Naira: A Parallel^2 Haskell Compiler

Sahalu Junaidu Tony Davie Kevin Hammond

Division of Computer Science, University of St. Andrews
{sahl,ad,kh}@dcs.st-and.ac.uk

Abstract

Naira is a compiler for a parallel dialect of Haskell, compiling to a dataflow-inspired parallel abstract machine. Unusually (perhaps even uniquely), Naira has itself been parallelised using state-of-the-art tools developed at Glasgow and St Andrews. This paper reports initial performance results that have been obtained using the GranSim simulator, both for the top-level pipeline and for individual compilation stages. Our results show that a modest but useful degree of parallelism can be achieved even for a distributed memory machine.

1 Introduction

The Naira compiler was written to explore the problems of parallelising a modern functional language compiler [Juna97]. It compiles from a subset of Haskell [HPW92] to a RISC-like target language that has been extended with special parallel constructs [Osth93]. The front end of the compiler comprises about 5K lines of Haskell code organised in 18 modules.

This paper explores the process of parallelising this compiler using state-of-the-art profiling tools that were developed at Glasgow and St Andrews [HLP95]. Our initial results are promising, indicating that acceptable speedups can be achieved within individual compiler passes, notably the type inference pass. There is, however, a sequential nub caused by file I/O and parsing which limits the overall speedup that can be obtained.

The rest of this paper is structured as follows. Section 2 describes our general approach to parallelising the compiler, giving performance results for both the top-level pipeline and individual compilation passes. Section 3 describes related work. Finally Section 4 concludes.

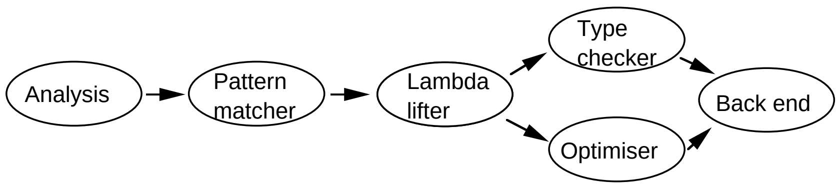

Figure 1: The Structure of the Top-Level Pipeline

```
analThisMod fileNm  stPM stTE stCG exptNames name impVals
              aTree tTree dats syns combs locals = result
  where
    defs       = mkDefs fileNm stPM combs
    liftedDefs = lLift fileNm stPM defs
    typeList   = tcModule fileNm stTE exptNames tTree  syns liftedDefs
    intLang    = optimiseParseTree fileNm exptNames stCG  aTree liftedDefs
    result     = showModule name impVals dats exptNames (intLang, typeList)
    strat  res =  parList rnf combs            `par`
                    parList rnf defs       `par`
                    parList rnf liftedDefs `par`
                    parList rnf typeList   `par`
                    parList rnf intLang    `par`
                    ()
```

Figure 2: The Top-Level Compiler Function: `analyseModule`

2 Parallelisation

We use a top-down parallelisation methodolody, as outlined in [THLP98], starting with the top-level pipeline, then proceeding to parallelise successive pipeline stages. We concentrate on parallelising the four main compiler passes — the pattern matcher, lambda lifter, type checker, and the optimiser. These passes are parallelised in a data-oriented fashion by annotating the intermediate data structures used to communicate results between the passes. We have experimented with two common data structures for these intermediate structures: lists and binary trees.

2.1 Unique Name Servers

Data dependency can be a significant hindrance to exploiting parallelism effectively. In Naira, unique name servers are used to help break data dependencies and so expose additional parallelism. Our early experiences with some name supply mechanisms suggest that a simple name server similar to that of Hancock [Peyt87] is acceptable, and more complex name servers such as those described by Augustsson *et al.* [ARS94] are not needed.

2.2 The Top-Level Pipeline

The overall top-level pipeline structure of the compiler is as depicted in Figure 1. The first, analysis pass consists of the lexical analyser and the parser. The next four passes implement the pattern matching compiler, the lambda lifter, the type checker and the intermediate language optimiser respectively. The detailed organisation and implementation of these passes are described elsewhere [Juna97].

Each compiler pass operates on an intermediate parse-tree which is modified to produce the input to the next compiler pass. The outputs of the type-checker and optimiser passes are merged within the final back-end pass. The pipeline is parallelised by defining data-oriented evaluation strategies [THLP98] on these intermediate structures. Choosing the correct strategy turns out to be surprisingly subtle, since we need to avoid introducing excessive speculative evaluation, with its consequent negative effect on performance.

Figure 2 shows the function, `analyseModule`, that implements this top-level pipeline. It is called immediately following symbol table construction, and passes its arguments to each compiler pass as appropriate. The evaluation strategy `strat` sparks five parallel tasks, one for each of the pipeline phases shown in Figure 1. One disadvantage of using strategies in this form (through the `using` combinator) is that every intermediate value must be named. To avoid this, we can use two binary combinators, (`$`|) and (`$`||), for sequential and parallel function application, respectively [THLP98]. The second argument in each case is a strategy to be applied respectively before, or in parallel with, the function application.

Using these combinators, the code for `analyseModule` can be written more concisely, but perhaps less intuitively, as in Figure 3.

In order to evaluate the compiler, we experimented with two different machine configurations: a

```
analyseModule fileNm  stPM stTE stOpt exptNames name impVals
              aInfo tInfo dats syns combs =
    showModule name impVals dats exptNames $||
                        parPair (parList rnf) (parList rnf) $
    fork (optimiseParseTree fileNm exptNames stOpt  aInfo,
         tcModule fileNm stTE exptNames tInfo  syns) $||
                                              parList rnf $
    lLift fileNm stPM      $|| parList rnf $
    mkDefs fileNm stPM $|| parList rnf $
    combs
fork (f, g) inp = (f inp, g inp)
```

Figure 3: `analyseModule` rewritten using Pipeline Strategies

Module	**Avg. Par.**	**Speedup**	*Module*	**Avg. Par.**	**Speedup**
MyPrelude	4.0 (3.0)	3.85 (2.91)	MatchUtils	3.4 (2.6)	3.32 (2.55)
DataTypes	4.3 (3.0)	4.17 (2.90)	Matcher	2.7 (2.1)	2.58 (2.06)
PrintUtils	1.5 (1.5)	1.44 (1.43)	LambdaUtils	2.1 (2.0)	1.98 (1.87)
Printer	2.1 (1.3)	2.08 (1.30)	LambdaLift	2.4 (2.1)	2.36 (2.07)
Tables	3.6 (2.7)	3.49 (2.59)	TCheckUtils	3.5 (3.2)	3.46 (3.19)
LexUtils	2.9 (1.9)	2.79 (1.83)	TChecker	1.8 (1.8)	1.76 (1.75)
Lexer	1.6 (1.5)	1.50 (1.45)	OptmiseUtils	1.5 (1.4)	1.48 (1.43)
SyntaxUtils	3.2 (2.1)	3.21 (2.09)	Optimiser	1.1 (1.1)	1.07 (1.06)
Syntax	1.3 (1.3)	1.24 (1.23)	Main	1.7 (1.7)	1.64 (1.63)

Table 1: Top-level Pipeline: Speedup and Parallelism

low-latency shared-memory configuration, and a medium-latency distributed memory configuration [Juna97]. Both configurations were for 32-processor machines. Our test input comprised the 18 source modules for the Naira compiler itself.

Our experiments reveal that as well as being less concise, the original version of `analyseModule` is also less efficient than the new version. For our 18 sample input modules, we found that the second version was up to 20% more efficient than the first. There were, however, two cases where

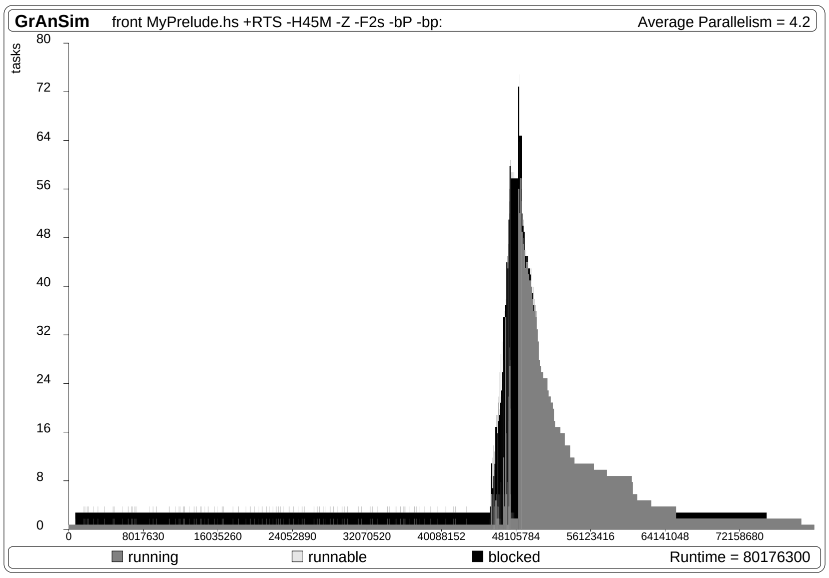

Figure 4: Top-Level Pipeline compiling `myPrelude` (Distributed Memory Machine)

the version using (`$||`) was inferior. Overall speedup relative to the sequential case ranges from 1.07 to 4.17 (mean: 2.41) for the shared-memory configuration, or 1.06 to 3.19 (mean: 1.96) for the distributed memory configuration. In cases the shared-memory configuration yields marginally greater speedup than its distributed-memory counterpart. The full set of results for all 18 modules is given in Table 1 (shared-memory results are shown in-line, distributed-memory results are parenthesised).

Compared with experimental results that have been previously achieved [THL+96, THLP98], these results are quite encouraging. These earlier studies achieved average parallelism of 1.2 on a database-type problem [THL+96] and about 2.5 on the Lolita natural language parser [THLP98] at the same top-level parallelisation stage.

We now consider how to parallelise each compiler pass. Each of subsections 2.3–2.6 considers a single pass. Section 2.7 considers the overall effect of combining all the passes.

2.3 The Pattern Matching Compiler

The pattern matching compiler transforms function definitions that use equational patterns into equivalent ones involving case expressions with simple variable patterns., as described by

```
mkDefs fileNm env [] = []
mkDefs fileNm env l  =
     mkAppend  $|| parPair (parList rnf)(parList rnf) $
     fork2 (checkAdjDefs fileNm env,
           mkDefs fileNm env) $|| parPair (parList rnf)(parList rnf) $
     partition (sameDef (head l)) $|| parList rwhnf $ l
```

Figure 5: The Pattern Matching Compiler: `mkDefs`

```
lLift fileNm stPM defs =
  id $|| parList rnf $
  lifter.triplet1 $||
        parTriple (parList rnf)(parList rnf)(parList rnf) $
  scopeAnalysis fileNm stPM [] [] initNS 1 $|| parList rnf $ defs
```

Figure 6: The Lambda Lifter: `lLift`

Peyton Jones [Peyt87]. This transformation is primarily performed for efficiency purposes.

Once the definitions within a module have been parsed, the pattern matching transformation can be applied to each of these definition independently. The pattern matching compiler is implemented using the function `mkDefs` (Figure 5), whose arguments are the file name, a pattern matching symbol table and the list of definitions output from the parser.

This definition of `mkDefs` provides an initial top-level parallelisation in which the analysis of each binding proceeds in parallel with that of the others. In order to provide additional finer-grained parallelism, we have tried three further parallelisation steps: compiling local pattern definitions in parallel with their top-level parents; parallelising heavily used auxiliary functions and changing the parse tree representation to be a list rather than a binary tree. None of these made any significant difference to the overall performance, based on our 18 input modules.

2.4 The Lambda Lifter

The lambda lifter is fairly conventional, comprising a scope analyser, a renamer, a dependency analyser, and the final lifting operation. As usual, this transformation is relevant only for (mutually) recursive top level bindings and for function definitions which have local definitions.

```
tcModule fileNm env exptNames tyList syns defs    =
        tyList ++ topDefsTypes
  where
  (ns0,ns1)      = split initNS
  tIds           = map defId defs
  auxEnv         = mkTypeVars tIds ns0
  topDefsTypes   =
     tcTopDefs fileNm env auxEnv exptNames initSubs ns1 syns defs
```

Figure 7: The Type Checker: `tcModule`

The two main functions, `scopeAnalysis` and `lifter`, which collectively perform the bulk of the computation in the lambda lifting process are combined into a two-stage pipeline. The first stage performs scope analysis, also incorporating the renamer and dependency analyser. The second stage (`lifter`) computes the transitive closure of each function's free variables and performs appropriate substitutions. Parallelising this pipeline leads to a modest performance improvement: average parallelism improves to 1.1 overall for eight of our input modules with consequent modest speedup.

Because the renamer simply associates an identifier with a small integer, and since only locally defined identifiers are renamed (the parser would have reported any name clashes amongst top level identifiers), it performs very little work, and is thus not suitable for parallelisation. Although, in comparison, the dependency analyser performs a relatively large amount of computation, since it is based on sequential graph algorithms, unfortunately it cannot be easily parallelised.

In the second pipeline stage, the lambda lifter collects free variables, forms and solves equations [John87] in order to determine the complete set of free variables for each function. It transpires that for our sample programs, this is not an expensive process, since each local binding contains very few local definitions, and so it is not worth parallelising. This is a consequence of separating the source definitions into minimal dependency groups in order to aid type checking [Peyt87].

We conclude that, as for the pattern matching compiler, there is minimal scope for parallelisation within this compiler pass.

Module	**Avg. Par.**	**Speedup**	*Module*	**Avg. Par.**	**Speedup**
MyPrelude	?? (3.3)	?? (3.26)	MatchUtils	?? (3.3)	?? (3.23)
DataTypes	?? (2.2)	?? (2.21)	Matcher	?? (2.1)	?? (2.07)
PrintUtils	?? (1.4)	?? (1.40)	LambdaUtils	?? (1.7)	?? (1.63)
Printer	?? (2.0)	?? (2.01)	LambdaLift	?? (2.2)	?? (2.16)
Tables	?? (2.4)	?? (2.33)	TCheckUtils	?? (2.2)	?? (2.14)
LexUtils	?? (2.3)	?? (2.22)	TChecker	?? (1.8)	?? (1.71)
Lexer	?? (1.5)	?? (1.43)	OptimiseUtils	?? (1.2)	?? (1.15)
SyntaxUtils	?? (3.2)	?? (3.15)	Optimiser	?? (1.2)	?? (1.23)
Syntax	?? (1.2)	?? (1.23)	Main	?? (1.6)	?? (1.61)

Table 2: Type Checker (Initial Parallelisation): Speedup and Parallelism for Distributed Memory

2.5 The Type Checker

Cost-centre profiling [SaPe95] reveals that, as is often the case, the type checker is the most expensive part of the compiler, both in terms of space usage and runtime. In fact it is more expensive than all the other compiler passes put together. This is largely because the type checking process incorporates complex sub-algorithms such as unification. The effectiveness of our overall parallelisation therefore depends significantly on how much useful parallelism can be extracted from the type checker.

The function `tcModule` (Figure 7) is used to implement the type inference algorithm for a collection of definitions in a module. The first three arguments to this function are the file name, the type environment and a list of exported values. The final arguments contain the types of imported values, a list of type synonyms and the list of definitions in the module.

As in standard polymorphic type checking algorithms, `tcModule` initially associates each bound name with an assumed type creating an auxiliary environment, `auxEnv`. These assumed types usually become specialised as unifications and substitutions are performed. Inferred types are also checked against user-declared type signatures in accordance with the usual Haskell rules [HPW92].

The type checker can be parallelised using a parallel name server and by distributing substitutions to avoid sequentialising the inference process. For intance, to type check two quantities

```
tcLocalDefs fileNm env subs ns syns []                             = ([],[],subs)
tcLocalDefs fileNm env subs ns syns (VDef(IdPat id) args e:defs) =
    (id:idsL, infTy:tysL,subs4) `using` strat
  where
    (ns0,ns1)         = split ns
    (infTy,subs1)     = typeCheck fileNm id (mkLam args e) env subs syns ns1
    (idsL,tysL,subs4) = tcLocalDefs fileNm env subs ns0 syns defs
    strat res         = parTriple rnf (parList rnf) rwhnf res
```

Figure 8: Type Inference for Local Definitions: `tcLocalDefs`

```
mkUnify syns t1 t2 = (subs,theTy)
   where subs  = unify (OK []) (expandSynonyms syns t1)(expandSynonyms syns t2)
         theTy = mkTheType subs t1 t2
```

Figure 9: Type Unification: `mkUnify`

d_1 and d_2, we analyse them simultaneously in the current type environment, each returning a type and a substitution record. If a variable v common to both d_1 and d_2 is assigned (possibly different) types t_1 and t_2 from these two independent operations, t_1 and t_2 will be unified in the presence of the resulting substitutions and the unified type will be associated with v.

Table 2 shows the average parallelism and speedup figures that we obtain from this initial parallelisation (due to lack of time, only distributed-memory results are available in this draft paper). These results reveal promising speedup, similar to those for the top-level pipeline.

There are three obvious avenues for further exploitation of parallelism. These are:

1. inside local definitions;
2. on calls to frequently used functions; and
3. at other expression constructs.

We will consider each of these in turn.

The first step is to add strategic code to the function `tcLocalDefs` (Figure 8) so as to create additional parallel threads to infer the types of the locally defined identifiers. The strategic code

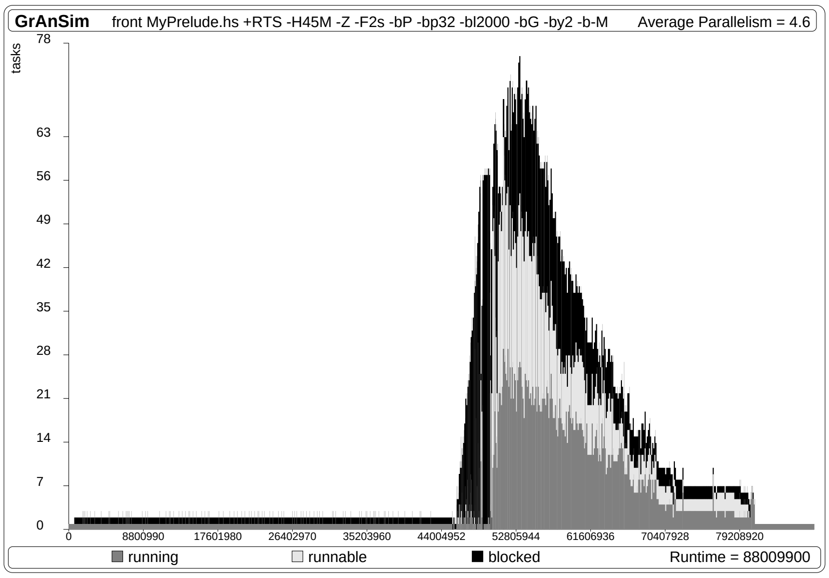

Figure 10: Type Checking `myPrelude` – Step 2 (Distributed Memory Machine)

`parTriple rnf (parList rnf) rwhnf res` ensures the creation of parallel tasks for `res`, the result of `tcLocalDefs`. This modification leads to a modest increase in parallel activity, but no significant increase in speedup [Juna97].

In the second step, we introduce parallelism in the unification algorithm. It is clear that type unification is one of the most costly operations during type inference, and so should yield significant parallelism. Figure 9 shows the sequential unification algorithm. The function `expandSynonyms` ensures that type synonyms within the types being unified are replaced before unification proceeds, while `mkTheType` obtains the unified type on successful unification or generates an error message on unification failure. We introduce parallelism here by sparking a child task to carry out unification on sub-trees, by applying the strategic code below to *each call* of `mkUnify` inside the type checker.

Figure 10 shows the activity profile that results when compiling `MyPrelude` using this setup. Overall performance is slightly improved compared with the first parallelisation step. In the third and final step, we compose substitutions in parallel as for `mkUnify`. This is important since substitutions are composed very frequently in the Naira compiler. The corresponding speedup results are shown in Table 3. Overall speedups range from 1.11 to 4.24 (mean: 2.43) for our shared-memory configuration; 0.75 to 2.05 (mean: 2.05) for the distributed-memory configuration. It is not clear why the Optimiser module yields a slow-down for the distributed-

Module	**Avg. Par.**	**Speedup**	*Module*	**Avg. Par.**	**Speedup**
MyPrelude	6.3 (6.1)	3.70 (3.57)	MatchUtils	6.4 (6.0)	3.39 (3.17)
DataTypes	7.1 (4.2)	4.24 (2.55)	Matcher	6.8 (5.1)	2.64 (1.93)
PrintUtils	2.2 (1.9)	1.48 (1.23)	LambdaUtils	3.4 (3.2)	2.01 (1.87)
Printer	3.4 (3.1)	2.13 (1.95)	LambdaLift	4.2 (6.8)	2.39 (2.06)
Tables	6.6 (5.1)	3.58 (2.71)	TCheckUtils	6.1 (4.4)	3.26 (2.58)
LexUtils	9.1 (7.3)	2.85 (2.40)	TChecker	2.7 (2.7)	1.66 (1.64)
Lexer	5.2 (4.6)	1.60 (1.40)	OptimiseUtils	2.6 (3.2)	1.50 (1.41)
SyntaxUtils	4.5 (4.3)	3.26 (3.10)	Optimiser	1.2 (4.5)	1.11 (0.75)
Syntax	1.9 (1.8)	1.26 (1.23)	Main	5.5 (5.2)	1.67 (1.51)

Table 3: Type Checker (Final Parallelisation): Speedup and Parallelism

memory machine, but communications costs presumably dominate this particular computation.

2.6 The Optimiser

The optimisation pass specialises general function applications (using arity information to short-circuit argument satisfaction checks [Peyt92]) and transforms case-expressions to a simplified form better suited for code generation.

One task is created to collect arity information for the module in parallel with optimising the module (producer/consumer parallelism). Since once arity information is available there are no data dependencies between the defintions, all parse tree simplications can then be performed in parallel.

Once again, it transpires that this compiler pass is not computationally intensive, and so there is little advantage to the parallelisation process. Indeed, the resulting activity profiles are very similar to those for the pattern matcher and lambda lifter (Sections 2.3 and 2.4).

Module	**Avg. Par.**	**Speedup**	*Module*	**Avg. Par.**	**Speedup**
MyPrelude	7.9 (7.7)	3.97 (3.88)	MatchUtils	6.9 (4.7)	3.43 (2.35)
DataTypes	8.3 (8.1)	4.37 (4.28)	Matcher	7.4 (7.3)	2.63 (2.61)
PrintUtils	2.6 (2.5)	1.60 (1.56)	LambdaUtils	4.4 (4.3)	2.40 (2.23)
Printer	3.7 (3.7)	2.18 (2.16)	LambdaLift	3.4 (3.4)	2.40 (2.37)
Tables	7.1 (7.2)	3.56 (3.55)	TCheckUtils	5.6 (5.3)	3.80 (3.74)
LexUtils	10.1 (9.6)	2.88 (2.78)	TChecker	2.9 (2.8)	2.18 (1.57)
Lexer	5.5 (5.3)	1.61 (1.58)	OptimiseUtils	3.5 (3.5)	1.55 (1.49)
SyntaxUtils	4.7 (2.7)	3.26 (1.86)	Optimiser	2.4 (2.3)	1.12 (1.08)
Syntax	2.0 (2.0)	1.27 (1.26)	Main	5.5 (5.1)	1.73 (1.51)

Table 4: Overall Parallelisation

2.7 Overall Parallelisation

In the preceding sections we have shown how to parallelise the top-level pipeline and our four main compiler passes. In this section we consider the effect of combining all our parallel optimisations.

Table 4 shows the results that are obtained when all our parallelisations were used. Compared with the results of Table 3, we observe that there is some interference between individual parallelisations, and so the overall performance is not as high as might be predicted. Overall speedups range from 1.12 to 4.37 (mean: 2.46) for the shared-memory configuration, or 1.08 to 4.28 (mean: 2.32) for the distributed-memory configuration when all parallelisation code is enabled. This represents a slight improvement over simply parallelising the top-level pipeline. Once again performance is not significantly degraded for the distributed-memory configuration.

3 Related Work

While there have been many successful attempts to produce parallelising compilers for pure functional languages (e.g. [THM$^+$96, NSvEP91, Sked91]), we know of no similar attempt to parallelise a complete working compiler. There have, however, been a few attempts to consider individual compiler stages. For example, Hammond has described a parallel type inference algorithm based on using monads to exploit parallelism within type graphs at a finer granularity than that described here [Hamm90].

4 Conclusions and Further Work

Using the GranSim simulator, we have demonstrated that speedup can be achieved within a functional language compiler, even in the relatively harsh environment of a distributed memory machine. As expected, the overall speedup we have achieved so far is useful rather than dramatic. Unusually, the speedups achieved for the shared-memory configuration are not significantly greater than for the distributed-memory configuration. While this may reflect a good parallel decomposition with low communication overheads, it may also indicate that hard data dependencies (probably within the top-level pipeline) comprise a major limitation on the overall parallel potential of the problem. This would repay further investigation.

Clearly the most important parallelisation step from the results we have obtained so far is the parallelisation of the top-level pipeline. While we had hoped to achieve better results from parallelising individual compiler passes (and may still do so with further effort), this does support the contention that it is possible to achieve modest performance improvement for modest effort.

Of the individual compiler passes, the type checker was clearly the most productive from a parallel perspective, yielding performance equivalent to that obtained from parallelising the top-level pipeline in isolation. This is because its cost dominates that of the overall compilation process. Other passes are relatively cheap, and therefore give less overall improvement. Unfortunately, it has so far proved impossible to combine the speedup obtained from the type-checker with that obtained from the top-level pipeline.

Careful study of the parallelism profiles reveals that file I/O and parsing accounts for a significant part of the remaining sequential component to the computation (and therefore by Amdahl's law represents a major limitation on further optimisation). Parallelising I/O can be quite difficult, and is probably beyond the scope of the work reported here.

Our experiences with parallelising individual compiler passes has shown that even with the tools available it is quite hard to understand and predict the performance of the compiler. Small changes in the parallelisation code can also lead to significant changes in parallel behaviour for some inputs. Clearly, there is a need for even better parallel performance monitoring tools. Parallel cost-centre profiling may be a step in this direction [HHLT97].

We note that Naira is an experimental compiler rather than a state-of-the-art optimised compiler like the Glasgow Haskell compiler. While it would be interesting to start from the basis of a compiler such as this, and our results would naturally be more directly useful to a number of real users, the fact that GHC is already highly optimised for *sequential* compilation inevitably makes it a much harder proposition than Naira to parallelise successfully. We hope that the directions we have explored will help focus any available parallelisation effort in production compilers such as this.

It remains for us to investigate further to determine whether our performance results can be easily improved upon, and to demonstrate similar speedups in a real system rather than a simulation.

References

[ARS94] L Augustsson, M Rittri and D Synek, “On Generating Unique Names”. *Journal of Functional Programming*, **4**(1), pp.117–123, January, 1994.

[HHLT97] K Hammond, CV Hall, H-W Loidl and PW Trinder, “Parallel Cost Centre Profiling”. In *1997 Glasgow Workshop on Functional Programming*, Ullapool, Scotland, September 1997.

[HLP95] K Hammond, H-W Loidl and AS Partridge, “Visualising Granularity in Parallel Programs: A Graphical Winnowing System for Haskell”. In *HPFC'95—Conf. on High Performance Functional Computing*, pp. 208-221, Denver, CO, April 10-12, 1995.

[HPW92] P Hudak, SL Peyton Jones, and PL Wadler (eds.), “Report on the Programming Language Haskell Version 1.2”. *ACM SIGPLAN Notices* **27**(5), May 1992.

[Hamm90] K Hammond, “Efficient Type Inference Using Monads”. In *Draft Proceedings, 1990 Glasgow FP Workshop*, Ullapool, Scotland, August 1990.

[John87] T Johnsson, *Compiling Lazy Functional Languages.* Ph.D. Thesis, Department of Computer Science, Chalmers University of Technology, G oteborg, Sweden, 1987.

[Juna97] S Junaidu *A Parallel Functional Language Compiler for Message Passing Multicomputers.* Forthcoming PhD thesis, School of Mathematical and Computational Sciences, St Andrews University, Scotland, 1997.

[NSvEP91] EGJMH Nöcker, JEW Smetsers, MCJD van Eekelen and MJ Plasmeijer, "Concurrent Clean". In *Proc. PARLE '91*, Springer-Verlag LNCS 506, pp. 202-219.

[Osth93] G Ostheimer, *Parallel Functional Programming for Message Passing Multiprocessors.* PhD Thesis, Department of Mathematical and Computational Sciences, St. Andrews University, Scotland, 1993.

[Peyt87] SL Peyton Jones, *The Implementation of Functional Programming Languages.* Prentice Hall International, 1987.

[Peyt92] SL Peyton Jones, "Implementing Functional Languages on Stock Hardware: the Spineless Tagless G-machine". *Journal of Functional Programming*, **2**(2), pp. 127-202, 1992.

[SaPe95] PM Sansom and SL Peyton Jones, "Time and Space Profiling for Non-strict Higher Order Functional Languages". In *Proc. 22nd ACM Symposium on Principles of Programming Languages*, San Francisco, Carlifornia, January 1995.

[Sked91] S Skedzielewski, "Sisal". In *Parallel Functional Languages and Compilers*, Frontier Series, ACM Press, New York 1991.

[THL$^+$96] PW Trinder, K Hammond, H-W Loidl, SL Peyton Jones, and J Wu, "A Case Study of Data-intensive Programs in Parallel Haskell". In *Proc. 1996 Glasgow Workshop on Functional Programming 1996*, Ullapool, Scotland, July 8-10.

[THM$^+$96] PW Trinder, K Hammond, JS Mattson Jr., AS Partridge and SL Peyton Jones, "GUM: A Portable Parallel Implementation of Haskell". In *PLDI '96 — Programming Languages Design and Implementation*, pp. 78-88, Philadelphia, PA, May 1996.

[THLP98] PW Trinder, K Hammond, HW Loidl and SL Peyton Jones "Algorithm + Strategy = Parallelism". To appear in *Journal of Functional Programming*, 1998.