Automatic Parallelisation using Effect Tracking and Cost Analysis

Part II Computer Science Project Proposal

April 13, 2023

Project Originator:	**The Dissertation Author and Alan Mycroft**
Project Supervisor:	**Alan Mycroft**
Director of Studies:	**Ramsey Faragher**
Overseers:	**Robert Mullins and Marcelo Fiore**

Introduction

Since 2006, Moore's law has begun to slow down and we can no longer solely rely on improvements to transistor technology for computational performance gains year upon year. 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. However, there have been some limiting factors in the progress of parallel processing - the most prominent of which being the difficulty humans face reasoning about concurrent systems[1]. Human brains are designed for sequential calculations [citation pending] which makes it very hard for us to design and understand systems where there are multiple threads of independent, interlocking calculations 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 very difficult and often opt to avoid it when it isn't strictly necessary. This may potentially cause problems in the future as programmers will either have to embrace concurrency, despite its current flaws, or potentially miss out on the yearly code performance speed ups which have been previously observed in the last few decades.

In this project, I will propose a new sequential programming language and a compiler, written in OCaml, that compiles these sequential programs 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 compile my sequential language to the Go programming language, and automatically parallelise the code during this compilation process using one of Go's concurrency primitives; goroutines (lightweight threads managed by the Go runtime designed to run functions in parallel). I will design a simple sequential programming language with Go-like syntax, which will henceforth be provisionally dubbed Kautuka after the woven Indian "thread"[2]. The main tradeoff for my proposed solution is that Kautuka's type system will be very verbose and require a lot of extra type annotations from the user. I will implement effect tracking[3] into Kautuka and develop an original approach to cost analysis using a custom type system and execution-time measurements of standard functions.

[1]We assume that concurrency refers specifically to parallelism concurrency in this paper

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

[3]The act of tracking all the effects (such as side effects) that could potentially be introduced by a function

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.

Structure of the Project

There are 5 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: explained later in the Evaluation chapter

My proposed idea is that we want to parallelise sets of functions if 2 criteria are met:

The first being that functions in parallel sets don't contain any conflicting side effects which can be determined through effect tracking based upon [1].

And second being that there is sufficient cost benefit in parallelising the sets of functions to outweigh any overheads introduced by creating threads and higher cache miss rates. I have devised a new type system which allows us to track the minumum and maximum value[4] of variables (henceforth referred to as bounds), and inference to update these bounds throughout the program. We also measure the expected runtime of standard functions with different sized inputs, allowing us to estimate the minimum and maximum runtime of functions in our code using [2]. Based upon this analysis, we can determine if there is a way to schedule functions into independent threads to gain a speed improvement over pure sequential code while avoiding conflicting side effects.

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

Success Criteria

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

1. Design the syntax for my custom sequential language 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.
2. Implement effect tracking for Kautuka. In our core project, this is limited to tracking changes to non-local state.
3. Implement cost analysis for Kautuka. This involves creating a type system to track the minumum and maximum values of variables. And also measuring the runtime of standard functions based upon inputs of different sizes. Hence we can calculate the estimated minumum and maximum runtime of functions.
4. Parallelisation based upon this analysis. We want to compile functions into parallel threads if there are no interfering side effects and we deem there to be sufficient cost benefit.
5. Evaluate whether our Kautuka program compiled to parallelised Go gives any runtime improvements over the semantically equivalent non-parallelised Go code. And also evaluate whether our Kautuka program is semantically equivalent to the Go code it is being compiled to.

Possible Extensions

1. Add IO support to Kautuka, namely interactions with console IO and the file system. This involves extending the: translation, effect tracking and cost analysis stages.
2. Add support for first-class functions and higher-order functions to Kautuka. This primarily involves extending both the effect tracking and cost analysis stages.
3. Add support for the data structures structs and enums to Kautuka. This primarily involves extending the translation and cost analysis stages.
4. Seamlessly integrate Kautuka programs with existing Go codebases. This involves developing a tool that allows us to build our project containing both Kautuka and Go files into a project containing only Go files, which can then be executed by the standard Go compiler.
5. Give support for different levels of optimisation. Higher levels of optimisation have the potential for a greater improvement in code performance but are less reliable.

Evaluation

Quantitative

Evaluate the execution time of my custom language compared to the equivalent non-parallelised 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.

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 (eg. 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 [1] and carry out further reading. For cost analysis, read [2] 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

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: Complete my test suite for compilation to abstract syntax trees and Go. Carry out runtime evaluation

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.

Milestone: Finish core project implementation

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

Finish progress report for the deadline Fri 3 Feb 2023

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)

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

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

Make improvements to dissertation based upon this feedback.

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

Finish improvements to dissertation and submit to 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.

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

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] 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).

[2] 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`.