

The Manticore Project

(Status Report)

<http://manticore.cs.uchicago.edu>

John Reppy
University of Chicago

People

The Manticore project is a joint project between the University of Chicago and the Toyota Technological Institute.

Matthew Fluet	Toyota Technological Institute at Chicago
Nic Ford	University of Chicago
Mike Rainey	University of Chicago
Adam Shaw	University of Chicago
Yingqi Xiao	University of Chicago

with help from

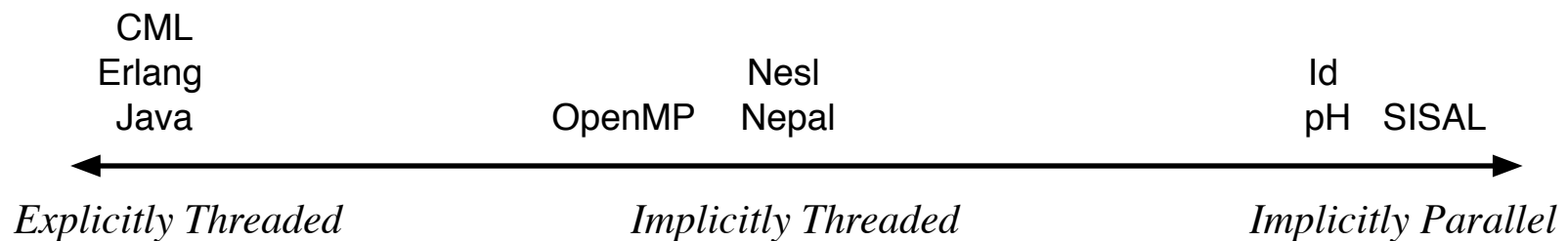
Jon Riehl	University of Chicago
Ridg Scott	University of Chicago

Background and motivation

- In the past, processor clock speeds increased regularly, which meant that sequential performance increased regularly.
- Because of heat and power issues, further increases in clock speeds will be small.
- Instead, microprocessors are becoming multiprocessors.
- Commodity computers support parallelism at multiple levels: SIMD, SMT, multicore, and small-scale SMP.
- Likewise, there are many applications that run on commodity hardware that exhibit parallelism at multiple levels.

Background and motivation *(continued ...)*

- The Manticore project is our effort to address the programming needs of commodity applications running on the commodity hardware of 2010.
- To maximize the performance of these applications, we need a parallel language that supports parallelism at multiple levels.
- We call this property *heterogeneous parallelism*.



Project overview

The Manticore project has three major aspects:

1. Language design for heterogeneous parallel programming
2. Nested schedulers for heterogeneous parallelism.
3. Compiler and runtime support for a high-performance implementation of parallel applications in Manticore.

Language design

The initial design of Manticore is purposefully conservative. It can be described as the combination of three distinct aspects:

- A mutation free subset of SML (no refs or arrays, but includes exceptions).
- Language mechanisms for *implicitly-threaded* parallel programming.
- Language mechanisms for *explicitly-threaded* parallel programming (*a.k.a.* concurrent programming).

Language design *(continued ...)*

Manticore provides several light-weight syntactic forms for introducing parallel computation. These forms are *hints* to the compiler and runtime that a computation is a good candidate for parallel execution.

- *Parallel arrays* provide fine-grain data-parallel computations over sequences.
- *Parallel tuples* provide a basic fork-join parallel computation.
- *Parallel bindings* provide data-flow parallelism with cancelation of unused subcomputations.
- *Parallel choice* provides speculative parallelism.

Parallel arrays

We support fine-grained data-parallel computation using a *parallel array comprehension* form (NESL/Nepal/DPH):

```
[ : exp | pati in expi where pred : ]
```

For example, the parallel point-wise summing of two arrays:

```
[ : x+y | x in xs, y in ys : ]
```

NOTE: zip semantics, not Cartesian-product semantics.

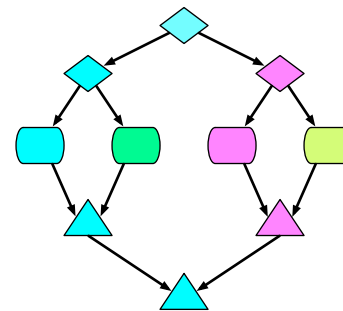
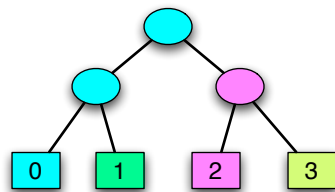
Parallel tuples

Parallel tuples provide fork-join parallelism. For example, consider summing the leaves of a binary tree.

```
datatype tree = LF of long | ND of tree * tree
```

```
fun treeAdd (LF n) = n
```

```
| treeAdd (ND(t1, t2)) = (op +) (| treeAdd t1, treeAdd t2 |)
```

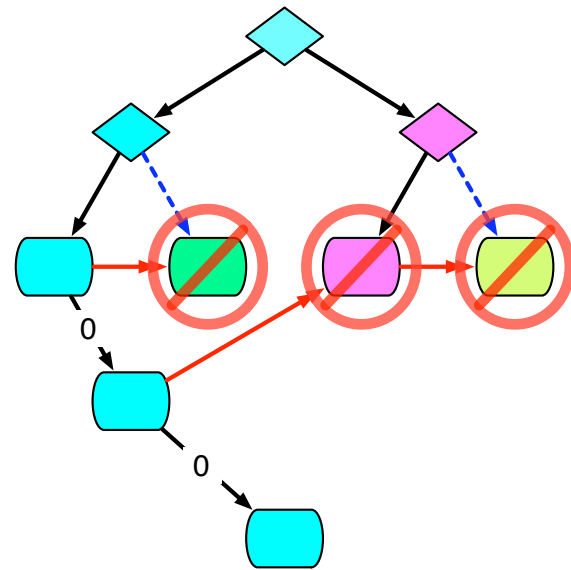
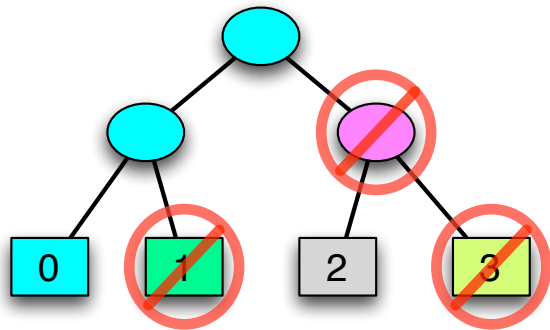


Parallel bindings

Parallel bindings provide more flexibility than parallel tuples. For example, consider computing the product of the leaves of a binary tree.

```
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
```

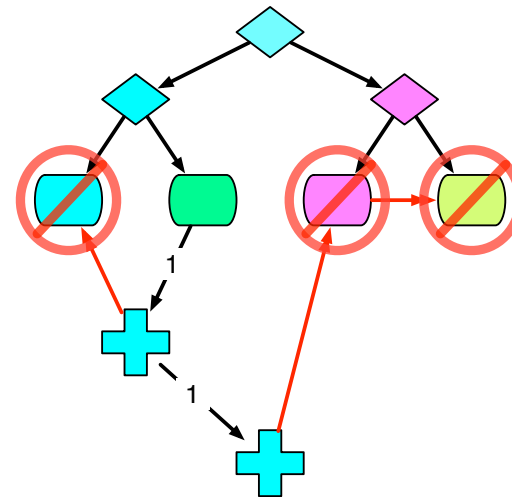
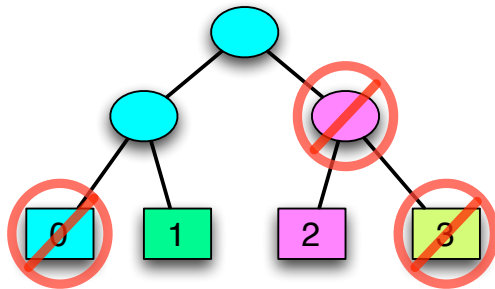
Parallel bindings (continued ...)



Parallel choice

Parallel choice supports speculative parallelism when we want the quickest answer (e.g., search problems). For example, consider picking a leaf of the tree:

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



Explicit threading

Course-grain parallelism and explicit concurrency is supported via CML-style primitives.

Explicit thread creation

```
spawn exp
```

and channels for message passing

```
type 'a chan  
val channel : unit -> 'a chan  
val send : ('a chan * 'a) -> unit  
val recv : 'a chan -> 'a
```

We also have CML-style event combinators.

Nested schedulers for heterogeneous parallelism

The runtime model is designed to support multiple scheduling policies in a common framework.

It is based on heap-allocated first-class continuations and has three distinct notions of process abstraction:

Fibers are unadorned threads of control. A suspended fiber's state is represented as a continuation.

Threads correspond to language-level threads and have an ID.

Virtual Processors (VProcs) are an abstraction of a computational resource. In our current implementation, each VProc is hosted by its own *pthread*.

Scheduler actions

A *scheduler action* is a function that implements context switching for a VProc.

```
type fiber = unit cont
```

```
datatype signal
```

```
= STOP
```

```
| PREEMPT of fiber
```

```
type action = signal cont
```

VProc action-stack transitions

The runtime model defined three operations on a VProc's action stack:

```
val run : action -> fiber -> 'a
```

pushes the action on the stack and starts running the fiber.

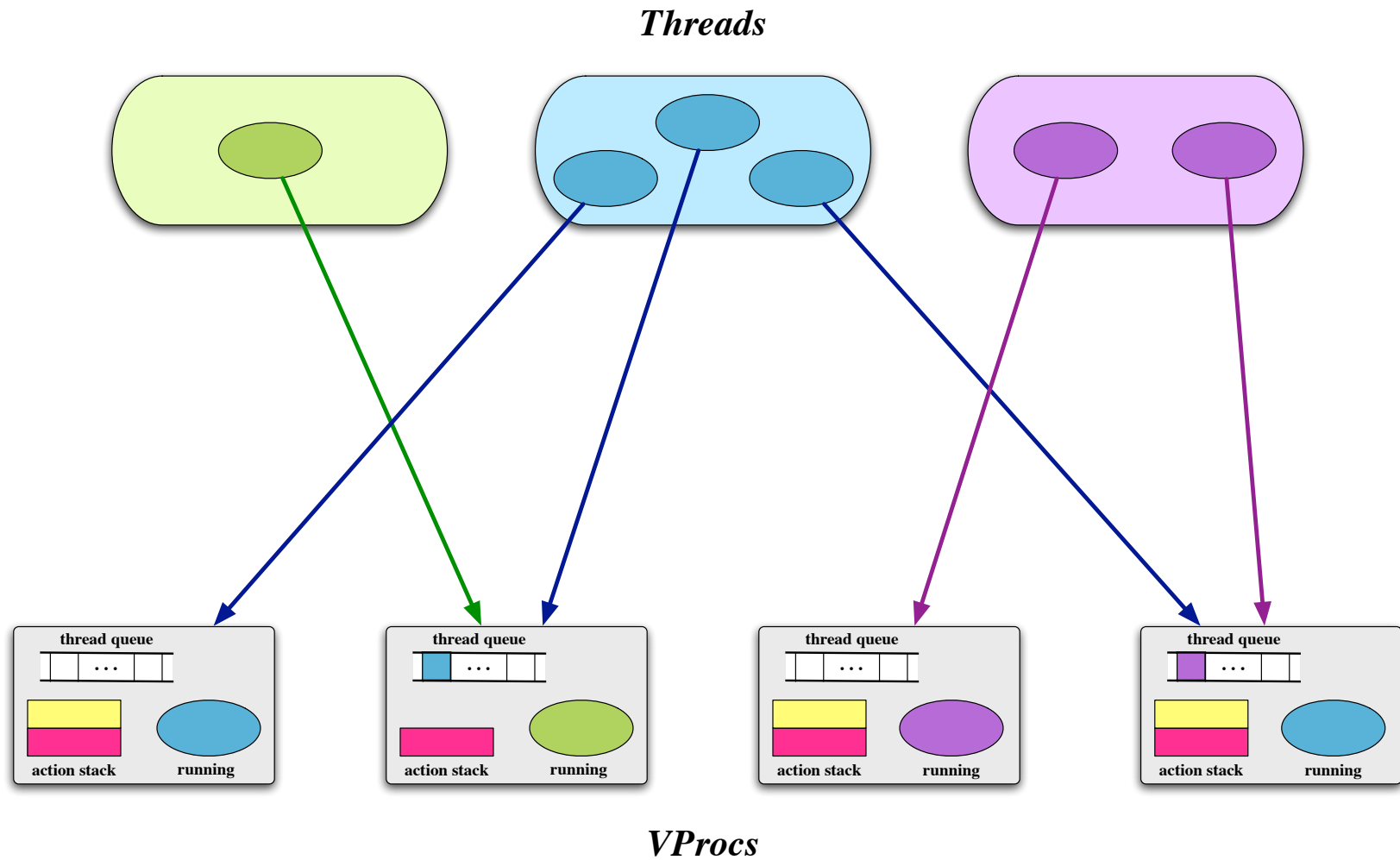
Preemption

captures the current state as a fiber and delivers a preemption signal to the top action on the stack (which is popped)

```
val forward : signal -> 'a
```

delivers the signal to the top action on the stack (which is popped)

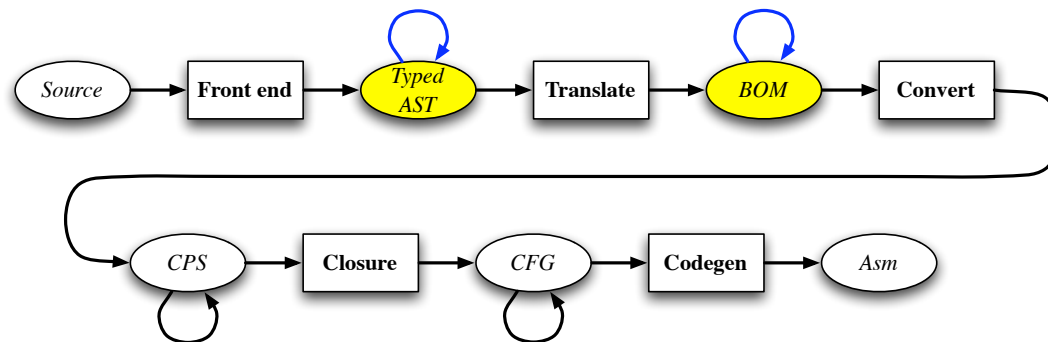
Schedulers are distributed across VProcs



The Manticore compiler

The compiler is organized as a series on transformations between IRs:

- **Typed AST** — explicitly-typed, polymorphic, abstract-syntax tree.
- **BOM** — direct-style, normalized λ -calculus.
- **CPS** — continuation-passing-style λ -calculus.
- **CFG** — first-order control-flow graph



AST optimizations

- Flattening of nested-data-parallelism (AOS to SOA).
- Introduction of futures for other implicitly-threaded parallel constructs.
- Compilation of pattern matching.

Introducing futures

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

BOM optimizations

- Standard functional-compiler optimizations (*e.g.*, uncurrying, inlining, and contraction).
- High-level operator expansion.
- Domain-specific optimization.

High-level operations

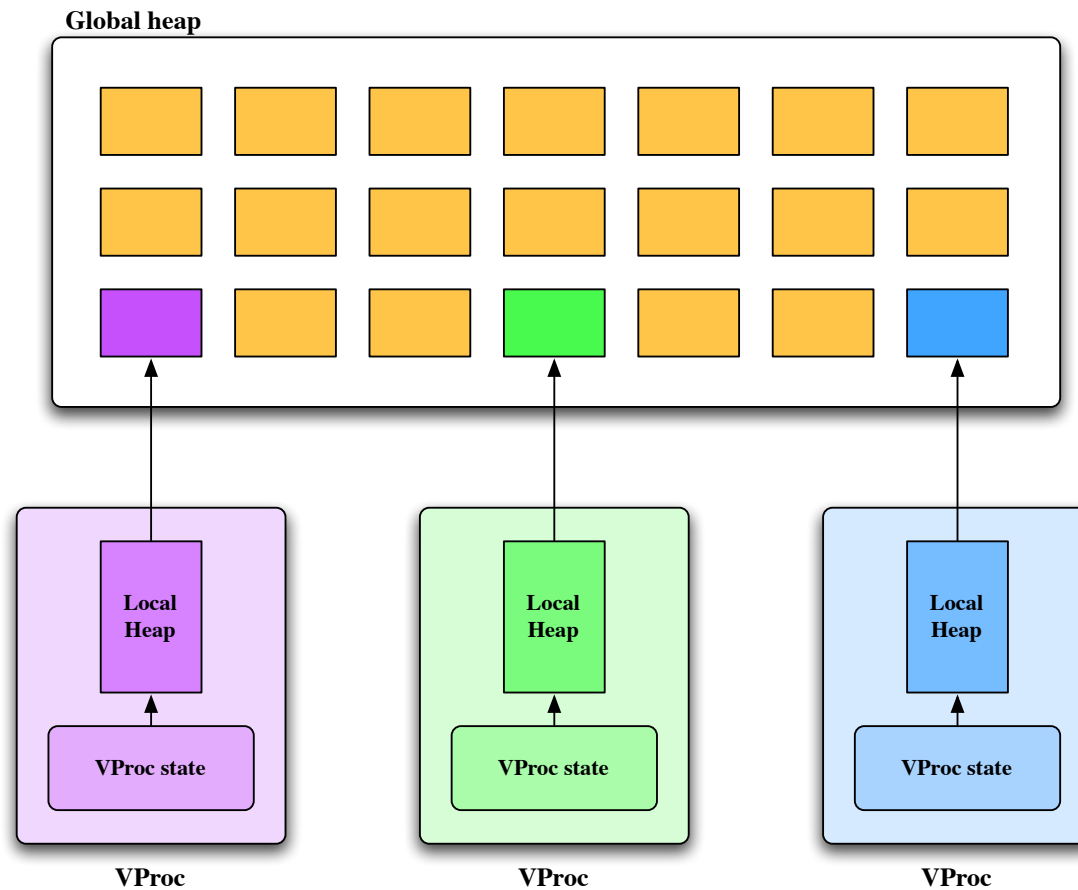
- High-level operators provide a uniform way of dealing with special cases: one syntactic form (application), but global name space.
- Used to implement concurrency and parallel features (*e.g.*, spawn, future, and touch).
- Used to import scheduler code into a program and to implement scheduler operations (*e.g.*, run and forward).
- Rewriting to implement DSOs (*e.g.*, fusion)

Runtime-system overview

- VProcs are implemented as pthreads.
- The GC is a combination of the Appel Semi-generational collector and the Doligez-Leroy-Gonthier parallel collector.
- Each VProc has a local heap that can be independently collected.

Heap architecture

Goal: avoid synchronization and communication between VProcs.



Implementation status

- A front-end for “mini-Manticore” is implemented.
- All of the IRs (AST, BOM, CPS, and CFG) are implemented (with invariant checkers), as are the translations between IRs.
- Code generation for the x86-64 is implemented using the MLRisc framework.
- The runtime system (but not global GC) is implemented on Linux and Mac OS X.
- A number of basic BOM optimizations are implemented.

Implementation status *(continued ...)*

- We are currently working on AST to AST transforms for parallel constructs.
- Protocols for the CML operations have been designed, but not coded yet.
- Rewriting for high-level operators is currently being implemented.
- Stress testing and debugging.
- Applications and performance tuning.

Summary

- Manticore combines ideas in language design to support heterogeneous parallelism on commodity hardware.
- We have a flexible runtime model for supporting heterogeneous scheduling policies.
- Work on a Manticore compiler and runtime system is ongoing and we expect a release later this year.