Automatic Parallelisation using Effect Tracking and Cost Analysis

Part II Computer Science Project Proposal

Dylan Moss, dm894

October 14, 2022

Project Originator:	Dylan Moss and Alan Mycroft
Project Supervisor:	Alan Mycroft
Director of Studies:	Ramsey Faragher
Overseers:	Robert Mullins and Marcelo Fiore

1 Introduction

Since 2006, improvements made to single-core processor performance have begun to stagnate as we begin to approach theoretical limits in advancements to transistor technology. This has led to an uptake in parallel processing, the act of splitting computations across multiple processors running simultaneously, in order to increase computational throughput by utilising more than one processor at once. However, there have been some limiting factors in the progress of parallel processing - the most prominent of which being the difficulties humans face when attempting to reason about concurrent systems (we assume that concurrency refers specifically to parallelism concurrency in this proposal). Human brains seem to be designed for sequential calculations [1] which makes it very hard for us to design and understand systems containing multiple threads of independent, interlocking calculations which are running in parallel. In order to try and aid programmers, modern programming languages often include concurrency primitives to reduce the cognitive load needed to design these sorts of systems, but despite this, many programmers still find concurrency difficult and often opt to avoid it when it is not strictly necessary. This may cause problems in the future as programmers will either have to embrace concurrency, despite its current flaws, or potentially miss out on the yearly performance improvements in computation which have been observed in previous decades.

In this project, I propose a new sequential programming language and a compiler, written in OCaml, which compiles this sequential language to parallelised code at the function level. This means that users can gain the benefits of parallel programming while avoiding many complexities surrounding concurrency. I will design a simple sequential programming language with Go-like syntax, which is provisionally dubbed Kautuka after the woven Indian "thread"[1]. I will compile Kautuka to the Go programming language, automatically parallelising the code during this compilation process using the Go's concurrency primitives: channels and goroutines[2].

When compiling from sequential Kautuka to parallel Go code, we expect that the parallel code acts in the same way as the sequential code, regardless of any non-determinism introduced by threads. Hence any instructions which may be affected by running in parallel, namely effects, should be statically analysed before compilation to ensure that effects in different threads do not interfere. And secondly, we expect that the compiler makes an attempt to produce parallel Go code which is computationally faster than the equivalent sequential Go code. Hence we introduce a static analysis: effect tracking and cost analysis into the compiler to ensure these expectations are met. As part of this project, I will implement a novel approach to cost analysis by designing a type system based upon tracking lower and upper bounds of values. My approach to cost analysis and effect tracking is explored in further detail in the next section.

[1]`https://en.wikipedia.org/wiki/Kautuka`

[2]Goroutines are lightweight threads managed by the Go runtime designed to run functions in parallel

2 Structure of the Project

There are five main components to this project:

- Translation: Building a compiler from Kautuka to sequential Go
- Effect tracking: Implementing effect tracking into Kautuka
- Cost analysis: Implementing cost analysis into Kautuka
- Automatic parallelisation: Extending the Translation step to compile Kautuka to parallelised Go based upon the previous two steps
- Evaluation: refer to Evaluation section

Translation

This involves designing the syntax and semantics of my custom language: Kautuka. Once the syntax and semantics have been formalised, I can use OCaml's lexing and parsing libraries to compile Kautuka to an AST, along with tests to ensure this has been carried out correctly. Finally, I can develop a tool which converts this AST to Go code, and write tests to check that the produced Go code is semantically equivalent to the original Kautuka program.

Effect tracking

I will extend the compiler by introducing a basic effect system based upon [2]. This will allow me to track the effects which may potentially be introduced by calls to user-defined functions, in order to ensure that parallel threads do not contain conflicting effects. The core project will only focus on the effects of mutating non-local state, but this can be extended to reasoning about console IO and file system effects in my project extensions. However, this project will not focus on pointers and aliases, and so we will assume that pointers of the same type will conflict unless we are explicitly told otherwise by the user.

Cost analysis

A large part of my project will be implementing an original approach to cost analysis by designing a custom type system which allows us to better estimate the time it takes to run functions. The type system will track the minimum and maximum value of variables[3] (henceforth referred to as bounds), and uses inference to update these bounds throughout the program. By measuring the execution time of standard functions for different input size, we can estimate the minimum and maximum execution time of all the standard function calls in a given program. The novelty of this idea is that tracking the lower and upper bounds of variables does not seem to have been used to statically estimate the execution times of functions before. However, tracking the intervals of integer variables seems to be well researched in the

[3]Value refers to the numerical value for int types and size for string and array types

field of scientific computing and so I intend to apply some of these ideas to the cost analysis in my project. When determining the runtime of all user-declared functions in the program, I intend to use [3] as a reference for dealing with conditions and loops. Hence, this analysis can be used to estimate how much time would be saved or lost by parallelising sets of functions, taking into consideration any costs incurred by introducing threads. The main tradeoff for my proposed approach is that Kautuka's type system is quite verbose and require a lot of extra type annotations from the user surrounding estimated bounds of variables. In my extensions, we can explore cost analysis in relation to higher-order and first-class functions, also using [3] as a reference, which will add significant complexity to this analysis.

Automatic parallelisation

Effect tracking allows us to identify combinations of function sets which do not contain any interfering effects. We can then use the cost analysis to determine the best way to schedule functions into separate threads in order to gain the most benefit; this involves determining whether parallelisation will give us any benefit at all compared to compiling functions purely sequentially. If we decide to carry out parallelisation, we will compile the program to Go; making use of goroutines to parallelise functions and channels to pass results out of successfully terminated goroutines (similar to *futures*[4]).

3 Evaluation

Quantitative

Evaluate the execution time of Kautuka compared to the equivalent sequential Go code. The examples will be handcrafted and well-suited to potential parallelisation optimisation as that is the intended use case of my program.

Qualitative

Ensure that my compiled language is semantically equivalent to the sequential Go code it is emulating, using the same corpus of examples as above.

4 Starting Point

I have limited prior experience in implementing lexers and parsers, and no prior experience in implementing effect systems or cost analysis. However I have studied Semantics of Programming Languages and Compiler Construction which should aid with building the lexer and parser stages of my compiler. Furthermore I have read ahead on the Optimising Compilers course regarding effect systems.

[4] `https://en.wikipedia.org/wiki/Futures_and_promises`

5 Success Criteria

For the project to be deemed a success, the following must be successfully completed:

1. Design the syntax for Kautuka and then develop a compiler from Kautuka to Go. Kautuka should support basic control flow, iterators and (non-first-class and non-higher-order) functions. My compiler will not include type checking, this stage will be carried out when the Go code is compiled further using the standard Go compiler.
2. Implement effect tracking for Kautuka. In the core project, this is limited to tracking modifications of non-local state.
3. Implement cost analysis for Kautuka. This involves creating a type system to track the minimum and maximum values of variables, and then measuring the runtime of standard functions based upon inputs of different sizes. Hence we can calculate the estimated minimum and maximum runtime of functions.
4. Parallelise the code during the compilation process based upon this analysis. We want to compile functions into parallel threads if there are no interfering effects and we deem there to be sufficient cost benefit.
5. Compare the runtime of a Kautuka program compiled to parallelised Go with the equivalent sequential Go code. And also evaluate whether a given Kautuka program is semantically equivalent to the Go code it is compiled to.

6 Possible Extensions

Possible extensions include:

1. Add IO support to Kautuka, namely interactions with console IO and the file system.
2. Add support for first-class functions and higher-order functions to Kautuka.
3. Add support for data structures, such as structs and enums, to Kautuka.
4. Seamlessly integrate Kautuka programs with existing Go codebases. This involves developing a tool that allows us to run a project containing both Kautuka and Go files using a combination of both my compiler and the standard Go compiler.

7 Timetable and Milestones

Weeks 1 to 2 (17 Oct 22 - 30 Oct 22)

Proposal Submitted

Learn best software development practices in OCaml and research tools available for creating the frontend of a compiler (e.g. lexer and parser tools).

Carry out research into effect tracking and cost analysis. For effect tracking, re-read the Optimising Compilers lectures regarding effects systems [2] and carry out further reading. For cost analysis, read [3] and research other papers for more inspiration on how best to implement this into my compiler.

Familiarise myself with Go syntax. Begin designing the syntax for Kautuka.

Begin writing the Introduction draft in dissertation.

Weeks 3 to 4 (31 Oct 22 - 13 Nov 22)

Finish designing the syntax for Kautuka.

Implement the lexer and parser for the language and write tests to verify that the output abstract syntax trees are correct. Write a compiler to convert the output abstract syntax tree into Go code, which at this point will closely resemble the source code we started with.

Finish writing the Introduction draft in dissertation.

Milestone: Given a Kautuka program, generate the corresponding abstract syntax tree

Weeks 5 to 6 (14 Nov 22 - 27 Nov 22)

Design and formalise effect tracking for my language. Then implement this into the compiler to generate lists of effects for all functions in the program. Add more tests to verify that this has been done correctly.

Begin writing the Implementation draft in dissertation (specifically adding my formalised Kautuka syntax and effect tracking inference rules).

Milestone: Given a Kautuka program, determine the list of effects for all of the functions

Weeks 7 to 8 (28 Nov 22 - 11 Dec 22)

End of Michaelmas Term - start of Christmas holidays.

Design and formalise my proposed bound-based type system. Then implement this into the compiler. Collect data surrounding the execution time for the standard functions and the overheads involved with creating threads.

Continue writing the Implementation draft in dissertation.

Milestone: Extend Kautuka with my bound-based type system, represent these bounds in the generated abstract syntax tree

Weeks 9 to 11 (12 Dec 22 - 1 Jan 22)

Finish collecting data surrounding execution times and create an algorithm to predict how long functions will take on inputs of different sizes. Using my type system and the collected data, now extend the compiler to estimate the runtime of the user's functions. Finally we can compile functions into different threads if there is sufficient cost benefit.

Start my evaluation.

Finish writing the Implementation draft in dissertation.

Take time off for Christmas

Milestone: Estimate the runtime of all user defined Kautuka functions. Compile Kautuka to parallelised Go based upon the analysis

Weeks 12 to 13 (2 Jan 22 - 15 Jan 22)

Add better type system inference and create further tests to ensure that the program is compiled correctly.

Evaluate the runtime of Kautuka compared to the semantically equivalent, sequential Go code.

Begin writing the Evaluation draft in dissertation.

Milestone: Finish core project implementation and pass all success criteria

Weeks 14 to 15 (16 Jan 22 - 29 Jan 22)

End of Christmas holidays - start of Lent Term.

Slack time to finish core project implementation.

Make a start on progress report.

Weeks 16 to 17 (30 Jan 22 - 12 Feb 22)

Finish progress report for the deadline Fri 3 Feb 2023. And create progress report presentation for the deadline Wed 9 Feb 2023.

Finish writing the Evaluation draft in dissertation.

Milestone: Submit progress report

Weeks 18 to 19 (13 Feb 22 - 26 Feb 22)

Make a start on extensions if time permits. Mainly focus on extensions 1 and 4.

Start Conclusion draft in dissertation.

Weeks 20 to 21 (27 Feb 22 - 12 Mar 22)

Continue with extensions if time permits. Now focus on the remaining extensions.

Finish Conclusion draft in dissertation.

Milestone: Finish dissertation draft

Weeks 22 to 23 (13 Mar 22 - 26 Mar 22)

Slack time to finish the dissertation draft.

Weeks 24 to 25 (27 Mar 22 - 9 Apr 22)

Submit draft of the dissertation to my supervisor and DoS for review by the deadline Fri 7 Apr 2023. Begin exam revision and review code repository in the meantime (ensure that my code is clear, readable and well structured).

Make improvements to the dissertation based upon this feedback.

Weeks 26 to 27 (10 Apr 22 - 23 Apr 22)

Finish improvements to the dissertation and submit to my supervisor a second time for final review.

Make last adjustments and focus on making the dissertation presentable and easy to read.

Weeks 28 - 29 (24 Apr 22 - 7 May 22)

Slack time for any extra additional evaluation and tests that may be discovered during final reviews of my dissertation, and start revision for exams.

Milestone (28 April 22): Submit Dissertation (2 weeks in advance of final deadline)

8 Resource Declaration

I will be using my personal laptop (Dell XPS 15 7590 - 2.6GHz i7, 16GB RAM) as my primary machine for software development. As a backup, I will use my secondary personal laptop and resources provided by SRCF (student run computing facility). I will continuously backup my code and dissertation with Git version control and carry out periodic backups to Google Drive.

References

[1] Mariano Sigman and Stanislas Dehaene. “Brain Mechanisms of Serial and Parallel Processing during Dual-Task Performance”. In: *Journal of Neuroscience* 28.30 (2008), pp. 7585–7598. ISSN: 0270-6474. DOI: 10.1523/JNEUROSCI.0948-08.2008. eprint: https://www.jneurosci.org/content/28/30/7585.full.pdf. URL: https://www.jneurosci.org/content/28/30/7585.

[2] Timothy M. Jones. *Optimising Compilers Lecture 13 Effect Systems*. 2022. URL: https://www.cl.cam.ac.uk/teaching/2122/OptComp/slides/lecture13.pdf (visited on 10/07/2022).

[3] Ben Wegbreit. “Mechanical Program Analysis”. In: *Commun. ACM* 18.9 (Sept. 1975), pp. 528–539. ISSN: 0001-0782. DOI: 10.1145/361002.361016. URL: https://doi.org/10.1145/361002.361016.