Improving Implicit Parallelism

José Manuel Calderón Trilla Colin Runciman
University of York, York UK
jmct@jmct.cc

Abstract

Using static analysis techniques compilers for lazy functional languages can be used to identify parts of a program that can be legitimately evaluated in parallel and ensure that those expressions are executed concurrently with the main thread of execution. These techniques can produce improvements in the runtime performance of a program, but are limited by the static analyses' poor prediction of runtime performance. This paper outlines the development of a system that uses iterative profile-directed improvement *in addition to* well-studied static analysis techniques. This allows us to achieve higher performance gains than through static analysis alone.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers; D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming

General Terms Languages, Algorithms, Design, Performance

Keywords Implicit Parallelism, Lazy Functional Languages, Automatic Parallelism, Strictness Analysis, Projections, Iterative Compilation, Feedback Directed Compilation

1. Introduction

> I thought the "lazy functional languages are great for implicit parallelism" thing died out some time ago [1]
>
> Ben Lippmeier

Advocates of purely functional programming languages often cite easy parallelism as a major benefit of abandoning mutable state [2, 3]. This idea drove research into the theory and implementation of compilers that take advantage of *implicit parallelism* in a functional program. For lazy functional languages this can be seen to be at odds with the goal of only evaluating expressions when they are needed.

The ultimate goal of writing a program in a functional style, and having the compiler find the implicit parallelism, still requires work. We believe there are several reasons why previous work into implicit parallelism has not achieved the results that researchers have hoped for. Chief amongst those reasons is that the static placement of parallel annotations is not sufficient for creating well-performing parallel programs [4–7]. This paper explores one route to improvement: the compiler can use runtime profile data to improve initial decisions about parallelism in much the same way a programmer would manually tune a parallel program.

Haskell'15, September 3-4, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3808-0/15/09...$15.00
http://dx.doi.org/10.1145/2804302.2804308

```
gcd x y = if y == 0
          then x
          else if x > y
               then gcd (x - y) y
               else gcd x (y - x)

fromto x y = if x > y
             then []
             else x : fromto (x + 1) y

map f []     = []
map f (x:xs) = f x : map f xs

relPrime x y = gcd x y == 1

filter p []     = []
filter p (x:xs) = if p x
                  then x : filter p xs
                  else filter p xs


length []     = 0
length (x:xs) = 1 + (length xs)

sum []     = 0
sum (x:xs) = x + (sum xs)

euler n = let xs = fromto 1 n
          in length (filter (relPrime n) xs)

main = print (sum (map euler (fromto 1 1000)))
```

Figure 1: Source listing for `SumEuler`

Additionally, when research into implicit parallelism was more common, the work was often based on novel architectures or distributed systems, not commodity hardware [4, 8]. Research was unable to keep up with huge improvements in sequential hardware. Today most common desktop workstations are parallel machines; this steers our motivation away from the full utilisation of hardware. Many programmers today write sequential programs and run them on parallel machines. We argue that even modest speedups are worthwhile if they occur 'for free'.

Imagine that we have written the program in Figure 1. We might study its structure and decide to introduce some `par` annotations. When the program is compiled and executed, if we find that performance is still not satisfactory, we might study a profile (e.g. as provided by threadscope [9]), return to the source for the

program and adjust the placement of parallel annotations. This is the approach advocated by [9] and [10]. In contrast, many of the previous attempts at automating parallelism only analyse the program statically and do not adjust any parallel annotations after runtime data is gathered. This would be equivalent to a programmer never adjusting annotations after profiling the program.

By using runtime feedback, we can have the compiler *be generous* when introducing parallelism into the program. The profiling data will then point to the places where parallel evaluation underperforms and the compiler will *disable* the parallelism in these places.

We have designed and implemented an experimental compiler for implicit parallelism with profile-driven improvement. The source language of the compiler is F-Lite [11], a non-strict functional core language including ADTs, in addition to integer primitives, type checking, parametric polymorphism, pattern-matching, and higher-order functions.

1.1 Contributions

The contributions of our work are as follows:

- A fresh implementation of Hinze's projection-based strictness analysis [12] used to derive evaluation strategies
- A scheme for introducing parallelism into a program in conjunction with these strategies
- Simple rules for evaluating the effectiveness of parallelism
- A technique to disable `par` annotations
- Search strategies to improve initial `par` placements

This paper presents an overview of the design of our compiler and some of the design decisions that were made. It also presents the results of experiments using a range of small benchmark programs.

1.2 Roadmap

§2 presents a high-level overview of our approach to implicit parallelism. §3 explains the advantages of performing defunctionalisation on the source program. §4 motivates our use of a projection-based strictness analysis. §5 describes the correspondence between projections and strategies which allows us to generate parallel strategies based on the projections provided by the strictness analysis. §6 describes the initial placement of `par` annotations in the source program. §7 introduces the technique used for utilising the runtime profiling to disable some of the introduced parallelism along with possible additional search techniques. §8 presents some experimental results and discussion of those results. Lastly, §9 contains our conclusions and thoughts on possible future work.

2. Overview

In this section we present the overall picture of our technique. Much of the discussion will center around the code presented in Figure 2. In order to understand the code, it is useful to understand the architecture of the compiler.

2.1 Compiler Stages

The compiler is organised into 8 main phases, as follow:

1. Parsing
2. Defunctionalisation
3. Projection based Strictness Analysis
4. Generation of strategies
5. Placement of `par` annotations
6. G-Code Generation
7. Execution
8. Feedback and iteration

The parsing of the source language and sequential G-Code generation are done in the standard way and will not be discussed further. The rest of the paper will discuss the other phases.

2.2 A Program Before Iteration

The code listed in Figure 2 is the resulting core representation of the program in Figure 1 after our analysis and transformations. Specifically, the program has passed through compiler stages 1-5.

Before we dive into the program itself, note the following points:

- We only present the functions that have changed as a result of transformation
- We have replaced auto-generated names with easier to read names
- Functions ending in 'SN' are derived strategies
- Functions with an underscore in the name are the result of defunctionalisation

Taking a look at the program we can see several of the core ideas.

Defunctionalisation: The application of `map` to `euler` has been replaced by a call to the specialised `map_euler` function. We no longer have the functions `map` or `filter` instead we have specialised versions of these functions (e.g. `map_euler`)

Introduction of parallelism: Two calls to `par` have been introduced in the `main` function. Our work uses the traditional style for the parallel combinator [13]

```
par :: a -> b -> b
par x y = y
```

The first argument is *sparked* off to be evaluated in parallel and the function returns the second argument. This style is what allows us to easily *switch off* a particular `par` which causes that switched off `par` to act like `flip const`. This is explored further in §7.2.

Each application of `par` introduced by our compiler takes the following form: `par (s x) e` where `(s x)` is the application of derived strategy `s` to a variable `x` and `e` is an expression containing `x` as a free variable. In a *top-level definition* like `main`, `x` will be a name introduced by a `let` expression. For `par`s within the strategies themselves, `x` will be a name introduced by case analysis.

Demand Analysis and Strategies: The compiler has introduced a number of strategies into the program. These strategies are derived based on the results of a demand analysis. A simple example from the program is the transformed version of `length`. Because `+` (for non-lazy integers) requires both arguments to be fully evaluated, it is safe to evaluate the arguments to `+` in parallel to the execution of its body. In order to benefit from the parallel evaluation, the structure must be shared. The introduction of the name `len` accomplishes this. Because the type of `len` is `Int` evaluating the expression to WHNF is sufficient to evaluate the value fully.

Looking at the body of `euler` where `length` is called we see a similar pattern. In this case the demand analysis determines that it is safe the evaluate the *spine* of the list passed to `length`. The expression is given the name `ys` and the strategy `eulerListS1` is derived based on this information. Notice that the elements of the list passed to `eulerListS1` are ignored in its body.

There are cases where there is a strict demand on an expression but we do not introduce parallel evaluation of the expression. This can be seen in the first argument to `+` in the body of `length`. The rules for which subexpressions are considered *definitely* not worthwhile are discussed in §6.

```
main  =
   let eulerList =
       let xs = fromto 1 1000
       in par (fix mainListS1 xs) (map_euler xs)
   in par (fix mainListS2 eulerList) (sum eulerList);

rwhnf x = seq x Unit;

mainListS1 f xs = case xs of
    Cons y ys -> par (rwhnf y) (seq (f ys) Unit);
    Nil       -> Unit;

mainListS2 f xs = case xs of
    Cons y ys -> par (rwhnf y) (seq (f ys) Unit);
    Nil       -> Unit;

sum xs = case xs of
    Nil       -> 0;
    Cons y ys -> let rest = sum ys
                 in (par (rwhnf rest) ((y + rest)));

map_euler xs = case xs of
    Nil       -> Nil;
    Cons y ys -> Cons (euler y) (map_euler ys);

euler x = let ys = filter_relPrime x (fromto 1 x)
          in par (fix eulerListS1 ys) (length ys);

eulerListS1 f xs = case xs of
    Cons y ys -> seq (f ys) Unit;
    Nil       -> Unit;

length xs = case xs of
    Nil       -> 0;
    Cons y ys -> let len = length ys
                 in (par (rwhnf len) ((1 + len)));

filter_relPrime x y = case y of
    Nil       -> Nil;
    Cons z zs ->
        let b = relPrime x z
        in (par (filter_relPrimeS1 b)
                (if b
                 then Cons z (filter_relPrime x zs)
                 else filter_relPrime x zs));

filter_relPrimeS1 x = case x of
    True -> Unit;
    False -> Unit;

relPrime x y = let z = gcd x y
               in (par (rwhnf z) ((z == 1)));
```

Figure 2: Core representation of `SumEuler` after defunctionalisation, demand analysis, and the introduction of initial `par` sites along with their associated strategies.(Auto-generated names have been replaced for better readability)

Iterative Improvement: Just because an expression is *able* to be evaluated in parallel does not mean that doing so is beneficial! This is one of the critical problems in implicit parallelism [4, 5, 9]. To combat this we run the program as presented in Figure 2 and collect statistics about the amount of productive work each `par` is responsible for. The `par`s that do not introduce a worthwhile amount of parallelism (see discussion in §7) are disabled, freeing the program from incurring the overhead of managing threads for tasks with insufficient granularity [1].

Earlier we mentioned the `par`s in the bodies of `length` and `euler`. We picked these examples because while they are safe, they are not likely to be worthwhile. In the body of `length` the parallel evaluation of `len` only evaluates what + will immediately evaluate anyway. Giving us no benefit from parallelism. The parallel evaluation of `ys` in the body of `euler` also suffers from a similar issue.

So why introduce this parallelism? Because the granularity and possible interference of parallel threads is difficult to know *statically* at compile time. If we err on the side of generosity with our `par` annotations we can then use *runtime* profiling to gather information about the granularity and interference of threads.

As we would hope, our runtime system does determine that these two `par`s (and some others) are not worthwhile and disables them, improving performance.

Now that we have presented the high-level view of our work we can explore each of the stages in depth and discuss our reasons for certain design decisions.

3. Defunctionalisation

After parsing the next stage of the compiler applies a defunctionalising transformation to the input programs. Our defunctionalisation method is limited in scope, but sufficient for our purposes. It specialises higher-order functions defining separate instances for different functional arguments. We are careful to preserve sharing during this transformation. Here we give our motivation for introducing this transformation.

3.1 Why We Defunctionalise

Central to our design is the concept of `par` placement within a program. Each `par` application can be identified by its *position* in the AST. In a higher-order program basing our parallelism on the location of a `par` would very likely lead to undesirable consequences. For example, a common pattern in parallel programs is to introduce a parallel version of the `map` function

```
parMap :: (a -> b) -> [a] -> [b]
parMap f []     = []
parMap f (x:xs) = let y = f x
                  in y `par` y : parMap f xs
```

There is inevitably some overhead associated with evaluation of a `par` application, and of sparking off a fresh parallel thread. So if the computation `f x` is inexpensive, the parallelism may not provide any benefit and could even be detrimental. As `parMap` may be used throughout a program it is possible that there are both useful and detrimental parallel applications for various functional arguments: `parMap f` may provide useful parallelism while `parMap g` may cost more in overhead than we gain from any parallelism. Unfortunately when this occurs we are unable to switch off the `par` for `parMap g` without losing the useful parallelism of `parMap f`. This is because the `par` annotation is within the body of `parMap`. By specialising `parMap` we create two separate functions: `parMap_f` and `parMap_g`, with distinct `par` annotations in *each* of the instances of `parMap`.

[1] This can be seen as a more extreme variation of Clack and Peyton Jones' "Evaluate and die!" model of parallelism [14]: Evaluate *a lot* or die!

$$f\ e_1 \ldots e_{i-1}\ (g\ e'_1 \ldots e'_m)\ e_{i+1} \ldots e_{\#f} \qquad 0 \leq m < \#g$$
$$\Longrightarrow\ f_{\langle i,g,m \rangle}\ e_1 \ldots e_{i-1}\ e'_1 \ldots e'_m\ e_{i+1} \ldots e_{\#f} \tag{1}$$

$$f\ x_1 \ldots\ x_n\ = e$$
$$\Longrightarrow\ f_{\langle i,g,m \rangle}\ x_1 \ldots x_{i-1}\ y_1 \ldots y_m\ x_{i+1} \ldots x_n \tag{2}$$
$$= e[x_i/g\ y_1 \ldots y_m]$$

Figure 3: Rules for Defunctionalisation. $\#f$ and $\#g$ represent the arities of the functions. (1) refers to the transformation at the *call site*, (2) describes the transformation of the definition, creating a new version of f that has been specialised at its ith argument with function g and m arguments to g.

```
parMap_f []     = []
parMap_f (x:xs) = let y = f x
                  in y `par` y : parMap_f xs

parMap_g []     = []
parMap_g (x:xs) = let y = g x
                  in y `par` y : parMap_g xs
```

After defunctionalisation we can determine the usefulness of parallelism in each case independently. The plan is to deactivate the `par` for the inexpensive computation, `g x`, without affecting the parallel application of the worthwhile computation, `f x`.

3.2 How We Defunctionalise

Our defunctionaliser makes the following set of assumptions:

- Algebraic data structures are first-order (no functional components)
- The patterns on the left-hand side of a declaration have been compiled into case expressions
- Functions may have functional arguments but their definitions must be arity-saturated and return data-value results
- No explicit lambdas in the program, but partial applications are permitted

With these assumptions in mind, the rules for defunctionalisation are presented in Figure 3. These rules are applied to the AST in a *bottom up* fashion. This allows the transformation to assume that the arguments to partially applied functions (like e'_1 in (1)) have already been defunctionalised.

Example: Take `reverse` defined as an instance of `foldl`:

```
reverse xs  =  foldl (flip Cons) Nil xs
```

this becomes

```
reverse xs  =  foldl_flip_Cons Nil xs

foldl_flip_Cons z xs
    = case xs of
        Nil       -> z
        Cons y ys ->
            foldl_flip_Cons (flip_Cons z y) ys

flip_Cons xs x = Cons x xs
```

□

Another important benefit of applying defunctionalisation to the program is that it allows the use of Hinze's projection-based strictness analysis [12], which we discuss next.

4. Demand Analysis

In lazy languages evaluation should only occur when necessary. This apparently sensible rule can be at odds with the goals of performance through parallelism: if we have parallel processing resources, we wish to use them to do as much work as possible to shorten execution time [6].

Call-by-need semantics forces the compiler to take care in deciding which sub-expressions can safely be executed in parallel. Having a simple parallelisation heuristic such as 'compute all arguments to functions in parallel' can alter the semantics of a non-strict language, introducing non-termination or runtime errors that would not have occurred during a sequential execution.

The process of determining which arguments are required for a function is known as *strictness analysis* [15]. Since the early 1980's such analysis has been widely used for reducing the overheads of laziness [16]. In this section we provide a brief overview of the two predominant techniques for strictness analysis: *abstract interpretation* and *projection-based* analysis. We then motivate our decision to use a projection-based analysis.

4.1 Abstract Interpretation

Mycroft introduced the use of abstract interpretation for performing strictness analysis on call-by-need programs over thirty years ago [15]. Strictness analysis as originally described by Mycroft was only capable of dealing with a two-point domain (values that are definitely needed, and values that may or may not be needed). This works well for types that can be represented by a flat domain (Integer, Char, Bool, etc.)[2] but falls short on more complex data structures. For example, even if we find that a function is strict in a list argument, we can only evaluate up to the first `cons` safely. For many functions on lists, evaluating the entire list, or the spine of the list, is safe; canonical examples are `sum` and `length`.

In order to accommodate this type of reasoning, Wadler developed a *four-point domain* for the abstract interpretation of list-processing programs [17]. However, when extended in the natural way for general recursive data structures, the size of the domains made finding fix-points prohibitively costly.

4.2 Projections

This explosion in cost motivated Wadler and Hughes to propose using *projections* from domain theory to analyse strictness [18].

Projection-based analysis provides two benefits over abstract interpretation: the ability to analyse functions over arbitrary structures, and a correspondence with parallel strategies [13, 19]. This allows us to use the projections provided by our analysis to produce an appropriate function to compute the strict arguments in parallel.

Strictness analysis by abstract interpretation asks "When passing $\perp$ as an argument is the result of the function call $\perp$?". Projection-based strictness analysis instead asks "If there is a certain degree of demand on the result of this function, what degree of demand is there on its arguments?".

What is meant by 'demand'? As an example, the function `length` requires that the input list be finite, but no more. We can therefore say that `length` *demands* the spine of the argument list. The function `append` is a more interesting example:

```
append :: [a] -> [a] -> [a]
append []     ys = ys
append (x:xs) ys = x : append xs ys
```

By studying the function we can tell that the first argument must be defined to the first cons, but we cannot know whether the second argument is ever needed.

[2] Any type that can be represented as an enumerated type.

However, what if the *result* of `append` needs to be a finite list? For example:

```
lengthOfBoth :: [a] -> [a] -> Int
lengthOfBoth xs ys = length (append xs ys)
```

In this case *both* arguments to `append` must be finite. Projections can be used to formalise this type of context [12, 18].

4.2.1 Semantics of Projections

Given a domain D, a projection on D is a continuous function $\pi : D \to D$ that satisfies

$$\pi \sqsubseteq ID \tag{3}$$
$$\pi \circ \pi = \pi \tag{4}$$

Equation (3) ensures that a projection can not add any information to a value, i.e. all projections approximate the identity function. Idempotence (4) ensures that projecting the same demand twice on a value has no additional effect. This aligns with our intuition of demand. If we demand that a list is spine-strict, demanding spine-strictness again does not change the demand on the list.

Because we want the introduction of parallelism to be semantics-preserving we use the following safety condition for projections:

$$\gamma \circ f = \gamma \circ f \circ \pi \tag{5}$$

Given a function $f : X \to Y$, and demand γ on the *result* of f, projection-based analysis propagates the demand given by γ to the arguments of f. This results in the demand on the *arguments* of f given by π. The analysis aims to find the *smallest* π for each γ, but approximating towards ID (as it is always safe to project the identity).

Demands on Primitives: On unlifted base types, such as unboxed integers, there are two demands, ID and BOT, with the following semantics

$$ID\, x = x \tag{6}$$
$$BOT\, x = \bot \tag{7}$$

When an expression is in a BOT context it means that non-termination is inevitable. You can safely evaluate an expression in this context because there is no danger of *introducing* non-termination that is not already present.

Demands on Lifted Types: Haskell's non-strict semantics means that most types we encounter are *lifted* types. Lifted types represent possibly unevaluated values. Given a demand π on D, we can form two possible demands on $D_\perp$, $\pi!$ and $\pi?$; strict lift and lazy lift respectively. To paraphrase Kubiak et al.: $\pi!$ means we will definitely need the value demanded by this projection, and we will need π's worth of it [20]. $\pi?$ does not tell us whether we need the value or not, but if we *do* need the value, we will need it to satisfy π's demand.

Demands on Products: A projection representing a demand on a product can be formed by using the $\otimes$ operator with the following semantics

$$\langle \pi_1 \otimes \cdots \otimes \pi_n \rangle \perp = \perp$$
$$\langle \pi_1 \otimes \cdots \otimes \pi_n \rangle \langle x_1, \ldots, x_n \rangle = \langle \pi_1 x_1, \ldots, \pi_n x_n \rangle$$

Demands on Sums: If projections are functions on a domain, then $\oplus$, the operator that forms projections on sum-types performs the case-analysis.

$$\begin{array}{lr}
d ::= BOT & \text{Bottom (hyperstrict)} \\
\mid ID & \text{Top (the identity)} \\
\mid \langle d_1 \otimes d_2 \cdots \otimes d_n \rangle & \text{Products} \\
\mid [d_1 \oplus d_2 \cdots \oplus d_n] & \text{Sums} \\
\mid \mu\beta.d & \text{Recursive Demands} \\
\mid d? & \text{Strict Lift} \\
\mid d! & \text{Lazy Lift}
\end{array}$$

Figure 4: Abstract Syntax for Contexts of Demand

$$[ID_{True} \oplus ID_{False}]\, True = True$$
$$[ID_{True} \oplus BOT_{False}]\, False = \bot$$

Figure 4 presents a suitable abstract syntax for projections representing demand. This form was introduced by Kubiak et al. and used in Hinze's work on projection-based analyses [12, 20]. We have omitted the details on the representation of context variables (for polymorphic demands), for a comprehensive discussion we suggest Chapter 6 of Hinze's dissertation [12].

In short, projections representing demand give us information about how defined a value must be to satisfy a function's demand on that value. Knowing that a value is definitely needed, and to what degree, allows us to evaluate the value before entering the function.

4.2.2 Example Projections

Because our primitives can be modelled by a flat domain (just ID and BOT), our lattice of projections corresponds with the two-point domain used in abstract interpretation. □

For pairs of primitive values, possible contexts include:

$$[\langle ID? \otimes ID? \rangle] \tag{8}$$
$$[\langle ID! \otimes ID? \rangle] \tag{9}$$

As Haskell's types are sums of products, pairs are treated as sums with only one constructor. For product types each member of the product is lifted. Context 8 is the top of the lattice for pairs, accepting all possible pairs. Context 9 requires that the first member be defined but does not require the second element. This is the demand that `fst` places on its argument. □

For polymorphic lists there are 7 principal contexts; 3 commonly occurring contexts are:

$$\mu\beta.[ID \oplus \langle \gamma? \otimes \beta? \rangle] \tag{10}$$
$$\mu\beta.[ID \oplus \langle \gamma? \otimes \beta! \rangle] \tag{11}$$
$$\mu\beta.[ID \oplus \langle \gamma! \otimes \beta! \rangle] \tag{12}$$

Here μ binds the name for the 'recursive call' of the projection and γ is used to represent an appropriate demand for the element type of the list. An important point is that this representation for recursive contexts restricts the representable contexts to *uniform projections*: projections that define the same degree of evaluation on each of their recursive components as they do on the structure as a whole. The detailed reason for this restriction is given on page 89 of Hinze [12]. This limitation does not hinder the analysis significantly as many functions on recursive structures are themselves uniform.

With this in mind Context 10 represents a lazy demand on the list, Context 11 represents a *tail strict* demand, and Context 12 represents a *head and tail* strict demand on the list. □

It will be useful to have abbreviation for a few of the contexts on lists. These abbreviation are presented in Figure 5.

ID: accepts all lists

T (tail strict): accepts all finite lists

H (head strict): accepts lists where the head is defined

HT: accepts finite lists where every member is defined

Figure 5: Four contexts on lists as described in [18].

We can now say more about the strictness properties of `append`. The strictness properties of a function are presented as a *context transformer* [12].

$$\begin{array}{lcl} append(ID) & \rightarrow & ID!; ID? \\ append(T) & \rightarrow & T!; T! \\ append(H) & \rightarrow & H!; H? \\ append(HT) & \rightarrow & HT!; HT! \end{array}$$

This can be read as "If the demand on the result of `append` is ID then the first argument is strict with the demand ID and the second argument is lazy, but if it *is* needed, it is with demand ID. □

Following Hinze [12] we construct projections for every user-defined type. Each projection represents a specific strategy for evaluating the structure, as we shall define in section 5. This provides us with the ability to generate appropriate parallel strategies for arbitrary types. Using a projection-based strictness analysis, we avoid the exponential blowup of domains required for abstract interpretation.

5. Deriving Strategies from Projections

One of the reasons that projections were chosen for our strictness analysis is their correspondence to parallel strategies. Strategies are functions whose sole purpose is to force the evaluation of specific parts of their arguments [13, 19]. All strategies return the unit value `()`. Strategies are not used for their computed result but for the evaluation they force along the way.

5.1 Some Examples

The type for strategies is defined as `type Strategy a = a -> ()`.

The simplest strategy, named `r0` in the original paper [19], which performs no reductions is defined as `r0 x = ()`. The strategy for weak head normal form is only slightly more involved: `rwhnf x = x ‘seq‘ ()`

□

The real power comes when strategies are used on data-structures. Take lists for example. Evaluating a list sequentially or in parallel provides us with the following two strategies

```
seqList s []     = ()
seqList s (x:xs) = s x ‘seq‘ (seqList s xs)

parList s []     = ()
parList s (x:xs) = s x ‘par‘ (parList s xs)
```

Each strategy takes another strategy as an argument. The provided strategy is what determines how much of each element to evaluate. If the provided strategy is `r0` the end result would be that only the spine of the list is evaluated. On the other end of the spectrum, providing a strategy that evaluates a value of the list-item type `a` fully would result in list's spine *and* elements being evaluated.

□

$$\begin{array}{l} \mathcal{C} :: [\![Context]\!] \rightarrow Names \rightarrow Exp \\ \mathcal{C}\,[\![c?]\!]\,\phi \quad = \lambda x \rightarrow () \\ \mathcal{C}\,[\![c!]\!]\,\phi \quad = [\![c]\!]\,\phi \\ \mathcal{C}\,[\![\mu\beta.c]\!]\,\phi \quad = fix\ (\lambda n \rightarrow [\![c]\!]\,(n : \phi)) \\ \mathcal{C}\,[\![\beta]\!]\,(n : \phi) \quad = n \\ \mathcal{C}\,[\![[cs]]\!]\,\phi \quad = \lambda x \rightarrow Case\ x\ of\ \mathcal{A}\,[\![cs]\!]\,\phi \\ \mathcal{C}\,[\![c]\!]\,\phi \quad = \lambda x \rightarrow x\ `seq`\ () \end{array}$$

$$\begin{array}{l} \mathcal{A} :: [\![(Constructor, Context)]\!] \rightarrow Names \rightarrow (Pat, Exp) \\ \mathcal{A}\,[\![(C_n, ID)]\!]\,\phi \quad = (C_n, ()) \\ \mathcal{A}\,[\![(C_n, BOT)]\!]\,\phi \quad = (C_n, ()) \\ \mathcal{A}\,[\![(C_n, \langle cs \rangle)]\!]\,\phi \quad = (C_n\ vs, \mathcal{F}\,[\![ss]\!]\,\phi) \\ \quad where\ ss = filter\ (isStrict \circ fst)\ \$\ zip\ cs\ vs \\ \quad\quad vs = take\ (length\ cs)\ freshVars \end{array}$$

$$\begin{array}{l} \mathcal{F} :: [\![[(Context, Exp)]]\!] \rightarrow Names \rightarrow Exp \\ \mathcal{F}\,[\![[]]\!]\,\phi \quad = () \\ \mathcal{F}\,[\![((c, v) : [])]\!]\,\phi \quad = App\ (Fun\ \text{“}seq\text{”})\ [App\ (\mathcal{C}\,[\![c]\!]\,\phi)\ [v], ()] \\ \mathcal{F}\,[\![((c, v) : cs)]\!]\,\phi \quad = App\ (Fun\ \text{“}par\text{”})\ [App\ (\mathcal{C}\,[\![c]\!]\,\phi)\ [v], ls] \\ \quad where\ ls = \mathcal{F}\,[\![cs]\!]\,\phi \end{array}$$

Figure 6: Rules to generate strategies from demand contexts

Already we can see a correspondence between these strategies and the contexts shown in Figure 5. The T context (tail strict) corresponds to the strategy that only evaluates the spine of the list, while the HT context corresponds to the strategy that evaluates the spine and all the elements of a list.

Recalling the program in Figure 2 the generated function `mainListS1` corresponds to a strategy for evaluating a list in a HT context. In our derived strategies it is not necessary to pass additional strategies as arguments because the demand on a structure's elements makes up part of the context that describes the demand on that structure.

5.2 Derivation Rules

Because projections *already* represent functions on our value domain, translating a projection into a usable strategy only requires that we express the projection's denotation in a programmed definition. The rules we use are shown in Figure 6. Rule $\mathcal{C}$ constructs a strategy for all of the context constructors except for products. This is because product types are only found within constructors in the source language and are therefore wrapped in sums as constructor-tag context pairs. These pairs are handled by the $\mathcal{A}$ rule.

One aspect of strategies that does *not* directly correspond to a context is the choice between `seq` and `par`. Every context can be fully described by both sequential and parallel strategies. When a constructor has two or more fields, it can be beneficial to evaluate *some* of the fields in parallel. It is not clear, generally, which fields should be evaluated in parallel and which should be evaluated in sequence. As shown in rule $\mathcal{F}$ we evaluate all fields in parallel *except* for the last field in a structure. This means that if a structure has only one field then its field will be evaluated using `seq`.

5.3 Specialising on Demand

The key reason for performing a strictness analysis in our work is to know when it is *safe* to perform work before it is needed. This work can then be sparked off and performed in parallel to the main thread of execution. Using the projection-based analysis allows us to know not only *which* arguments are needed, but *how much* (structurally) of each argument is needed. We convert the projections into strategies and then spark off those strategies in parallel.

Assume that our analysis determines that function `f` is strict in both of its arguments. This allows us to convert

```
f e1 e2
```

into

```
let a = e1
    b = e2
in (s1 a) `par` (s2 b) `seq` (f a b)
```

where `s1` and `s2` are the strategies generated from the projections on those expressions.

5.3.1 Different Demands on the Calling Function

If a function has different demands on its result at different calling sites, that is dealt with 'for free' using the transformation above. However, there may be multiple *possible* demands *at the same call site*.

This can happen when there are different demands on the calling function, for example:

```
func x = f e1 e2
```

Different demands on the result of `func` may mean different demands on the result of `f`. This in turn means that different transformations would be appropriate. Assume this results in having two different demands on `f`. One demand results in the first transformation (`funcD1`) and the other results in the second (`funcD2`). How do we reconcile this possible difference?

5.3.2 Specialisation by Demand

To accommodate this possibility we can clone the function `func`. One clone for each demand allows us to have the 'more parallel' version when it is safe, and keep the 'less parallel' version in the appropriate cases. Note, we do not have to clone all functions with versions for every possible demand. Instead we can do the following for each function:

1. Determine which demands are actually present in the program
2. In the body of the function, do the different demands result in differing demands for a specific function call?
3. If no, no cloning
4. If yes, clone the function for each demand and re-write the call-sites to call the appropriate clone

Applying the above procedure to our hypothetical expression would result in the following

```
funcD1 x = let a = e1
               b = e2
           in s1 a `par` s2 b `seq` f a b

funcD2 x = let a = e1
           in s1 a `par` f a e2
```

6. Introducing `pars`

After our strictness analysis and demand specialisation are complete we are ready to introduce parallelism into our program. The approach taken is to apply the generated strategies to *the strict* arguments of a function.

Example: Take the famous `fib`

```
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = fib (n - 2) + fib (n - 1)
```

The results of the strictness analysis show us that both arguments to + have the same demand: ID!. We therefore evaluate the recursive calls to `fib` in parallel:

```
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = let a = fib (n - 2)
            b = fib (n - 1)
        in (s1 a) `par` (s2 b) `seq` a + b
```

There are two points to consider in the transformed program. One is that by lifting subexpressions into `let` bindings we preclude the possibility of certain compiler optimisations. The sharing of values is essential for parallel strategies to be beneficial. In particular, thunk elimination becomes more difficult.

The other point is that we utilise the common technique of combining `pars` and `seqs` in order to prevent collisions between threads. This is not always possible and as we will see in §9 can be detrimental.

□

6.1 Granularity

It is possible to rely solely on the results of the strictness analysis to determine which sub-expressions should be evaluated in parallel. However, an expression being needed does not necessarily mean that evaluating that expression will be worthwhile. This is known as the *granularity* problem [4]. We use a simple oracle to determine whether a subexpression should be evaluated in parallel. Recall that our oracle should be generous in 'allowing' subexpressions to be evaluated in parallel. Our iterative improvement *reduces* the amount of parallelism introduced by static analysis. As the oracle's only job is to determine whether a subexpression is 'worth' the overhead of parallel evaluation it has the type `type Oracle = Exp -> Bool`. The two trivial oracles are

```
allYes :: Oracle
allYes = const True

allNo :: Oracle
allNo = const False
```

`allNo` clearly defeats the purpose of an auto-parallelising compiler, but `allYes` can serve as a stress-test for the iterative process. The oracle used in our results returns `True` if the expression contains a non-primitive function call, `False` otherwise.

```
mediumOracle e = or $ map f (universe e)
  where
    f (App (Fun n) as)
        | n `elem` prims = False
        | otherwise      = True
    f _ = False
```

Here, `universe` takes an expression e and provides a list of all the valid subexpressions of e, reaching the leaf nodes of the AST.

The transformation we apply is simple, for each function application $f\ e_1 \ldots e_n$:

1. Gather all the strict argument expressions to a function
2. Pass each expression to the oracle
3. Give a name (via let-binding) to each of the oracle-approved expressions
4. Before calling f spark the application of the derived strategy to the appropriate binding
5. If there are multiple arguments that are oracle approved, ensure that the last argument has its strategy applied with `seq`

We now have the necessary mechanisms in place for the *introduction* of parallelism into a program.

7. Iterative Compilation

In this section we will look at the techniques we use to *disable* some of the parallelism that has been introduced into our programs. This involves recording a significant amount of runtime information and a method for safely switching off the `par` annotations.

Our runtime system is designed in the tradition of projects like GranSim [21]. The goal is to have as much control of the execution substrate as possible. This allows us to investigate certain trade-offs while ensuring that we minimise confounding variables.

7.1 Logging

The runtime system maintains records of the following global statistics:

- Number of reduction cycles
- Number of sparks
- Number of blocked threads
- Number of active threads

These statistics are useful when measuring the overall performance of a parallel program, but tell us very little about the usefulness of the threads themselves.

In order to ensure that the iterative feedback system is able to determine the overall 'health' of a thread, it is important that we collect some statistics pertaining to each individual thread. We record the following metrics for each thread:

- Number of reduction cycles
- Number of sparks generated
- Number of threads blocked by this one
- Which threads have blocked the current thread

This allows us to reason about the productivity of the threads themselves. An ideal thread will perform many reductions, block very few other threads, and be blocked rarely. A 'bad' thread will perform few reductions and be blocked for long periods of time.

7.2 Switchable `pars`

In order to take advantage of runtime profiles we must be able to adapt the compilation based on any new information. One choice is to recompile the program completely and create an oracle that uses the profiles. This way the oracle can better decide which subexpressions to parallelise. Our approach is to modify the runtime system so that it is able to *disable* individual `par` annotations. When a specific `par` in the source program is deactivated it no longer creates any parallel tasks while still maintaining the semantics of the program. The method has two basic steps:

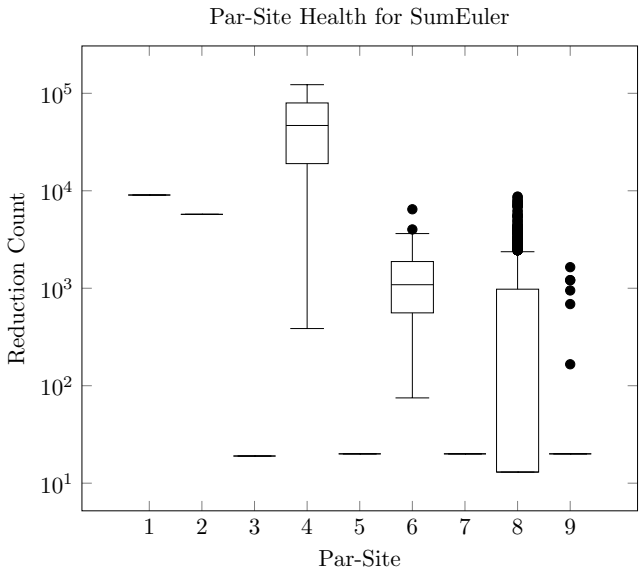


Figure 7: Statistics on the number of reductions carried out by the threads a `par` site sparks off

- `par`'s are identified via the instruction `PushGlobal "par"` and each par is given a unique identifier.
- When a thread creates the heap object representing the call to `par` the runtime system looks up the status of the `par` using its unique identifier. If the `par` is 'on' execution follows as normal. If the `par` is 'off' the thread will ignore the G-Code instruction `Par`.

7.3 Iteration

While we do record a variety of statistics, our current approach focuses on reduction count as a guide to determine which `par`-sites are beneficial to the program. The reasoning is simple[3], our motivation for parallelism is to do more work at once, so measuring the amount of work undertaken by each thread might be a good metric.

Because we record how productive each thread in a program is and we keep track of which `par` site created each thread, we can easily visualise how useful each `par` site is. Figure 7 gives an overview of the health of each `par`-site. For each site we record the total number of reductions carried out. The plot shows us the statistics for this data with the median (line), inter-quartile range (IQR, box), and $\pm 1.5 * \text{IQR}$ (whiskers). Statistical outliers are shown as independent points. The `par`sites that only show a line as their plot either have only one child thread (the case for `par`-site 1) or have little variance in the distribution of reduction counts.

After every execution of the program, turn off the `par` site whose threads have the lowest average reduction count. In the case of the execution statistics displayed in Figure 7 we would disable `par` site 2, allowing us to avoid the overhead of all the unproductive threads it sparked off. Then repeat this process until switching a `par` site off increases the overall runtime of the program.

[3] As we will see in §9, too simple.

8. Experimental Results

In this section we present some preliminary results and point out certain patterns that appear in our data.

First we introduce our benchmark programs and state the number of `par` sites that were introduced statically:

SumEuler: `SumEuler` is a common parallel functional programming benchmark first introduced with the work on the $\langle \nu, G \rangle$-Machine in 1989 [22]. The program computes the sum of mapping the euler-totient function of a list. As this functional idiom is known to be highly parallelisable; this benchmark can be seen as a 'sanity-check' on our technique (9 `par` sites).

Queens + Queens2: We benchmark two versions of the `nQueens` program. Both versions use a backtracking algorithm to search for possible solutions. `Queens2` is a purely symbolic version that represents the board as a list of lists (10 `par` sites for `Queens` and 24 for `Queens2`).

SodaCount: Solves a word search problem for a given grid of letters and a list of keywords (15 `par` sites).

Tak: Small recursive numeric computation that calculates a Takeuchi number (2 `par` sites).

Taut: Determines whether a given predicate expression is a tautology. Attempts all assignments of Boolean values to the variables in the given expression (15 `par` sites).

MatMul: List of list matrix multiplication (7 `par` sites).

8.1 Overheads

Whether an expression is worthwhile to evaluate in parallel is directly tied to *cost* of creating a parallel task. In order to account for this we ran all of our experiments with the simulated overhead cost at 3 settings. We chose the upper and lower bounds based on benchmarking parallel functions using the Criterion library [23].

For each program we set our runtime system to simulate 4, 8, and 16 cores. First, let us examine Figure 1 which displays the results of setting the cost of task creation to 10 reductions.

Table 1: Speedups relative to sequential computation when the cost of sparking a task is set to 10 reductions. The number of runs corresponds to the number of `par` sites that have been switched off.

Program	4-core		8-cores		16-cores	
	Runs	Final	Runs	Final	Runs	Final
SumEuler	6	3.77	6	6.84	6	10.27
Queens	5	1.30	5	1.37	5	1.41
Queens2	22	3.91	22	7.74	22	15.07
SodaCount	3	2.42	3	4.72	3	8.95
Tak	1	3.39	1	6.79	1	13.58
Taut	4	1.00	0	1.00	9	1.00
MatMul	2	1.02	2	1.07	2	1.10

Already there are a few interesting results. `SumEuler` performs as expected and manages to eliminate the majority of the introduced `par` sites. Recalling Figure 2, the `par`s that survive the iterative improvement are the two in the `main` function and the `par` in `mainListS2`. The second `par` in `main` and the strategy `mainListS2` are, taken together, equivalent to applying `parMap euler` over the input list. When this program is parallelised explicitly, that `parMap` is usually the only addition to the program [22]. It is reassuring that our technique converges on the same result.

The two implementations of `nQueens` vary drastically in their improvement, with the more symbolic solution (`Queens2`) achieving much better results. Search problems are known to be problematic for techniques involving strictness analysis and usually benefit from the introduction of *speculative* parallelism [4].

`Taut` was chosen as a benchmark program specifically because the program (as written) did not have many opportunities for parallelism. Had our technique managed to find any useful parallelism, we would have been surprised.

`MatMul` is, to us, the most surprising of the results so far. Matrix multiplication is famously parallelisable and yet our implementation barely breaks even! Notice that of the 7 `par` sites in `MatMul`, only 2 are being switched off. The `par` setting that the iterative improvement converged on was not the optimal setting (we know there is at least 2 superior settings). This convergence on local maxima is something we will discuss in §9.

While the results in Figure 1 are revealing, it could be argued that an overhead of 10 reductions to spark off a thread is unrealistically low. Therefore we repeat the experiments with the more realistic 100 reduction overhead (Figure 2) and the pessimistic case of 1000 reduction overheads (Figure 3).

Table 2: Speedups relative to sequential computation when the cost of sparking a task is set to 100 reductions. The number of runs corresponds to the number of `par` sites that have been switched off.

Program	4-core		8-cores		16-cores	
	Runs	Final	Runs	Final	Runs	Final
SumEuler	6	3.74	6	6.81	6	10.23
Queens	5	1.29	5	1.37	5	1.41
Queens2	22	3.83	22	7.57	22	14.76
SodaCount	3	2.17	3	4.23	3	8.02
Tak	1	2.36	1	4.71	1	9.42
Taut	9	1.00	0	1.00	9	1.00
MatMul	2	0.93	2	1.06	2	1.09

The results in Figure 2 mostly align with what we would expect to happen if creating a parallel task incurred higher overheads: we see reduced speedup factors and adding more cores is less likely to benefit.

The key point to take away from this set of results is that while lower speedups are achieved, the *same* `par` sites are eliminated in the same number of iterations [4].

Now we try the same experiment again but with the less realistic 1000 reduction overhead to create a new thread.

Table 3: Speedups relative to sequential computation when the cost of sparking a task is set to 1000 reductions. The number of runs corresponds to the number of `par` sites that have been switched off.

Program	4-core		8-cores		16-cores	
	Runs	Final	Runs	Final	Runs	Final
SumEuler	6	3.51	6	6.40	6	9.73
Queens	5	1.26	5	1.35	5	1.40
Queens2	22	3.14	22	6.22	22	12.18
SodaCount	12	1.85	3	2.08	1	1.39
Tak	1	0.57	1	1.15	1	2.32
Taut	12	1.00	12	1.00	7	1.00
MatMul	5	1.00	5	1.00	5	1.01

While the speedups are now much more moderate (when there is a speedup at all) these results are interesting for a few reasons.

In particular, the number of cores now has a greater influence on how many `par` sites are worthwhile. `SodaCount`, for instance,

[4] Except for `Taut`, which in the 4-core case now takes 9 runs to determine that there is no parallelism in the program.

now eliminates 12 of its 15 `par` annotations in the case of 4-core execution. This fits with our intuition that when there are fewer processing units the threads require coarser granularity to be worthwhile. In the cases of 8 and 16-core executions we observe that fewer `par` sites are disabled, reinforcing this intuition.

`MatMul` also sees a jump in the number of disabled `par` sites. Sadly, this results in even worse performance for `MatMul`, which should be a highly parallelisable program.

8.2 Static vs. Iterative

While the results presented in Figures 1, 2, and 3 are promising for preliminary results they are based on an admittedly simple search heuristic. Part of our argument is that static analysis *alone* is not sufficient for good gains from implicit parallelism. Figure 8 presents a selection of results that show how the iterative improvement affects the static placement of `par` annotations.

Even in the cases where the final speedup is lower than anticipated, such as `Queens` in Figure 8b, the program still benefits from the iterative improvement. `Queens2` sees the highest payoff from iterative improvement. Many of the `par`s introduced by the static analysis do not contribute significantly to the computation even though it is semantically safe to introduce them. The iterative loop converges on the few `par` sites that make a significant difference.

8.3 Comparison to GHC

While the results above are encouraging we would like to see how the resulting programs perform when compiled by a modern high-performance Haskell compiler. To do this we extract the final `par` settings from each program and translate that to Haskell suitable for compilation by GHC. For the versions parallelised by hand we use the `par` placements found in the literature [10, 22].

Table 4: Speedups compared to the sequential program as compiled by GHC for both manually and automatically parallelised versions

Program	4-core	
	Hand	Auto
SumEuler	3.32	3.31
Queens	1.76	0.97
Queens2	2.29	0.61
SodaCount	1.25	0.64
Tak	1.77	1.64
MatMul	1.75	0.80

As Table 4 makes clear, the results are not impressive. In fact, except for `SumEuler` and `Tak`, all of the parallel benchmarks performed *worse* than their sequential counterparts.

While some of the results in Table 4 are disappointing, it is important to remember that if the iterative compiler was hosted in GHC itself, we would never see speedups below a factor of 1. This is because in the worst case the compiler can switch off all of the parallelism that was introduced into the program (or even discard the modified program and use the original input program). Additionally, hosting the iterative process on GHC itself would provide realistic overhead costs, allowing the compiler to eliminate `par`s more accurately.

However, we feel that not all hope is lost. There are a few recurring issues in the generated program. A common issue is that the generated strategies will not be what forces the evaluation of a value. Take the following example as an illustration

```
foo n = let ys = gen n n
        in par (tailStrict1 ys) (bar ys)
```

In the function `foo` we spark off a strategy that is meant to force the spine of the list `ys`, the catch is that GHC's `par` is fast enough for `bar ys` to be what forces the evaluation of `ys`. So we're paying the overhead and reaping none of the benefits. In some programs changing a `par` like the one found in `foo` to a `seq` is enough to solve the issue and make the parallel version competitive with the manually parallelised version.

9. Conclusions

We hope we have motivated the key design choices and ideas behind our compiler: utilising defunctionalisation in §3, and the use of projections over other strictness analysis methods in §4. Moreover, that we have shown that there is a natural correspondence between projections and strategies §5 that allows us to generate parallel strategies from the results of our strictness analysis.

While projection-based strictness analysis does provide a useful foundation for the introduction of parallelism, the results in §8 show that static analysis alone does not provide the desired speedups. Additionally, Table 4 shows that determining `par`-site health by reduction count alone is too naive. Thread collisions that may not happen on a simulated system are able to significantly hamper parallel performance and should be taken into account. The results suggest that better performance gains would be attained by hosting the iterative framework on GHC itself.

9.1 Related Work

There is a wealth of prior work on parallelism, both implicit and explicit, for functional languages. Here we provide a quick discussion of the work that is most closely related to our own.

9.1.1 Work on Demand Analysis

Much of the early work on strictness analysis as a means to achieve implicit parallelism focused on the abstract interpretation approach because the work on projections had not been fully developed when implicit parallelism was a more active research area. In particular, the work on the "Automatic Parallelization of Lazy Functional Programs" [5] only used two and four-point domains (as described in [17]) in their strictness analysis. This limits the ability of the compiler to determine the neededness of more complex structures.

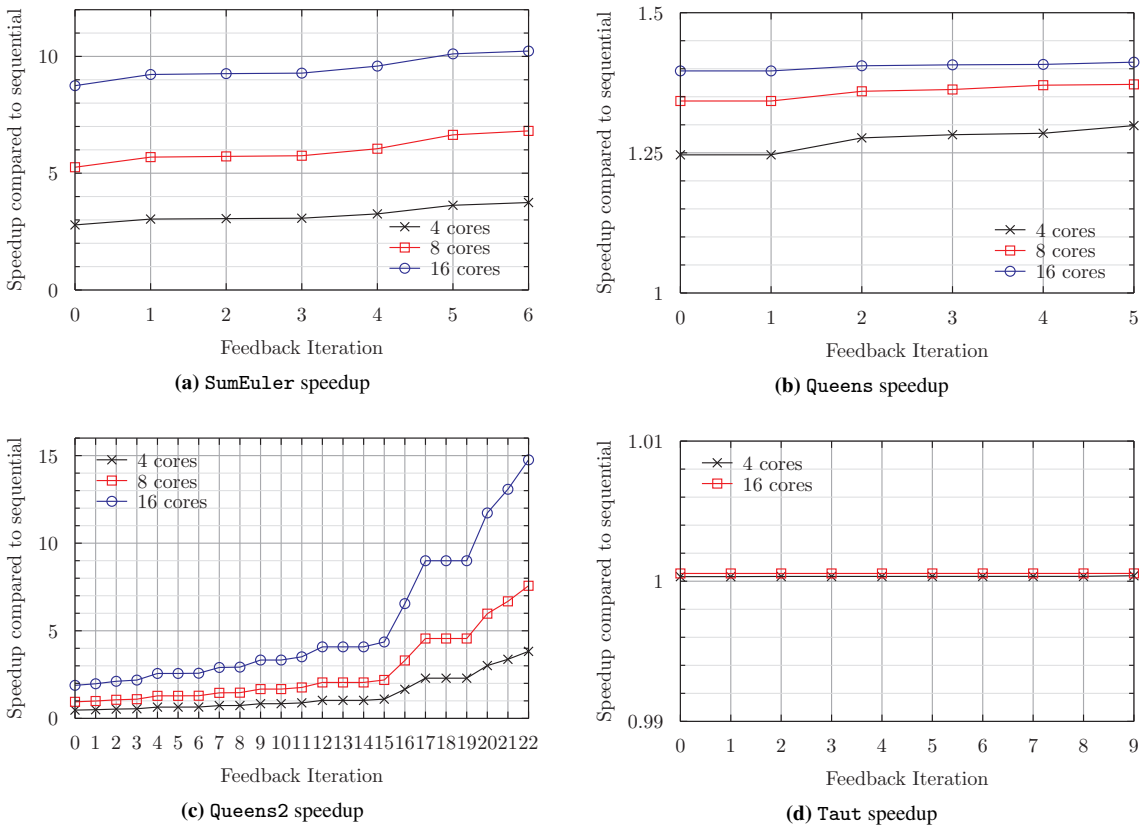
More recent work on demand analysis for GHC is able to analyse higher-order functions, but is restricted to single-constructor types [24]. This is sufficient for the eliminating unnecessary allocations, but does not provide enough information for the derivation of parallel strategies.

Hinze's work on projection-based strictness analysis came after work on implicit parallelism fell out of favour [4, 12]. To our knowledge we are the first to apply this work to the implicit parallelism.

9.1.2 Blind Search

By representing a program's `par` settings as a bit string we can experiment with standard search techniques. Many algorithms can optimise a function using only a fitness function that takes a bitstring as an input. In our case, the compiler would set the `par` switches according to the bitstring and use the wall-clock time as a fitness function. We explored two such algorithms in an earlier paper showing that even simple search algorithms, such as hill-climbing, can achieve promising results [25]. The downside to these techniques is that they require a large number of iterations to converge. The technique presented here allows us to get similar results in linear (in the length of the bitstring) time.

As we attempt to scale our technique to larger programs it is likely that blind search technique will run into the combinatorial explosion inherent in bitstring-based search techniques.



(a) `SumEuler` speedup

(b) `Queens` speedup

(c) `Queens2` speedup

(d) `Taut` speedup

Figure 8: The improvement to parallel speedup as `par`s are disabled

9.1.3 Implicit Parallelism

As the quote from the start of the paper alludes to, there are few modern attempts at implicit parallelism for lazy functional languages. One significant exception to this is the work undertaken in 2007 by Harris and Singh [7]. The results were mostly positive (in that most benchmarks saw an improvement in performance) but were not to the degree desired. Since this research was published we have seen no other attempt in this line of research within the lazy functional programming community.

The approach presented here can be seen as the inverse of the approach by Harris et al. [7] where the compiler starts with *no* parallelism and the iterative feedback seeks to add parallel evaluation. Their work attempted to use runtime profile data to introduce parallel annotations into the program based on heap allocations. In short, when viewing the parallel execution of a program as a tree, their method seeks to expand the tree based on previous executions of the program. Our goal is to develop a system that begins with a program that has *too much* parallelism and uses runtime data to prune the execution tree.

9.1.4 Semi-implicit Parallelism

Research into semi-implicit parallelism for lazy functional languages is still an active research area. The work on Repa frees the programmer from worrying about parallelism in array computations by hiding the parallel details behind an API [26]. GPGPU parallelism is also well studied and has resulted in the popular high-performance Haskell library Accelerate [27].

9.2 Future Work

`par` Health: Other forms of penalties need to be introduced. Blocking other threads, being blocked for extended periods of time, creating too many parallel threads (or not enough) could all be measures that incur a penalty. One simple but effective metric would be to penalise `par` sites that spark strategies that are not the first to evaluate their arguments. When another thread forces the evaluation of a structure before the strategy that was meant to, we lose the benefit of parallel strategies. These penalties would factor in to the `par` site's 'health' and determine which site should be disabled before the next iteration.

Specialisation: One area that we expect to explore is the use of other forms of specialisation. Defunctionalisation specialises higher-order functions to first-order ones. Other possibilities include specialising polymorphic functions into their monomorphic versions and specialising functions based on their call-depth. Specialising based on call-depth is a common technique in hand written programs. Automating the process could lead to significant improvements in recursive numeric programs (such as `Tak`).

Path Analysis: One goal of our work is to determine when it is appropriate to use parallelism in a strategy. Currently, we spark constructor fields in a left-to-right order but we believe that performing a path analysis would aid in this task [28]. Path analysis can also help reduce the chances of thread collisions statically. By determining which expression will force a value first we can avoid one of the central shortcomings of the approach outlined in this paper.

References

[1] B. Lippmeier, "[Haskell] Implicit parallel functional programming," https://mail.haskell.org/pipermail/haskell/2005-January/015213.html, Jan 2005, [Online; accessed 13-March-2015].

[2] R. J. M. Hughes, "The Design and Implementation of Programming Languages," Ph.D. dissertation, Programming Research Group, Oxford University, July 1983.

[3] S. L. Peyon Jones, "Parallel Implementations of Functional Programming Languages," *Comput. J.*, vol. 32, no. 2, pp. 175–186, Apr. 1989.

[4] K. Hammond and G. Michelson, *Research Directions in Parallel Functional Programming*. Springer-Verlag, 2000.

[5] G. Hogen, A. Kindler, and R. Loogen, "Automatic Parallelization of Lazy Functional Programs," in *ESOP'92*. Springer, pp. 254–268.

[6] G. Tremblay and G. R. Gao, "The Impact of Laziness on Parallelism and the Limits of Strictness Analysis," in *Proceedings High Performance Functional Computing*, 1995, pp. 119–k133.

[7] T. Harris and S. Singh, "Feedback Directed Implicit Parallelism," in *ICFP '07: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*. New York, NY, USA: ACM, 2007, pp. 251–264.

[8] S. P. Jones, C. Clack, and J. Salkind, "GRIP: A High Performance Architecture for Parallel Graph Reduction," in *Functional Programming Languages and Computer Architecture: Third International Conference (Portland, Oregon)*. Springer Verlag, 1987.

[9] D. Jones, Jr., S. Marlow, and S. Singh, "Parallel Performance Tuning for Haskell," in *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*. New York, NY, USA: ACM, 2009, pp. 81–92.

[10] C. Runciman and D. Wakeling, "Profiling Parallel Functional Computations (Without Parallel Machines)," in *Functional Programming, Glasgow 1993*. Springer, 1994, pp. 236–251.

[11] M. Naylor and C. Runciman, "The Reduceron Reconfigured," in *ICFP '10: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. New York, NY, USA: ACM, pp. 75–86.

[12] R. Hinze, "Projection-based Strictness Analysis: Theoretical and Practical Aspects," 1995, Inaugural Dissertation, University of Bonn.

[13] P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones, "Algorithm + Strategy = Parallelism," *J. Funct. Program.*, vol. 8, no. 1, pp. 23–60, Jan. 1998.

[14] C. Clack and S. Peyton Jones, "The Four-Stroke Reduction Engine," in *Proceedings of the 1986 ACM conference on LISP and functional programming*. ACM, 1986, pp. 220–232.

[15] A. Mycroft, "The Theory and Practice of Transforming Call-by-Need Into Call-by-Value," in *International Symposium on Programming*. Springer, 1980, pp. 269–281.

[16] S. Peyton Jones, P. Sestoft, and J. Hughes, "Demand Analysis," 2006, unpublished draft.

[17] P. Wadler, "Strictness Analysis on Non-Flat Domains," in *Abstract Interpretation of Declarative Languages*. Ellis Horwood, 1987, pp. 266–275.

[18] P. Wadler and R. J. M. Hughes, "Projections for Strictness Analysis," in *Functional Programming Languages and Computer Architecture*. Springer, 1987, pp. 385–407.

[19] S. Marlow, P. Maier, H. Loidl, M. Aswad, and P. Trinder, "Seq No More: Better Strategies for Parallel Haskell," in *Proceedings of the third ACM Haskell symposium on Haskell*. ACM, 2010, pp. 91–102.

[20] R. Kubiak, J. Hughes, and J. Launchbury, "Implementing Projection-Based Strictness Analysis," in *Functional Programming, Glasgow 1991*. Springer, 1992, pp. 207–224.

[21] H. W. Loidl, "Granularity in Large-Scale Parallel Functional Programming," Ph.D. dissertation, PhD thesis, Department of Computing Science, University of Glasgow, 1998.

[22] L. Augustsson and T. Johnsson, "Parallel Graph Reduction with the $\langle v, G\rangle$-Machine," in *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture*, ser. FPCA '89. New York, NY, USA: ACM, 1989, pp. 202–213.

[23] B. O'Sullivan, "Criterion: A Haskell Microbenchmarking kibrary," https://hackage.haskell.org/package/criterion, 2009.

[24] I. Sergey, D. Vytiniotis, and S. Peyton Jones, "Modular, Higher-order Cardinality Analysis in Theory and Practice," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '14. New York, NY, USA: ACM, 2014, pp. 335–347.

[25] J. M. Calderón Trilla, S. Poulding, and C. Runciman, "Weaving Parallel Threads: Searching for Useful Parallelism in Functional Programs," in *Proceedings of the Symposium on Search-Based Software Engineering*, 2015.

[26] G. Keller, M. M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier, "Regular, Shape-Polymorphic, Parallel Arrays in Haskell," in *ICFP '10: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, vol. 45, no. 9. ACM, 2010, pp. 261–272.

[27] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover, "Accelerating Haskell Array Codes with Multicore GPUs," in *Proceedings of the 6th Workshop on Declarative Aspects of Multicore Programming*. ACM, 2011, pp. 3–14.

[28] A. Bloss, "Path Analysis and the Optimization of Nonstrict Functional Languages," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 3, pp. 328–369, 1994.