),
           wrap (choose (
                   guard g2,
                   wrap (guard g3, w3)
                 ), w4)
         )
```

where the $\text{bev}_i$ are base events, the $\text{g}_i$ are guard functions, and the $\text{w}_i$ are wrapper functions. The leftmost portion of Figure 3

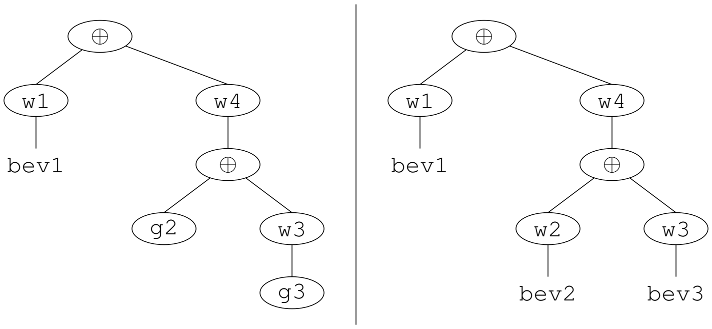

Figure 3: An event value

gives a pictorial representation of this event value, where `choose` nodes are labeled with ⊕, and `wrap` nodes are labeled with the wrapper functions, and `guard` nodes are labeled with the guard functions. When a thread evaluates `sync ev`, each of guard functions at the leaves is evaluated to an event value, which replaces the `guard` node; if the resulting event value has additional guard functions at the leaves, then they are evaluated and replaced, until the final event value has only base events at the leaves. The rightmost portion of Figure 3 gives a pictorial representation of this final event value. The thread blocks until one of the $\texttt{bev}_i$ is enabled by some other thread offering a matching communication. If multiple $\texttt{bev}_i$ are enabled by matching communications, then one is chosen nondeterministically. Once a pair of matching communications are chosen, the sender's base event returns `()` and the receiver's base event returns the message. The wrapper functions on the path from the selected base event to the root are applied to the result, producing the result of the synchronization. For example, if `bev2` is selected, with result `v`, then the result of `sync ev` is `w4(w2(v))`.

We can use the `choose` and `wrap` combinators to give more robust implementations of the `addStrms` and `copyStrm` combinators:

```
fun addStrms (inCh1, inCh2, outCh) =
  forever () (fn () =>
    let
       val (a, b) =
         sync (choose (
           wrap (recvEvt inCh1, fn a => (a, recv inCh2)),
           wrap (recvEvt inCh2, fn b => (recv inCh1, b))
         ))
    in
       send (outCh, a + b)
    end)

fun copyStrm (inCh, outCh1, outCh2) =
  forever () (fn () =>
    let val x = recv inCh
    in
       sync (choose (
         wrap (sendEvt (outCh1, x), fn () => send (outCh2, x)),
         wrap (sendEvt (outCh2, x), fn () => send (outCh1, x))
       ))
    end)
```

In the revised `addStrms` combinator, we are choosing between the operation of receiving a message on `inCh1` and the operation of receiving a message on `inCh2`; in each case, we use the `wrap` combinator to associate the action of receiving a message on the other channel. Similarly, in the revised `copyStrm` combinator, we are choosing between the operation of sending a message on `outCh1` and the operation of sending a message on `outCh2`; in each case, we use the `wrap` combinator to associate the action of sending the value on the other channel. Using these revised combinators in the implementation of the Fibonacci network avoids the subtle correctness issue; more importantly, it frees clients that use the revised `addStrms` and `copyStrm` combinators from needing to know their specific behavior (*i.e.*, the revised `addStrms` and `copyStrm` combinators are more abstract).

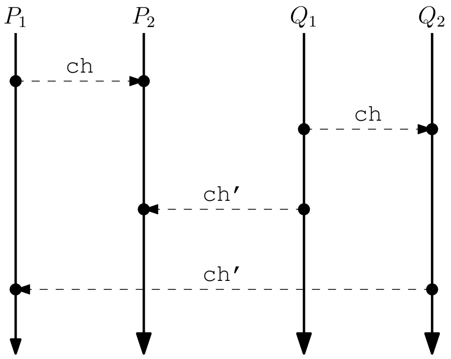

Figure 4: A swap mismatch

### 3.3.2 Example — Swap channels

A simple example of a communication abstraction that uses all of the event combinators in its implementation is the *swap channel*. This is a new type of channel that allows two processes to swap values when they rendezvous. We define the following abstract interface to swap channels:

```
signature SWAP_CHANNEL =
  sig
    type 'a swap_chan
    val swapChannel : unit -> 'a swap_chan
    val swapEvt : 'a swap_chan * 'a -> 'a event
  end
```

The operation `swapChannel` creates a new swap channel; the operation `swapEvt` is used to simultaneously send and receive a value on a swap channel. When two processes communicate on a swap channel, each sends a value and each receives a value; it is important that exactly two processes swap values.

Because swap channels provide symmetric message passing and the implementation is based on the asymmetric message-passing operations, each thread in a swap must offer to send a message and to receive a message on the same channel. The `choose` combinator suffices for this purpose. Once one thread has sent a value (and the other thread has received the value), a value must be sent in the other direction to complete the swap. We cannot use the channel on which the first value was sent, because other threads synchronizing on a `swapEvt` are trying to send and receive on that channel. We also cannot use another (dedicated) channel to complete the swap. For example, Figure 4 shows a swap mismatch. Threads $P_1$ and $P_2$ are matched (by sending and receiving on `ch`) and thread $Q_1$ and $Q_2$ are matched (by also sending and receiving on `ch`), but the values sent to complete the swap are mismatched: the value sent by $P_2$ on `ch'` (meant for $P_1$) is received by $Q_1$ and the value sent by $Q_2$ also on `ch'` (meant for $Q_2$) is received by $P_1$.

To avoid this problem, we allocate a fresh channel to complete the second phase of the swap operation each time the swap operation is executed. The implementation is as follows:

```
structure SwapChannel : SWAP_CHANNEL =
  struct
    datatype 'a swap_chan = SC of ('a * 'a chan) chan

    fun swapChannel () = SC (channel ())

    fun swapEvt (SC ch, msgOut) =
      guard (fn () =>
             let
               val inCh = channel ()
             in
               choose (
                 wrap (recvEvt ch, fn (msgIn, outCh) =>
                   (send (outCh, msgOut) ; msgIn)),
                 wrap (sendEvt (ch, (msgIn, inCh)), fn () =>
                   recv inCh)
               )
             end)
  end
```

A swap channel is represented by a single channel on which is sent both the value communicated in the first phase and a (private) channel on which is sent the value communicated in the second phase. By making the channel for the second phase private to instance of the swap operation ensures that there is never a mismatch when sending the value in the second phase. However, this channel for the second phase must be allocated after the synchronization begins (because it must private to this instance of the swap operation) and before the communication of the first value. This is precisely the behavior of the `guard` combinator.

The swap-channel abstraction illustrates several important programming techniques. The use of dynamically allocating a new channel to serve as the unique identifier for an operation is a common idiom. (Note that we used this technique in the implementation of updatable storage cells.) It is also an example that uses all of the event combinators: `choose`, `wrap`, and `guard`. Finally, it is a example that shows the utility of the event abstraction: clients that use the `SWAP_CELL` abstraction may treat the `swapEvt` operation as though it were a base event — it may be used in `choose`, `wrap`, and `guard` combinators.

## 3.4 Additional First-class Synchronous Operations

We can extend the set of primitive first-class synchronous operations discussed above with additional combinators, base-event constructors, and various miscellaneous operations.

### 3.4.1 Simple Base-event Constructors

Recall that the `recvEvt` and `sendEvt` base-event constructors are enabled when there is a matching communication being offered by another synchronizing thread. One can imagine two extreme cases of `recvEvt` — one in which there is *always* another thread offering the matching communication and one in which there is *never* another thread offering the matching communication.

It is useful to realize these two extremes as (primitive) base-event constructors:

```
val alwaysEvt : 'a -> 'a event
val neverEvt  : 'a event
```

The `alwaysEvt` constructor is used to build an event value that is always enabled for synchronization. The `neverEvt` constructor is used to build an event value that is never enabled for synchronization. Because a `neverEvt` can never be chosen for synchronization, it is the identity for the `choose` combinator; hence, it is useful for choosing from a list of event values:

```
val chooseList : 'a event list -> 'a event =
  fn l => List.foldl choose neverEvt l
```

(Note that the nondeterminism in `choose` makes it an associative and symmetric operator; the choice of fold direction is arbitrary.)

Note that one cannot implement a reliable polling mechanism by combining `alwaysEvt` and `choose`. For example, the following function does not accurately poll a channel for input:

```
fun recvPoll (ch : 'a chan) : 'a option =
  sync (choose (
    alwaysEvt NONE,
    wrap (recvEvt ch, fn x => SOME x)
  ))
```

Although it will never block, it may return NONE even when there is a matching communication available on ch. This is because the choice of enabled events to be returned by the synchronization is nondeterministic — the alwaysEvt may be chosen over the recvEvt.

### 3.4.2 Negative Acknowledgements

When programming in the client-server style, we can characterize servers as either idempotent or not idempotent. A server that is idempotent is one such that the handling of a given request is independent of other requests. A server that is not idempotent may be servicing multiple requests at once (having accepted multiple requests), where each client is blocking on the receive of a reply from the server. If a client receives the reply, then (due to the synchronous nature of message passing) the server knows that the client has completed the request-reply protocol. However, if the client uses the receive of the reply in a choose combinator and another event is chosen for synchronization, then the server cannot know that the client will never complete the request-reply protocol.

To ensure the correct semantics in this kind of situation, we need a mechanism for *negative acknowledgements*. The following event combinator provides such a mechanism:

```
val withNack : (unit event -> 'a event) -> 'a event
```

This combinator behaves like the guard combinator — it takes a function (which we continue to call the guard function) whose evaluation is delayed until synchronization time. The difference is that the function is applied to an *abort event*, which is enabled only if the event returned by the guard function is *not* chosen in the synchronization.

### 3.4.3 External Synchronous Events

As noted earlier, concurrent programming is a useful programming paradigm for interactive systems (*e.g.*, graphical-user interfaces). An interactive system must deal with multiple (asynchronous) input streams (*e.g.*, keyboard, mouse, network), multiple (asynchronous) output streams (*e.g.*, display, audio, network). Similarly, an interactive system often provides multiple services, where each service is largely independent, having is own internal state and control-flow. In sequential languages, these issues are often dealt with through complex event loops and callback functions.

Here, we describe the interface to various kinds of external synchronous events. By using first-class synchronous events, we can treat these external events using the same framework as internal sychronization.

**Intput/Output** An important form of external event is the availability of user input. A very simplistic account of input/output for a console application would be to take the standard input, output, and error streams to be character channels:

```
val stdInCh : char channel
val stdOutCh : char channel
val stdErrCh : char channel
```

Of course, it is not sensible to send on stdInCh or to receive on stdOutCh or stdErrCh. A better interface is to expose the streams as events:

```
val stdInEvt : char event
val stdOutEvt : char -> unit event
val stdErrEvt : char -> unit event
```

We can naturally extend this style of interface to accomodate events for reading from and writing to files, sending and receiving on a network socket, *etc.*

In practice, one builds a higher-level I/O library above these primitive operations, much as the Standard ML Basis Library [35] builds imperative I/O and stream I/O levels above the primitive I/O level, which provides abstractions of the underlying operating system's unbuffered I/O operations.

**Timeouts** Timeouts are another important example of external synchronous events. Most concurrent languages provide special mechanisms for "timing out" on a blocking operation. Using the framework of events, one can give base-event constructors for synchronizing on time events:

```
val timeOutEvt : time -> unit event
val atTimeEvt  : time -> unit event
```

The time type represents both absolute time and durations of time intervals. The timeOutEvt constructor takes a time value t (representing a time interval) and returns an event that becomes enabled at a time t units *relative* to the time at which the synchronization is performed. For example, a thread that evalutates the following expression will be delayed for one second:

```
sync (timeOutEvt (timeFromSeconds 1))
```

The atTimeEvt constructor takes a time value t (representing an absolute time) and returns an event that becomes enabled at time t.

Note that synchronization on time values is, by necessity, approximate. The granularity of the underlying system clock, scheduling delays in both the underlying operating system and the Manticore scheduler tend to delay synchronization on a time value slightly.

The fact that both input/output and timeouts are represented by events allows threads to combine them with other synchronous operations. For example, suppose that the program has prompted the user to enter Y or N, but wishes to proceed as though the user had entered N after a 10 second delay. This could be expressed by the following event:

```
choose (
  wrap (timeOutEvt (timeFromSeconds 10), fn () => #"N"),
  stdInEvt
)
```

## 3.5 Examples

We conclude with a few more examples of useful abstractions built atop the first-class synchronous operations.

### 3.5.1 Buffered Channels

The send and recv operations provide synchronous communication — both sender and receiver block until there is a matching communcation. It is sometimes useful to support asynchronous communication — a sender does not block (its message is buffered in the channel) and a receiver blocks until there is an available message. This buffering of communication can be useful when a cyclic communication pattern is required (as in the Fibonacci process network).

We define the following abstract interface to buffered channels:

```
signature BUFFERED_CHAN =
  sig
    type 'a buffered_chan
    val bufferedChan : unit -> 'a buffered_chan
    val bufferedSend    : 'a buffered_chan * 'a -> unit
    val bufferedRecvEvt : 'a buffered_chan -> 'a event
  end
```

As described above, a buffered channel consists of a queue of messages. The send operation adds a message to the queue without blocking the sender. The recvEvt operations attempts to take a message from the queue; if the queue is empty, it blocks until some other thread sends a message.

The implementation of buffered channels is similar to the implementation of the updatable storage cells: each time a buffered channel is created, a server thread is spawned to service requests to send on and receive on the channel.

```
structure BufferedChan : BUFFERED_CHAN =
  struct
    datatype 'a buffered_chan =
      BC of {inCh: 'a chan, outCh: 'a chan}

    fun bufferedSend (BC {outCh, ...}, x) = send (outCh, x)
    fun bufferedRecvEvt (BC {inCh, ...}) = recvEvt inCh

    fun bufferedChan () =
      let
         val (inCh, outCh) = (channel (), channel ())
         fun loop ([], []) = loop ([recv inCh], [])
           | loop ([], rear) = loop (rev rear, [])
           | loop (front as frontHd::frontTl, rear) =
               let
                 val (front', rear') =
                   sync (choose (
                     wrap (recvEvt inCh, fn y =>
                       (front, y::rear)),
                     wrap (sendEvt (outCh, frontHd), fn () =>
                       (frontTl, rear))
                   ))
               in
                 loop (front', rear')
               end
         val _ = spawn (loop ([], []))
      in
         BC {inCh = inCh, outCh = outCh}
      end

  end
```

### 3.5.2 Futures

Futures are a common mechanism for specifying parallel computation. (Indeed, many of the mechanisms in Section 4 can be seen as special cases of futures.) The future construct takes a computation, creates a (logically) separate thread and returns a placeholder (called a *future cell*) for the computation's result. The act of reading a value from a future cell is called *toucing*. If a thread attempts to touch a future, before the computation of its value is completed, then the thread blocks.

Implementing futures is straightforward. Since touching a future is a synchronous operation, we represent a future cell as an event value and we use `sync` to touch a value. We define the following abstract interface to buffered channels:

```
signature FUTURE =
  sig
    datatype 'a result = VAL of 'a | EXN of exn
    val future : ('a -> 'b) -> 'a -> 'b result event
  end
```

Because the evaluation of a future might result in a raised exception, we introduce the `result` type constructor to distinguish evaluation to a value (`VAL`) from evaluation to a raised exception (`EXN`). The implementation is quite simple:

```
structure Future : FUTURE =
  struct
    datatype 'a result = VAL of 'a | EXN of exn
    fun future f x =
      let
        val ch = channel ()
        val _ = spawn (
          let
            val r = (VAL (f x)) handle exn => EXN exn
          in
            forever () (fn () => send (ch, r))
          end)
      in
        recvEvt ch
      end
  end
```

To create a future, we create a channel on which to communicate the future result and spawn a thread to evaluate the computation and then repeatedly send the result on the channel. The future cell is represented by the event that receives the result on the channel.

Note that this is not a particularly efficient implementation of futures. A more efficient implementation can be build using synchronizing shared-memory (*e.g.*, *M-variables* and *I-variables* [55]), which themselves fit naturally into the framework of first-class synchronous events.

### 3.6 Conclusion

This section has discussed the explicit-concurrency mechanisms of Manticore, based upon those of Concurrent ML (CML). A much longer exposition, including detailed descriptions of non-trivial applications (a software build system; a concurrent window system; a distributed tuple-space), can be found in the book-length treatment of CML [71].

## 4 Implicit Parallelism in Manticore

### 4.1 Introduction

Manticore provides implicitly-threaded parallel versions of a number of sequential forms. These constructs can be viewed as *hints* to the compiler about which computations are good candidates for parallel execution; the semantics of (most of) these constructs is sequential and the compiler and/or runtime system may choose to execute them in a single thread.

There are number of distinct reasons for introducing implicitly-threaded parallel constructs into a language (in addition to the explictly-threaded concurrency constructs of Section 3). As noted in Section 1, parallel programming languages must provide mechanisms at multiple levels of parallelism, both because applications exhibit parallelism at multiple levels and because hardware requires parallelism at multiple levels to maximize performance. The implicitly-threaded parallel constructs are much better suited for expressing fine-grained parallelism (as might be executed using SIMD instructions). In a sense, the implicitly-threaded parallel constructs ease the burden for both the programmer and the compiler: the programmer is able to utilize simple parallel constructs, which efficiently (in terms of program text) express the desired parallelism, and the compiler is able to analyze and optimize these constructs, yielding programs that efficiently (in terms of time and computational resources) execute. Although the implicitly-threaded parallel constructs are necessarily more specific (and therefore less expressive) than the explicitly-threaded parallel constructs, this does not diminish their utility. Rather, they express common idioms of parallel computation and their limited expressiveness allows the compiler and runtime system to better manage the parallel computation.

Manticore introduces a number of implicitly-threaded parallel constructs:

- parallel arrays
- parallel tuples
- parallel bindings
- parallel cases

In addition to these implicitly-threaded parallel constructs visible in the source language, there is a general-purpose cancellation mechanism that is used to stop the (parallel) execution of computations when their results are guaranteed never to be demanded.

As noted above, the implicitly-threaded parallel constructs provide a parallel execution of a sequential semantics. Having a sequential semantics is useful in two ways: it provides the programmer with a deterministic programming model and it formalizes the expected behavior of the compiler. Specifically, the compiler must verify that the individual sub-computations in a parallel computation do not send or receive messages before executing the computation in parallel. Furthermore, if a sub-computation raises an exception, the runtime code must delay delivery of that exception until it has verified that all sequentially prior computations have terminated. Both of these restrictions require program analysis to implement efficiently.

## 4.2 Parallel Arrays

Support for parallel computations on arrays and matrices is common in parallel languages. The reason for this is that operations on arrays and matrices naturally express *data parallelism*, in which a single computation is performed in parallel across a large number of data elements. In Manticore, we support such computations using the nested parallel array mechanism inspired by NESL [7, 6, 8] and developed further by Nepal [15, 16, 49] and Data Parallel Haskell (DPH) [18, 17].

As might be expected, the type constructor `'a parray` is used to generate the types of parallel arrays, which are immutable sequences that can be computed in parallel. An important feature of parallel arrays is that they may be nested (*i.e.*, one can construct a parallel array of parallel arrays); multi-dimensional arrays need not be rectangular, which means that many irregular data structures can be represented. Furthermore, Manticore (like Nepal and DPH, but unlike NESL) supports parallel arrays of arbitrary types, admitting arrays of floating-point numbers, user-defined datatypes (*e.g.*, polymorphic lists or trees), functions, *etc.* Based on the parallel array element type and the parallel array operations, the compiler will map parallel array operations onto the appropriate parallel hardware (*e.g.*, operations on parallel arrays of floating-point numbers may be mapped onto SIMD instructions).

### 4.2.1 Parallel-Array Introduction

There are three basic expression forms that yield a parallel array. The simplest is an explicit enumeration of the expressions to be evaluated in parallel to yield the elements of the parallel array:

```
[| e1, ..., en |]
```

Thus, this parallel-array expression form constructs a parallel array of $n$ elements, where the **`[| |]`** delimiters alert the compiler that the $\texttt{e}_i$ may be evaluated in parallel.

Integer sequences are a common data structure in parallel algorithms. Manticore provides a parallel-array expression form for conveniently expressing integer sequences:

```
[| el to eh by es |]
```

This parallel-array expression form constructs a parallel array of integers, where the first element is `el`, the successive elements are `el + es`, `el + 2 * es`, ..., and the last element is `el + n * es` for some `n` such that `el + n * es <= eh`. For example, the expression

```
[| 1 to 31 by 10 |]
```

is equivalent to the expression

```
[| 1, 11, 21, 31 |]
```

If the step expression ("**`by`** `es`") is omitted, then it naturally defaults to `1`.

The final expression form that creates a parallel array is a *parallel-array comprehension*, which provides a concise description of a parallel loop. In its full generality, a parallel-array comprehension has the form:

```
[| e | x1 in ea1, ..., xn in ean where ep |]
```

where `e` is an expression that computes the elements of the array, the $\texttt{ea}_i$ are parallel-array expressions that provide inputs to `e`, and `ep` is a boolean expression that filters the input. (If the filter expression ("**`where`** `ep`" is omitted, then it naturally defaults to `true`.) If the input arrays $\texttt{ea}_i$ have different lengths, all are truncated to the length of the shortest input, and they are processed, in parallel, in lock-step.[4]

Parallel-array comprehensions can be used to specify both SIMD parallelism that is mapped onto vector hardware (*e.g.*, Intel's SSE instructions) and SPMD parallelism where parallelism is mapped onto multiple cores. For example, to double each

[4] This behavior is known as *zip semantics*, since the comprehension loops over the zip of the inputs. Both NESL and Nepal have zip semantics, but Data Parallel Haskell [18] has *Cartesian-product semantics* where the iteration is over the product of the inputs.

positive integer in a given parallel array of integers `nums`, one would use the following expression:

```
[| 2 * n | n in nums where n > 0 |]
```

This expression can evaluated efficiently in parallel using vector instructions. Two additional examples are the definitions of *parallel map* and *parallel filter* combinators; the former applies a function to each element of an array in parallel, while the latter discards elements of an array that do not satisfy a predicate:

```
fun mapP f xs = [| f x | x in xs |]
fun filterP p xs = [| x | x in xs where p x |]
```

Parallel-array comprehensions are first-class expressions; hence, the expressions defining the source parallel arrays of a comprehension can themselves be parallel-array comprehensions. For example, the main loop of a ray tracer generating an image of width `w` and height `h` can be written

```
[| [| trace(x,y) | x in [| 0 to w-1 |] |]
                 | y in [| 0 to h-1 |] |]
```

This parallel comprehension within a parallel comprehension is an example of *nested data parallelism*.

The sequential semantics of expression forms that create (and eliminate) parallel arrays is defined by mapping them to lists (see [33] or [75] for details). The main subtlety in the parallel implementation is that if an exception is raised when computing its $i$th element, then we must wait until all preceding elements have been computed before propagating the exception.

### 4.2.2 Parallel-Array Elimination

There are a number of basic expression forms that consume a parallel array. The parallel-array comprehension described above is one such form. Another simple elimination form is the subscript operator that extracts a single element of a parallel array:

```
ea ! ei
```

where `ea` is a parallel-array expression and `ei` is an integer expression. Parallel arrays are indexed by zero; if the index is outside the range of the array, then the `Subscript` exception is raised.

An important expression form that consumes a parallel array is a *parallel-array reduction*, which provides a concise description of a parallel loop. This operation is available through a combinator with the signature:

```
val reduceP : ('a * 'a -> 'a) -> 'a -> 'a array -> 'a
```

The expression `reduceP f b ea` is similar to folding the function `f` over the elements of `ea` using the base value `b`. The difference is that the function is applied in parallel to the elements, using a tree-like decomposition of the array elements. Hence, it is important that the function `f` is an associative function with `b` as a left zero.

An obvious application of a parallel-array reduction is to sum all of the elements of a parallel array of integers:

```
fun sumP a = reduceP (fn (x, y) => x + y) 0 a
```

Note that + is an associative function with `0` as a (left and right) zero.

### 4.2.3 Additional Parallel-Array Operations

Before turning to some more examples, we describe a number of additional parallel-array operations. Although these operations can be implemented in terms of the expression forms and operations given above, most of them have direct implementations, for efficiency.

Since parallel arrays are finite data structures, it is often useful to query the number of elements in the parallel array. The `lengthP` operation has the signature:

```
val lengthP : 'a parray -> int
```

One possible implementation of `lengthP` is the following:

```
fun lengthP a = sumP (mapP (fn _ => 1) a)
```

Although parallel-array comprehensions may be used to express many compuations, it is sometimes necessary to explicitly decompose a computation and explicitly combine the results. Hence, it is useful to be able to concatenate and flatten parallel arrays:

```
val concatP : 'a parray * 'a parray -> 'a parray
val flattenP : 'a parray parray -> 'a parray
```

Because concatP is an associative function with [| |] as a (left and right) zero, we can implement flattenP using reduceP and concatP:

```
fun flattenP a = reduceP concatP [| |] a
```

The concatenation of parallel arrays can be expressed as a comprehension:

```
fun concatP (a1, a2) =
  let
    val l1 = lengthP a1
    val l2 = lengthP a2
  in
    [| if i < l1 then a1 ! i else a2 ! (i - l1)
       | i in [| 0 to (l1 + l2 - 1) |] |]
  end
```

### 4.2.4 Examples

Parallel arrays are a natural representation for images:

```
type pixel = int * int * int
type img = pixel parray parray
```

We assume that a pixel represents the red, green, and blue components, each of which is in the range 0 to 255. Many image transformations can be expressed as a computation that is applied to each pixel of an image. For example, to convert a color image to a gray-scale image, we simply need to convert each color pixel to a gray-scale pixel:

```
fun rgbPixToGrayPix ((r, g, b) : pixel) : pixel =
  let
     val m = (r + g + b) / 3
  in
     (m, m, m)
  end
fun rgbImgToGrayImg (img : img) : img =
  [| [| rgbPixToGrayPix pix | pix in row |] | row in img |]
```

We can express the entire family of pixel-to-pixel transformations with a higher-order function:

```
fun xformImg (xformPix: pixel -> pixel) (img : img) : img =
  [| [| xformPix pix | pix in row |] | row in img |]
```

Operations on vectors and matrices are classic motivating examples for nested data parallelism. A parallel array can be used to represent both dense and sparse vectors:

```
type vector = real parray
type sparse_vector = (int * real) parray
```

A sparse matrix can be can be represented naturally as an array of rows, where each row is a sparse vector:

```
type sparse_matrix = sparse_vector parray
```

To multiply a sparse matrix by a dense vector, we simply compute the dot product for each row:

```
fun dotp (sv: sparse_vector) (v: vector) : real =
  sumP [| x * (v!i) | (i,x) in sv |]
fun smvm (sm: sparse_matrix) (v: vector) : vector =
  [| dotp (row, v) | row in sm |]
```

Note that smvm exprsses a nested parallel computation: dotp is applied to each row of the sparse matrix in parallel, while dotp is itself a parallel operation (comprised of both a parallel-array comprehension and a parallel-array reduction).

The quicksort algorithm is a common example in the nested data parallelism literature. We can implement quicksort in

Manticore as follows:

```
fun quicksort (a: int parray) : int parray =
  if lengthP a < 2
    then a
    else let
           val pivot = ns ! 0
           val ss = [| filterP cmp a
                       | cmp in [| fn x => x < pivot,
                                   fn x => x = pivot,
                                   fn x => x > pivot |] |]
           val rs = [| quicksort a | a in [| ss!0, ss!2 |] |]
           val sorted_lt = rs!0
           val sorted_eq = ss!1
           val sorted_gt = rs!1
         in
           flattenP [| sorted_lt, sorted_eq, sorted_gt |]
         end
```

In this implementation, the argument parallel array a is partitioned into elements less than, equal to, and greater than the pivot element. Note the use of a parallel-array comprehension over an array of comparison functions, which is another example of nested data parallelism. The arrays of less-than and greater-than elements are recursively sorted in parallel by using another parallel comprehension. Finally, the sorted arrays of elements are flattened into a single array of sorted elements.

## 4.3 Parallel Tuples

The parallel arrays of the previous section provide a very regular form of parallelism. However, it is sometimes more convenient to express a less regular form of parallelism. Parallel tuples are similar in spirit to the explicit-enumeration parallel-array expression form. The parallel-tuple expression form provides a hint to the compiler that the elements of the tuple may be evaluated in parallel:

```
(| e1, ..., en |)
```

Thus, this parallel-tuple expression form constructs a tuple of $n$ elements, where the (| |) delimiters alert the compiler that the $e_i$ may be evaluated in parallel.

A parallel tuple expresses a simple *fork/join* form of parallelism; each of the tuple components is evaluated in parallel and the computation of the tuple result blocks until all the sub-expressions are fully evaluated. Like parallel arrays, they enable the expression of computations with a high degree of parallelism in a very concise manner. Unlike parallel arrays, they support heterogeneous parallelism, because a tuple may be comprised of heterogeneous types and heterogeneous computations. Parallel tuples can thus avoid some awkwardness that can arise when using parallel arrays exclusively. For example, here is a revised quicksort implementation that uses both parallel arrays and parallel tuples to more naturally express the computation:

```
fun quicksort (a: int parray) : int parray =
  if lengthP a < 2
    then a
    else let
           val pivot = ns ! 0
           val (sorted_lt, sorted_eq, sorted_gt) =
             (| quicksort (filterP (fn x => x < pivot) a),
                filterP (fn x => x = pivot),
                quicksort (filterP (fn x => x > pivot) a) |)
         in
           flattenP [| sorted_lt, sorted_eq, sorted_gt |]
         end
```

The sequential semantics of parallel tuples is trivial: the expressions are evaluated in left-to-right order, just as they are for a (non-parallel) tuple. The implication for the parallel implementation is similar to that for parallel arrays: if an exception is raised when computing its $i$th element, then the implementation must wait until all preceding elements have been computed before propagating the exception.

Parallel tuples are convenient for expressing recursive functions, where the recursive calls can be evaluated in parallel. For example, here is a function to compute the binomial coefficient:

```
fun add (a, b) = a + b
fun choose (n, k) =
  if n = k then 1
  else if k = 0 then 1
  else add (| choose (n - 1, k), choose (n - 1, k - 1) |)
```

Similarly, here is a function to sum the leaves of a binary tree:

```
datatype tree = Lf of int | Br of tree * tree
fun trAdd t =
  case t of
     Lf i => i
   | Br (t1, t2) => add (| trAdd t1, trAdd t2 |)
```

As noted above, the implicitly-threaded parallel constructs are *hints* to the compiler about which computations are good candidates for parallel execution. As demonstrated by the previous examples, parallel tuples make it very easy to express parallel computations; indeed, they can often express more parallelism that can be effectively utilized. An important problem is to determine when parallel execution is likely to be profitable: the compiler and runtime must determine when the overhead of starting a parallel execution does not outweigh the benefits of parallel execution (else, sequential execution would be more efficient). By integrating analyses and transformations into the compiler and runtime system, we preserve a simple source langauge but provide sophisticated runtime behavior.

For example, the `trAdd` function concisely expresses the fact that one may sum the branches of a binary tree in parallel. However, it should be clear that summing *all* the branches of a binary tree in parallel would have poor performance: a balanced binary tree of depth $N$ would induce the creation of $2^N - 2$ parallel sub-computations. Realizing each of these sub-computations as an independent thread would quickly result in more threads than physical processors, and many threads would be blocked waiting for the completion of sub-computations. Even reifying each of these sub-computations as a unit of work for a collection of work-stealing threads would induce an unacceptable overhead. In order to achieve high-performance executions, it is necessary to ensure that there is sufficient (sequential) computation to warrant the overhead of parallel execution. For example, we might wish to transform the `treeAdd` function as follows:

```
datatype tree = Tr of int * tree’
     and tree’ = Lf’ of int | Br’ of tree * tree
fun Lf n = Tr (1, Lf’ n)
fun Br (t1 as Tr (d1, _), t2 as Tr (d2, _)) =
  Tr (max(d1, d2) + 1, Br’ (t1, t2))

fun trAdd (Tr (d,t’)) =
   if (d < 16 orelse numIdleProcs () < 2)
     then tr’Add_seq t’
     else tr’Add_par t’
and trAdd_seq (Tr (_,t’)) = tr’Add_seq t’
and tr’Add_seq t’ =
  case t’ of
     Lf’ i => n
   | Br’ (t1, t2) => add ( trAdd_seq t1, trAdd_seq t2 )
and tr’Add_par t’ =
  case t’ of
     Lf’ i => n
   | Br’ (t1, t2) => add (| trAdd t1, trAdd t2 |)
```

Under this transformation, the `tree` datatype maintains the depth of binary tree. We use the depth of the binary tree to ensure that any binary tree of depth less than 16 is summed as a sequential computation (`trAdd_seq` and `trAdd’_seq`). Similarly, we suppress parallel execution when there are insufficient computational resources available (`numIdleProcs () < 2`). An important point is that we would like this transformation to be *automatically* generated by the compiler. The original `trAdd` function is only two lines long and manifestly correct; the translated function above is significantly more complex. Making the analysis and transformation a duty compiler helps to ensure that the transformed program is semantically equivalent to the original program.

### 4.4 Parallel Bindings

Parallel arrays and tuples provide a fork-join pattern of computation, but in some cases more flexible scheduling is desirable. In particular, we may wish to execute some computations speculatively. Manticore provides a parallel binding form

```
pval p = e
```

that spawns the evaluation of the expression e as a parallel thread. The sequential semantics of a parallel binding are similar to lazy evaluation: the binding is only evaluated (and only evaluated once) when one of the variables bound in the pattern p is demanded by the evaluation of some expression in the main thread of control. One important subtlety in the semantics of parallel bindings is that any exceptions raised by the evaluation of the binding must be postponed until one of the variables is touched, at which time the exception is raised at the point of the touched variable.

The distinguishing characteristic of the parallel-binding declaration form is that the spawned computation may be canceled before completion. When a (simple, syntactic) program analysis determines the program points at which point a spawned computation is guaranteed never to be demanded, the compiler inserts a corresponding cancellation. Note these sites can only be located in conditional expression forms.

The following function computes the product of the leaves of a tree:

```
datatype tree = Lf of int | Br of tree * tree
fun trProd t=
  case t of
     Lf i => i
   | Br (t1, t2) =
     let
       pval p1 = trProd t1
       pval p2 = trProd t2
     in
       if p1 = 0
         then 0
         else p1 * p2
    end
```

This implementation short-circuits when the product of the left subtree of a Br variant evaluates to zero. Note that if the result of the left product is zero, we do not need the result of the right product. Therefore its subcomputation and any descendants may be canceled. The short-circuiting behavior is not explicit in the function; rather, it is implicit in the semantics of the parallel-binding declaration form that when control reaches a point where the result of an evaluation is known to be unneeded, the resources devoted to that evaluation are freed and the computation is abandoned.

The analysis to determine when a future is subject to cancellation is not as straightforward as it might seem. The following example includes two parallel bindings linked by a common computation:

```
let
  pval x = f 0
  pval y = (| g 1, x |)
in
  if b then x else h y
end
```

In the conditional expression here, while the computation of y can be canceled in the **then** branch, the computation of x cannot be canceled in either branch. Our analysis must respect this dependency and similar subtle dependencies.

We will give more examples of the use of parallel bindings in Section 4.7. However, as a very simple example, we note that the behavior of parallel tuples may be encoded using parallel bindings; in particular, we encode `(| e1, ..., en |)` as

```
let
  pval x1 = e1
  ...
  pval xn = en
in
   (x1, ..., xn)
end
```

## 4.5 Parallel Cases

The parallel-case expression form is a nondeterministic counterpart to Standard ML's sequential-case expression form. In a parallel-case expression, the discriminants are evaluated in parallel and the match rules may include wildcard patterns that match even if their corresponding discriminants have not yet been fully evaluated. Thus, a parallel case expression nondeterministically takes any match rule that matches after sufficient discriminants have been evaluated. The parallel-case expression form leverages the familiar pattern-matching idom and is flexible enough to express a variety of non-deterministic parallel mechanisms.

Unlike the other implicitly-threaded mechanisms, the parallel-case exprssion form is nondeterministic. We can still give a sequential semantics, but it requires including a source of non-determinism (*e.g.*, McCarthy's **amb** [51]), in the sequential language.

In many respects, the parallel-case expression form is syntactically similar to the sequential-case expression form:

```
pcase e1 & ... & en of
    pp11 & ... & pp1n => e'1
  | ...
  | ppm1 & ... & ppmn => e'm
  | otherwise => e
```

The metavariable `pp` denotes a *parallel pattern*, which is either

- a nondeterministic wildcard **?**,
- a handle pattern **handle** p, or
- a pattern p,

where p in the latter two cases signifies a conventional SML pattern. Furthermore, **pcase** expressions include an optional **otherwise** branch (which must be the last branch) which has a special meaning as discussed below.

A *nondeterministic wildcard pattern* can match against a computation that is either finished or not. It is therefore different than the usual SML wildcard, which matches against a finished computation, albeit one whose result remains unnamed. Nondeterministic wildcards can be used to implement short-circuiting behavior. Consider the following parallel-case branch:

```
| false & ? => 9
```

Once the constant pattern `false` has been matched with the result of the first discriminant's computation, the running program need not wait for the second discriminant's computation to finish; it can immediately return `9`.

A *handle pattern* catches an exception if one is raised in the computation of the corresponding discriminant. It may furthermore bind the raised exception to a pattern for use in subsequent computation.

We can transcribe the meaning of **otherwise** concisely, using SML/NJ-style or-patterns in our presentation for brevity. An **otherwise** branch can be thought of as a branch of the form:

```
| (_ | handle _) & ... & (_ | handle _) => e
```

The fact that every position in this pattern is either a deterministic wildcard or a handle means it can only match when all computations are finished. It also has the special property that it takes lowest precedence when other branches also match the evaluated discriminants. In the absence of an explicit **otherwise** branch, a parallel-case expression is evaluated as though it were specified with the following branch:

```
| otherwise => raise Match
```

To illustrate the use of parallel case expressions, we consider parallel choice. A parallel choice expression `e1 |?| e2` nondeterministically returns either the result of `e1` or `e2`. This is useful in a parallel context, because it gives the program the opportunity to return whichever of `e1` or `e2` evaluates first.

We might wish to write a function to obtain the value of some lear of a given tree:

```
datatype tree = Lf of int | Br of tree * tree
fun trPick t =
  case t of
     Lf i => i
   | Br (t1, t2) = (trPick t1) |?| (trPick t2)
```

This function evaluates `trPick(t1)` and `trPick(t2)` in parallel. Whichever evaluates to a value first, loosely speaking, determines the value of the choice expression as a whole. Hence, the function is likely, but not required, to return the value of the shallowest leaf in the tree. Furthermore, the evaluation of the discarded component of the choice expression—that is, the one whose result is not returned—is canceled, as its result is known not to be demanded. If the computation is running, this cancellation will free up computational resources for use elsewhere. If the computation is completed, this cancellation will be a harmless idempotent operation.

The parallel choice operator is a derived form in Manticore, as it can be expressed as a **pcase** in a straightforward manner. The expression `e1 |?| e2` is desugared to:

```
pcase e1 & e2 of
    x & ? => x
  | ? & x => x
```

Parallel case gives us yet another to write the `trProd` function:

```
datatype tree = Lf of int | Br of tree * tree
fun trProd t =
  case t of
     Lf i => i
   | Br (t1, t2) =>
       (pcase trProd t1 & trProd t2 of
           0 & ? => 0
         | ? & 0 => 0
         | p1 & p2 => p1 * p2)
```

This function will short-circuit when either the first or second branch is matched, implicitly canceling the computation of the other sub-tree. Because it is nondeterministic as to which of the matching branches is taken, a programmer should ensure that all branches that match the same discriminants yield acceptable results. For example, if `trProd(t1)` evaluates to `0` and `trProd(t2)` evaluates to `1`, then either the first branch or the third branch may be taken, but both will yield the result `0`.

As a third example, consider a function to find a leaf value in a tree that satisfies a given predicate. The function should return an `int option` to account for the possibility that no leaf value in the tree match the predicate. We might mistakenly write the following code:

```
fun trFind (p, t) =
  case t of
     Lf i => if p i then SOME i else NONE
   | Br (t1, t2) => (trFind (p, t1)) |?| (trFind (p, t2))
```

In the case where the predicate `p` is not satisfied by any leaf values in the tree, this implementation will always return `NONE`, as it should. However, if the predicate is satisfied at some leaf, the fuction will nondeterministically return either `SOME n`, for a satisfying `n`, or `NONE`. In other words, this implementation will never return a false positive, but it will, nondeterministically, return a false negative. The reason for this is that as soon as one of the operands of the parallel choice operator evaluates to `NONE`, the evaluation of the other operand might be canceled, even if it were to eventually yield `SOME n`.

A correct version of `trFind` may be written as follows:

```
fun trFind (p, t) =
  case t of
     Lf i => if p i then SOME i else NONE
   | Br (t1, t2) =>
       (pcase trFind (p, t1) & trFind (p, t2) of
           SOME i & ? => SOME i
         | ? & SOME i => SOME i
         | NONE & NONE => NONE)
```

This version of `trFind` has the desired behavior. When either `trFind(p, t1)` or `trFind(p, t2)` evaluates to `SOME n`, the function returns that value and implicitly cancels the other evaluation. The essential computational pattern here is a parallel abort mechanism, a common device in parallel programming.

A parallel case can also be used to encode a short-circuiting parallel boolean conjunction expression. We first consider some possible alternatives. We can attempt to express parallel conjunction in terms of parallel choice using the following strategy. We mark each expression with its originating position in the conjunction; after making a parallel choice between the two marked expressions, we can determine which result to return. Thus, we can write an expression that always assumes the correct value, although it may generate redundant computation:

```
datatype z = L | R
val r = case (e1, L) |?| (e2, R) of
           (false, L) => false
          | (false, R) => false
         | (true, L) => e2
         | (true, R) => e1
```

This expression exhibits the desired short-circuiting behavior in the first two cases, but in the latter cases it must restart the other computation, having canceled it during the evaluation of the parallel choice expression. So, while this expression always returns the right answer, in non-short-circuiting cases its performance is no better than sequential, and probably worse.

We encounter related problems when we attempt to write a parallel conjunction in terms of **pval**, where asymmetries are inescapable.

```
val r =
  let
    pval b1 = e1
    pval b2 = e2
  in
    if (not b1)
      then false
      else e2
  end
```

This short-circuits when `e1` is false, but not when `e2` is false. We cannot write a parallel conjunction in terms of **pval** such that either subcomputation causes a short-circuit when false.

The **pcase** mechanism offers the best encoding of parallel conjunction:

```
val r =
  pcase e1 & e2 of
      false & ? => false
    | ? & false => false
    | true & true => true
```

Only when both evaluations complete and are `true` does the expression as a whole evaluate to `true`. If one constituent of a parallel conjunction evaluates to `false`, the other can be safely canceled. As soon as one expression evaluates to `false`, the other is canceled, and `false` is returned. As a convenience, Manticore provides |**andalso**| as a derived form for this expression pattern.

In addition to |**andalso**|, we provide a variety of other similar derived parallel forms whose usage we expect to be common. Examples include |**orelse**|, `|*|` (parallel multiplication, short-circuiting with `0`), and parallel maximum and minimum operators for numeric types. Because Manticore has a strict evaluation semantics for the sequential core language, such operations cannot be expressed as simple functions: to obtain the desired parallelism, the subcomputations must be unevaluated expressions. Thus, it may be desirable to provide a macro facility that enables a programmer to create her own novel syntactic forms in the manner of these operations.

## 4.6 Exceptions

The interaction of exceptions and parallel constructs must be considered in the implementation of the parallel constructs. Raises and exception handlers are first-class expressions, and, hence, they may appear at arbitrary points in a program, including in a parallel construct. For example, the following is a legal parallel-array expression:

```
[| 2+3, 5-7, raise A |]
```

Evaluating this parallel array expression should raise the exception `A`.

Note the following important detail. Since the compiler and runtime system are free to execute the subcomputations of a parallel array expression in any order, there is no guarantee that the first **raise** expression observed during the parallel execution corresponds to the first **raise** expression observed during a sequential execution. Thus, some compensation is required to ensure that the sequentially first exception in a given parallel array (or other implicitly-threaded parallel construct) is raised whenever multiple exceptions could be raised. Consider the following minimal example:

```
[| raise A, raise B |]
```

Although the exception `B` might be raised before `A` during a parallel execution, `A` must be the exception observed to be raised by the context of the parallel array expression in order to adhere to the sequential semantics. Realizing this behavior in this and other parallel constructs requires our implementation to include compensation code, with some runtime overhead.

In choosing to adopt a strict sequential core language, Manticore is committed to realizing a precise exceptions semantics in the implicitly-threaded parallel features of the language. This is in contrast to an imprecise exception semantics [59] that arise from a lazy sequential language. While a precise semantics requires a slightly more restrictive implementation of the implicitly-threaded parallel features than would be required with an imprecise semantics, we believe that support for exceptions and the precise semantics is crucial for systems programming. Furthermore, implementing the precise exception semantics is not particularly onerous.

It is possible to eliminate some or all of the compensation code with the help of program analyses. There already exist various well-known analyses for identifying exceptions that might be raised by a given computation [80, 48]. If, in a parallel array expression, it is determined that no subcomputation may raise an exception, then we are able to omit the compensation code and its overhead. As another example, consider a parallel array expression where all subcomputations can raise only one and the same exception.

```
[| if x<0 then raise A else 0,
   if y>0 then raise A else 0 |]
```

The full complement of compensation code is unnecessary here, since any exception raised by any subcomputation must be the exception A.

Although exception handlers are first-class expressions, their behavior is orthogonal to that of the parallel constructs and mostly merit no special treatment in the implementation.

Note that when an exception is raised in a parallel context, the implementation should free any resources devoted to parallel computations whose results will never be demanded by virtue of the control-flow of raise. For example, in the parallel tuple

```
(| raise A, fact(100), fib(200) |)
```

the latter two computations should be abandoned as soon as possible.

## 4.7 Examples

We consider a few examples to illustrate the use and interaction of our language features in familiar contexts. We choose examples that stress the parallel binding and parallel case mechanisms of our design, since examples exhibiting the use of parallel arrays and comprehensions are covered well in the existing literature.

### 4.7.1 A Parallel Typechecking Interpreter

First we consider an extended example of writing a parallel typechecker and evaluator for a simple model programming language. The language in question, which we outline below, is a pure expression language with some basic features including boolean and arithmetic operators, conditionals, let bindings, and function definition and application. A program in this language can, as usual, be represented as an expression tree. Both typechecking and evaluation can be implemented as walks over expression trees, in parallel when possible. Furthermore, the typechecking and evaluation can be performed in parallel with one another. In our example, failure to type a program successfully implicitly cancels its simultaneous evaluation.

While this is not necessarily intended as a realistic example, one might wonder why parallel typechecking and evaluation is desirable in the first place. First, typechecking constitutes a single pass over the given program. If the program involves, say, recursive computation, then typechecking might finish well before evaluation. If it does, and if there is a type error, the presumably doomed evaluation will be spared the rest of its run. Furthermore, typechecking touches all parts of a program; evaluation might not.

Our language includes the following definition of types:

```
datatype ty = NatTy | BoolTy | ArrowTy of ty * ty
```

For the purposes of yielding more useful type errors, we assume each expression consists of a location (some representation of its position in the source program) and a term (its computational part). These are represented by the following datatype definition:

```
datatype term =
   NatTerm of int
 | AddTerm of exp * exp
 | BoolTerm of bool
 | IfTerm of exp * exp * exp
 | VarTerm of var
 | LetTerm of var * exp * exp
 | LamTerm of var * ty * exp
 | AppTerm of exp * exp
 ...
withtype exp = loc * term
```

We assume that variables in the parse tree are uniquely identified.

For typechecking, we need a function that checks the equality of types. When we compare two arrow types, we can compare the domains of both types in parallel with comparison of the ranges. Furthermore, if either the domains or the ranges turn out to be not equal, we can cancel the other comparison. Here we encode this, in the `ArrowTy` case, as an explicit short-circuiting parallel computation:

```
fun tyEq (ty1, ty2) =
  case (ty1, ty2) of
    (BoolTy, BoolTy) => true
  | (NatTy, NatTy) => true
  | (ArrowTy (ty1a, ty1r), ArrowTy (ty2a, ty2r)) =>
      (pcase tyEq (ty1a, ty2a) & tyEq (ty1r, ty2r) of
         false & ? => false
       | ? & false => false
       | true & true => true)
  | _ => false
```

In practice, we could use the parallel-and operator `|andalso|` for the `ArrowTy` case

```
tyEq (ty1a, ty2a) |andalso| tyEq (ty1r, ty2r)
```

which would desugar into the expression explicitly written above.

We present a parallel typechecker as a function `typeOfExp` that consumes an environment (a map from variables to types) and an expression. It returns either a type, in the case that the expression is well-typed, or an error, in the case that the expression is ill-typed. We introduce a simple union type to capture the notion of a value or an error.

```
datatype 'a res = Ans of 'a | Err of loc
```

The signature of `typeOfExp` is

```
val typeOfExp : env * exp -> ty res
```

We consider a few representative cases of the `typeOfExp` function. To typecheck an `AddTerm` node, we can simultaneously check both subexpressions. If the first subexpression is not of type `NatTy`, we can record the error and implicitly cancel the checking of the second subexpression. The function behaves similarly if the first subexpression returns an error. Note the use of a sequential **`case`** inside a **`pval`** block to describe the desired behavior.

```
fun typeOfExp (G, e as (loc, term)) =
  case term of
     NatTerm _ => Ans NatTy
   | AddTerm (e1, e2) =
       let
         pval rty2 = typeOfExp (G, e2)
       in
         case typeOfExp (G, e1) of
            Ans NatTy =>
              (case rty2 of
                  Ans NatTy => Ans NatTy
                | Ans _ => Err (locOf e2)
                | Err loc => Err loc)
          | Ans _ => Err (locOf e1)
          | Err loc => Err loc
       end
```

The conditional case is similar to the add case. Its first component must have type `BoolTy`, and its second and third components must have the same type as one another.

```
   | BoolTerm _ => Ans BoolTy
   | IfTerm (e1, e2, e3) =
       let
         pval rty2 = typeOfExp (G, e2)
         pval rty3 = typeOfExp (G, e3)
       in
         case typeOfExp (G, e1) of
            Ans BoolTy =>
              (case (rty2, rty3) of
                  (Ans ty2, Ans ty3) =>
                    if tyEq (ty2, ty3)
                      then Ans ty2
                      else Err (locOf e)
                | (Err loc, _) => Err loc
                | (_, Err loc) => Err loc)
          | Ans _ => Err (locOf e1)
          | Err loc => Err loc
       end
```

In the `Apply` case, we require an arrow type for the first subexpression and the appropriate domain type for the second.

```
   | ApplyTerm (e1, e2) =
       let
         pval rty2 = typeOfExp (G, e2)
       in
         case typeOfExp (G, e1) of
            Ans (ArrowTy (ty11, ty12)) =>
              (case rty2 of
                  Ans ty2 =>
                    if tyEq (ty2, ty11)
                      then Ans ty12
                      else Err (locOf e2)
                | Err loc => Err loc)
          | Ans _ => Err (locOf e1)
          | Err loc => Err loc
       end
```

Of course, when there are no independent subexpressions, no parallelism is available:

```
| VarTerm var =>
    (case envLookup (G, var) of
        NONE => Err (locOf e)
      | SOME ty => Ans ty)
| LamTerm (var, ty, e) =>
    (case typeOfExp (envExtend (G, (var, ty)), e) of
        Ans ty’ => Ans (ArrowTy (ty, ty’))
      | Err loc => Err loc)
```

However, the representation of the environment (*e.g.*, as balanced binary tree) may enable parallelism in the `envLookup` and `envExtend` functions.

Throughout these examples, the programmer rather than the compiler is identifying opportunities for parallelism.

We have also written the `typeOfExp` function to report the earliest error when one exists. If we wished to report any error when multiple errors exist, then we could use a parallel case:

```
| ApplyTerm (e1, e2) =
    (pcase typeOfExp (G, e1) & typeOfExp (G, e2) of
        Ans ty1 & Ans ty2 =>
          (case ty1 of
              ArrowTy (ty11, ty12) =>
                if tyEq (ty11, ty2)
                  then Ans ty12
                  else Err (locOf e2)
            | _ => Err (locOf e1))
      | Err loc & ? => Err loc
      | ? & Err loc => Err loc)
```

For evaluation, we need a function to substitute a term for a variable in an expression. Substitution of closed terms for variables in a pure language is especially well-suited to a parallel implementation. Parallel instances of substitution are completely independent, so no subtle synchronization or cancellation behavior is ever required. Parallel substitution can be accomplished by means of our simplest parallel construct, the parallel tuple. We show a few cases here.

```
fun substExp (t, x, e as (p, t’)) =
  (p, substTerm (t, x, t’))
and substTerm (t, x, t’) =
  case t’ of
     NumTerm n => NumTerm n
   | AddTerm (e1, e2) =>
       AddTerm (| substExp (t, x, e1),
                  substExp (t, x, e2) |)
   | BoolTerm b => BoolTerm b
   | IfTerm (e1, e2, e3)
       IfTerm (| substExp (t, x, e1),
                 substExp (t, x, e2),
                 substExp (t, x, e3) |)
   (* ... *)
```

Like the parallel typechecking function, the parallel evaluation function simultaneously evaluates subexpressions. Since we are not interested in identifying the first runtime error (when one exists), we use a parallel case:

```
exception EvalError
fun evalExp (p, t) =
  case t of
     NumTerm n => NumTerm n
   | AddTerm (e1, e2) =>
       (pcase evalExp e1 & evalExp e2 of
           NumTerm n1 & NumTerm n2 => NumTerm (n1 + n2)
         | otherwise => raise EvalError)
```

The IfTerm case is notable in its use of speculative evaluation of both branches. As soon as the test completes, the abandoned branch is implicitly canceled.

```
| IfTerm (e1, e2, e3) =>
    let
      pval v2 = evalExp e2
      pval v3 = evalExp e3
    in
      case evalExp e1 of
         BoolTerm true => v2
       | BoolTerm false => v3
       | _ => raise EvalError
    end
```

We conclude the example by wrapping typechecking and evaluation together into a function that runs them in parallel. If the typechecker discovers an error, the program implicitly cancels the evaluation. Note that if the evaluation function raises a EvalError exception before the typechecking function returns an error, it will be harmlessly canceled. If the typechecking function returns any type at all, we simply discard it and return the value returned by the evaluator.

```
fun typedEval e : term res =
  pcase typeOfExp (emptyEnv, e) & evalExp e of
     Err loc & ? => Err loc
   | Ans _ & v => Ans v
```

### 4.7.2 Parallel Game Search

We now consider the problem of searching a game tree in parallel. This has been shown to be a successful technique by the Cilk group for games such as Pousse [5] and chess [19].

For simplicity, we consider the game of tic-tac-toe. Every tic-tac-toe board is associated with a score: 1 if X holds a winning position, ~1 if O holds a winning position, and 0 otherwise. We use the following polymorphic rose tree to store a tic-tac-toe game tree.

```
datatype 'a rose_tree =
   RoseTree of 'a * 'a rose_tree parray
```

Each node contains a board and the associated score, and every path from the root of the tree to a leaf encodes a complete game.

A player is either of the nullary constructors X or O; a board is a parallel array of nine player options, where NONE represents an empty square.

```
datatype player = X | O
type board = player option parray
```

Extracting the available positions from a given board is written as a parallel comprehension as follows:

```
fun availPositions (b: board) : int parray =
  [| i | s in b, i in [| 0 to 8 |] where isNone s |]
```

Generating the next group of boards given a current board and a player to move is also a parallel comprehension:

```
fun succBoards (b: board, p: player) : board parray =
  [| mapP (fn j => if i = j then SOME p else b!j) [| 0 to 8 |]
     | i in availPositions b |]
```

With these auxiliaries in hand we can write a function to build the full game tree using the standard minimax algorithm, where each player assumes the opponent will play the best available move at the given point in the game.

```
fun maxP a = reduceP (fn (x, y) => max (x, y)) ~1 a
fun minP a = reduceP (fn (x, y) => min (x, y)) 1 a
fun minimax (b: board, p: player) : board rose_tree =
  if gameOver b
    then RoseTree ((b, boardScore b), [| |])
    else let
           val ss = succBoards (b, p)
           val ch = [| minimax (b, flipPlayer p) | b in ss |]
           val chScores = [| treeScore | t in ch |]
         in
           case p of
              X => Rose ((b, maxP chScores), ch)
            | O => Rose ((b, minP chScores), ch)
         end
```

Note that at every node in the tree, all subtrees can be computed independently of one another, as they have no interrelationships. Admittedly, one would not write a real tic-tac-toe player this way, as it omits numerous obvious and well-known improvements. Nevertheless, as written, it exhibits a high degree of parallelism and performs well relative both to a sequential version of itself in Manticore and to similar programs in other languages.

Using alpha-beta pruning yields a somewhat more realistic example. We implement it here as a pair of mutually recursive functions, `maxT` and `minT`:

```
fun maxT (b, alpha, beta) =
  if gameOver board
    then RoseTree ((b, boardScore b), [| |])
    else let
           val ss = succBoards (b, p)
           val t0 = minT (ss!0, alpha, beta)
           val alpha' = max (alpha, treeScore t0)
           fun loop i =
             if i = lengthP ss
               then [| |]
               else let
                      pval ts = loop (i + 1)
                      val ti = minT (ss!i, alpha', beta)
                    in
                      if (treeScore ti) >= beta
                        then [| ti |] (* prune *)
                        else concatP ([| ti |], ts)
                    end
            val ch = concatP ([| t0 |], loop 1)
            val chScores = [| treeScore | t in ch |]
          in
            Rose ((b, maxP chScores), ch)
          end
 and minT (b, alpha, beta) = (* symmetric *)
```

Alpha-beta pruning is an inherently sequential algorithm, so we must adjust it slightly. This program prunes subtrees at a particular level of the search tree if they are at least as disadvantageous to the current player as an already-computed subtree. (The sequential algorithm, by contrast, considers every subtree computed thus far.) We compute one subtree sequentially as a starting point, then use its value as the pruning cutoff for the rest of the sibling subtrees. Those siblings are computed in parallel by repeatedly spawning computations in an inner loop by means of **`pval`**. Pruning occurs when the implicit cancellation of the **`pval`** mechanism cancels the evaluation of the right siblings of a particular subtree.

## 4.8 Conclusion

This section has discussed the implicit-parallelism mechanisms of Manticore. Although many of these mechanisms appear simple, that is a significant contribution to their appeal — they provide light-weight syntactic hints of available parallelism, relieving the programmer from the burden of orchestrating the computation. Furthermore, since **val** declaration bindings and **case** expressions are essential idioms in a functional programmer's repertoire, providing implicitly-threaded forms allows parallelism to be expressed in a familiar style.

# 5 Implementation of Manticore

## 5.1 Overview

Our initial implementation of the Manticore system, which consists of a compiler and a runtime system, targets the 64-bit version of the x86 architecture (*a.k.a.* X86-64 or AMD64) under the Linux and MacOS X operating systems.

As is typical in implementations of high-level languages, our implementation consists of a number of closely related components: a compiler (written in Standard ML) for the Manticore language, a runtime kernel (written in C and assembler) that implements garbage collection and various machine-level scheduler operations, and a framework for nested schedulers that provides the implementation of language-level parallel constructs. This scheduler framework is implemented using one of the compiler's intermediate representations as the programming language (specifically, it uses the BOM IR, which can be thought of as a low-level language in the style of ML). The combination of the runtime kernel and the scheduling framework define the runtime system for the Manticore language.

Two important themes of our implementation are the use of first-class continuations and our notions of process abstraction. We discuss these topics before describing the compiler, runtime kernel, and scheduling framework in the following sections.

### 5.1.1 Continuations

Continuations are a well-known language-level mechanism for expressing concurrency [79, 44, 68, 76]. Continuations come in a number of different strengths or flavors.

1. *First-class* continuations, such as those provided by SCHEME and SML/NJ, have unconstrained lifetimes and may be used more than once. They are easily implemented in a continuation-passing style compiler using heap-allocated continuations [2], but map poorly onto stack-based implementations.

2. *One-shot* continuations [11] have unconstrained lifetimes, but may only be used once. The one-shot restriction makes these more amenable for stack-based implementations, but their implementation is still complicated. In practice, most concurrency operations (but not thread creation) can be implemented using one-shot continuations.

3. *Escaping* continuations[5] have a scope-limited lifetime and can only be used once, but they also can be used to implement many concurrency operations [64, 28]. These continuations have a very lightweight implementation in a stack-based framework; they are essentially equivalent to the C library's `setjmp`/`longjmp` operations.

In Manticore, we are using continuations in the BOM IR to express concurrency operations. For our prototype implementation, we are using heap-allocated continuations *à la* SML/NJ [2]. Although heap-allocated continuations impose some extra overhead (mostly increased GC load) for sequential execution, they provide a number of advantages for concurrency:

- Creating a continuation just requires allocating a heap object, so it is fast and imposes little space overhead ($<$ 100 bytes).

- Since continuations are *values*, many nasty race conditions in the scheduler can be avoided.

- Heap-allocated first-class continuations do not have the lifetime limitations of escaping and one-shot continuations, so we avoid prematurely restricting the expressiveness of our IR.

- By inlining concurrency operations, the compiler can optimize them based on their context of use [28].

### 5.1.2 Process Abstractions

Our infrastructure has three distinct notions of process abstraction. At the lowest level, a *fiber* is an unadorned thread of control. We use unit continuations to represent the state of suspended fibers.

Surface-language *threads* (those created with **spawn**) are represented as fibers paired with a thread ID. Since threads may initiate implicit-parallel computations, a thread may consist of multiple fibers.

Lastly, a *virtual processor* (vproc) is an abstraction of a hardware processor resource. A vproc runs at most one fiber at a time, and furthermore is the only means of running fibers. The vproc that is currently running a fiber is called the *host vproc* of the fiber.

[5] The term "escaping continuation" is derived from the fact that they can be used to *escape*.

## 5.2 The Manticore Compiler

As is standard, the Manticore compiler is structured as a sequence of translations between intermediate languages (IRs). There are six distinct IRs in our compiler:

1. Parse tree — the product of the parser.
2. AST — an explicitly-typed abstract-syntax tree representation.
3. BOM — a direct-style normalized $\lambda$-calculus.
4. CPS — a continuation-passing-style $\lambda$-calculus.
5. CFG — a first-order control-flow-graph representation.
6. MLTree — the expression tree representation used by the MLRISC code generation framework [39].

With the exception of the parse tree, each of these representations has a corresponding collection of optimizations. In the case of the MLTree representation, MLRISC provides a number of application-independent optimizations, such as register allocation and peephole optimization, which we do not discuss.

### 5.2.1 AST Optimizations

We use a series of transformations on the AST representation to simplify the program for later stages. These include transformations that implement the threading policies for the implicitly-parallel constructs, which we describe briefly below (Section 5.3); a more detailed account of these transformations is given in other publications [75, 32]. Lastly, we compile nested patterns to simpler decision trees using Pettersson's technique [58].

### 5.2.2 BOM Optimizations

The BOM representation is a normalized direct-style $\lambda$-calculus where every intermediate result is bound to a variable and all arguments are variables.[6] This representation has several notable features:

- It supports first-class continuations with a binding form that reifies the current continuation. This mechanism is used to express the various operations on threads (see Section 5.5) [79, 63, 71].
- It includes a simplified form of SML datatypes with simple pattern matching. These are included to allow BOM code to be independent of datatype representations.
- It includes *high-level operators*, which are used to abstract over the implementation of various higher-level operations, such as thread creation, message passing, parallel loops, *etc.* We are also working on a *domain-specific* rewriting system for high-level operators that we will use to implement various optimizations [60].
- It also includes atomic operations, such as *compare-and-swap* (**cas**).

In this paper, however, we use SML syntax to write BOM code, since it is more compact than the actual syntax. Continuations are supported in the BOM IR by the **cont** binding form for introducing continuatsions and the **throw** exprssion form for applying continuations. The **cont** binding:

```
let cont k x = e in body end
```

binds `k` to the first-class continuation:

```
fn x => (throw k' e)
```

where `k'` is the continuation of the whole expression.The scope of `k` includes both the expression `body` and the expression `e` (*i.e.*, `k` may be recursive). Continuations have indefinite extent and may be used multiple times.[7]

A couple of examples will help illustrate this mechanism. The traditional `callcc` function can be defined as

```
fun callcc f = let cont k x = x in f k end
```

Here we use the **cont** binding to reify `callcc`'s return continuation. The **cont** binding form is more convenient than

[6] BOM is the successor of the BOL intermediate representation used in the Moby compiler. The name "BOL" does not stand for anything; rather, it is the lexical average of the acronyms "ANF" and "CPS" [66].

[7] The syntax of our continuation mechanism is taken from the Moby compiler's BOL IR [66], but our continuations are first-class, whereas BOL's continuations are a restricted form of one-shot continuations known as *escaping continuations*.

callcc, since it allows us to avoid the need to nest callcc's in many places. For example, we can create a fiber (unit continuation) from a function as follows:

```
fun fiber f =
  let
    cont k () = ( f () ; stop () )
  in
    k
  end
```

where stop (defined in Section 5.5) returns control to the scheduler.

After translation from AST, we apply a suite of standard optimizations, such as contraction, uncurrying, and inlining [2]. We then apply a series of refinement passes that first apply rewrite operations to the high-level operators [60] and then expand the operators with their definitions, whgich are loaded from external files (one can think of high-level operators as similar to hygienic macros). We then apply contraction to the resulting code before doing another refinement pass. Our plan is to use the high-level operations and rewriting to implement optimizations such as fusion [18]. We also plan to implement analysis and optimization passes to specialize the concurrency primitives [14, 67], which are also represented as high-level operations.

High-level operations play a key rôle in the implementation of the parallelism and concurrency primitives. The initial operations are introduced during the translation from AST to BOM. For example, the Manticore expression

```
spawn e
```

is translated into the following BOM code:

```
let
  fun f_thnk (z: unit) : unit = e′
  val tid : tid = @spawn (f_thnk)
in
  tid
end
```

where e′ is the translation of e to BOM. Here the compiler has defined a function f_thnk that evaluates the expression e′ and applies the high-level operation **@spawn** to create a new thread of control. The **@spawn** operation is defined as small BOM code fragment stored in an external file. When the compiler is ready to expand **@spawn**, it loads the definition of **@spawn**, which is

```
fun @spawn (f : unit -> unit) : tid =
  let
    cont fiber () = ( f () ; @stop () )
    val tid : tid = @new_tid ()
    val _ : unit = @enq_with_tid (tid, fiber)
  in
    tid
  end
```

The implementation of **@spawn** uses the **cont** binder to create a continuation for the new thread (fiber). Notice also that **@spawn** has other high-level operations in its definition (**@stop**, **@new_tid**, and **@enq_with_tid**). These will be expanded in subsequent passes over the BOM code. In a few cases, high-level operations expand to calls to the runtime kernel.

By defining these operations in external files, it easy to modify the implementation of scheduling mechanisms, message-passing protocols, *etc.* An alternative design would be to program these mechanisms in the surface language, but our surface language is lacking continuations (needed to represent threads) and mutable memory (needed to represent scheduling queues, *etc.*). On the other hand, by using the BOM IR for this implementation, we can take advantage of garbage collection, cheap memory allocation, and higher-order programming. These features would not be readily available if we coded the high-level operations in the runtime kernel. Furthermore, the optimizer can work on the combination of the application code and the implementations of high-level operations.

### 5.2.3 CPS Optimizations

The translation from direct style to CPS eliminates the special handling of continuations and makes control flow explicit. We use the Danvy-Filinski CPS transformation [20], but our implementation is simplified by the fact that we start from a normalized direct-style representation. We plan to implement a limited collection of optimizations on the CPS representation, since some transformation, such as inlining return continuations are much easier in CPS than direct style.

### 5.2.4 CFG Optimizations

The CPS representation is converted to CFG by closure conversion. Our current implementation uses a simple flat-closure conversion algorithm, but we plan to implement the Shao-Appel closure conversion algorithm [73] at a future date. We apply two transformations to the CFG: the first is specializing calling conventions for known functions. The second is adding explicit heap-limit checks to program. Because heap-limit tests are used as "safe-points" for preemption (see Section 5.4), it is necessary to guarantee that there is at least one check on every loop, even those that do not allocate. We use a feedback-vertex set algorithm [36] taken from the SML/NJ compiler to place the checks. Finally, we generate X86-64 (*a.k.a.* AMD64) assembly code from the CFG using the MLRISC framework [39, 38].

## 5.3 Implementing Implicitly-threaded Parallel Constructs

To sketch the important points of our implementation of the implicitly-threaded parallel constructs, we describe some of the simple AST-to-AST rewrites. For clarity, we present the transformed program fragments in SML syntax.

To implement the implicitly-threaded parallel features of Manticore, we define two polymorphic SML datatypes, `result` and the `rope`, and we introduce MultiLisp-style futures [40]. These datatypes, an abstract future type, and the signatures of the core future operations are as follows:

```
datatype 'a result = VAL of 'a | EXN of Exn
datatype 'a rope =
   Leaf of 'a vector
 | Cat of 'a rope * 'a rope

type 'a future
val future : (unit -> 'a) -> 'a future
val poll : 'a future -> 'a result option
val touch : 'a future -> 'a
val cancel : 'a future -> unit
```

### 5.3.1 Utility Data Structures

**Future** A future value is a handle to a (lightweight) computation that may be executed in parallel to the main thread of control. There are three elimination operations on futures. The `poll` operation returns `NONE` if the computation is still being evaluated and `SOME` value or exception—in the form of a `result`—if the computation has evaluated to a result value or a raised exception. The `touch` operation demands the result of the future computation, blocking until the computation has completed. The behavior of `touch` is equivalent to the following (inefficient) implementation:

```
fun touch fut =
  case poll fut of
     NONE => touch fut
   | SOME (VAL v) => v
   | SOME (EXN exn) => raise exn
```

Finally, the `cancel` operation terminates a future computation, releasing any computational resources being consumed if the computation is still being evaluated and discarding any result value or raised exception if the computation has been fully evaluated. It is an error to `poll` or `touch` a future value after it has been canceled. It is not an error to `cancel` a future value multiple times or to cancel a future that has been touched. Many of the translations below depend on this property of the `cancel` operation.

Futures and future operations are implemented by means of a flexible scheduling framework [61, 31], described in Section 5.5. This framework is capable of handling the disparate demands of the various heterogeneous parallelism mechanisms and capable of supporting a diverse mix of scheduling policies. Futures are just one of the mechanisms implemented with this framework.

Recall that the implicitly-threaded parallel constructs may be nested arbitrarily. Thus, a single future created to evaluate (one subcomputation of) an implicitly-threaded parallel construct may, in turn, create multiple futures to evaluate nested constructs. Futures, then, may be organized into a tree, encoding parent-child relationships. If a future is canceled, then all of its child futures must also be canceled.

The implementation of futures makes use of standard synchronization primitives provided by the scheduling framework; *e.g.*, I-variables [4] are used to represent future results. Cancellation is handled by special "cancellable" data structures that record parent-child relationships and the cancellation status of each future; a nestable scheduler action is used to poll for cancellation when a future computation is preempted.

Finally, note that a number of the program transformations below use futures in a stylized manner. For example, some futures are guaranteed to be touched at most once, a fact the compiler can exploit. Similarly, some futures will all be explicitly canceled together. The scheduling framework makes it easy to support these special cases with decreased overhead.

**Ropes** Parallel arrays are implemented via ropes [10]. Ropes, originally proposed as an alternative to strings, are immutable balanced binary trees with vectors of data at their leaves. Read from left to right, the data elements at the leaves of a rope constitute the data of the parallel array it represents. Ropes admit fast concatenation and, unlike contiguous arrays, may be efficiently allocated in memory even when very large. One disadvantage of ropes is that random access to individual data elements requires logarithmic time. Nonetheless, we do not expect this to present a problem for many programs, as random access to elements of a parallel array will in many cases not be needed. However, a Manticore programmer should be aware of this representation.

As they are physically dispersed in memory, ropes are well-suited to being built in parallel, with different processing elements simultaneously working on different parts of the whole. Furthermore, ropes embody a natural tree-shaped parallel decomposition of common parallel array operations like maps and reductions. Note the rope datatype given above is an oversimplification of our implementation for the purposes of presentation. In our prototype system, rope nodes also store their depth and data length. These values assist in balancing ropes and make length and depth queries constant-time operations.

### 5.3.2 Parallel Tuples

A future is created for each element of the parallel tuple, except the first. Since the first element of the tuple will be the first element demanded, and the main thread will block until the first element is available, there is no need to incur the overhead of a future for the computation of the first element. To manage computational resources properly, when an exception is raised during the evaluation of a parallel tuple, it is necessary to install an exception handler that will cancel any running futures before propagating the exception. Thus, a parallel tuple **(|** e1, e2, ..., en **|)** is rewritten to the following:

```
let
  val futn = future (fn _ => en)
  ...
  val fut2 = future (fn _ => e2)
in
  (e1, touch fut2, ..., touch futn)
  handle exn => (cancel fut2 ; ... ; cancel futn ; raise exn)
end
```

Note that if the expression ei (and touch futi raises an exception), then the cancellation of fut2, . . ., futn will have no effect. Since we expect exceptions to be rare, we choose this transformation rather than one that installs custom exception handlers for e1 and each touch futi. Also note that the expressions are registered as futures in the order en, . . ., e2, which works well with a work stealing policy for futures [12, 40].

### 5.3.3 Parallel Arrays

Operations on parallel arrays are translated to operations on ropes. A simple parallel-array comprehension, such as that in the implementation of mapP, is translated to a function that maps a computation over a rope:

```
fun mapR f r =
  case r of
     Leaf vec => Leaf (mapV f vec)
   | Cat (r1, r2) =>
       let
         val fut2 = future (fn _ => mapR f r2)
         val m1 = (mapR f r1)
                  handle exn => (cancel fut2 ; raise exn)
         val m2 = touch fut2
       in
         Cat (m1, m2)
       end
```

Note that if mapR f r1 raises an exception, then the future evaluating the map of the right half of the rope is canceled, and the exception from the map of the left half is propagated. If mapR fr2 raises an exception, then it will be propagated by the touch fut2. By this mechanism, we preserve the sequential semantics of exceptions.

The maximum length of the vector at each leaf is controlled by a compile-time option; its default value is 256. Altering the maximum leaf length, currently a compile-time option, can affect the execution time of a given program. If the leaves store very little data, then ropes become very deep. Small leaves correspond to the execution of many futures. This leads to good load balancing when applying the mapped function to an individual element is relatively expensive. By contrast, large leaves correspond to the execution of few futures; this is advantageous when applying the mapped function to an individual element is relatively cheap. Allowing the user to vary the maximum leaf size on a per-compilation basis gives some rough control over these tradeoffs. A more flexible system would allow the user to specify a maximum leaf size on a per-array basis, although many decisions remain about how to provide such a facility.

As another example, the parallel-array reduction operation reduceP is implemented by a reduceR operation:

```
fun reduceR f b r =
  case r of
     Leaf vec => reduceV f b vec
   | Cat (r1, r2) =>
       let
         val fut2 = future (fn _ => reduce f b r2)
         val v1 = (reduceR f b r1)
                  handle exn => (cancel fut2 ; raise exn)
         val v2 = touch fut2
       in
         f (v1, v2)
       end
```

Again, compensation code ensures that the sequential semantics of exceptions is preserved.

Note that in both examples, the compensation code is subject to elimination by program analysis when the function is known never to raise an exception.

## 5.4 Runtime Kernel

Our runtime kernel is implemented in C with a small amount of assembly-code glue between the runtime and generated code.

**Vprocs** Each vproc is hosted by its own POSIX thread (pthread). Typically, there is one vproc for each physical processor core. We use the Linux and MacOS X processor affinity extensions to bind pthreads to distinct processors. For each vproc, we allocate a local memory region of size $2^k$ bytes aligned on a $2^k$-byte boundary (currently, $k = 20$ and fixed at compile-time). The runtime representation of a vproc is stored in the base of this memory region and the remaining space is used as the vproc-local heap. Thus, the host_vproc primitive may be implemented by clearing the low $k$ bits of the allocation pointer.

One important design principle that we follow is minimizing the sharing of mutable state between vprocs. By doing so, we minimize the amount of expensive synchronization needed to safely read and write mutable state by multiple vprocs. This improves the parallel performance of a program, because each vproc spends the majority of time executing without needing to coordinate with other vprocs. Secondary effects, such as avoiding cache updates and evictions on the physical processors due to memory writes, further improve the parallel performance and are enabled by minimizing the sharing of mutable state.

We distinguish between three types of vproc state: fiber-local state, which is local to each individual computation; vproc-local state, which is only accessed by code running on the vproc; and global state, which is accessed by other vprocs. The thread-atomic state, such as machine registers, is protected by limiting context switches to "safe-points" (*i.e.*, heap-limit checks).

**Fiber-local storage** Our system supports dynamically-bound per-fiber storage. This storage is used to provide access to scheduler data structures, thread IDs, and other per-fiber information. Each vproc has a pointer to keep track of the current fiber-local storage (FLS). The following operations are used to allocate FLS and to get and set the host vproc's current FLS pointer.

```
type fls
val newFls : unit -> fls
val setFls : fls -> unit
val getFls : unit -> fls
```

To support flexibility, we provide an association-list-style mechanism for accessing the attributes in FLS:

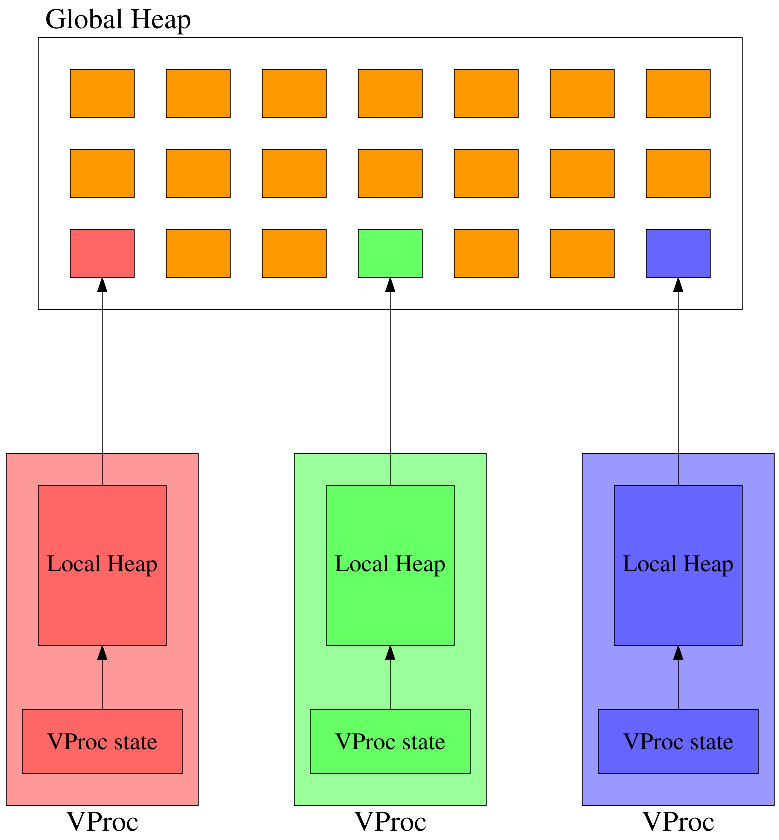

Figure 5: Organization of the Manticore heap

```
        val getFromFls : fls * 'a tag -> 'a option ref
```

In keeping with our use of SML syntax, we use phantom types to type these operations. The tag values that are used as lookup keys are globally defined.

**Garbage Collector** For the initial implementation, we have adopted a simple, yet effective, garbage collection strategy. While there has been significant work on parallel and concurrent garbage collection algorithms, such garbage collectors demand more compiler support than our initial implementation can presently provide. Our garbage collector might best be described as a "locally-concurrent/globally-sequential" collector. It is based on the approach of Doligez, Leroy, and Gonthier [24, 23]. The heap is organized into a fixed-size local heap for each vproc and a shared global heap (Figure 5). The global heap is simply a collection of chunks of memory, each of which may contain many heap objects. Each vproc has a dedicated chunk of memory in the global heap. Heap objects consist of one or more pointer-sized words with a pointer-sized header. The heap has the invariant that there are no pointers into the local heap from either the global heap or another vproc's local heap.

Each local heap is managed using Appel's "semi-generational" collector [1]. Each local heap is divided into a nursery area and an old-data area. New objects are allocated in the nursery; when the nursery area is exhausted, a minor collection copies live data in the nursery area to the old-data area, leaving an empty (but slightly smaller) nursery area (Figure 6). When the resulting nursery is too small, a major collection promotes the live data in the old-data to the global heap, leaving a (larger) empty nursery and the most recently minor-collected young-data (Figure 7). (Keeping the young-data in the local heap is consistent with the generational hypothesis: the most recently allocated data is likely to become garbage soonest.) Objects are promoted to the vproc's dedicated chunk of memory in the global heap; dedicating a chunk of memory to each vproc ensures that the major collection of a vproc local heap can proceed without locking the global heap. Note that the heap invariant allows a vproc to collect its local heap completely independently from the other vprocs. As noted above, avoiding expensive synchronizations with other vprocs improves the parallel performance of a program. Synchronization is only required when a vproc's dedicated chunk of memory is exhausted, and a fresh chunk of memory needs to be allocated and added to the global heap.

Thus, each vproc performs a certain amount of local garbage collection, during which time other vprocs may be executing either mutator code or performing their own local garbage collection. When a global garbage collection is necessary, all vprocs synchronize on a barrier, after which the initiating vproc (alone) performs the global garbage collection. While the

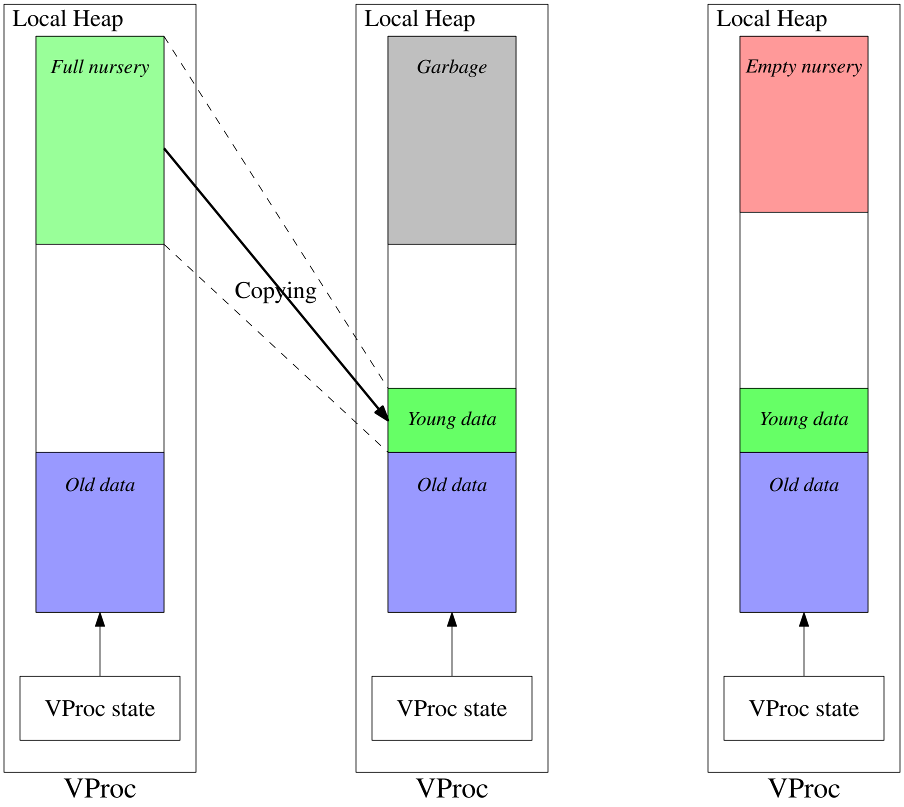

Figure 6: Minor garbage collection

sequential global garbage collection fails to take advantage of the parallelism available in a multiprocessor system, we expect that the concurrent local garbage collections will handle a significant portion of garbage-collection work. Furthermore, the implementation of the garbage collector remains quite simple. Eventually, we plan to experiment with more sophisticated garbage collection algorithms for the global heap.

Enforcing the heap invariant requires *promoting* objects that might become visible to other vprocs to the global heap. For example, if a thread is going to send a message, then the message must be promoted first, since the receiver may be running on a remote vproc. When objects are promoted, the reachable local data is copied to the global heap, but forward pointers are left so that sharing is reestablished when future promotions or collections encounter pointers to the promoted objects.

**Preemption** We implement preemption by synchronizing preempt signals with garbage-collection tests as is done in SML/NJ [69]. We dedicate a pthread to periodically send `SIGUSR2` signals to the vproc pthreads. (Sinals can be masked locally on a vproc, which we do to avoid preemption while holding a spin lock.) Each vproc has a signal handler that sets the heap-limit register to zero, which causes the next heap-limit check to fail and the garbage collector to be invoked. At that point, the computation is in a safe state, which we capture as a continuation value that is passed to the current scheduler on the vproc. The one downside to this approach is that the compiler must add heap-limit checks to non-allocating loops. An alternative that avoids this extra overhead is to use the atomic-heap transactions of Shivers *et al.* [77], but that technique requires substantial compiler support.

**Startup and Shutdown** One challenging part of the implementation is initialization and clean termination. When a program initially starts running, it is single threaded and running on a single vproc. Before executing the user code, it enqueues a thread on every vproc that installs the default scheduler. After initialization, each of the vprocs, except the initial one, will be idle and waiting for a fiber to be added to their secondary queues. If at any point, all of the vprocs go idle, then the system shuts down.

**Scheduling Queues** Each vproc maintains a scheduling queue. This queue is actually split into a locally accessible queue that has no synchronization overhead and a globally accessible queue that is protected by a mutex lock. As part of handling preemption, the vproc moves any threads in the global queue into the vproc's local queue. We provide the following operations for operating on a vproc's scheduling queue:

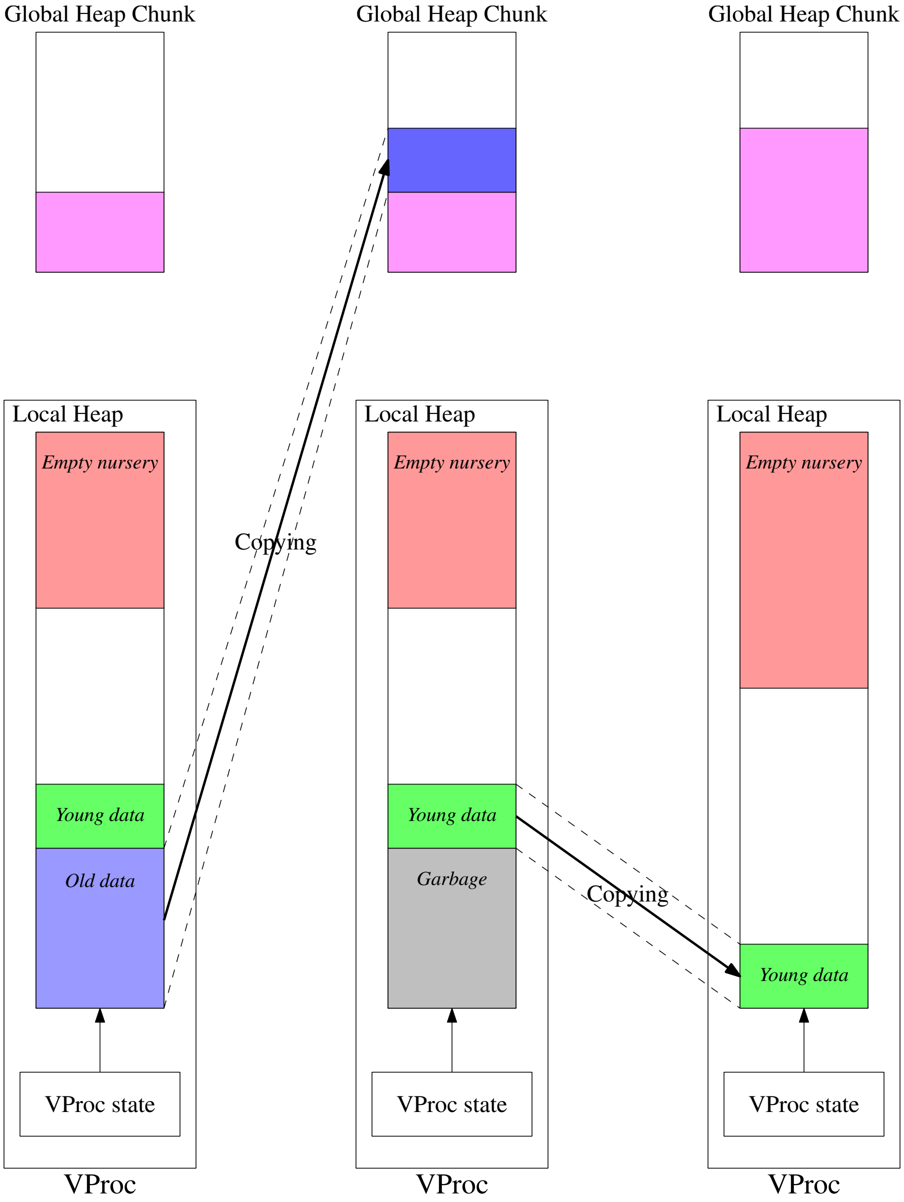

Figure 7: Major garbage collection

```
val enq : fiber -> unit
val deq : unit -> fiber
val enqOnVP : vproc * fiber -> unit
```

The first two operations work on the host vproc's local queue and only require that signals be masked locally. If both the locally accessible and the gobally accessible queue are empty, then the deq operation causes the vproc to go idle until there is work for it. The enqOnVP operation enqueues a fiber on a remote vproc. Note that there is no way to dequeue a thread from a remote vproc. This asymmetric design can complicate the implementation of load-balancing, but makes local queue operations (the common case) significantly faster. Also note that, because the enqueued fiber is made visible to another vproc, enqOnVP requires the fiber to have been promoted.

## 5.5 An Infrastructure for Nested Schedulers

Supporting parallelism at multiple levels poses interesting technical challenges for an implementation. While it is clear that a runtime system should minimally support thread migration and some form of load balancing, we choose to provide a richer infrastructure that serves as a uniform substrate on which an implementor can build a wide range of parallelism mechanisms with complex scheduling policies. Our infrastructure can support both explicit parallel threads that run on a single processor and groups of implicit parallel threads that are distributed across multiple processors with specialized scheduling disciplines. For example, workcrews [78], work stealing [9], lazy task creation [54], engines [43] and nested engines [25] are abstractions providing different scheduling policies, each of which is expressible using the constructs provided by our infrastructure [61]. Finally, our infrastructure provides the flexibility to experiment with new parallel language mechanisms that may require new scheduling disciplines. As our present work focuses on the design and implementation of a flexible scheduling substrate, we explicitly leave questions of if and how end-programmers can write their own schedulers and how end-programmers express scheduling policies to future work.

We note that the various scheduling policies often need to cooperate in an application to satisfy its high-level semantics (*e.g.*, real-time deadlines in multimedia applications). Furthermore, to best utilize the underlying hardware, these various scheduling policies should be implemented in a distributed manner, whereby a conceptually global scheduler is executed as multiple concrete schedulers on multiple processing units. Programming and composing such policies can be difficult or even impossible under a rigid scheduling regime. A rich notion of scheduler, however, permits both the nesting of schedulers and different schedulers in the same program, thus improving modularity, and protecting the policies of nested schedulers. Such nesting is precisely what is required to efficiently support heterogeneous parallelism.

In this section, we sketch the design of an infrastructure for the modular implementation of nested schedulers that support a variety of scheduling policies (a more detailed description can be found in other publications [61, 31]). Our approach is similar in philosophy to the microkernel architecture for operating systems; we provide a minimum collection of compiler and runtime-kernel mechanisms to support nested scheduling and then build the scheduling code on top of that infrastructure.

We present the infrastructure here using SML for notational convenience, but note that schedulers are actually implemented as high-level operations in the BOM IR of the compiler. Specifically, user programs do not have direct access to the scheduling operations or to the underlying continuation operations. Rather, the compiler takes care of importing and inlining schedulers into the compiled program. Indeed, a number of the "primitive" scheduling operations are implemented as high-level operations, and, hence, are themselves inlined into compiled programs. This exposes much of the low-level operational behavior of schedulers to the optimizers, while preserving a high-level interface for writing schedulers.

### 5.5.1 Scheduling Operations

At the heart of our infrastructure are scheduler actions. A scheduler action is a function that takes a signal and performs the appropriate scheduling activity in response to that signal.

```
datatype signal = STOP | PREEMPT of fiber
type action = signal -> void
```

At a minimum, we need two signals: STOP that signals the termination of the current fiber and PREEMPT that is used to asynchronously preempt the current fiber. When the runtime kernel preempts a fiber it reifies the fiber's state as a continuation that is carried by the preempt signal. The signal type could be extended to model other forms of asynchronous events, such as asynchronous exceptions [50]. As a scheduler action should never return, its result type (void) is one that has no values.

Each vproc has its own stack of scheduler actions. The top of a vproc's stack is called the *current* scheduler action. When a vproc receives a signal, it handles it by popping the current action from the stack, setting the signal mask, and throwing the signal to the current action. The operation is illustrated in Figure 8; here we use red in the mask box to denote when signals are masked.

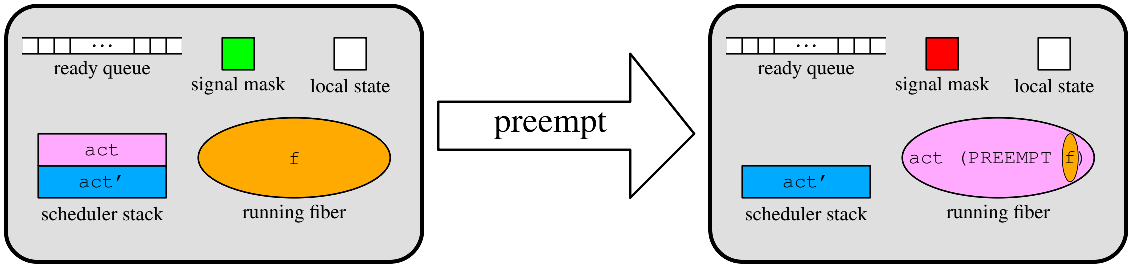

Figure 8: The effect of preemption on a VProc

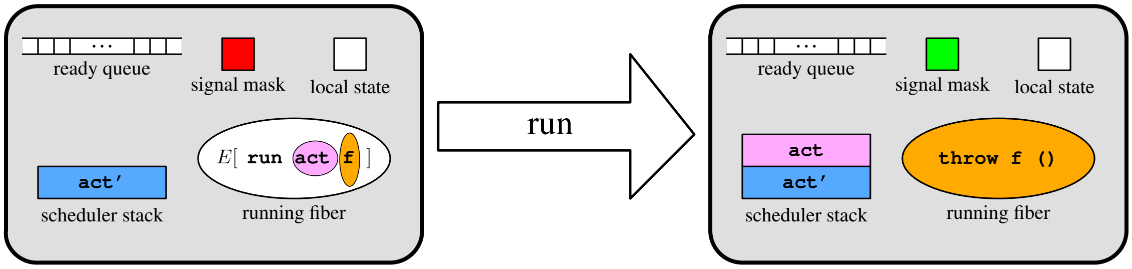

Figure 9: The effect of `run` on a VProc

There are two operations in the infrastructure that scheduling code can use to affect a vproc's scheduler stack directly.

```
val run : action * fiber -> 'a
val forward : signal -> 'a
```

Both operations should never return, so their result types may be instantiated to arbitrary types.

The operation `run (act, f)` pushes `act` onto the host vproc's action stack, clears the vproc's signal mask, and throws to the fiber `f` (see Figure 9). The `run` operation requires that signals be masked, since it manipulates the vproc's action stack. The other operation is the operation `forward sgn`, which sets the signal mask and forwards the signal `sgn` to the current action (see Figure 10). The `forward` operation is used both in scheduling code to propagate signals up the stack of actions and in user code to signal termination, which means that signals may, or may not, be masked when it is executed. For example, a fiber exit function can be defined as follows:

```
fun stop () = forward STOP
```

Another example is the implementation of a yield operation that causes the current fiber to yield control of the processor, by forwarding its own continuation:

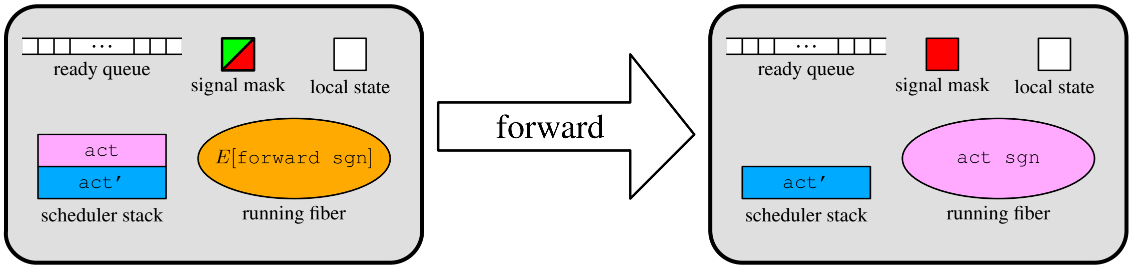

Figure 10: The effect of `forward` on a VProc

```
fun preempt k = forward (PREEMPT k)
fun yield () =
  let
    cont k x = x
  in
    preempt k
  end
```

The last part of our framework is the operations used to map a parallel computation across multiple vprocs. We have already introduced the enqOnVP operation to enqueue a fiber on a specific vproc. Using this operation we can implement an explicit migration function that moves the calling computation to a specific vproc:

```
fun migrateTo vp =
  let
    val fls = getFls ()
    cont k x = ( setFls fls ; x )
  in
    enqOnVP (vp, k) ;
    stop ()
  end
```

Note that the migrated computation takes its fiber-local storage.

We also provide a mechanism for assigning vprocs to computations. A parallel computation is a group of fibers running on separate vprocs; the scheduling framework uses FLS to distinguish between different parallel computations. Computations request additional vprocs by using the provision operation; this operation either returns SOME vp, where vp is a vproc that is not already assigned to the computation, or returns NONE, indicating that no additional vprocs are available for the computation. To balance workload evenly between threads, the runtime system never assigns a vproc to a given group twice and attempts to balance the number of groups assigned to each vproc. When a computation is finished with a vproc, it uses the release operation to alert the runtime that it is done with the vproc.

### 5.5.2 Scheduler Utility Functions

To avoid the danger of asynchronous preemption while scheduling code is running, the forward operation masks preemption and the run operation unmasks preemption on the host vproc. We also provide operations for explicitly masking and unmasking preemption on the host vproc.

```
val mask   : unit -> unit
val unmask : unit -> unit
```

On top of the basic scheduling framework described above, we implement a few operations that are common to many schedulers. For example, we have a function for passing preemptions up the action stack:

```
fun atomicYield () = ( yield () ; mask () )
```

Some of our scheduling code makes use of concurrent queues [52], which have the interface below.

```
type 'a queue
val emptyQ : unit -> 'a queue
val addQ : 'a queue * 'a -> unit
val remQ : 'a queue -> 'a option
```

### 5.5.3 Default Scheduler

With our primitives in hand, we can implement a simple round-robin scheduling policy for fibers in the scheduling queue. The scheduler action below embodies this policy. At runtime, each vproc executes its own instance of this scheduler.

```
cont dispatch () = run (roundRobin, deq ())
and roundRobin sgn =
  case sgn of
     STOP => dispatch ()
   | PREEMPT k =>
       let
         val fls = getFls ()
         cont k' () = ( setFls flst; throw k () )
       in
         enq k' ;
         dispatch ()
       end
```

We have also tested proportional-share policies using engines [43] and nested engines [25]. Nested engines, although somewhat complicated to implement with first-class continuations alone, have a compact implementation using operators in our runtime framework [61, 62]. Empirical studies on scheduling behavior of CML-like threads also exist [42, 71].

### 5.5.4 Load Balancing

We conclude this section by briefly describing how we extend our round-robin scheduler with load balancing. The design of this load-balancing scheduler is based on the following observation of typical CML programs. Often, these programs consist of a large number of threads (possibly thousands), but the bulk of these threads are *reactive* (*i.e.*, they spend most of their life waiting for messages). In such a system, only a handful of threads carry enough computational load to benefit from migration. Thus, we have optimized for fast context switching between local threads and have made thread migration bear the synchronization costs. Our implementation of vproc queues reflects this design, as we have no operations for dequeing from remote vprocs.

Our load-balancing scheduler performs thread migration by using the following protocol. When vprocs go idle, they periodically spawn *thief* threads on other vprocs. Thieves (executing on the remote vproc) can safely inspect the remote vproc queues, and if they observe sufficient load, then they can migrate threads back to the idle vproc. One complication of this protocol, of course, is that if we are not careful, thieves could end up migrating other thieves! We avoid this situation, and similar situations that arise with our schedulers for implicit parallelism, by pinning a thread to a processor. We use fiber-local storage to mark a thread as pinned.

Although we have not performed an empirical evaluation of this scheduler, we have benefited by testing it on synthetic workloads. Since it makes extensive use of our scheduling operations, the scheduler has proved to be a good stress test for the compiler and runtime code. In the future, we plan to incorporate ideas from the growing body of research on OS-level scheduling for multicore processors into our thread-scheduling approach [26].

### 5.5.5 Gang Scheduling

In this section, we describe a simple scheduler for data parallelism that is based on futures [40] with gang scheduling. Note that using the implementation sketched in Section 5.3, we can use a restricted form of future that we call *one-touch futures*. By fixing the number of fibers touching a future to one, we can utilize a lighter-weight synchronization protocol than in the general case.

There are only two opportunities for evaluating a future. If there is insufficient parallelism, then the fiber that originally created the future touches (and evaluates) it using the `touch1` function. Otherwise, an instance of the gang scheduler steals the future using the `future1StealAndEval` function, which evaluates it in parallel.

```
val futureStealAndEval : 'a future -> unit
```

The primary difference between touching and stealing a future is that the former blocks if the future is being evaluated (after having been stolen), while the latter is a nop if the future is being evaluated (after having been touched). We elide further discussion of the synchronization protocol, as it is mostly straightforward and irrelevant to our scheduling code.

The entry point for our gang scheduler is the `future` operation. As can be seen below, it initializes the future cell and adds it to the scheduler's ready queue.

```
fun future thnk =
  let
    val fut = newFutureCell thnk
  in
    addQ (getReadyQueue (),
          fiber (fn () => futureStealAndEval fut))
  end
```

Notice that what we have put on the ready queue is actually a *suspended fiber* for evaluating the future. As we build more advanced scheduling features, the advantage of using this uniform representation will become clear. The operation for getting the ready queue is shown the code below; it returns the queue by querying FLS.

```
fun getReadyQueue () =
  case !(getFromFls(getFls(), #futRdyQ)) of
     NONE => futuresScheduler()
    | SOME gq => gq
```

If the queue is not available, then the scheduler needs to be initialized.

The following shows our initialization and scheduling code, which follows the gang policy.

```
fun futuresScheduler () =
  let
    val gq = initGangQueue()
    val fls = getFls()
    cont gsAction sgn =
      let
        cont dispatch () =
          case remQ(gq) of
             NONE => ( atomicYield() ;
                       throw dispatch () )
           | SOME k => ( setFls(fls) ;
                         run(gsAction, k) )
      in
        case sgn of
           STOP => throw dispatch ()
         | PREEMPT k => ( addQ(gq, k) ;
                          atomicYield() ;
                          throw dispatch () )
       end
  in
    schedulerStartup(gsAction) ;
    gq
  end
```

Workers obtain work from the single, shared queue `gq`. The scheduler action `gsAction` embodies a parallel instance of our gang scheduler. This action loops indefinitely, stealing futures from the ready queue. When none are available, the scheduler yields to its parent scheduler, thus giving other schedulers a chance to run. Similarly, when the evaluation of a stolen future is preempted, the gang scheduler returns the in-progress future evaluation to the ready queue (to be stolen by another instance of the scheduler) and yields to its parent scheduler.

Related work on supporting data-parallel arrays in the Data Parallel Haskell, however, has noted deficiencies of the gang policy. Specifically, on NUMA machines, poor data locality can become an issue, degrading performance for memory-intensive computations [18].

The only remaining piece of this implementation is the code responsible for initializing the scheduler on some collection of vprocs. This operation is important for heterogeneous programs, as deciding how many vprocs should be available to a scheduler affects other running schedulers. Many alternatives are available, including a wide variety of job scheduling techniques [27]. We have prototyped a job scheduler for our framework, but have yet to evaluate its performance. We expect that finding efficient and reliable job scheduling policies will be a significant focus of our future work. In our current implementation, however, this function just spawns the given scheduler on all vprocs in the system.

### 5.6 Conclusion

This section has sketched some of the highlights of the implementation of the Manticore system. Much of our current and future research activities are focused on implementation techniques for high-level parallel languages. The characteristics of a multicore architecture demand that an implementation be cognizant of issues related to the preservation of sequential semantics in parallel constructs, the granularity of parallel computations, the scheduling of concurrent and parallel threads, the affinity of data, *etc.* We are exploring a collection of implementation techniques that combine static program analyses, compiler transformations, and dynamic runtime policies. By using a hybrid approach that combines static and dynamic techniques, the implementation can get the best of both approaches: *i.e.*, using static information to reduce runtime overhead and using dynamic techniques to improve the quality of information used to make runtime decisions.

## 6 Conclusion

These notes have described Manticore, a language (and implementation) for heterogeneous parallelism, suporting parallelism at multiple levels. By combining explicit concurrency and implicit parallelism into a common linguistic and execution framework, we hope to better support applications that might run on commodity processors of the near future, such as multimedia processing, computer games, small-scale simulations, *etc.* As a statically-typed, strict, functional language, Manticore (like other functional languages) emphasizes a value-oriented and mutation-free programming model, which avoids entanglements between separate threads of exection.

We have made steady progress on a prototype implementation of the Manticore language. A significant portion of the implementation is completed, and we have been able to run examples of moderate size (*e.g.* a parallel ray tracer). Some of the more novel features (*e.g.*, the `pcase` expression form) have only preliminary implementations, without significant optimization.

## References

[1] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software: Practice and Experience*, 19(2):171–183, 1989.

[2] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, Cambridge, England, 1992.

[3] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent programming in ERLANG (2nd ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996.

[4] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, October 1989.

[5] Reid Barton, Dan Adkins, Harald Prokop, Matteo Frigo, Chris Joerg, Martin Renard, Don Dailey, and Charles Leiserson. Cilk Pousse, 1998. Viewed on March 20, 2008 at 2:45 PM.

[6] Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, March 1996.

[7] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, 1994.

[8] Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of NESL. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 213–225, New York, NY, May 1996. ACM.

[9] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

[10] Hans-J. Boehm, Russ Atkinson, and Michael Plass. Ropes: an alternative to strings. *Software: Practice and Experience*, 25(12):1315–1330, December 1995.

[11] Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. Representing control in the presence of one-shot continuations. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '96)*, pages 99–107, New York, NY, May 1996. ACM.

[12] F. Warren Burton and M. Ronan Sleep. Executing functional programs on a virtual tree of processors. In *Functional Programming Languages and Computer Architecture (FPCA '81)*, pages 187–194, New York, NY, October 1981. ACM.

[13] Martin Carlisle, Laurie J. Hendren, Anne Rogers, and John Reppy. Supporting SPMD execution for dynamic data structures. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, March 1995.

[14] Richard Carlsson, Konstantinos Sagonas, and Jesper Wilhelmsson. Message analysis for concurrent programs using message passing. *ACM Transactions on Programming Languages and Systems*, 28(4):715–746, July 2006.

[15] Manuel M. T. Chakravarty and Gabriele Keller. More types for nested data parallel programming. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 94–105, New York, NY, September 2000. ACM.

[16] Manuel M. T. Chakravarty, Gabriele Keller, Roman Leshchinskiy, and Wolf Pfannenstiel. Nepal – Nested Data Parallelism in Haskell. In *Proceedings of the 7th International Euro-Par Conference on Parallel Computing*, volume 2150 of *Lecture Notes in Computer Science*, pages 524–534, New York, NY, August 2001. Springer-Verlag.

[17] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Gabriele Keller. Partial Vectorisation of Haskell Programs. In *Proceedings of the ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming*, New York, NY, January 2008. ACM.

[18] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data Parallel Haskell: A status report. In *Proceedings of the ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming*, pages 10–18, New York, NY, January 2007. ACM.

[19] Don Dailey and Charles E. Leiserson. Using Cilk to write multiprocessor chess programs. *The Journal of the International Computer Chess Association*, 2002.

[20] Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

[21] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation*, pages 137–150, December 2004.

[22] Erik D. Demaine. Higher-order concurrency in Java. In *Proceedings of the Parallel Programming and Java Conference (WoTUG20)*, pages 34–47, April 1997. Available from `http://theory.csail.mit.edu/~edemaine/papers/WoTUG20/`.

[23] Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Conference Record of the 21st Annual ACM Symposium on Principles of Programming Languages (POPL '94)*, pages 70–83, New York, NY, January 1994. ACM.

[24] Damien Doligez and Xavier Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ml. In *Conference Record of the 20th Annual ACM Symposium on Principles of Programming Languages (POPL '93)*, pages 113–123, New York, NY, January 1993. ACM.

[25] R. Kent Dybvig and Robert Hieb. Engines from continuations. *Computer Languages*, 14(2):109–123, 1989.

[26] Alexandra Fedorova. *Operating System Scheduling for Chip Multithreaded Processors*. PhD thesis, Department of Computer Science, Harvard University, Boston, MA, 2007.

[27] Dror G. Feitelson. Job scheduling in multiprogrammed parallel systems. Research Report RC 19790 (87657), IBM, October 1994. Second revision, August 1997.

[28] Kathleen Fisher and John Reppy. Compiler support for lightweight concurrency. Technical memorandum, Bell Labs, March 2002. Available from `http://moby.cs.uchicago.edu/`.

[29] Matthew Flatt and Robert B. Findler. Kill-safe synchronization abstractions. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '04)*, pages 47–58, June 2004.

[30] Matthew Fluet, Nic Ford, Mike Rainey, John Reppy, Adam Shaw, and Yingqi Xiao. Status Report: The Manticore Project. In *Proceedings of the 2007 ACM SIGPLAN Workshop on ML*, pages 15–24, New York, NY, October 2007. ACM.

[31] Matthew Fluet, Mike Rainey, and John Reppy. A scheduling framework for general-purpose parallel languages. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, pages 241–252, New York, NY, September 2008. ACM.

[32] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Implicitly-threaded parallelism in Manticore. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, pages 119–130, New York, NY, September 2008. ACM.

[33] Matthew Fluet, Mike Rainey, John Reppy, Adam Shaw, and Yingqi Xiao. Manticore: A heterogeneous parallel language. In *Proceedings of the ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming*, pages 37–44, New York, NY, January 2007. ACM.

[34] Emden R. Gansner and John H. Reppy. *A Multi-threaded Higher-order User Interface Toolkit*, volume 1 of *Software Trends*, pages 61–80. John Wiley & Sons, 1993.

[35] Emden R. Gansner and John H. Reppy, editors. *The Standard ML Basis Library*. Cambridge University Press, Cambridge, England, 2004.

[36] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[37] Jean-Luc Gaudiot, Tom DeBoni, John Feo, Wim Bohm, Walid Najjar, and Patrick Miller. The Sisal model of functional programming and its implementation. In *Proceedings of the 2nd AIZU International Symposium on Parallel Algorithms / Architecture Synthesis (pAs '97)*, pages 112–123, Los Alamitos, CA, March 1997. IEEE Computer Society Press.

[38] Lal George and Andrew Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, 18(3):300–324, May 1996.

[39] Lal George, Florent Guillame, and John Reppy. A portable and optimizing back end for the SML/NJ compiler. In *Fifth International Conference on Compiler Construction*, pages 83–97, April 1994.

[40] Robert H. Halstead Jr. Implementation of multilisp: Lisp on a multiprocessor. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 9–17, New York, NY, August 1984. ACM.

[41] Kevin Hammond. *Parallel SML: a Functional Language and its Implementation in Dactl*. The MIT Press, Cambridge, MA, 1991.

[42] Carl Hauser, Cristian Jacobi, Marvin Theimer, Brent Welch, and Mark Weiser. Using threads in interactive systems: A case study. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, pages 94–105, December 1993.

[43] Christopher T. Haynes and Daniel P. Friedman. Engines build process abstractions. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 18–24, New York, NY, August 1984. ACM.

[44] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Continuations and coroutines. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 293–298, New York, NY, August 1984. ACM.

[45] H. Peter Hofstee. Cell broadband engine architecture from 20,000 feet. Available at `http://www-128.ibm.com/developerworks/power/library/pa-cbea.html`, August 2005.

[46] Mark P. Jones and Paul Hudak. Implicit and explicit parallel programming in Haskell. Technical Report Research Report YALEU/DCS/RR-982, Yale University, August 1993.

[47] Xavier Leroy. *The Objective Caml System (release 3.00)*, April 2000. Available from `http://caml.inria.fr`.

[48] Xavier Leroy and François Pessaux. Type-based analysis of uncaught exceptions. *ACM Transactions on Programming Languages and Systems*, 22(2):340–377, 2000.

[49] Roman Leshchinskiy, Manuel M. T. Chakravarty, and Gabriele Keller. Higher order flattening. In V. Alexandrov, D. van Albada, P. Sloot, and J. Dongarra, editors, *International Conference on Computational Science (ICCS '06)*, number 3992 in LNCS, pages 920–928, New York, NY, May 2006. Springer-Verlag.

[50] Simon Marlow, Simon Peyton Jones, Andrew Moran, and John Reppy. Asynchronous exceptions in Haskell. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '01)*, pages 274–285, June 2001.

[51] John McCarthy. A Basis for a Mathematical Theory of Computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, Amsterdam, 1963.

[52] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 267–275, New York, NY, May 1996. ACM.

[53] MLton. Concurrent ML. Available at `http://mlton.org/ConcurrentML`.

[54] Eric Mohr, David A. Kranz, and Robert H. Halstead Jr. Lazy task creation: a technique for increasing the granularity of parallel programs. In *Conference record of the 1990 ACM Conference on Lisp and Functional Programming*, pages 185–197, New York, NY, June 1990. ACM.

[55] Rishiyur S. Nikhil. *ID Language Reference Manual*. Laboratory for Computer Science, MIT, Cambridge, MA, July 1991.

[56] Rishiyur S. Nikhil and Arvind. *Implicit Parallel Programming in pH*. Morgan Kaufmann Publishers, San Francisco, CA, 2001.

[57] Kunle Olukotun and Lance Hammond. The future of microprocessors. *ACM Queue*, 3(7), September 2005. Available from `http://www.acmqueue.org`.

[58] Mikael Pettersson. A term pattern-match compiler inspired by finite automata theory. In *Fourth International Conference on Compiler Construction*, volume 641 of *Lecture Notes in Computer Science*, pages 258–270, New York, NY, October 1992. Springer-Verlag.

[59] Simon Peyton Jones, Alastair Reid, Fergus Henderson, Tony Hoare, and Simon Marlow. A semantics for imprecise exceptions. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '99)*, pages 25–36, New York, NY, May 1999. ACM.

[60] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: Rewriting as a practical optimization technique in GHC. In *Proceedings of the 2001 Haskell Workshop*, pages 203–233, September 2001.

[61] Mike Rainey. The Manticore runtime model. Master's thesis, University of Chicago, January 2007. Available from `http://manticore.cs.uchicago.edu`.

[62] Mike Rainey. Prototyping nested schedulers. In *Redex Workshop*, September 2007.

[63] Norman Ramsey. Concurrent programming in ML. Technical Report CS-TR-262-90, Department of Computer Science, Princeton University, April 1990.

[64] Norman Ramsey and Simon Peyton Jones. Featherweight concurrency in a portable assembly language. Unpublished paper available at `http://www.cminusminus.org/abstracts/c--con.html`, November 2000.

[65] William T. Reeves. Particle systems — a technique for modeling a class of fuzzy objects. *ACM Transactions on Graphics*, 2(2):91–108, 1983.

[66] John Reppy. Optimizing nested loops using local CPS conversion. *Higher-order and Symbolic Computation*, 15:161–180, 2002.

[67] John Reppy and Yingqi Xiao. Specialization of CML message-passing primitives. In *Conference Record of the 34th Annual ACM Symposium on Principles of Programming Languages (POPL '07)*, pages 315–326, New York, NY, January 2007. ACM.

[68] John H. Reppy. First-class synchronous operations in Standard ML. Technical Report TR 89-1068, Department of Computer Science, Cornell University, December 1989.

[69] John H. Reppy. Asynchronous signals in Standard ML. Technical Report TR 90-1144, Department of Computer Science, Cornell University, Ithaca, NY, August 1990.

[70] John H. Reppy. CML: A higher-order concurrent language. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '91)*, pages 293–305, New York, NY, June 1991. ACM.

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

[72] George Russell. Events in Haskell, and how to implement them. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, pages 157–168, September 2001.

[73] Zhong Shao and Andrew W. Appel. Efficient and safe-for-space closure conversion. *ACM Transactions on Programming Languages and Systems*, 22(1):129–161, 2000.

[74] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213, New York, NY, 1995. ACM.

[75] Adam Shaw. Data parallelism in Manticore. Master's thesis, University of Chicago, July 2007. Available from `http://manticore.cs.uchicago.edu`.

[76] Olin Shivers. Continuations and threads: Expressing machine concurrency directly in advanced languages. In *Proceedings of the Second ACM SIGPLAN Workshop on Continuations (CW '97)*, New York, NY, January 1997. ACM.

[77] Olin Shivers, James W. Clark, and Roland McGrath. Atomic heap transactions and fine-grain interrupts. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming*, pages 48–59, New York, NY, September 1999. ACM.

[78] Mark T. Vandevoorde and Eric S. Roberts. Workcrews: an abstraction for controlling parallelism. *International Journal of Parallel Programming*, 17(4):347–366, August 1988.

[79] Mitchell Wand. Continuation-based multiprocessing. In *Conference Record of the 1980 ACM Conference on Lisp and Functional Programming*, pages 19–28, New York, NY, August 1980. ACM.

[80] Kwangkeun Yi. An abstract interpretation for estimating uncaught exceptions in Standard ML programs. *Sci. Comput. Program.*, 31(1):147–173, 1998.

[81] Cliff Young, Lakshman YN, Tom Szymanski, John Reppy, Rob Pike, Girija Narlikar, Sape Mullender, and Eric Grosse. Protium, an infrastructure for partitioned applications. In *Proceedings of the Twelfth IEEE Workshop on Hot Topics in Operating Systems (HotOS-XII)*, pages 41–46, January 2001.

[82] Lukasz Ziarek, Philip Schatz, and Suresh Jagannathan. Stabilizers: a modular checkpointing abstraction for concurrent functional programs. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming*, pages 136–147, New York, NY, September 2006. ACM.