Status Report: The Manticore Project

http://manticore.cs.uchicago.edu

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

Nic Ford
Mike Rainey
John Reppy
Adam Shaw
Yingqi Xiao
University of Chicago
{nford,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 fine-grain, implicitly threaded, parallel constructs. We have been working on an implementation of Manticore for the past six months; this paper gives an overview of our design and a report on the status of the implementation effort.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language classifications — Concurrent, distributed, and parallel languages; Applicative (functional) languages; D.3.3 [*Programming Languages*]: Language Constructs and Features — Concurrent programming structures; D.3.4 [*Programming Languages*]: Processors — Compilers; Run-time environments

General Terms Languages

Keywords ML, concurrent languages, data-parallel languages

1. Introduction

The laws of physics and the limitations of instruction-level parallelism have forced microprocessor architects to develop new multicore processor designs. As a result, parallel computing is becoming widely available on commodity hardware. Ideal applications for this hardware, such as multimedia processing, computer games, and small-scale simulations, can exhibit parallelism at multiple levels with different granularities, which means that a homogeneous language design will not take full advantage of the 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. On the other hand, 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. 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 is a status report on the Manticore project at the University of Chicago and TTI-C, which addresses the topic of *language design and implementation for multicore processors*. 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, 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 [OH05]. It is quite likely that these processors will also have special-purpose vector units, similar to those of the IBM Cell processor [Hof05]. 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

ML'07, October 5, 2007, Freiburg, Germany.

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 [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, coarse-grain parallelism, and fine-grain parallelism.

Manticore's concurrency mechanisms are based on Concurrent ML (CML) [Rep99], 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 [BCH+94, Ble96, BG96] and Nepal [CK00, CKLP01, LCK06].

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 collection 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, lightweight 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 manner.

In the remainder of this paper, we describe the current status of the Manticore project. We sketch the design of the Manticore language in Section 2. In Section 3, we give an overview of our implementation effort, which we then examine in more detail: the Manticore compiler (Section 4), the infrastructure for nested schedulers (Section 5), and the runtime system (Section 6).

2. The Manticore language

Parallelism mechanisms can be roughly grouped into three catagories:

- *implicit parallelism*, where the compiler and runtime system are responsible for partitioning the computation into parallel threads. Examples of this approach include Id [Nik91], pH [NA01], and Sisal [GDF+97].
- *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 [Ble96] and Nepal [CKLP01].
- *explicit threading*, where the programmer explicitly creates parallel threads. Examples of this approach include CML [Rep99] and Erlang [AVWW96].

These different design points represent a trade-off 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. In Manticore, we have both implicitly threaded parallel mechanisms and explicitly threaded mechanisms. We see the former as supporting fine-grained parallel computation, whereas the latter are for coarse-grain parallel tasks and explicit concurrent programming. These parallelism mechanisms are built on top of a sequential functional language. In the sequel, we discuss each of these in turn, starting with the sequential base.

2.1 Sequential programming

Manticore is based on a subset of Standard ML (SML). The main difference is that Manticore does not have mutable data (*i.e.*, reference cells and arrays). 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 our implicitly threaded primitives, which we discuss below, 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 [Ham91, Rep91, JH93, NA01, DG04]. In essence, successful parallel languages rely on notions of *separation*; mutation-free functional programming gives data separation for free.

While SML does have its warts, our research focus is on parallel computation, so we have resisted tinkering with the sequential language. The Manticore Basis, however, differs significantly from the SML Basis [GR04]. For example, we have a fixed set of numeric types — `int`, `long`, `integer`, `float`, and `double` — instead of SML's families of numeric modules. In our current implementation, we have further restricted the language to a subset of Core SML (*i.e.*, SML without modules).

2.2 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 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.

Parallel arrays Support for parallel computation on arrays and matrices is common in parallel languages. In Manticore, we support such computations using the nested parallel-array mechanism inspired by NESL [Ble96] and Nepal [CKLP01]. A parallel-array expression has the form

`[|`e_1`, ..., `e_n`|]`

which constructs an array of n elements. "`[| |]`" brackets tell the compiler that the e_i may be evaluated in parallel.

Parallel-array values may also be constructed using a *parallel comprehension syntax*, which provides a concise description of a parallel loop. A comprehension has the general form

[1] Shaw's Master's paper [Sha07] provides a rigorous account of the semantics of these mechanisms.

```
[| 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.[2] 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 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 |]
```

This expression can easily be evaluated using vector instructions. 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. For example, the main loop of a ray tracer can be written as

```
[| [| trace(i, j) | j in [| 0 to wid-1 |] |]
   | i in [| 0 to ht-1 |] |]
```

The syntax

```
[| e1 to en |]
```

defines an array initialized to a sequence of integer values. Using this notation, we can define a parallel tabulate function:

```
fun tabulateP (n, f) =
      [| f(i) | i in [| 0 to n-1 |] |]
```

The sequential semantics of parallel arrays is defined by mapping them to lists (see [FRR+07] or [Sha07] for details). The main subtlety in the parallel implementation is that if an exception is raised when computing the ith element, then we must wait until all preceding elements have been computed before propagating the exception.

Parallel tuples Just as parallel arrays provide a hint to the compiler that evaluating the elements of an array in parallel can be productive, a parallel tuple hints that the elements of a tuple can be evaluated in parallel. The basic form is

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

which describes a fork-join evaluation of the e_i in parallel. The result is a normal tuple value. For example, here is a parallel summing of the leaves of a binary tree:

```
datatype tree = LF of int | ND of tree * tree
fun treeAdd (LF n) = n
  | treeAdd (ND(t1, t2)) =
      (op +)(|treeAdd t1, treeAdd t2|)
```

While this example could have been coded using parallel arrays, it is slightly more convenient to use parallel tuples. Parallel tuples also have the advantage that the individual subcomputations can have different types.

The sequential semantics of parallel tuples is trivial: they are just evaluated as tuples. The implication for the sequential semantics is similar to that for parallel arrays: if an exception is raised in some element of the tuple, we must wait until all elements to the left have been computed prior to raising the exception.

Parallel bindings Parallel arrays and tuples provide a fork-join pattern of computation, but in some cases we want more scheduling flexibility. In particular, we may choose to execute some computations speculatively. Manticore has a parallel binding form

```
pval pattern = expression
```

that starts the evaluation of the expression as a parallel thread. The sequential semantics of a parallel binding are similar to lazy evaluation, where the binding is only evaluated when one of its variables is needed. In the parallel implementation, we use eager evaluation for parallel bindings, but computations are cancelled when control reaches a point where their result is guaranteed not to be needed. For example, the following program computes the product of the leaves of a tree in parallel.

```
fun treeMul (LF n) = n
  | treeMul (ND(t1, t2)) = let
      pval b = treeMul t2
      val a = treeMul t1
      in
        if (a = 0) then 0 else a*b
      end
```

Note that if the result of the left product is zero, then we do not need the result of the right product and that subcomputation (and its descendants) may be cancelled. The other 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.

Parallel choice In some situations, we may want to find one of possibly many different solutions (*e.g.*, the 8-queens problem). In these cases, we can use Manticore's parallel choice operator to run two computations in parallel, returning the result of the first to complete. For example, the following function returns a leaf from a tree; if the tree is unbalanced, then it can be faster than picking a specific path:

```
fun treePick (LF n) = n
  | treePick (ND(t1, t2)) =
      treePick t1 |?| treePick t2
```

Unlike the other implicitly-threaded mechanisms, parallel choice is nondeterministic. We can still give a sequential semantics, but it requires including a source of non-determinism, such as McCarthy's **amb**, in the sequential language. One of the design questions that we have struggled with is whether to make parallel choice an implicit or explicitly threaded operation. In the end, we decided that it was most useful as a fine-grained operation.

2.3 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 [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.

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.

3. Implementation Overview

We are nearing the completion of an initial implementation of Manticore that will provide a testbed for future research in both language design and implementation techniques. Our initial imple-

[2] 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 [CLP+07] has *Cartesian-product* semantics where the iteration is over the product of the inputs.

mentation targets the 64-bit version of the x86 architecture (*a.k.a.* X86-64 or AMD64) and we hope to have a public release ready by end of the summer of 2007.

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 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, scheduling framework, and runtime kernel in the following sections.

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[3] 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 the BOM 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].

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

3.2 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* 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, and can be obtained obtained by the `host_vproc` operator.

4. 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 λ-calculus.
4. CPS — a continuation-passing-style λ-calculus.
5. CFG — a first-order control-flow-graph representation.
6. MLTree — the expression tree representation used by the ML-RISC code generation framework [GGR94].

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 in this paper.

4.1 The front end

In previous work [FRR$^+$07], we described our intention to adapt the HaMLet SML compiler [Ros], extended with syntax for our data-parallel and concurrency operations, to serve as a parser and typechecker for our compiler. While we had hoped that this approach would speed the construction of the initial Manticore compiler, we were stymied by the fact that the HaMLet compiler does not produce a typed representation of the program. Thus we decided that it was too difficult to extract an explicitly-typed abstract syntax tree representation from HaMLet for translation through the rest of the intermediate representations.

As an alternative, we extended the front-end for a subset of core-SML that is used as a compiler project at the University of Chicago. This front-end accepts most of Core ML extended with our concurrency primitives and parallelism annotations. The lexer and parser of the compiler are implemented using the ml-ulex and ml-antlr tools from SML/NJ. The resulting parse tree is typechecked using the imperative version of the Hindley-Milner algorithm. Because we currently do not have a module system, our Basis Library is necessary organized as a flat namespace and so it differs from that of SML [GR04], but with a simple prelude, we can compile any sequential Manticore program using an SML compiler.

4.2 AST optimizations

We use a series of transformations on the AST representation to simplify the program for later stages. These include a *flattening* transformation in the style of Keller's work [Kel99] that eliminates the data-parallel constructs in favor of lower-level operations. We have extended this transformation to account for exceptions. We also introduce futures with cancellation to implement the threading policies for the other implicitly-parallel constructs. For example,

the `treeMul` function from Section 2 is transformed to the following form that has explicit management of futures:

```
fun treeMul (LF n) = n
  | treeMul (ND(t1, t2)) = let
      val b = future (treeMul t2)
      val a = treeMul t1
      in
        if (a = 0)
          then (cancel b; 0)
          else a * touch b
      end
```

Shaw's Masters paper gives a rigorous account of these transformations [Sha07]. Lastly, we compile nested patterns to simpler decision trees using Pettersson's technique [Pet92].

4.3 BOM optimizations

The BOM representation is a normalized direct-style λ-calculus where every intermediate result is bound to a variable and all arguments are variables.[4] 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) [Wan80, Ram90, Rep99].
- 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 [PTH01].

We are currently working on the implementation of the BOM optimizer, so we will describe our design, although not all of it is implemented yet. After translation from AST, we apply a suite of standard optimizations, such as contraction, uncurrying, and inlining [App92]. We then apply a series of refinement passes that first apply rewrite operations to the high-level operators [PTH01] and then expand the operators with their definitions. 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 [CLP+07]. We also plan to implement analysis and optimization passes to specialize the concurrency primitives [CSW06, RX07], 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:

```
fun fiber ( / exh) = ê
let tid = @spawn (fiber)
    return tid
```

where $\hat{e}$ is the translation of e to BOM. Here the compiler has defined a function `fiber` that evaluates the expression e[5] 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

```
define inline @spawn (f) =
    cont fiber _ =
        cont handler (ex) = @dispatch(host_vproc)
        let _ = apply f( / handler)
          @dispatch(host_vproc)
    let vp = host_vproc
    let tid = @new-tid(vp)
    do @enqueue (vp, tid, fiber)
        return (tid)
```

The implementation of `@spawn` uses the **`cont`** binder to create a continuation for the new thread (`fiber`) and to define a catch-all exception handler for the thread. Notice also that `@spawn` has other high-level operations in its definition (`@dispatch`, `@new-tid`, and `@enqueue`). 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 BOM 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.

4.4 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 [DF92], 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.

4.5 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 [SA00] 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 6), 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 [GJ79] 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 [GGR94, GA96].

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

[4] 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" [Rep02].

[5] In BOM, function arguments are partitioned into normal arguments (to the left of the "/") and continuation arguments (the exception handler continuation `exh` on the right of the "/").

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 [VR88], work stealing [BL99], lazy task creation [MKH90], engines [HF84] and nested engines [DH89] are abstractions providing different scheduling policies, each of which is expressible using the constructs provided by our infrastructure [Rai07]. Finally, our infrastructure provides the flexibility to experiment with new parallel language mechanisms that may require new scheduling disciplines.

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 Rainey's Master's paper [Rai07]). 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.

5.1 The scheduler-action stack

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 [MJMR01]. 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 1; here we use dark grey in the mask box to denote when signals are masked.

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

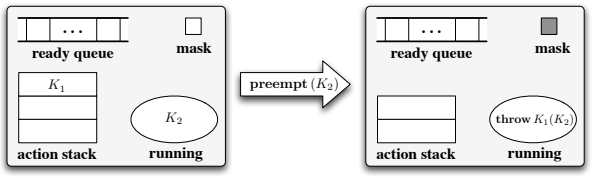

Figure 1. The effect of preemption on a vproc

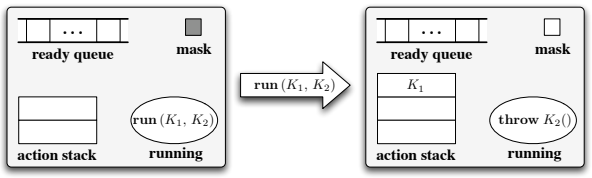

Figure 2. The effect of `run` K_1 K_2 on a vproc

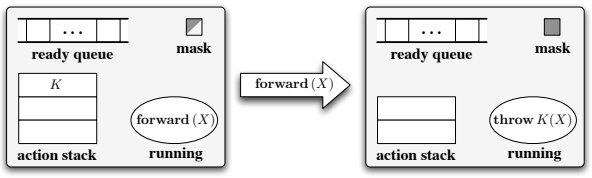

Figure 3. The effect of `forward` X on a vproc

```
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 (`K_1`,`K_2`)` pushes K_1 onto the host vproc's action stack, clears the vproc's signal mask, and throws to the continuation K_2 (see Figure 2). The run operation requires that signals be masked, since it manipulates the vproc's action stack. The other operation is the operation `forward` X, which sets the signal mask and forwards the signal X to the current action (see Figure 3). 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 thread 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 to its parent scheduler.

```
fun yield () = callcc (fn k => forward (PREEMPT k))
```

5.2 Scheduling queues

In addition to the scheduler stack, each vproc has a queue of ready fibers that is used for scheduling. The `enq` operation takes a suspended fiber and adds it to the scheduler queue, while the `deq` operation removes the next fiber from the queue.

```
val enq : fiber * tid -> unit
val deq : unit -> fiber * tid
```

If the queue is empty, then the `deq` operation causes the vproc to go idle until there is work for it.

5.3 Miscellaneous

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
```

Schedulers for speculative computations may require a mechanism to asynchronously signal fibers (executing speculative compuations) when they become unnecessary. Hence, we provide an operation to trigger a preemption on a vproc.

```
val signalVP : vproc -> unit
```

5.4 The default scheduler

We have already shown how the language-level **spawn** primitive is implemented using the BOM continuations. Language-level threads use a simple round-robin scheduler built on top of the per-vproc scheduling queues. This scheduler is installed at the bottom of each action stack and is called the *default scheduler*.

```
fun switch STOP = dispatch ()
  | switch (PREEMPT k) = (
      enq (k, get_tid());
      dispatch () )
and dispatch () = let
    val (fiber, tid) = deq ()
    in
      set_tid tid;
      run (switch, fiber)
    end
```

On a `STOP` signal, it runs the next fiber in the queue and on a `PREEMPT` signal, it enqueues the preempted fiber and then runs the next fiber. It uses the `get_tid` and `set_tid` operations to update the host vproc's current thread ID.[6]

5.5 Provisioning parallel computations

The last part of our infrastructure are the operations used to map a parallel computation across multiple vprocs. The operation `enqVP` enqueues a fiber on a named vproc:

```
val enqVP : vproc * fiber * tid -> unit
```

Using this operation we can implement an explicit migration function that moves the calling computation to a specific vproc.

```
fun migrateTo vp = callcc (fn k => (
      enqVP (vp, k, get_tid());
      stop ()))
```

We also provide a mechanism for assigning vprocs to computations. The basic parallel computation is a group of fibers running on separate vprocs; the scheduling infrastructure provides the mechanism of *group IDs* to distinguish between different parallel computations.

```
type gid
val newgid : unit -> gid
val provision : gid -> vproc option
val release : gid * vproc -> unit
```

When initiating a parallel computation, the `newgid` operation is used to create a unique group ID for the computation. This ID is passed to the `provision` operation to request additional vprocs. This operation either returns a vproc that is not already assigned to the computation or else the constant `NONE` to signal that no additional processing resources are available for the group. To balance workload evenly between threads, the runtime kernel 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 report to the runtime kernel that it is done with the vproc.

[6] To simplify the code, we have left the host vproc implicit, but in the actual implementation, it is an extra argument to the various operations like **deq**.

6. 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). We use the Linux processor affinity extension 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, $k = 20$). 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. The `hostVP` primitive is implemented by clearing the low k bits of the allocation pointer.

One important design principle that we follow is minimizing sharing of mutable state between vprocs. We distinguish between three types of vproc state: thread-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).

Scheduling queues Each vproc has two scheduling queues: a primary queue that is vproc local and a secondary queue that is globally accessible. In our framework we distinguish between enqueueing a fiber on the host vproc's scheduling queue (`enq`) and enqueueing it on a remote vproc's queue (`enqVP`). This distinction allows us to keep operations on the primary queue local, which means we can avoid expensive synchronization. The secondary queue is protected with traditional locking and is accessed periodically to move fibers down to the primary queue (or when the primary queue is empty).

Preemption We implement preemption by synchronizing preempt signals with garbage-collection tests as is done by Reppy [Rep90]. We dedicate a pthread to periodically send `SIGUSR2` signals to the vproc pthreads. 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 wrapped in the `PREEMPT` signal and passed to the topmost signal-action handler. 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.* [SCM99], 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.

Garbage collector For the initial implementation, we have adopted a simple, yet effective, garbage collection strategy. 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 [DL93, DG94]. The heap is organized into a fixed-size local heap for each vproc and a shared global heap. 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. We enforce 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 [App89]. 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. When the resulting nursery is too small, a major collection promotes the the live data in the old-data to the global heap, leaving an empty nursery and old-data areas. 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. 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.

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 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.

Since the execution of mutator code must preserve the garbage collector invariant, the transfer of data between vprocs requires that data to be promoted to the global heap. For example, when one vproc enqueues a fiber on another vproc's secondary queue, the fiber will become reachable from both vprocs, and hence must be allocated in the global heap. Note that such a promotion requires transitively promoting any object reachable from the fiber. Similar promotion is necessary to implement message-passing operations. When promoting objects, we install forward pointers in their headers, which preserves sharing if the objects are promoted multiple times or survive a major collection.

7. Related work

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]. Like Manticore, these languages have functional sequential cores and parallel arrays. To this mix, Manticore adds explicit parallelism, which neither NESL or Nepal support. The languages Id [Nik91], pH [NA01], and Sisal [GDF+97] represent another approach to implicit parallelism in a functional setting that does not require user annotations. The explicit concurrency mechanisms in Manticore are taken from Concurrent ML (CML) [Rep99]. While 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. Erlang is a similar language that has a mutation-free sequential core with message passing [AVWW96] that has parallel implementations [Hed98], but no support for fine-grain parallel computation.

Chapel [CCZ04] and Fortress [Ste06] are imperative languages with high-level parallel constructs. Chapel has `forall` loops and `forall` expressions, and Fortress has aggregates and comprehensions, both of which are similar to our parallel comprehensions. Arrays in both languages are operated on in parallel by default, similar to our parallel arrays. The main differences between these languages and ours seem to be their imperative nature and their focus on supporting object-oriented programming. Because both languages have shared mutable state, both have a notion of atomicity, which is a topic our design does not currently address.

Programming parallel hardware effectively is difficult, but there have been a few recent successes. Google's MapReduce programming model [DG04] has been a great success in processing large datasets in parallel. Sawzall is a system for analysis of large datasets distributed over disks or even machines [PDGQ05]. Brook for GPUs [BFH+04] is a C-like language which allows the programmer to use a GPU as a stream co-processor.

8. Status and future work

We began our implementation effort in December of 2006. We started with the backend of the compiler and the runtime system; since then, we have been working our way forward. We have implemented a parser, pretty printer, and type checker for each IR. Having an external representation for each IR has has allowed us to compile and execute test programs throughout development and to exercise our runtime system.

As noted previously, we have not implemented any significant optimizations yet, so we are not in a position to benchmark performance. Rather, we have focused on moving quickly towards an end-to-end implementation. We have recently begun testing the compilation of front-end programs through to executable code. The front-end recognizes all of the language constructs and primitives, but we have yet to implement the full complement of AST transformations to support them.

In addition to the scheduling framework and runtime system described here, we previously developed a prototype that supports writing schedulers and applications in C [Rai07]. We used assembly routines to implement one-shot continuations (essentially like `setjmp`/`longjmp`). The purpose of this first implementation was primarily as a "proof of concept." It allowed us to test our framework as a platform for writing schedulers. Most of the schedulers we have experimented with have been prototyped in this implementation. It also demonstrates that our design carries over to implementations based on traditional stacks, although our experience has been that using heap-allocated continuations greatly simplifies the implementation.

We hope to have a public release ready by end of the summer of 2007. In the immediate future, we plan to explore additional analyses and optimizations to support the parallel constructs described in Section 2. Looking further ahead, we plan to experiment with language support for shared state and compiler support for heterogeneous processors (e.g., special-purpose vector units).

References

[App89] Appel, A. W. Simple generational garbage collection and fast allocation. *SP&E*, **19**(2), 1989, pp. 171–183.

[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, pp. 4–14.

[BFH+04] Buck, I., T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. *SIGGRAPH '04*, **23**(3), July 2004, pp. 777–786.

[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.

[CCZ04] Callahan, D., B. L. Chamberlain, and H. P. Zima. The Cascade High Productivity Language. In *HIPS '04*, Los Alamitos, CA, April 2004. IEEE Computer Society Press, pp. 52–60.

[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.

[CLP+07] Chakravarty, M. M. T., R. Leschchinski, S. Peyton Jones, G. Keller, and S. Marlow. Data Parallel Haskell: A status report. In *DAMP '07*, New York, NY, January 2007. ACM, pp. 10–18.

[CSW06] Carlsson, R., K. Sagonas, and J. Wilhelmsson. Message analysis for concurrent programs using message passing. *ACM TOPLAS*, **28**(4), July 2006, pp. 715–746.

[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/`.

[DF92] Danvy, O. and A. Filinski. Representing control: A study of the CPS transformation. *MSCS*, **2**(4), 1992, pp. 361–391.

[DG94] Doligez, D. and G. Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *POPL '94*, New York, NY, January 1994. ACM, pp. 70–83.

[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. *Comp. Lang.*, **14**(2), 1989, pp. 109–123.

[DL93] Doligez, D. and X. Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ml. In *POPL '93*, New York, NY, January 1993. ACM, pp. 113–123.

[FF04] Flatt, M. and R. B. Findler. Kill-safe synchronization abstractions. In *PLDI '04*, June 2004, pp. 47–58.

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

[FRR+07] Fluet, M., M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Manticore: A heterogeneous parallel language. In *DAMP '07*, New York, NY, January 2007. ACM, pp. 37–44.

[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.

[GJ79] Garey, M. R. and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[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.

[GR04] Gansner, E. R. and J. H. Reppy (eds.). *The Standard ML Basis Library*. Cambridge University Press, Cambridge, England, 2004.

[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.

[Hof05] Hofstee, H. P. Cell broadband engine architecture from 20,000 feet. Available at `http://www-128.ibm.com/developerworks/power/library/pa-cbea.html`, August 2005.

[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.

[Kel99] Keller, G. *Transformation-based Implementation of Nested Data Parallelism for Distributed Memory Machines*. Ph.D. dissertation, Technische Universität Berlin, Berlin, Germany, 1999.

[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.

[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.

[OH05] Olukotun, K. and L. Hammond. The future of microprocessors. *ACM Queue*, **3**(7), September 2005. Available from `http://www.acmqueue.org`.

[PDGQ05] Pike, R., S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with sawzall. *Scientific Programming Journal*, **13**(4), 2005, pp. 227–298.

[Pet92] Pettersson, M. A term pattern-match compiler inspired by finite automata theory. In *CC '92*, vol. 641 of *LNCS*, New York, NY, October 1992. Springer-Verlag, pp. 258–270.

[PTH01] Peyton Jones, S., A. Tolmach, and T. Hoare. Playing by the rules: Rewriting as a practical optimization technique in GHC. In *Proceedings of the 2001 Haskell Workshop*, September 2001, pp. 203–233.

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

[Ram90] Ramsey, N. Concurrent programming in ML. *Technical Report CS-TR-262-90*, Dept. of C.S., Princeton University, April 1990.

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

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

[Rep90] Reppy, J. H. Asynchronous signals in Standard ML. *Technical Report TR 90-1144*, Dept. of CS, Cornell University, Ithaca, NY, August 1990.

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

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

[Rep02] Reppy, J. Optimizing nested loops using local CPS conversion. *HOSC*, **15**, 2002, pp. 161–180.

[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*, New York, NY, January 2007. ACM, pp. 315–326.

[SA00] Shao, Z. and A. W. Appel. Efficient and safe-for-space closure conversion. *ACM TOPLAS*, **22**(1), 2000, pp. 129–161.

[SCM99] Shivers, O., J. W. Clark, and R. McGrath. Atomic heap transactions and fine-grain interrupts. In *ICFP '99*, New York, NY, September 1999. ACM, pp. 48–59.

[Sha07] Shaw, A. Data parallelism in Manticore. Master's dissertation, University of Chicago, July 2007. Available from `http://manticore.cs.uchicago.edu`.

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

[Ste06] Steele Jr., G. L. Parallel programming and code selection in Fortress. In *PPoPP '06*, New York, NY, March 2006. ACM, p. 1. Keynote talk; slides available from `http://research.sun.com/projects/plrg/CGOPPoPP2006public.pdf`.

[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 *LFP '80*, New York, NY, August 1980. ACM, pp. 19–28.

[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.