Manticore: A heterogeneous parallel language

http://manticore.cs.uchicago.edu

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

Mike Rainey
John Reppy
Adam Shaw
Yingqi Xiao
University of Chicago
{mrainey,jhr,adamshaw,xiaoyq}@cs.uchicago.edu

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, we combine CML-style explicit concurrency with NESL/Nepal-style data-parallelism. In this paper, we describe and motivate the design of the Manticore language. We also describe a flexible runtime model that supports multiple scheduling disciplines (*e.g.*, for both fine-grain and course-grain parallelism) in a uniform framework. Work on a prototype implementation is ongoing and we give a status report.

1. Introduction

We believe that existing general-purpose languages do not provide adequate support for parallel programming, while existing parallel languages, which are largely targeted at scientific applications, do not provide adequate support for general-purpose programming. This state of affairs must change. The laws of physics and the limitations of instruction-level parallelism are forcing microprocessor architects to develop new multicore processor designs, which means that parallel computing is coming to commodity hardware. We need new languages to maximize application performance on these new processors.

Our thesis is that parallel languages must provide mechanisms for multiple levels of parallelism, both because applications exhibit parallelism at multiple levels and because the hardware requires parallelism at multiple levels to maximize performance. For example, consider a networked flight simulator. Such an application might use data-parallel computations for particle systems [Ree83] to model natural phenomena such as rain, fog, and clouds. At the same time it might use parallel threads to preload terrain and compute level-of-detail refinements, and use SIMD parallelism in its physics simulations. The same application might also use explicit concurrency for user interface and network components. Programming such applications will be challenging without language support for parallelism at multiple levels.

This paper describes a new research project at the University of Chicago and TTI-C addressing the topic of *language design and implementation for multicore processors*. Our emphasis is on applications that might run on commodity processors, such as multimedia processing, computer games, small-scale simulations, *etc.* These applications can exhibit parallelism at multiple levels with different granularities, which means that a homogeneous approach will not take advantage of all of the hardware resources. A language that provides data parallelism but not explicit concurrency will be inconvienent for the development of the networking and GUI components of a program. On the other hand, a language that provides concurrency but not data parallelism will be ill-suited for components of a program that demand fine-grain SIMD parallelism, such as image processing and particle systems. Instead, we propose a *heterogeneous* parallel language, called Manticore, that combines support for parallel computation at different levels into a common linguistic and execution framework.

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 [Ham91, Rep91, JH93, NA01]. 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) [Rep99], which provides support for threads and synchronous message passing. Although CML was not designed with parallelism in mind (in fact, its original implementation is inherently not parallel), we believe that it will provide good support for coarse-grain parallelism. In this respect, Manticore is similar to Erlang, which has a mutation-free sequential core with message passing [AVWW96]. Erlang has parallel implementations [Hed98], but no support for fine-grain parallel computation. Manticore's support for fine-grain parallelism is influenced by previous work on nested data-parallel languages, such as NESL [BCH+94, Ble96, BG96] and Nepal [CK00, CKLP01, LCK06]. From these languages, Manticore adopts parallel arrays and parallel-array comprehensions.

In addition to language design, we are exploring a unified runtime substrate for Manticore that can handle the disparate demands

Presented at DAMP 2007 January 16, 2007; Nice France
ACM 2007.

of explicit concurrency and various fine-grain parallel programming mechanisms. This substrate will provide a foundation for rapidly experimenting with alternative parallelism mechanisms. We have also been developing techniques for implementing CML's message-passing operations in a multiprocessor setting. This work includes new protocols for the operations and a program analysis and optimization techniques to improve the performance of message-passing programs [RX07].

2. The Manticore language

As noted above, the Manticore language provides the programmer with both explicit mechanisms for concurrency and coarse-grain parallelism and implicit mechanisms for fine-grain parallelism.[1] For concurrency and coarse-grain parallelism, explicit mechanisms can be an effective technique, but for fine-grain parallelism they are burdensome to the programmer and may impose excessive overhead. In this section, we sketch the major features of our language design, demonstrate how different levels of parallelism may be used in a simple example, and discuss possible future directions for the design.

Briefly, the design of the Manticore language combines three distinct components: a sequential base language using functional programming features, drawn from a (large) subset of SML; explicit concurrent programming mechanisms using threads and synchronous message passing, drawn from CML [Rep91, Rep99]; and implicit parallel programming mechanisms using nested data-parallel constructs, drawn from NESL [Ble96] and Nepal [CKLP01].

In the sequential base language (and, by extension, the Manticore language as a whole), we include important features from SML, such as datatypes, polymorphism, type inference, and higher-order functions, but simplify the design by supporting only a simple module system and by removing a number of non-essential elements. Most importantly, we remove mutable reference and array types, so the concurrency mechanisms drawn from CML are the only stateful operations in Manticore.[2] As many researchers have observed, using a mutation-free computation language greatly simplifies the implementation and use of parallel features [Ham91, Rep91, JH93, NA01, DG04]. In essence, successful parallel languages rely on notions of *separation*; mutation-free functional programming gives data separation for free.

The explicit concurrent programming mechanisms presented in Manticore serve two purposes: they support concurrent programming, which is an important feature for systems programming [HJT+93], 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; 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.

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 [GR93], a distributed tuple-space implementation [Rep99], a system for implementing partitioned applications in a distributed setting [YYS+01], and a higher-level library for software checkpointing [ZSJ06]. CML-style primitives have also been added to a number of other languages, including HASKELL [Rus01], JAVA [Dem97], OCAML [Ler00], and SCHEME [FF04]. We believe that this history demonstrates the effectiveness of CML's approach to concurrency.

At the heart of the implicit parallel programming mechanisms presented in Manticore are *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 have parallel arrays of parallel arrays). Furthermore, Manticore (like Nepal, but unlike NESL) supports parallel arrays of arbitrary types including arrays of floats, functions, trees, *etc.* Based on the parallel array element type, the compiler will map parallel array operations onto the appropriate parallel hardware (*e.g.*, operations on parallel arrays of floats may be mapped onto SIMD instructions).

Parallel array values are constructed using a *parallel comprehension syntax*, which provides a concise description of a parallel computation. [3] A comprehension has the general form

```
[: e | x1 in e1, ..., xn in en where p :]
```

where e is the expression that computes the elements of the array, the e_i are array-valued expressions used as inputs to e, and p is an optional boolean-valued expression that filters the input. If the input arrays have different lengths, they are truncated to the length of the shortest input. For example, to double each positive integer in a given parallel array of integers `nums`, one would use the following parallel comprehension:

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

Another example is the definition of a *parallel map* combinator that maps a function across an array in parallel.

```
fun mapP f xs = [: f x | x in xs :]
```

The computation of elements in a comprehension can themselves be defined by comprehensions. We give an example of this pattern below in Figure 1 and other examples can be found in Blelloch's work [Ble96].

Comprehensions can be used to specify both SIMD parallelism that is mapped onto vector hardware (*i.e.*, Intel's SSE instructions) and SPMD parallelism where parallelism is mapped onto multiple cores.

An important feature of parallel arrays is that they have a sequential semantics, defined by mapping arrays to lists. The general comprehension form from above can be translated into the following sequential list code:

```
let fun f (x1::r1, ..., xn::rn, l) =
          f(r1, ..., rn, if p̂ then ê::l else l)
      | f (_, ..., _, l) = rev l
in f(ê1, ..., ên, []) end
```

where $\widehat{p}$, $\widehat{e}$, *etc.*, are the translated subexpressions.

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,

[1] We classify parallelism/concurrency mechanisms as either *explicit*, where the programmer manages thread creation, or *implicit*, where the compiler and runtime system manage thread creation.

[2] Note that we do *not* describe the sequential language as side-effect free, since it still supports exceptions.

[3] Some implicitly parallel languages, such as SISAL [GDF+97], Id [Nik91], and pH [NA01], allow independent computations to be executed in parallel with no programmer annotations, but most require programmer annotations to mark which computations are good candidates for parallel execution. For Manticore, we have chosen the latter approach because we believe that it will more easily coexist with the explicit parallel mechanisms.

```
structure GrayServer : sig
    type pixel = int * int * int (* RGB encoding *)
    type img   = [: [: pixel :] :]
    val convert : img -> img event
  end = struct
    type pixel = int * int * int
    type img   = [: [: pixel :] :]
    fun rgbToG ((r,g,b) : pixel) : pixel = let
          val m = (r + g + b) div 3
          in
            (m, m, m)
          end
    fun imgToGray img =
          [: [: rgbToG pix | pix in row :]
             | row in img :]
    fun convert img = let
          val replCh = channel()
          in
            spawn (send (replCh, imgToGray img));
            recvEvt replCh
          end
  end
```

Figure 1. An gray-scale converter

the compiler must verify that the individual sub-computations in a data-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.

To demonstrate how the different concurrent- and parallel-programming mechanisms can be used in combination, we present a simple, but illustrative, example. Consider the implementation of a service for converting color images into gray-scale images. This computation is inherently data parallel, but an application may also want to process multiple images in parallel (*e.g.*, if the service were web-based). The code in Figure 1 is a Manticore module that implements such a service. An image is represented as a parallel array of parallel arrays of pixels (the "`[: :]`" brackets double as a type constructor). The `imgToGray` function converts an image by using a nested comprehension. This conversion process is presented to clients as an asynchronous operation (the `convert` function). When the `convert` function is called on an image, a new thread is spawned to do the conversion and an event value is returned that the client can later synchronize on to acquire the image.

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 [MKH90, CHRR95, BL99]. 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 [Nik91]) and *software transactional memory* (STM) [ST95].

3. A runtime model for Manticore

Supporting parallelism at multiple levels poses interesting technical challenges for the implementation. We need a framework that 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. Furthermore, we want the flexibility to experiment with new parallel language mechanisms that may require new scheduling disciplines.

In this section, we describe an efficient and general runtime model for implementing scheduling disciplines (a more detailed description can be found in Rainey's Master's paper [Rai07]). This model, which uses first-class continuations [Wan80, Rey93] to represent suspended computations, provides a simple, but flexible, interface between the runtime system and the language implementation. The runtime-system infrastructure supports both per-processor and nested schedulers.[4] As we demonstrate below, it is capable of supporting both explicit and implicit threading models in a unified framework. We present the model using SML for notational convenience, but it is actually implemented as part of the compiler's internal representation. Specifically, user programs do not have direct access to the scheduling operations or to the underlying continuation operations.

3.1 Continuations

Continuations are a well-known language-level mechanism for expressing concurrency [Wan80, HFW84, Rep89, Shi97]. 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 [App92], but map poorly onto stack-based implementations.
2. *One-shot* continuations [BWD96] 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 [RP00, FR02]. 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 our compiler's IR to express concurrency operations. For our prototype implementation, we are using heap-allocated continuations *à la* SML/NJ [App92]. 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 [FR02].

[4] Regehr coined the term "*general, heterogeneous schedulers*" for similar scheduler hierarchies [Reg01].

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

3.2 Fibers, threads, and virtual processors

Our runtime model has three distinct notions of process abstraction. At the lowest level, a *fiber* is an unadorned thread of control, which is represented as a unit continuation.

```
type fiber = unit cont
```

The `fiber` operator takes a function value, and creates a fiber that, when run, calls the function before stopping.

```
val fiber : (unit -> unit) -> fiber
```

Note that this operator can be directly implemented with first-class continuations (but not with one-shot continuations).

A surface-language *thread* (*i.e.*, one created by `spawn`) is initially mapped to a fiber paired with a unique thread ID (`tid`).

```
type thread = tid * fiber
```

In addition to having an ID, threads are differentiated from fibers by the fact that they may create additional fibers to run data-parallel computations. Thus at run time, a thread consists of a `tid` and one or more fibers.

Lastly, a *virtual processor* (vproc) is an abstraction of a hardware processor resource. The runtime model represents a vproc with the `vproc` type. A vproc runs at most one fiber at a time, and furthermore is the only means of running fibers. The vproc for the currently running fiber is called the *host vproc*, and is obtained by the `hostVP` operator.

```
val hostVP : unit -> vproc
```

The runtime model provides a mechanism for assigning vprocs to threads. When applied to the desired number of processors, `provision` returns a list of vprocs that are available for a thread (which may be fewer than the number requested). The complementary `release` operator informs the runtime system that a thread is finished with some vprocs.

```
val provision : int -> vproc list
val release   : vproc list -> unit
```

To balance workload evenly between threads, the runtime system never assigns a vproc to a given thread twice. Additionally, the runtime system considers load and possibly even processor affinity when assigning vprocs.

3.3 Scheduling infrastructure

Our scheduling infrastructure is a low-level substrate for writing schedulers. It directly encodes all scheduling that occurs at run time, and does not rely on external or fixed schedulers. Our approach to scheduling is inspired by Shivers' proposal for exposing hardware concurrency using continuations [Shi97], but we have extended it to support nested schedulers and multiple processors. To support a variety of scheduling disciplines, the infrastructure provides mechanisms that divide a vproc's time among multiple fibers and mechanisms that divide and synchronize parallel computations among multiple vprocs. The former mechanisms are described in detail here.

A *scheduler action* is a function that implements context switching for a vproc. By defining different functions, we can implement different scheduling policies. Scheduler actions have the type

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

where the `signal` type represents the events that are handled by schedulers. Here we have two — fiber termination and preemption — but this type could be extended to model other forms of asynchronous events, such as asynchronous exceptions [MJMR01]. A scheduler action should never return, so its result type (`void`) is one that has no values.

Our model supports nesting of schedulers (*e.g.*, a data-parallel scheduler runs on top of a thread-level scheduler) by giving each vproc a stack of scheduler actions. The top of a vproc's stack is the scheduler action for the current scheduler on that vproc. When a vproc receives a signal, it handles it by popping the current scheduler action from the stack and applying it to the signal. Figure 2 gives a pictorial description of the operations on a vproc's action stack, which we describe below.

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

```
val run : action -> fiber -> void
val forward : signal -> void
```

The `run` primitive initiates the execution of a fiber. It takes a scheduler action that implements the scheduling policy for the fiber and the fiber itself, pushes the action on the scheduler-action stack, and then runs the fiber. The expression "`forward` *sig*" sends the signal to the host vproc, which means that topmost signal action is popped from the stack and applied to the *sig*. Our model uses this operation to implement the `stop` function for fiber termination.

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

Preemption is generated by a hardware event, such as as timer interrupt. When a vproc is preempted, it reifies the continuation of the running fiber *k*, and then executes "`preempt` *k*," where the `preempt` function is defined as

```
fun preempt k = forward (PREEMPT k)
```

The vproc then handles the signal as usual; it applies the the current scheduler action to the preemption signal. Using `preempt`, we can define a function that yields the vproc to the current scheduler.

```
fun yield () = callcc (fn k => preempt k)
```

In addition to the scheduler stack, each vproc also has a queue of ready threads. These queues are used to schedule threads, and are used as a mechanism to dispatch threads on multiple vprocs. There are three operations on these queues:

```
val enqueue : thread -> unit
val dequeue : unit -> thread
val enqueueOnProc : (vproc * thread) -> unit
```

The first two operations apply to the host vproc's queue. The second operator blocks a vproc on an empty queue. It can be unblocked when another vproc puts a thread on its queue. The third operator puts a thread on another vproc's queue, and is the only mechanism for parallel dispatch in the runtime model.

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
```

3.4 Scheduling language-level threads

Language-level thread scheduling is round-robin, and is implemented by the following scheduler action:

```
fun switch STOP = dispatch()
  | switch (PREEMPT k) = (
      enqueue (getTid(), k); dispatch())
and dispatch () = let
      val (tid, k) = dequeue ()
      in
        setTid tid; run switch k
      end
```

The `dispatch` function runs the next thread from the vproc's queue and is also used in the implementation of language-level

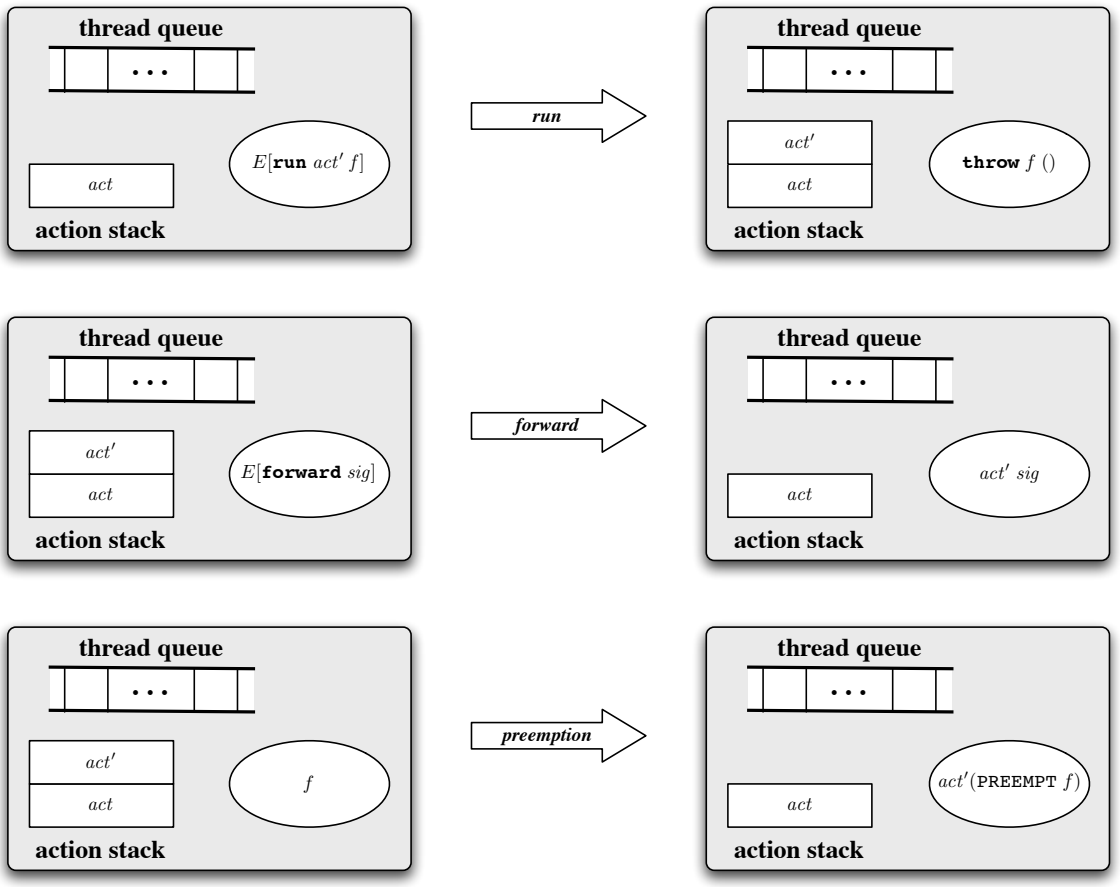

Figure 2. How `run`, `forward`, and preemption affect a vproc.

concurrency operations. Note that it invokes the thread using the `run` function with `switch` as the scheduler. This scheduler action is the first action on every vproc's stack. Our infrastructure can also support more complex time-sharing and priority-based policies, and can support migration policies, such as work stealing [CR95, BL99].

3.5 Scheduling data-parallel fibers

Data-parallel computations require multiple fibers running on multiple vprocs. There are a number of different ways to organize this computation, but we use a *workcrew* approach [VR88]. The compiler flattens the nested data parallelism into a flat operation [BG96, LCK06], and partitions it into a number of *jobs*. Each job should perform a significant chunk of the total work, employing SIMD parallelism when possible. The code in Figure 3 is a function used at runtime to schedule the jobs in parallel. It takes the number of processors, number of jobs, and a function for computing the ith job. The scheduler initializes itself by allocating a group of vprocs, and then applying each to the `init` function. This function takes a vproc, and enqueues on it a fiber that installs the scheduler action `dlpSwitch`.

Once it is initialized on a vproc, the `dlpSwitch` action acquires jobs from the work pool, and handles preemptions. The `STOP` signal is used to signal the completion of a job; if there are no more jobs available, then we relinquish the host vproc by releasing the vproc and then stopping. The last vproc to complete a job does not stop, but instead returns from the `forkN` function. When the scheduler receives a `PREEMPT` signal, it yields control to the parent scheduler. At some point in the future, the parent scheduler will resume the data-parallel scheduler. If, for example, the parent is the language-level thread scheduler, the thread scheduler will resume the data-parallel scheduler once it cycles through its ready queue. In this way, the host vproc can be multiplexed among both data-parallel and explicit-parallel computations.

3.6 Other scheduling disciplines

Our infrastructure is general enough to implement a wide variety of schedulers and we have sketched implementations of a number of different mechanisms [Rai07]. These include engines [HF84] and nested engines [DH89], which are an elegant mechanism that provide timed preemption for a collection of threads. Other examples include work stealing [MKH90, CHRR95, BL99] and wait-free cache-affinity work stealing [KD03]. We are also implementing schedulers that can adaptively provision vprocs via an extension to our signaling mechanism that is similar to *scheduler activations* [ABLL92].

In the long run, we believe that application-specific scheduling policies may be an important tool in maximizing parallel performance. Since implementing scheduling and load-balancing policies in general-purpose languages is error prone and tedious, we plan to explore domain-specific languages for programming schedulers. For example, the Bossa scheduler language ameliorates implemen-

```
fun forkN (nProcs, nJobs, job : int -> unit) =
      callcc (fn doneK => let
        val (cnt, done) = (ref 0, ref 0)
        fun dlpSwitch STOP = let
              val nextJob = fetchAndAdd(cnt, 1)
              in
                if (nextJob < nJobs) then
                  run dlpSwitch
                    (fiber (fn () => job nextJob))
                else if (fetchAndAdd(done,1) =
                                    nProcs-1)
                  then throw doneK ()
                  else (
                    release [hostVP ()];
                    stop())
              end
          | dlpSwitch (PREEMPT k) = (
                    yield ();
                    run dlpSwitch k )
        fun init vp = enqueueOnProc (vp,
              ( getTid(),
                fiber (fn () =>
                  run dlpSwitch (fiber stop))
              ))
        in
          List.app init (provision nProcs);
          stop()
        end)
```

Figure 3. Creating fibers for data-parallel computations

tation difficulties by using specialized abstractions for expressing policies and static checks of those policies [MLD05].

4. Multiprocessor CML

Concurrent ML is embedded in Standard ML of New Jersey. It is implemented uses the first-class continuations of SML/NJ and is inherently single threaded [Rep91, Rep99]. Thus, we are faced with developing a new multi-threaded implementation of CML's primitives suitable for modern multicore processors. The main challenge is the implementation of event synchronization, which involves a form of distributed agreement. In the general case, an event consists of a choice of channel communications and synchronization involves picking one of the enabled communications in the choice and executing it. What makes this problem difficult is that the other party involved in the communication may itself be involved in a choice, so we need a protocol that guarantees that both parties agree on the communication. Furthermore, a choice may involve multiple operations on the same channel, which makes deadlock avoidance a bit tricky.

The single-threaded implementation achieves this agreement by executing the following steps *atomically* [Rep99]:

- Poll the communication operations (*e.g.*, sends, recvs, *etc.*) in the the choice to see if they are enabled.
- If one or more operations are enabled, pick one and do it.
- Otherwise, enqueue continuations for each of the choices and dispatch another thread.

The single-threaded implementation relies on the global lock for correctness and, since there is only one processor, it does not hurt performance, but in a parallel implementation the global lock is a bottleneck.

An obvious first step is to give each channel its own lock, but to avoid deadlock when there are multiple operations on the same channel, we must either release the lock after polling the channel or use reentrant locks. We explored this approach, but found that the implementation was complicated. Instead, we have designed an optimistic protocol for implementing choice that has the following steps:

- First we poll channels for possible communications, which can be done in a *lock-free* way).
- If there are available communications, attempt to commit to one of them. This commit may fail because another thread has "stolen" the communication.
- If there are no available communications (or all attempts to commit failed), we block the thread on the channels. In this process, we may discover that a communication has become available, in which case we commit to it.

This protocol is optimized to the common case where a given channel is not shared between more than two threads, but it remains to be seen how well it works when there is contention for a shared channel.

Another aspect of our approach to implementing message passing is the development of a program analysis for detecting special patterns of channel usage. For example, our analysis can detect when a channel is only used by a single sender and single receiver in non-choice contexts. In such a case, the channel operations can be implemented using a single atomic compare-and-swap instruction, which is much faster than the general protocol. This program analysis and optimization technique, which we are implementing as part of the Manticore compiler, is discussed in full detail elsewhere [Xia05, RX07].

5. Status

We are currently working on an initial implementation of Manticore that will provide a testbed for future research in both language design and implementation techniques. Our initial implementation targets the 32 and 64-bit versions of the x86 architecture on Linux and we hope to have a public release ready by the Spring of 2007. This effort is proceeding along two tracks.

The first is a compiler and interpreter for the Manticore language. To speed the construction of this prototype, we have extended the HaMLet SML compiler [Ros] with syntax for our data-parallel and concurrency operations. We are using HaMLet as both a parser/typechecker for our compiler and as source-to-source translator that converts Manticore programs to CML programs. Although this translator does not support parallelism, it does allow us to gain experience with programming in Manticore. For our compiler, we are using the MLRISC framework for code generation and register allocation [GGR94, GA96].

We have also implemented a prototype of the runtime scheduler infrastructure described in Section 3. The implementation is written in C on Linux, with each vproc being represented by a POSIX thread. We are using this framework to gauge both the expressiveness of our model for writing schedulers and their performance. So far, we have implemented several data-parallel examples, but we have not yet made any performance measurements. We are also integrating the multiprocessor implementation of CML described above into the Manticore runtime system. Lastly, the runtime system model has been formalized as a parallel CEK machine [Rai07], which will provide a guide for both the compiler and runtime implementation efforts.

References

[ABLL92] Anderson, T. E., B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM TOCS*, **10**(1), February 1992, pp. 53–79.

[App92] Appel, A. W. *Compiling with Continuations*. Cambridge University Press, Cambridge, England, 1992.

[AVWW96] Armstrong, J., R. Virding, C. Wikström, and M. Williams. *Concurrent programming in ERLANG (2nd ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996.

[BCH+94] Blelloch, G. E., S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *JPDC*, **21**(1), 1994, p. 4=14.

[BG96] Blelloch, G. E. and J. Greiner. A provable time and space efficient implementation of NESL. In *ICFP '96*, New York, NY, May 1996. ACM, pp. 213–225.

[BL99] Blumofe, R. D. and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *JACM*, **46**(5), 1999, pp. 720–748.

[Ble96] Blelloch, G. E. Programming parallel algorithms. *CACM*, **39**(3), March 1996, pp. 85–97.

[BWD96] Bruggeman, C., O. Waddell, and R. K. Dybvig. Representing control in the presence of one-shot continuations. In *PLDI '96*, New York, NY, May 1996. ACM, pp. 99–107.

[CHRR95] Carlisle, M., L. J. Hendren, A. Rogers, and J. Reppy. Supporting SPMD execution for dynamic data structures. *ACM TOPLAS*, **17**(2), March 1995, pp. 233–263.

[CK00] Chakravarty, M. M. T. and G. Keller. More types for nested data parallel programming. In *ICFP '00*, New York, NY, September 2000. ACM, pp. 94–105.

[CKLP01] Chakravarty, M. M. T., G. Keller, R. Leshchinskiy, and W. Pfannenstiel. Nepal – Nested Data Parallelism in Haskell. In *Euro-Par '01*, vol. 2150 of *LNCS*, New York, NY, August 2001. Springer-Verlag, pp. 524–534.

[CR95] Carlisle, M. C. and A. Rogers. Software caching and computation migration in Olden. In *PPoPP '95*, New York, NY, July 1995. ACM, pp. 29–38.

[Dem97] Demaine, E. D. Higher-order concurrency in Java. In *WoTUG20*, April 1997, pp. 34–47. Available from `http://theory.csail.mit.edu/~edemaine/papers/WoTUG20/`.

[DG04] Dean, J. and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI '04*, December 2004, pp. 137–150.

[DH89] Dybvig, R. K. and R. Hieb. Engines from continuations. *Comput. Lang.*, **14**(2), 1989, pp. 109–123.

[FF04] Flatt, M. and R. B. Findler. Kill-safe synchronization abstractions. In *PLDI '04*, June 2004. (to appear).

[FR02] Fisher, K. and J. Reppy. Compiler support for lightweight concurrency. *Technical memorandum*, Bell Labs, March 2002. Available from `http://moby.cs.uchicago.edu/`.

[GA96] George, L. and A. Appel. Iterated register coalescing. *ACM TOPLAS*, **18**(3), May 1996, pp. 300–324.

[GDF+97] Gaudiot, J.-L., T. DeBoni, J. Feo, W. Bohm, W. Najjar, and P. Miller. The Sisal model of functional programming and its implementation. In *pAs '97*, Los Alamitos, CA, March 1997. IEEE Computer Society Press, pp. 112–123.

[GGR94] George, L., F. Guillame, and J. Reppy. A portable and optimizing back end for the SML/NJ compiler. In *CC'94*, April 1994, pp. 83–97.

[GR93] Gansner, E. R. and J. H. Reppy. *A Multi-threaded Higher-order User Interface Toolkit*, vol. 1 of *Software Trends*, pp. 61–80. John Wiley & Sons, 1993.

[Ham91] Hammond, K. *Parallel SML: a Functional Language and its Implementation in Dactl*. The MIT Press, Cambridge, MA, 1991.

[Hed98] Hedqvist, P. A parallel and multithreaded ERLANG implementation. Master's dissertation, Computer Science Department, Uppsala University, Uppsala, Sweden, June 1998.

[HF84] Haynes, C. T. and D. P. Friedman. Engines build process abstractions. In *LFP'84*, New York, NY, August 1984. ACM, pp. 18–24.

[HFW84] Haynes, C. T., D. P. Friedman, and M. Wand. Continuations and coroutines. In *LFP'84*, New York, NY, August 1984. ACM, pp. 293–298.

[HJT+93] Hauser, C., C. Jacobi, M. Theimer, B. Welch, and M. Weiser. Using threads in interactive systems: A case study. In *SOSP '93*, December 1993, pp. 94–105.

[JH93] Jones, M. P. and P. Hudak. Implicit and explicit parallel programming in Haskell. *Technical Report Research Report YALEU/DCS/RR-982*, Yale University, August 1993.

[KD03] Kurt Debattista, Kevin Vella, J. C. Wait-free cache-affinity thread scheduling. *IEEE Proceedings Software*, **150**(2), 2003, pp. 137–146.

[LCK06] Leshchinskiy, R., M. M. T. Chakravarty, and G. Keller. Higher order flattening. In V. Alexandrov, D. van Albada, P. Sloot, and J. Dongarra (eds.), *ICCS '06*, number 3992 in LNCS, New York, NY, May 2006. Springer-Verlag, pp. 920–928.

[Ler00] Leroy, X. *The Objective Caml System (release 3.00)*, April 2000. Available from `http://caml.inria.fr`.

[MJMR01] Marlow, S., S. P. Jones, A. Moran, and J. Reppy. Asynchronous exceptions in Haskell. In *PLDI '01*, June 2001, pp. 274–285.

[MKH90] Mohr, E., D. A. Kranz, and R. H. Halstead Jr. Lazy task creation: a technique for increasing the granularity of parallel programs. In *LFP'90*, New York, NY, June 1990. ACM, pp. 185–197.

[MLD05] Muller, G., J. L. Lawall, and H. Duchesne. A framework for simplifying the development of kernel schedulers: Design and performance evaluation. In *HASE '05*, October 2005, pp. 56–65.

[NA01] Nikhil, R. S. and Arvind. *Implicit Parallel Programming in pH*. Morgan Kaufmann Publishers, San Francisco, CA, 2001.

[Nik91] Nikhil, R. S. *ID Language Reference Manual*. Laboratory for Computer Science, MIT, Cambridge, MA, July 1991.

[Rai07] Rainey, M. The Manticore runtime model. Master's dissertation, University of Chicago, January 2007. Available from `http://manticore.cs.uchicago.edu`.

[Ree83] Reeves, W. T. Particle systems — a technique for modeling a class of fuzzy objects. *ACM TOG*, **2**(2), 1983, pp. 91–108.

[Reg01] Regehr, J. *Using Hierarchical Scheduling to Support Soft Real-Time Applications on General-Purpose Operating Systems*. Ph.D. dissertation, University of Virginia, 2001.

[Rep89] Reppy, J. H. First-class synchronous operations in Standard ML. *Technical Report TR 89-1068*, Dept. of CS, Cornell University, December 1989.

[Rep91] Reppy, J. H. CML: A higher-order concurrent language. In *PLDI '91*, June 1991, pp. 293–305.

[Rep99] Reppy, J. H. *Concurrent Programming in ML*. Cambridge University Press, Cambridge, England, 1999.

[Rey93] Reynolds, J. C. The discoveries of continuations. *LASC*, **6**(3-4), 1993, pp. 233–248.

[Ros] Rossberg, A. HaMLet. Available from `http://www.ps.uni-sb.de/hamlet`.

[RP00] Ramsey, N. and S. Peyton Jones. Featherweight concurrency in a portable assembly language. Unpublished paper available at `http://www.cminusminus.org/abstracts/c--con.html`, November 2000.

[Rus01] Russell, G. Events in Haskell, and how to implement them. In *ICFP '01*, September 2001, pp. 157–168.

[RX07] Reppy, J. and Y. Xiao. Specialization of CML message-passing primitives. In *POPL '07*, January 2007.

[Shi97] Shivers, O. Continuations and threads: Expressing machine concurrency directly in advanced languages. In *CW '97*, January 1997.

[ST95] Shavit, N. and D. Touitou. Software transactional memory. In *PODC '95*, New York, NY, 1995. ACM, pp. 204–213.

[VR88] Vandevoorde, M. T. and E. S. Roberts. Workcrews: an abstraction for controlling parallelism. *IJPP*, **17**(4), August 1988, pp. 347–366.

[Wan80] Wand, M. Continuation-based multiprocessing. In *LISP'80*, August 1980, pp. 19–28.

[Xia05] Xiao, Y. Toward optimization of Concurrent ML. Master's dissertation, University of Chicago, December 2005.

[YYS+01] Young, C., L. YN, T. Szymanski, J. Reppy, R. Pike, G. Narlikar, S. Mullender, and E. Grosse. Protium, an infrastructure for partitioned applications. In *HotOS-X*, January 2001, pp. 41–46.

[ZSJ06] Ziarek, L., P. Schatz, and S. Jagannathan. Stabilizers: a modular checkpointing abstraction for concurrent functional programs. In *ICFP '06*, New York, NY, September 2006. ACM, pp. 136–147.