An Iterative Compiler for Implicit Parallelism

Extended Abstract

José Manuel Calderón Trilla Colin Runciman
University of York, York UK
{jmct|colin}@cs.york.ac.uk

Abstract

Advocates of lazy functional programming languages often cite easy parallelism as a major benefit of abandoning mutable state [1]. This idea drove research into the theory and implementation of compilers that take advantage of implicit parallelism in a functional program. Using static analysis techniques compilers can attempt to identify where a program can benefit from parallelism and ensure that those expressions are executed concurrently with the main thread of execution [2, 3]. These techniques can produce improvements in the runtime performance of a program, but are limited by the static analyses' poor prediction of runtime performance. Our work is on the development of a system that uses feedback from runtime profiling *in addition to* well-studied static analysis techniques in order to achieve higher performance gains than through static analysis alone.

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

1. Introduction

The amenability of functional languages to parallelism has long been advertised [4, 5] but the ultimate goal of writing a program in a functional style and having the compiler find the implicit parallelism still requires work. Static analysis, when used alone, has underperformed in this endeavor [2, 3, 6, 7]. Our thought is that the compiler should incorporate runtime profile data into decisions about parallelism the same way a programmer would manually tune a parallel 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 `par` annotations that under-perform and the compiler will disable the parallelism they introduce.

1.1 Contributions

The main focus of our work has been the design and implementation of an experimental compiler that allows for implicit parallelism. The source language of the compiler is an enriched lambda calculus which is suitable for use as a functional core language in a larger compiler. The contributions of our work are as follows:

- The use of *switchable* `par` annotations[1]
- An implementation of Hinze's projection based strictness analysis [8]
- Utilising the correspondence between projections and strategies to introduce parallelism into a program
- Using search strategies to improve upon the initial `par` placement

This paper presents an overview of the design of our compiler and some of the design decisions that were made. As we are now beginning to run experiments, this paper also serves as a documented hypothesis for our results.

1.2 Compiler Pipeline

The compiler is composed of 5 main phases, illustrated in Figure 1

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 and G-Code generation are done in the standard way and will not be discussed further. The rest of the paper is organised by following the compiler pipeline as shown in figure 1. In §2 we explain the advantages of performing defunctionalisation. We motivate our use of a projection based strictness analysis [9] in §3. §4 is a description of the correspondence between projections and strategies [10] which allows us to generate parallel strategies based on the projections provided by the strictness analysis. The technique used for utilising the runtime profiling to switch off some of the introduced parallelism is described in§5 along with possible additional search techniques. Lastly, §6 contains our conclusions and thoughts on possible future work.

2. Defunctionalisaton

As mentioned above, the design of the compiler utilises a defunctionalising transformation on the input programs. Defunctionalisation

[1] Our `par` annotations take the familiar form of `par a b = b`, where the first argument is 'sparked off' in parallel and the function returns its second argument.

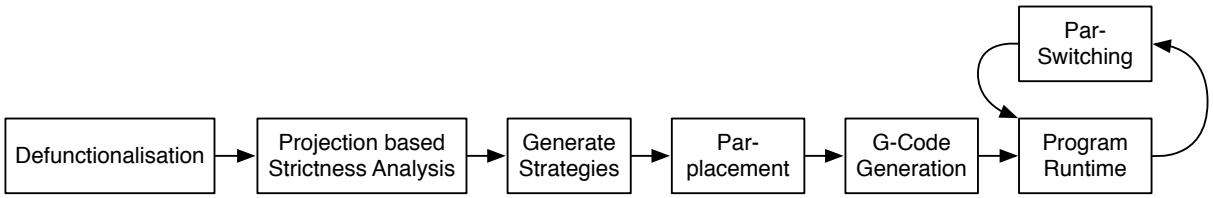

Figure 1. Compiler Pipeline After Parsing

specialises higher-order functions to the instances of their function arguments. Here we give our motivation for introducing this transformation.

Central to our design is the concept of `par` placement within a program. Each `par` is identified by its *position* in the AST. Due to the higher-order nature of our language, basing our parallelism on the location of a `par` can lead to undesirable circumstances. 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
```

This function allows us to use a common technique (mapping) with the possibility of performance gains through parallelism. However, when the computation `f x` is inexpensive, the parallelism may not provide any benefit or could even be detrimental. As `parMap` may be used throughout a program it is possible that there are both useful and detrimental instances of the function. For instance, `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 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 these functions we create two separate `parMap` functions: `parMapf` and `parMapg`. This now provides us with `par` annotations in *each* of the instances of `parMap`.

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

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

Because of defunctionalisation we are able to deactivate the `par` for the inexpensive computation, `g x`, without affecting the parallelism of the worthwhile computation, `f x`.

3. Strictness Analysis

The view of lazy languages (evaluation should only occur when necessary) can be at odds with the goals of performance through parallelism (do as much work as possible for faster 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. For example, in a strict language, the function below would not terminate due to having to evaluate $\perp$ before entering the function, while lazy languages can compute the correct result since they only evaluate expressions when they are needed:

```
squareFirst :: Int -> Int -> Int
squareFirst x ⊥ = x * x
```

The problem of knowing which arguments are required for a function is known as *strictness analysis* [11] and forms the core of the static analysis phase of the compiler. In this section we provide a brief overview of the two predominant techniques for strictness analysis, ideal[2] analysis and projection based analysis. We then motivate our decision to use a projection based analysis.

3.1 Ideal Analysis

The main idea behind abstract interpretation is that you can throw away information about your program that is not necessary for the property you are analysing. When dealing with the `Integer` type, it may not be necessary to know the actual value of an integer, but instead only *some* of the information about that integer. Mycroft's "The Theory and Practice of Transforming Call-by-need into Call-by-value" [11] introduced the use of abstract interpretation for performing strictness analysis on call-by-need programs.

In the case of strictness analysis, we only require information about how defined a value is, and do not need to know about its concrete value.

In short, when performing ideal analysis we only concern ourselves with the definedness of values when analysing the strictness properties of programs.

With the strictness information in hand we can annotate our program to execute strict arguments in parallel with the function. In short, the strictness analysis informs the *initial* placement of `par` annotations in a program. Basing the initial placement on strictness information is important because we aim for our compiler to maintain the semantics of the initial sequential program.

When using a safe analysis we may not be able to determine all of the needed arguments for a given function. However, we can be certain that any argument the analysis determines is needed is definitely needed. This safety is crucial in avoiding the introduction of $\perp$ where it would not have occurred in a sequential lazy implementation [12].

The strictness properties of a function can be defined more formally as follows: A function of the form

$$f\ n_1\ n_2\ \ldots\ n_n = e$$

is said to be strict in argument n_m iff

$$f\ \ldots\ \perp_m\ \cdots = \perp$$

3.2 Projections

Strictness analysis as originally described by Mycroft is only capable of dealing with a two-point domain (values that are definitely needed,

[2] This terminology is used by Hinze in [8] to differentiate between the two methods.

ID: accepts all lists

T (tail strict): accepts all finite lists

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

HT (H and T strict): accepts finite lists where every member is defined

Figure 2. Four contexts on lists as described in [9].

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.)[3] but falls short on more complex data structures. For example, if a list argument is needed for a function to terminate, we can only evaluate up to the first `cons` safely. However, there are many functions on lists where evaluating the entire list (or even just the spine) can be safe. The canonical examples are `length` and `sum`. When evaluating the length of the list it would be safe to have evaluated the spine (and only the spine) of the list beforehand. This makes intuitive sense, if the evaluation of the spine is non-terminating, then the evaluation of `length` would be non-terminating as well. The function `sum` extends the same premise to the spine *and* the elements of a list.

In order to accommodate this type of reasoning, Wadler developed the well known four-point domain for lists [12]. While this work allowed for analysis to be performed on functions accepting lists, it was not easily extended to functions on other data structures.

Another approach involved *projections* from domain theory. Projection based analysis provides two benefits over ideal based analysis: The ability to analyse functions over arbitrary structures, and a correspondence with parallel strategies [10, 13]. This allows use to use the projections provided by our analysis to produce an appropriate function to compute the strict arguments in parallel.

Ideally we could generate and utilise strategies on any arbitrary type. This would allow to compiler to annotate the needed expression with the maximal safe amount of reduction. This requires us to use a more sophisticated form of strictness analysis: projections [9].

Projections asks a slightly different question than the ideal analysis described above. If the above asks "When passing this argument as $\bot$ is the result of the function call $\bot$?" then projections ask "If there is a certain degree of demand for the result of this function, what degree of demand is there on its arguments?".

First let us explain what is meant by 'demand'. 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. However, what if the *result* of `append` needs to be a finite list? In other words the function calling `append` requires that its input list be finite.

A simple example of this is the following program

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

In this case *both* arguments to `append` must be finite. Projections allow us to make this distinction with the use of contexts [8, 9].

For lists we have the following contexts:

We can now say more about the strictness properties of `append`:

```
ID (append xs ys) = ID!(xs); ID(ys)
T (append xs ys)  = T!(xs); T!(ys)
H (append xs ys)  = H!(xs); H(ys)
HT (append xs ys) = HT!(xs); HT!(ys)
```

Here we use the convention from [8] of using `!` to denote the strictness of a context. `ID!` requires the list be defined to the first cons, whereas an expression in an `ID` context may not be needed.

Hinze's Work on Projections

Much of the work on strictness analysis as a means to achieve implicit parallelism focused on the ideal analysis approach. This was mostly an accident of timing, the work on projections had not been fully developed when implicit parallelism was a more active research area. In particular, the wonderful work on the "Automatic Parallelization of Lazy Functional Programs" [3] only used two and four-point domains (as described in [12]) in their strictness analysis. This limits the ability of the compiler to determine the neededness of more complex structures.

While projections were known as a possible technique for strictness analysis, the theory was much more complex and many of the details regarding the generality of the approach were not yet worked out. The work of Hinze [8] shows how projections can be used to determine the strictness information on complex data-types and sets the technique on a solid theoretical foundation that ensures its generality (in particular when working with polymorphic functions).

Using results from domain theory we are able to construct projections for every user-defined type, and furthermore each projection represents a specific strategy for evaluating the structure [8]. This provides us with the ability to generate appropriate parallel strategies for arbitrary types.

4. Projections and Strategies

As mentioned in the previous section, one of the reasons that projections were chosen for our strictness analysis is their correspondence to parallel strategies. The main idea behind strategies is that it is possible to write functions whose sole purpose is to force the evaluation of specific parts of a structure [10, 13]. An important point is that all strategies return `()`, having the type `type Strategy a = a -> ()`, which tells us that strategies are not used for their computed result but for the evaluation they force along the way.

The simplest strategy, named `r0` in [13], 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‘ ()`

Neither of these strategies are of much interest. The real power comes when strategies are used on nested data-structures. Take lists for example, evaluating a list sequentially or in parallel provides us with the following two strategies

```
seqList :: Strategy a -> Strategy [a]
seqList s []     = ()
seqList s (x:xs) = s x ‘seq‘ (seqList s xs)

parList :: Strategy a -> Strategy [a]
parList s []     = ()
parList s (x:xs) = s x ‘par‘ (parList s xs)
```

First notice that 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 type `a` fully would result in list's spine *and* elements being evaluated. Already we can see a correspondence between these strategies and

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

the contexts shown in figure 2. 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.

This correspondence allows us to generate strategies based on the results of our strictness analysis. Because the projection based approach gives us the ability to describe different levels of demand on arbitrary data-types, we then get all of the corresponding strategies to evaluate up to that demand, but no more.

One aspect of strategies that does not directly correspond a context is choice between `seq` and `par`. Every context can be fully described by both sequential and parallel strategies. One goal of our work is to determine when it is appropriate to use parallelism in a strategy. Every field of a constructor has the potential to be evaluated in parallel. When a constructor has one field, it is not usually beneficial to do so, but when the 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 parallelised and which should be evaluated in sequence. We currently rely on heuristics but we believe that performing a path analysis would aid in this task [14].

5. Iterative Compilation

We now have all of the building blocks for what we see as our contribution. 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.

Imagine that you were writing a parallel program. When writing the source code you may study the structure and then decide where to place `par` annotations. When the program is compiled and executed you find that the performance was not satisfactory. Normally, one would return to the source for the program and adjust the placement of parallel annotations. This is the approach advocated by [15] and [16]. However, many of the previous attempts at implicit 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.

There is one significant exception to this. In 2007 Harris and Singh published their results on a feedback directed implicit parallelism compiler [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 functional programming community.

The work in [7] 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 perhaps has *too much* parallelism and uses runtime data to prune the execution tree. We have implemented a few mechanisms to make this possible.

5.0.1 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 tells 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. For this reason we have used a similar system as that outlined in [16]. With the following metrics being recorded *for each thread*:

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

5.0.2 Transformation:

In order for the iterative feedback to be able to change the parallelization of a program, it must be able to determine which expressions can be transformed. The method we have devised is based on the idea that a specific `par` in the source program can be deactivated and therefore no longer create parallel tasks, while maintaining the semantics of the program. The method has two basic steps:

- `par`'s are identified via the G-Code 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`.

5.0.3 Iteration:

Using the runtime profile data we can experiment with different search methods. We can represent a program's `par` switches as a bit string with each `par`'s current setting stored in a bit.

One technique would be to ignore the runtime data altogether! There are lots of algorithms used for searching for optimal configurations of bit strings. In our case the fitness-function would simply be the overall running time of the program when run with a specific `par` setting.

However, while we feel that blind search techniques are worth exploring, guided search is more likely to produce results quickly. The first search heuristic could be very simple: After every execution, turn off the `par` site whose threads have the lowest average reduction count. Repeat this process until switching a `par` off increases the overall runtime of the program.

Another possibility is to determine an overhead penalty for sparking parallel threads. If the average reduction count for all the threads from a `par` site is less than the overhead penalty, the `par` is switched off. Other forms of penalties could also 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.

6. Conclusions

We hope we have motivated the key design choices and ideas behind our compiler: Utilising defunctionalisation in §2, and the use of projections over other strictness analysis methods §3. And that we have shown that there is a natural correspondence between projections and strategies §4 that allows us to generate parallel strategies from the results of our strictness analysis.

6.1 Future Work

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.

The first of these allows for the possibility that a polymorphic function that introduces parallelism may only provide a benefit when applied to arguments of a certain type. The depth-specialisation confronts the common granularity problem when writing recursive algorithms that introduce parallelism. The top-level call of the function may see huge benefits from its parallelism, but the lower level calls may not be as worthwhile (the `nfib` function is a good example of this, parallelising the recursive calls of `nfib 25` may be worthwhile, but not for `nfib 2`).

References

[1] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler, "A History of Haskell: Being Lazy with Class," in *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, ser. HOPL III. New York, NY, USA: ACM, 2007, pp. 12–1–12–55. [Online]. Available: http://doi.acm.org/10.1145/1238844.1238856

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

[3] G. Hogen, A. Kindler, and R. Loogen, "Automatic Parallelization of Lazy Functional Programs," in *ESOP'92*. Springer, 1992, pp. 254–268.

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

[5] S. L. Peyon Jones, "Parallel implementations of functional programming languages," *Comput. J.*, vol. 32, no. 2, pp. 175–186, Apr. 1989.

[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*. Citeseer, 1995, pp. 119–k133.

[7] T. Harris and S. Singh, "Feedback Directed Implicit Parallelism," *SIGPLAN Not.*, vol. 42, no. 9, pp. 251–264, Oct. 2007. [Online]. Available: http://doi.acm.org/10.1145/1291220.1291192

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

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

[10] 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. [Online]. Available: http://dx.doi.org/10.1017/S0956796897002967

[11] 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.

[12] P. Wadler, "Strictness Analysis on Non-Flat Domains," in *Abstract interpretation of declarative languages*. Ellis Horwood, 1987, pp. 266–275.

[13] 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.

[14] 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.

[15] D. Jones, Jr., S. Marlow, and S. Singh, "Parallel Performance Tuning for Haskell," in *Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell*, ser. Haskell '09. New York, NY, USA: ACM, 2009, pp. 81–92. [Online]. Available: http://doi.acm.org/10.1145/1596638.1596649

[16] N. Charles and C. Runciman, "An Interactive Approach to Profiling Parallel Functional Programs," in *Implementation of Functional Languages*. Springer, 1999, pp. 20–37.