Naira: A Parallel² Haskell Compiler

Sahalu Junaidu*, Antony Davie, and 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 graph-reducing parallel abstract machine with a strong dataflow influence. Unusually (perhaps even uniquely), Naira has itself been parallelised using state-of-the-art tools developed at Glasgow and St Andrews Universities. Thus Naira is a parallel, parallelising compiler in one. 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. We show that a modest but useful degree of parallelism can be achieved even for a distributed-memory machine. The simulation results have been verified on a network of distributed workstations using the GUM parallel implementation of Haskell.

1 Introduction

The Naira compiler was written to explore the problems of parallelising a modern functional language compiler [7]. It compiles from a substantial subset of Haskell [4] to a RISC-like target language that has been extended with special parallel constructs [10]. 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 Universities [3]. Our initial results are promising, indicating that acceptable speedups can be achieved within individual compiler passes, notably the type inference pass, and these results are confirmed by measurements on the GUM [17] system. There is, however, a large sequential component 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. Sections 2 and 3 respectively describe the parallelisation and analysis of the top-level pipeline and the individual passes. Section 4 describes related work. Finally Section 5 concludes.

2 The Top-Level Pipeline

We use a top-down parallelisation methodology, as in [18], starting with the top-level pipeline, then proceeding to parallelise successive pipeline stages. We

* Supported by the Federal Government of Nigeria under Federal Scholarship Scheme.
I also acknowledge the support of Islamic Relief, Makkah, Saudi Arabia.
concentrate on parallelising four main compiler passes — the pattern matcher, lambda lifter, type checker, and the optimiser. The parallelisation proceeded in a data-oriented fashion by annotating the intermediate data structures used to communicate results between the passes. We have experimented with lists and binary trees to represent the intermediate structures.

2.1 Structure of 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 is described elsewhere [7].

The two-way split after the lambda lifting pass indicates that the result of the lambda lifter can be piped simultaneously to both the type checker and the optimiser and that these latter two phases can proceed in parallel combining their results, using showModule, to produce the intermediate code.

2.2 Parallelising the Top-Level Pipeline: Version 1—Direct

Following the top-down parallelisation methodology outlined by Trinder et al [18], we start by parallelising the top-level pipeline of Naira before delving into the parallelisation of its main phases. Figure 2 shows the function, analyseModule, that implements this top-level pipeline. It is called immediately following symbol table construction, and forwards its arguments to each compiler pass as appropriate. The underlined definition, strat, is used to specify the parallel behaviour of analyseModule and is the only code that need to be added to ensure this parallelisation. This evaluation strategy, strat, sparks five parallel tasks using par (c `par` r creates a task to evaluate c, then continues executing r), one for each of the pipeline phases shown in Figure 1. The parList strategy applies its first argument (the rnf strategy