Implicitly-threaded Parallelism in Manticore

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

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

Abstract

The increasing availability of commodity multicore processors is making parallel computing available to the masses. Traditional parallel languages are largely intended for large-scale scientific computing and tend not to be well-suited to programming the applications one typically finds on a desktop system. Thus we need new parallel-language designs that address a broader spectrum of applications. In this paper, we present Manticore, a language for building parallel applications on commodity multicore hardware including a diverse collection of parallel constructs for different granularities of work. We focus on the implicitly-threaded parallel constructs in our high-level functional language. We concentrate on those elements that distinguish our design from related ones, namely, a novel parallel binding form, a nondeterministic parallel case form, and exceptions in the presence of data parallelism. These features differentiate the present work from related work on functional data parallel language designs, which has focused largely on parallel problems with regular structure and the compiler transformations — most notably, flattening — that make such designs feasible. We describe our implementation strategies and present some detailed examples utilizing various mechanisms of our language.

Categories and Subject Descriptors D.3.0 [*Programming Languages*]: General; D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages; Concurrent, distributed, and parallel languages; D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent programming structures

General Terms Design, Languages

Keywords implicitly-threaded parallelism, data parallelism, parallel binding, parallel case, exceptions

1. Introduction

The laws of physics and the limitations of instruction-level parallelism are forcing 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. Our design is targeted at these small to medium-scale parallel applications; the rationale for this focus has been stated at greater length in prior work [FRR+07].

A homogeneous language design is not likely to take full advantage of the hardware resources available. 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 to the components of a program that demand fine-grained parallelism, such as image processing and particle systems.

Our belief 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. Indeed, a number of research projects are exploring *heterogeneous* parallelism in languages that combine support for parallel computation at different levels into a common linguistic and execution framework. The Glasgow Haskell Compiler [GHC] has been extended with support for three different paradigms for parallel programming: explicit concurrency coordinated with transactional memory [PGF96, HMPH05], semi-implicit concurrency based on annotations [THLP98], and data parallelism [CLP+07], inspired by NESL [BCH+94, Ble96]. Manticore [FRR+07, FFR+07] incorporates both coarse-grained parallelism and fine-grained nested parallelism. Manticore's coarse-grained parallelism is based on Concurrent ML (CML) [Rep91], which provides explicit concurrency and synchronous-message passing. Manticore's fine-grained nested parallelism is based on previous nested data-parallel languages, such as NESL [BCH+94, Ble96, BG96] and Nepal [CK00, CKLP01, LCK06]), and provides data parallel arrays and parallel comprehensions, among other mechanisms.

After an overview of the Manticore language (Section 2), we present four main technical contributions:

- the **`pval`** binding form, for parallel evaluation and speculation (Section 3),
- the **`pcase`** expression form, for nondeterminism and user-defined parallel control structures (Section 4),
- the inclusion of exceptions and exception handlers in a data parallel context (Section 5), and
- our implementation strategies for all of the above, as well as our approach to handling exceptions in the context of parallel arrays (Section 7).

We exercise our design on a series of examples in Section 6. The examples are meant to highlight Manticore's suitability for irregular parallel applications, in contrast to the embarrassingly parallel examples that often appear in the data parallelism literature. We review related work and conclude in Sections 8 and 9.

ICFP'08, September 22–24, 2008, Victoria, BC, Canada.

2. An Overview of the Manticore Language

Parallel language mechanisms can be roughly grouped into three categories:

- *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 the mapping of computation onto parallel threads is left to the compiler and runtime system. Examples include NESL [Ble96] and its descendant Nepal [CKLP01], later renamed Data Parallel Haskell ("DPH" below).
- *explicit threading*, where the programmer explicitly creates parallel threads. Examples include CML [Rep99] and Erlang [AVWW96].

These different design points represent different trade-offs between programmer effort and programmer control. Automatic techniques for parallelization have proven effective for dense regular parallel computations (*e.g.*, dense matrix algorithms), but have been less successful for irregular problems. Manticore provides both implicit threading and explicit threading mechanisms. The former supports fine-grained parallel computation, while the latter supports coarse-grained parallel tasks and explicit concurrent programming. These parallelism mechanisms are built on top of a sequential functional language. In the sequel, we discuss each of these in turn, starting with the sequential base language. Space precludes giving a complete account of Manticore's language-design philosophy, goals, and target domain; we refer the interested reader to our previous publications [FRR+07, FFR+07].

2.1 Sequential Programming

Manticore's sequential core is based on a subset of Standard ML (SML). The main difference is the absence of mutable data (*i.e.*, reference cells and arrays) and, in the present language implementation, a module system. Manticore includes the functional elements of SML (datatypes, polymorphism, type inference, and higher-order functions) as well as exceptions. The interaction of exceptions and our implicit threading mechanisms adds some complexity to our design, as we discuss below, but we believe that an exception mechanism is necessary for systems programming. As many researchers have observed, using a mutation-free language greatly simplifies the implementation and use of parallel features [Ham91, Rep91, JH93, NA01, DG04]. In essence, mutation-free functional programming reduces interference and data dependencies. We recognize that the lack of mutable data has the potential to denigrate performance for certain common algorithms; nevertheless, we fell there is evidence of the success of languages that lack this feature (Erlang is one compelling example).

As the syntax and semantics of the sequential core language are largely orthogonal to the parallel language mechanisms, we have resisted tinkering with core SML. The Manticore Basis, however, differs significantly from the SML Basis Library [GR04]. In particular, we have a fixed set of numeric types — `short`, `int`, `long`, `float`, and `double` — instead of SML's families of numeric modules.

2.2 Explicitly-threaded parallelism

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

Further discussion of explicitly-threaded parallelism in Manticore is beyond the scope of this paper; other recent publications (notably [RX08]) provide more detail.

2.3 Implicitly-threaded Parallelism

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

Having a sequential semantics is useful in two ways: it gives the programmer a deterministic programming model and it formalizes the expected behavior of the compiler. It also requires the compiler to verify that the individual subcomputations in a parallel computation do not send or receive messages (if the computations are actually to be executed in parallel). Similarly, if a subcomputation raises an exception, the implementation must delay the delivery of the exception until all sequentially prior computations have terminated. Both of these restrictions can be efficiently implemented only through appropriate program analyses.

Parallel Arrays Support for parallel computations on arrays and matrices is common in parallel languages. In Manticore, we support such computations using the nested parallel array mechanism inspired by NESL and developed further by Nepal and DPH. A parallel array expression has the form

$$\texttt{[|}e_1\texttt{, ..., }e_n\texttt{|]}$$

which constructs an array of n elements. The delimiters `[| |]` alert the compiler that the e_i may be evaluated in parallel.

Parallel array values may also be constructed using *parallel comprehensions*, which allow concise expressions of parallel loops. A comprehension has the general form

$$\texttt{[|}\ e\ \texttt{|}\ p_1\ \texttt{in}\ e_1\texttt{, ...,}\ p_n\ \texttt{in}\ e_n\ \texttt{where}\ e_f\ \texttt{|]}$$

where e is some expression (with free variables bound in the p_i) computing the elements of the array, the p_i are patterns binding the elements of the e_i, which are array-valued expressions, and e_f is an optional boolean-valued expression over the p_i filtering the input. If the input arrays have different lengths, all are truncated to the length of the shortest input, and they are processed, in parallel, in lock-step.[1] For convenience, we also provide a parallel range form

$$\texttt{[|}\ e_l\ \texttt{to}\ e_h\ \texttt{by}\ e_s\ \texttt{|]}$$

which are useful in combination with comprehensions. (The step expression **`by`** e_s is optional, and defaults to **`by`** `1`.)

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 may use the following expression:

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

[1] 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.

```
datatype tree
  = Lf of int
  | Nd of tree * tree

fun trProd (Lf i) = i
  | trProd (Nd (tL, tR)) =
      op* (|trProd1 tL, trProd1 tR|)
```

Figure 1. Tree product with parallel tuples.

This expression can evaluated efficiently in parallel using vector instructions.

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

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

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

The sequential semantics of 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 its ith element, then we must wait until all preceding elements have been computed before propagating the exception. Section 7.2 describes our implementation strategy for this behavior.

Parallel Tuples Like the parallel array expression forms, the parallel tuple expression form hints to the compiler that the elements may be evaluated in parallel. The basic form is

$$(|e_1,\ \ldots,\ e_n|)$$

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

Figure 1 demonstrates the use of parallel tuples to compute the product of the leaves of a binary tree of integers. While this example could have been written with parallel arrays, it is more convenient to use parallel tuples.

An advantage of parallel tuples is that the elements need not all have the same type. In languages with only a parallel array construct, a programmer can evaluate expressions of different types by, for example, injecting them into an *ad hoc* union datatype, collecting them in a parallel array, and then projecting them out of that datatype, but this incurs uninteresting complexity in the program and adds runtime overhead.

The sequential semantics of parallel tuples is trivial: they are evaluated simply as (sequential) tuples. The implication for the parallel implementation is similar to that for parallel arrays: if an exception is raised when computing its ith element, then we must wait until all preceding elements have been computed before propagating the exception.

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

$$\texttt{pval}\ p\ \texttt{=}\ e$$

that launches the evaluation of the expression e as a parallel thread. The sequential semantics of a parallel binding are similar to lazy evaluation: the binding is only evaluated (and only evaluated once) when one of its bound variables is demanded. One important subtlety in the semantics of parallel bindings is that any exceptions raised by the evaluation of the binding must be postponed until one of the variables is touched, at which time the exception is raised at the point of the touched variable. In the parallel implementation, we use eager evaluation for parallel bindings, but computations are canceled when the main thread of control reaches a point where their result is guaranteed never to be demanded.

```
fun trProd (Lf i) = i
  | trProd (Nd (tL, tR)) = let
      pval pL = trProd2 tL
      pval pR = trProd2 tR
      in
        if (pL = 0) then 0
        else (pL * pR)
      end
```

Figure 2. Short-circuiting tree product with parallel bindings.

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

Parallel case is flexible enough to express a variety of nondeterministic parallel mechanisms, including, notably, the parallel choice operator `|?|`, which nondeterministically returns either of its two operands. A more detailed treatment of the syntax and semantics of parallel case is deferred until Section 4.

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

3. Parallel Bindings

The distinguishing characteristic of the parallel binding, or **`pval`** mechanism, is that its launched computation may be canceled before completion. When a (simple, syntactic) program analysis determines those program points where a launched computation is guaranteed never to be demanded, the compiler inserts a corresponding cancellation. Note that these sites can only be located in conditional expression forms.

As in Figure 1, the function in Figure 2 computes the product of the leaves of a tree. This version short-circuits, however, when the product of the left subtree of a `Nd` variant evaluates to zero. Note that if the result of the left product is zero, we do not need the result of the right product. Therefore its subcomputation and any descendants may be canceled. The short-circuiting behavior is not explicit in the function. Rather, it is implicit in the semantics of a parallel binding that when control reaches a point where the result of an evaluation is known to be unneeded, the resources devoted to that evaluation are freed and the computation is abandoned.

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

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

In the conditional expression here, while the computation of `y` can be canceled in the **`then`** branch, the computation of `x` cannot be canceled in either branch. Our analysis must respect this depen-

dency and similar subtle dependencies. We consider an example similar to this one in Section 7 in some detail.

There are many more examples of the use of parallel bindings in Section 6. We discuss the specific mechanisms (most importantly, *futures*) by which we realize their semantics in Section 7.

4. Parallel Case Expressions

The parallel case expression form has the following syntax:

```
pcase e & ... & e
 of π & ... & π => e
  | ...
  | otherwise => e
```

The metavariable π denotes a *parallel pattern*, which is either

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

where p in the latter two cases signifies a conventional SML pattern. Furthermore, **`pcase`** expressions include an optional **`otherwise`** branch, always last if present, which has a special meaning as discussed below.

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

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

Once the constant pattern `false` has been matched with the result of the first discriminant's computation, the running program need not wait for the second discriminant's computation to finish; it can return `false` straightaway. This branch is part of the expression of a short-circuiting parallel boolean conjunction, discussed below in more detail.

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

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

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

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

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

To illustrate the use of parallel case expressions, we consider parallel choice. (In fact, the parallel case expression was designed as a generalization of parallel choice.) A parallel choice expression `e1 |?| e2` nondeterministically returns either the result of `e1` or `e2`. This is similar to MultiLisp's *parallel or* (which is not the same as a parallel boolean disjunction). This is useful in a parallel context, because it gives the program the opportunity to return whichever of `e1` or `e2` evaluates first.

Employing the tree datatype from Figure 1, we might want to write a function to obtain the value of a leaf—any leaf—from a given tree.

```
fun trLeaf (Lf i) = i
  | trLeaf (Nd (tL, tR)) =
      trLeaf(tL) |?| trLeaf(tR)
```

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

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

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

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

```
fun trProd (Lf i) = i
  | trProd (Nd (tL, tR)) =
     (pcase trProd(tL) & trProd(tR)
       of 0 & ? => 0
        | ? & 0 => 0
        | pL & pR => pL * pR)
```

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

Is this the most efficient `trProd` function? Not necessarily. The implementation of the parallel case mechanism, as discussed in Section 7.6, induces more overhead than other mechanisms, and on small trees this function might run more slowly than lower-overhead counterparts.

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

```
fun trFindB (p, Lf i) = (* B for broken *)
      if p(i) then SOME(i)
      else NONE
  | trFindB (p, Nd (tL, tR)) =
      trFindB(p,tL) |?| trFindB(p,tR)
```

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

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

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

Figure 3. Encoding a parallel boolean conjunction with `pcase`.

```
fun trFind (p, Lf i) =
      if p(i) then SOME(i)
      else NONE
  | trFind (p, Nd (tL, tR)) =
      pcase trFind(p,tL) & trFind(p,tR)
       of SOME(n) & ? => SOME(n)
        | ? & SOME(n) => SOME(n)
        | NONE & x => x
        | x & NONE => x
```

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

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

```
datatype side = L | R

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

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

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

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

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

The **`pcase`** mechanism offers the best encoding of parallel conjunction (Figure 3). Only when both evaluations complete and are `true` does the expression as a whole evaluate to `true`. If one constituent of a parallel conjunction evaluates to `false`, the other can be safely canceled. As soon as one expression evaluates to `false`, the other is canceled, and `false` is returned. As a convenience, Manticore provides `|andalso|` as a derived form for this expression pattern.

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

5. Exceptions and Exception Handlers

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

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

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

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

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

Although `B` might be raised before `A` during a parallel execution, `A` must be the exception observed to be raised by the context of the parallel array expression in order to adhere to the sequential semantics. Realizing this behavior in this and other parallel constructs requires our implementation to include compensation code, with some runtime overhead. In the present work, we give details of our implementation of this behavior, compensation code included, in Section 7.

In choosing to adopt a strict sequential core language, Manticore is committed to realizing a precise exceptions semantics in the implicitly-threaded parallel features of the language. This is in contrast to an imprecise exception semantics [PRH+99] that arise from a lazy sequential language. While a precise semantics requires a slightly more restrictive implementation of the implicitly-threaded parallel features than would be required with an imprecise semantics, we believe that support for exceptions and the precise semantics is crucial for systems programming. Furthermore, as Section 7 will show, implementing the precise exception semantics is not particularly onerous.

It is possible to eliminate some or all of the compensation code with the help of program analyses. There already exist various well-known analyses for identifying exceptions that might be raised by a given computation [Yi98, LP00]. If, in a parallel array expression, it is determined that no subcomputation may raise an exception, then we are able to omit the compensation code and its overhead. As another example, consider a parallel array expression where all subcomputations can raise only one and the same exception.

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

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

Although exception handlers are first-class expressions, their behavior is orthogonal to that of the parallel constructs and mostly merit no special treatment in the implementation. At the present time, Manticore does not implement any form of flattening transformation on data parallel array computations. Once we incorporate flattening into our work, however, we will need to take particular account of exception handlers, since flattening and exception handlers cannot freely coexist [Sha07].

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

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

the latter two computations should be abandoned as soon as possible. Section 7 details our approaches when this and similar issues arise.

6. Examples

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

6.1 A Parallel Typechecking Interpreter

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

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

Our language includes the following definition of types.

```
datatype ty = Bool | Nat | Arrow of ty * ty
```

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

```
datatype term
  = N of int | B of bool | V of var
  | Add of exp * exp
  | If of exp * exp * exp
  | Let of var * exp * exp
  | Lam of var * ty * exp
  | Apply of exp * exp
    ...
withtype exp = loc * term
```

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

```
fun tyEq (Bool, Bool) = true
  | tyEq (Nat, Nat) = true
  | tyEq (Arrow(t,t'), Arrow(u,u')) =
     (pcase tyEq(t,u) & tyEq(t',u')
       of false & ? => false
        | ? & false => false
        | true & true => true)
  | tyEq _ = false
```

In practice, we could use the parallel and operator |andalso| for the Arrow case

```
tyEq(t,u) |andalso| tyEq(t',u')
```

which would desugar into the expression explicitly written above.

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

```
datatype 'a or_error
  = A of 'a
  | Err of loc
```

The signature of typeOf is

```
val typeOf : env * exp -> ty or_error
```

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

```
fun typeOf (G, (p, Add (e1,e2))) = let
      pval t2 = typeOf(G, e2)
      in case typeOf(G, e1)
        of A Nat => (case t2
             of A Nat => A Nat
              | A _ => Err(locOf(e2))
              | Err q => Err q)
         | A _ => Err(locOf(e1))
         | Err q => Err q
      end
```

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

```
  | typeOf (G, (p, Apply (e1, e2))) = let
     pval t2 = typeOf(G, e2)
     in case typeOf(G, e1)
      of A(Arrow(d,r)) => (case t2
           of A t => if tyEq(d,t) then A r
                     else Err p
            | Err q => Err q)
       | A _ => Err(locOf(e1))
       | Err q => Err q
     end
```

Where there are no independent subexpressions, no parallelism is available:

```
  | typeOf (G, (p, IsZero(e))) =
     (case typeOf(G,e)
       of A Nat => A Bool
        | _ => Err p)
```

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

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

```
fun subst (t, x, e as (p, t')) = (case t'
     of V(y) => if varEq(x,y) then (p,t) else e
      | Let(y,e1,e2) => if varEq(x,y)
          then (p, Let(y, subst(t,x,e1), e2))
          else (p, Let(|y, subst(t,x,e1),
                       subst(t,x,e2)|))
     ...
```

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

```
  | eval (p, Add(e1,e2)) =
     (pcase eval(e1) & eval(e2)
       of (N n1, N n2) => N(n1+n2)
        | otherwise => raise RuntimeError)
```

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

```
  | eval (p, If(e1, e2, e3)) =
     (pcase eval(e1) & eval(e2) & eval(e3)
       of B true & v & ? => v
        | B false & ? & v => v
        | otherwise => raise RuntimeError)
```

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

```
fun typedEval e =
  (pcase typeOf(emptyEnv,e) & eval(e)
    of (Err p, ?) => Err p
     | (A _, v) => A v)
```

6.2 Parallel Contract Checking

In this section, we extend the simple typed language of the previous section with parallel contract checking for a minimal contract system. To illustrate our point, we consider only the simplest kind of contracts, flat contracts on function arguments [Par72, Luc90].

To define the language, we extend the `term` datatype. A contract, in this model language, is a function that consumes a value and returns a boolean. Therefore, there is no distinguished contract variant in the datatype.

```
datatype term
  = ...
  | LamG of exp * var * ty * exp
  | Blame of loc
    ...
```

`LamG` is a special form for the representation of guarded functions. The first component is a contract for the function argument; the second, third, and fourth components are the function argument, the function argument's type, and the function body, respectively. Note that the variable is in scope only in the body, not the contract. If the argument to a `LamG` function fails to meet its contract, the location of the application supplying the argument will be blamed. This blame is reified in the form of a `Blame` expression naming the location of the contract violation.

As before, we define a function

```
eval : exp -> exp
```

to evaluate expressions in this language. We omit most of its predictable definition. The evaluation of expressions in this language is mostly standard, although care must be taken at each step to check if any `Blame`s have been generated, as we wish to report the earliest contract violation. In the `Add` case, for example, we evaluate both subexpressions in parallel by launching the evaluation of the second with **`pval`**.

```
  | eval (p, Add (e1, e2)) => let
     pval v2 = eval(e2)
     in case eval(e1)
      of Blame c => Blame c
       | N n1 => (case v2
          of Blame c => Blame c
           | N n2 => Add(n1+n2)
           | _ => raise RuntimeError)
       | _ => raise RuntimeError
     end
```

It is unnecessary to launch the evaluations of both operands with **`pval`**s; launching just one suffices for running both simultaneously. If neither evaluation blames anything, we proceed to evaluate the addition as usual. Here, a sequential case expression is employed to express the left bias of the contract-checking system. Note how the mixed presence of parallel and sequential constructs can be used to express a particular program.

When a contract must be checked, its evaluation is done in parallel with the evaluation of the expression as a whole. If the contract is not adhered to, the main evaluation is implicitly canceled; otherwise, it is returned.

```
  | eval (p, App (e1, e2)) = let
     pval v2 = eval(e2)
     in case eval(e1)
      of Blame c => c
       | Lam(x,_,b) => (case v2
          of Blame c => Blame c
           | _ => eval (subst(v2,x,b)))
       | LamG(c,x,_,b) => (case v2
          of Blame c => Blame c
           | _ => let
              pval res = eval(subst(v2,x,b))
              val chk = eval((p,App(c,v2)))
              in case chk
               of B false => Blame p
                | B true => res
                | _ => raise RuntimeError
              end)
  end
```

6.3 Parallel Game Search

We now consider the problem of searching a game tree in parallel. This has been shown to be a successful technique by the Cilk group for games such as Pousse [BAP+98] and chess [DL02].

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

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

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

```
fun maxT (board, alpha, beta) =
  if gameOver(board) then
    Rose ((board, boardScore board), [||])
  else let
    val ss = successors (board, X)
    val t0 = minT (ss!0, alpha, beta)
    val alpha' = max (alpha, treeScore t0)
    fun loop i =
      if (i = (plen ss)) then [||]
      else let
        pval ts = loop (i+1)
        val ti = minT (ss!i, alpha', beta)
        in
          if (treeScore ti) >= beta then
            [|ti|] (* prune *)
          else
            [|ti|] |@| ts
        end
    val ch = [|t0|] |@| loop(1)
    val maxScore = maxP [| treeScore t | t in ch |]
    in
      Rose ((board, maxScore), ch)
    end
```

Figure 4. The `maxT` half of parallel alpha-beta pruning. `!` is the subscript operator for parallel arrays. The parallel computation of `ts` can be canceled in the line marked `(* prune *)`.

A `player` is either of the nullary constructors `X` or `O`; a `board` is a parallel array of nine `player option`s, where `NONE` represents an empty square. Extracting the available moves from a given board is written as a parallel comprehension as follows:

```
fun allMoves b =
  [|i | s in b, i in [|0 to 8|] where isNone(s)|]
```

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

```
fun successors (b, p) =
  [| moveTo (b, p, i) | i in (allMoves b) |]
```

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

```
fun minimax (b : board, p : player) =
  if gameOver(b) then
    Rose ((b, boardScore b), [||])
  else let
    val ss = successors (b, p)
    val ch = [| minimax (b, other p) | b in ss |]
    val chScores = [| treeScore t | t in ch |]
    in
      case p
       of X => Rose ((b, maxP chScores), ch)
        | O => Rose ((b, minP chScores), ch)
    end
```

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

Using alpha-beta pruning yields a somewhat more realistic example. We implement it here as a pair of mutually recursive functions, `maxT` and `minT`. The code for `maxT` is shown in Figure 4, omitting some obvious helper functions; `minT`, not shown, is similar to `maxT`, with appropriate symmetrical modifications. Alpha-beta pruning is an inherently sequential algorithm, so we must adjust it slightly. This program prunes subtrees at a particular level of the search tree if they are at least as disadvantageous to the current player as an already-computed subtree. (The sequential algorithm, by contrast, considers every subtree computed thus far.) We compute one subtree sequentially as a starting point, then use its value as the pruning cutoff for the rest of the sibling subtrees. Those siblings are computed in parallel by repeatedly spawning computations in an inner loop by means of **`pval`**. Pruning occurs when the implicit cancellation of the **`pval`** mechanism cancels the evaluation of the right siblings of a particular subtree.

```
datatype 'a trap
  = Val of 'a
  | Exn of exn

datatype 'a rope
  = Leaf of 'a vector
  | Cat of 'a rope * 'a rope

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

Figure 5. Traps, ropes, and futures.

7. Implementation

To sketch the important points of our implementation, we express the transformations of the implicitly-threaded parallel constructs into more concrete mechanisms: specifically, as AST-to-AST rewrites. For clarity, we present the transformed program fragments in SML syntax in this paper.

To implement the implicitly-threaded parallel features of Manticore, we define two polymorphic SML datatypes, the `trap` and the `rope`, and we introduce MultiLisp-style futures [Hal84]. These datatypes, an abstract future type, and the signatures of the core future operations are given in Figure 5. These types and operations are sufficient to encode the language features presented above, as we demonstrate in this section.

7.1 Futures

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

```
fun touch f = (case poll f
  of NONE => touch f
   | SOME (Val v) => v
   | SOME (Exn e) => raise e)
```

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

Futures and future operations are implemented by means of a flexible scheduling framework [Rai07, FRR08] provided by the

Manticore compiler and runtime system. This framework is capable of handling the disparate demands of the various heterogeneous parallelism mechanisms and capable of supporting a diverse mix of scheduling policies. Futures are just one of the mechanisms implemented with this framework.

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

The implementation of futures makes use of standard synchronization primitives provided by the scheduling framework; *e.g.*, I-variables [ANP89] are used to represent future results. Cancellation is handled by special "cancellable" data structures that record parent-child relationships and the cancellation status of each future; a nestable scheduler action is used to poll for cancellation when a future computation is preempted.

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

7.2 Ropes

Parallel arrays are implemented via ropes [BAP95]. Ropes, originally proposed as an alternative to strings, are immutable balanced binary trees with vectors of data at their leaves. Read from left to right, the data elements at the leaves of a rope constitute the data of the parallel array it represents. Ropes admit fast concatenation and, unlike contiguous arrays, may be efficiently allocated in memory even when very large. One disadvantage of ropes is that random access to individual data elements requires logarithmic time. Nonetheless, we do not expect this to present a problem for many programs, as random access to elements of a parallel array will in many cases not be needed. However, a Manticore programmer should be aware of this representation.

As they are physically dispersed in memory, ropes are well-suited to being built in parallel, with different processing elements simultaneously working on different parts of the whole. Furthermore, ropes embody a natural tree-shaped parallel decomposition of common parallel array operations like maps and reductions. Note the rope datatype shown in Figure 5 is an oversimplification of our implementation for the purposes of presentation. In our prototype system, rope nodes also store their depth and data length. These values assist in balancing ropes and make length and depth queries constant-time operations.

7.3 Parallel Tuples

For our first transformation, we revisit an earlier example:

```
datatype tree
  = Lf of int
  | Nd of tree * tree

fun trProd (Lf i) = i
  | trProd (Nd (tL, tR)) =
      op* (|trProd1 tL, trProd1 tR|)
```

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

```
fun rmap (f, r) = (case r
    of Leaf v => Leaf (vmap (f, v))
     | Cat (r1, r2) => let
         val f2 = future (fn _ => rmap (f, r2))
         val m1 = rmap (f, r1)
                  handle e => (cancel f2; raise e)
         val m2 = touch f2
         in
           Cat (m1, m2)
         end)
```

Figure 6. Mapping a function over a rope in parallel.

Taking all this into account, the tree product code above is rewritten with futures as follows:

```
fun trProdT (Lf i) = i
  | trProdT (Nd (tL, tR)) = let
      val tup = let
        val fR = future (fn _ => trProdT tR)
        in
          (trProdT tl, touch fR)
          handle e => (cancel fR; raise e)
        end
      in
        op* tup
      end
```

In general, a parallel tuple `(|`e_1`, ..., `e_n`|)` is rewritten as follows:

```
let val f_2 = future (fn _ => e_2)
    ...
    val f_n = future (fn _ => e_n)
in
   (e_1, touch f_2, ..., touch f_n)
   handle e => (cancel f_2; ···; cancel f_n;
                raise e)
end
```

Note that if the expression e_i raises an exception (and `touch` f_i raises an exception), then the cancellation of $f_2, \ldots, f_i$ will have no effect. Since we expect exceptions to be rare, we choose this transformation rather than one that installs custom exception handlers for e_1 and each `touch` f_i.

7.4 Parallel Comprehensions

One of the most common operations on parallel arrays is to apply a function in parallel to all of its elements, as in

```
[| f x | x in a |]
```

for any function `f`. We transform this computation to the internal function `rmap`, whose definition is given in Figure 6. Note that if `rmap(f,r1)` raises an exception, then the future evaluating the map of the right half of the rope is canceled, and the exception from the map of the left half is propagated. If `rmap(f,r2)` raises an exception, then it will be propagated by the `touch f2`. By this mechanism, we raise the sequentially first exception as discussed above.

The maximum length of the vector at each leaf is controlled by a compile-time option; its default value is 256. Altering the maximum leaf length, currently a compile-time option, can affect the execution time of a given program. If the leaves store very little data, then ropes become very deep. Per the parallel decomposition shown in Figure 6, small leaves correspond to the execution of many futures. This leads to good load balancing when applying the mapped function to an individual element is relatively expensive. By contrast, large leaves correspond to the execution of few futures;

```
fun rreduce (f, z, r) = (case r
    of Leaf v => vreduce (f, z, v)
     | Cat (r1, r2) => let
         val f2 = future (fn _ => rreduce (f, z, r2))
         val v1 = rreduce (f, z, r1)
                  handle e => (cancel f2; raise e)
         val v2 = touch f2
         in
           f (b1, b2)
         end)
```

Figure 7. Reducing a function over a rope in parallel.

this is advantageous when applying the mapped function to an individual element is relatively cheap. Allowing the user to vary the maximum leaf size on a per-compilation basis gives some rough control over these tradeoffs. A more flexible system would allow the user to specify a maximum leaf size on a per-array basis, although many decisions remain about how to provide such a facility.

Another common operation on ropes is reduction by some associative operator $\oplus$. Compensation code to ensure that the sequentially first exception is raised is similar to that of `rmap`. A reduction of a parallel array is transformed to the internal function `rreduce`, whose definition is given in Figure 7.

7.5 Parallel Bindings

The implementation of parallel bindings introduces a future for each **pval** to be executed in parallel to the main thread of control and introduces cancellations when variables bound in the corresponding **pval** become unused on a control-flow path. Note that such control-flow paths may be implicit via a raised exception. As with parallel tuples, we do not introduce a future for a **pval** whose result is demanded by the main thread of control without any (significant) intervening computation.

To illustrate the implementation of parallel bindings, we present the translation of the function `trProd` from Figure 2.

```
fun trProdT (Lf i) = i
  | trProdT (Nd (tL, tR)) = let
      val fR = future (fn _ => trProdT tR)
      in (let
        val pL = trProdT tL
        in
          if (pL = 0) then (cancel fR; 0)
          else (pL * (touch fR))
        end)
        handle e => (cancel fR; raise e)
      end
```

Note that the translation introduces a future for only one of the **pval** computations, since `pL` is demanded immediately. The translation also inserts a cancellation of the future if an exception is raised that would exit the function body. As noted above, we expect exceptions to be rare, so we adopt a translation that may introduce redundant (but idempotent) cancellations (*i.e.*, canceling all introduced futures in a universal exception handler), rather than installing custom exception handlers to minimize the number of futures canceled.

Although the introduction of future cancellations is relatively straightforward, care must be taken to properly account for **pval** bound variables that are used in other **pval** computations. Consider the following example:

```
val r = let
  pval x = f 1
  pval y = g 2 + x
  in
    if h 3 then x else y
  end
```

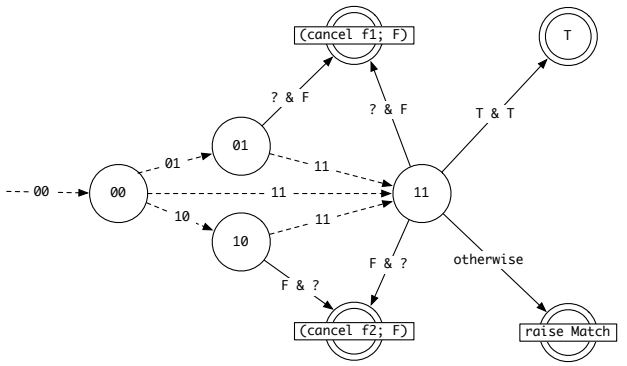

Figure 8. State machine for parallel conjunction. Completion transitions are dashed; match transitions are solid.

and its translation:

```
val rT = (let
  val fx = future (fn _ => f 1)
  val fy = future (fn _ => g 2 + touch fx)
  in
    if h 1
    then (cancel fy; touch fx)
    else ((*cancel fx;*) touch fy)
  end)
handle e => (cancel fy; cancel fx; raise e)
```

Note that touching `fx` twice, which might happen in certain executions of this code, is not problematic, since touches after the initial touch simply return the demanded value immediately.

We cannot include the commented `cancel fx`, as the result of `fx` might be demanded to satisfy the demand for the result of `fy`. In the **then** branch of the computation, `fx` is simply canceled as it should be. In the exception handler, `fy` is of necessity canceled before `fx`. Canceling in this order avoids the following problem: if `fx` were canceled, then touched in the computation of `fy` before `fy` were canceled, there would be an error.

7.6 Parallel Case

To implement each **pcase**, we construct a nondeterministic finite state machine. Recall the parallel conjunction from Section 4. The machine for the expression is shown in Figure 8. Each nonterminal state is labeled by a "completion bitstring" that represents which discriminant computations have been fully evaluated and which have not. Each terminal state corresponds to the body of a branch in the **pcase** expression. Naturally, the initial state is the non-terminal labeled `00`. There are two kinds of state transitions. For the first kind, there is a transition for each potential change to the completion bitstring; *i.e.*, the completed evaluation of additional discriminants. These "completion transitions" are shown in Figure 8 as dashed arrows. For the second kind, there is a transition for each branch in the **pcase** that can be tested given the discriminants that have been fully evaluated in the source state. These "match transitions" are shown in Figure 8 as solid arrows, labeled with the corresponding parallel patterns. When we follow a completion transition, we first test whether any of the match transitions can be taken. If so, one is taken (nondeterministically) and the appropriate branch body is evaluated to complete the evaluation of the **pcase**. If not, one of the completion transitions must be taken. Note that in the state labeled `11`, it is possible to match either of the branches `false & ?` or `? & false`; thus, there are match transitions for these branches. One detail not captured by Figure 8 is that the **otherwise** branch is only taken from state `11` if no

```
val f1 = future (fn () => ...)
val f2 = future (fn () => ...)

fun go() = (case (poll(f1), poll(f2))
 of (NONE, NONE) => state00()
  | (NONE, SOME t) => state01(t)
  | (SOME t, NONE) => state10(t)
  | (SOME t1, SOME t2) => state11(t1,t2))

and state00() = go()

and state01(t) = (case t
 of Val false => (cancel f2; false)
  | _ => go())

and state10(t) = (case t
 of Val false => (cancel f1; false)
  | _ => go())

and state11(t1,t2) = (case (t1,t2)
 of (Val false, _) => false
  | (_, Val false) => false
  | (Val true, Val true) => true
  | _ => raise Match)
```

Figure 9. An implementation of the state machine in Figure 8.

other match transitions are available (*i.e.*, the discriminants do not match any of the other branches).

The code in Figure 9 implements the state machine shown in Figure 8. Note the use of `poll` here induces some repeated tests and busy-waiting. In a mature implementation, there are various strategies one could employ to avoid this. For example, the future interface could be extended with a function

```
notify : 'a future * ('a trap -> unit) -> unit
```

that registers a function to be run upon completion of a future computation. Uses of `notify` and synchronizations would replace uses of `poll` in the expected ways.

8. Related work

Manticore's support for fine-grained parallelism is influenced by previous work on nested data-parallel languages, such as NESL [BCH+94, Ble96, BG96] and Nepal/DPH [CK00, CKLP01, LCK06]. Like Manticore, these languages have functional sequential cores and parallel arrays and comprehensions. To this mix, Manticore adds explicit parallelism, which neither NESL or DPH supports; neither does NESL or DPH have any analogs to our other mechanisms—parallel tuples, bindings, and cases. The NESL and DPH research has been directed largely at the topic of *flattening*, an internal compiler transformation which can yield great benefits in the processing of parallel arrays. Manticore is yet to implement flattening, although we expect to devote great attention to the topic as our work moves forward.

The Cilk programming language [BJK+95] is an extension of C with additional constructs for expressing parallelism. Cilk is an imperative language, and, as such, its semantics is different from Manticore's in some obvious ways. Some procedures in Cilk are modified with the **`cilk`** keyword; those are *Cilk procedures*. Cilk procedures call other Cilk procedures with the use of **`spawn`**. A spawned procedure starts running in parallel, and its parent procedure continues execution. In this way, spawned Cilk procedures are similar to Manticore expressions bound with `pval`. Cilk also includes a sophisticated abort mechanism for cancellation of spawned siblings; we have suggested some encodings of similar parallel patterns in Section 6 above.

Accelerator [TPO06] is an imperative data-parallel language that allows programmers to utilize GPUs for general-purpose computation. The operations available in Accelerator are similar to those provided by DPH's or Manticore's parallel arrays and comprehensions, except destructive update is a central mechanism. In keeping with the hardware for which it is targeted, Accelerator is directed towards regular, massively parallel operations on homogeneous collections of data, in marked contrast to the example presented in Section 6 above.

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 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-grained 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-grained parallel computation.

Programming parallel hardware effectively is difficult, but there have been a some important recent achievements. Google's MapReduce programming model [DG04] has been a success in processing large datasets in parallel. Sawzall, another Google project, is a system for analysis of large datasets distributed over disks or machines [PDGQ05]. (It is built on top of the aforementioned MapReduce system.) Brook for GPUs [BFH+04] is a C-like language which allows the programmer to use a GPU as a stream co-processor.

9. Conclusion

We have presented a design for a heterogeneous parallel functional language. In addition to explicitly-threaded CML-style language features not discussed in the present paper, we include proven implicitly-threaded features, such as parallel comprehensions, and novel ones, namely, parallel bindings and nondeterministic parallel cases. Since **`val`** bindings and **`case`** discrimination are essential idioms in a functional programmer's repertoire, providing implicitly-threaded forms allows parallelism to be expressed in a familiar style. We have furthermore demonstrated viable implementation strategies for the implicitly-threaded elements of our language.

We have been working on a prototype implementation of the Manticore language for the past eighteen months. Most parts of the implementation are working and we have been able to run examples of moderate size (*e.g.*, a parallel ray tracer translated from Id). The features described in this paper, however, do not have optimized implementations yet, which is why we omit performance measurements.

References

[ANP89] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: Data structures for parallel computing. *ACM TOPLAS*, **11**(4), October 1989, pp. 598–632.

[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.

[BAP95] Boehm, H.-J., R. Atkinson, and M. Plass. Ropes: an alternative to strings. *Software—Practice & Experience*, **25**(12), 1995, pp. 1315–1330.

[BAP+98] Barton, R., D. Adkins, H. Prokop, M. Frigo, C. Joerg, M. Renard, D. Dailey, and C. Leiserson. Cilk Pousse, 1998. Viewed on March 20, 2008 at 2:45 PM.

[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.

[BJK+95] Blumofe, R. D., C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.*, **30**(8), 1995, pp. 207–216.

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

[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.

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

[DL02] Dailey, D. and C. E. Leiserson. Using Cilk to write multiprocessor chess programs. *The Journal of the International Computer Chess Association*, 2002.

[FFR+07] Fluet, M., N. Ford, M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Status report: The Manticore project. In *ML '07*, New York, NY, October 2007. ACM, pp. 15–24.

[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.

[FRR08] Fluet, M., M. Rainey, and J. Reppy. A scheduling framework for general-purpose parallel languages. In *ICFP '08*, New York, NY, September 2008. ACM.

[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.

[GHC] GHC. The Glasgow Haskell Compiler. Available from `http://www.haskell.org/ghc`.

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

[Hal84] Halstead Jr., R. H. Implementation of multilisp: Lisp on a multiprocessor. In *LFP '84*, New York, NY, August 1984. ACM, pp. 9–17.

[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.

[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.

[HMPH05] Harris, T., S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. In *PPoPP '05*, New York, NY, June 2005. ACM, pp. 48–60.

[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.

[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.

[LP00] Leroy, X. and F. Pessaux. Type-based analysis of uncaught exceptions. *ACM Trans. Program. Lang. Syst.*, **22**(2), 2000, pp. 340–377.

[Luc90] Luckham, D. *Programming with Specifications*. Texts and Monographs in Computer Science. Springer-Verlag, 1990.

[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.

[Par72] Parnas, D. L. A technique for software module specification with examples. *CACM*, **15**(5), May 1972, pp. 330–336.

[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.

[PGF96] Peyton Jones, S., A. Gordon, and S. Finne. Concurrent Haskell. In *POPL '96*, New York, NY, January 1996. ACM, pp. 295–308.

[PRH+99] Peyton Jones, S., A. Reid, F. Henderson, T. Hoare, and S. Marlow. A semantics for imprecise exceptions. In *PLDI '99*, New York, NY, May 1999. ACM, pp. 25–36.

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

[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.

[RX08] Reppy, J. and Y. Xiao. Toward a parallel implementation of Concurrent ML. In *DAMP '08*, New York, NY, January 2008. ACM.

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

[THLP98] Trinder, P. W., K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + strategy = parallelism. *JFP*, **8**(1), January 1998, pp. 23–60.

[TPO06] Tarditi, D., S. Puri, and J. Oglesby. Accelerator: using data parallelism to program gpus for general-purpose uses. *SIGOPS Oper. Syst. Rev.*, **40**(5), 2006, pp. 325–335.

[Yi98] Yi, K. An abstract interpretation for estimating uncaught exceptions in standard ml programs. *Sci. Comput. Program.*, **31**(1), 1998, pp. 147–173.