

A C Compiler for Stack Machines

Mark Shannon

Submitted for the degree of MSc by Research
University of York
Department of Computer Science
July 2006

Abstract

Compiler design for stack machines, in particular register allocation, is an under researched area. In this thesis I present a framework for analysing and developing register allocation techniques for stack machines. Using this framework, I analyse previous register allocation methods and develop two new algorithms for global register allocation, both of which outperform previous algorithms, and lay the groundwork for future enhancements. Finally I discuss how effective global register allocation for stack machines can influence the design of high performance stack architectures.

Included with this thesis is a portable C compiler for stack machines, which incorporates these new register allocation methods. The C compiler and associated tools (assembler, linker and simulator) are included in the CD.

Acknowledgements

This work was done as part of the UFO project, to develop a soft-core stack processor utilising the latest research on stack based architectures. Its aim was to produce a C compiler which would produce high quality code for the UFO machine.

The research was funded by the Department of Trade and Industry through its Next Wave initiative, as part of the AMADEUS project.

I would like to thank my supervisor Chris Bailey for his help and support throughout this project, and for kindling my interest in stack machines.

Thanks also Christopher Fraser and David Hanson for providing the lcc C compiler free of charge. Without it this project could not have been a success.

Finally I would like to thank Ally Price and Huibin Shi for proof reading and listening to my ideas and ramblings.

Previous Publication

Elements of this thesis, notably the stack-region framework, appeared in a paper presented at EuroForth 2006[25].

Contents

1	Introduction	11
1.1	Stack Machines	11
1.1.1	The Stack	11
1.1.2	Advantages of the Stack Machine	12
1.1.3	History	12
1.1.4	Current Stack Machines	13
1.1.5	Stack Machines, More RISC Than RISC?	13
	Common Features	13
	Differences	14
1.1.6	Performance of Stack Machines	14
	Pros	14
	Cons	14
	In Balance	15
1.2	The Stack	15
1.2.1	An Abstract Stack Machine	15
1.2.2	Stack Manipulation	16
1.3	Compilers	17
1.3.1	Compilers for Stack Machines	17
1.3.2	Register Allocation for Stack Machines	18
1.4	Context	19
1.4.1	The UFO Architecture	19
1.5	Goal	20
1.6	Contribution	20
1.7	Summary	20
2	The Stack	21
2.1	The Hardware Stack	21
2.2	Classifying Stack Machines by Their Data Stack	22
2.2.1	Views of the Stack	22
2.2.2	Stack Regions	23
	The Evaluation Region	23
	The Parameter Region	24
	The Local Region	24

	The Transfer Region	25
	The Rest of the Stack	25
2.2.3	Using the Regions to do Register Allocation	25
	The E-stack	26
	The P-stack	26
	The L-stack	26
	The X-stack	26
2.2.4	How the Logical Stack Regions Relate to the Real Stack	27
2.2.5	Edge-Sets	27
2.3	An Example	28
2.4	Summary	32
3	The compiler	33
3.1	The Choice of Compiler	33
	3.1.1 GCC vs LCC	33
	3.1.2 A Stack Machine Port of GCC – The Thor Microprocessor	34
3.2	LCC	35
	3.2.1 Initial Attempt	35
3.3	The Improved Version	38
	3.3.1 The Register Allocation Phase	38
	3.3.2 Producing the Flow-graph	38
	3.3.3 Optimisation	38
	Tree Flipping	38
3.4	Structure of the New Back-End	39
	3.4.1 Program flow	39
	3.4.2 Data Structures	39
	3.4.3 Infrastructure	40
	3.4.4 Optimisations	40
3.5	LCC Trees	40
	3.5.1 Representing Stack Manipulation Operators as LCC Tree- Nodes.	40
	3.5.2 The Semantics of the New Nodes	42
	Stack	42
	Copy	42
	3.5.3 Tuck	43
	3.5.4 Representing the Stack Operations.	43
	Drop	43
	Copy	43
	Rotate Up	43
	Rotate Down	44
	Tuck	44
	3.5.5 Change of Semantics of Root Nodes in the LCC forest .	44
3.6	Additional Annotations for Register Allocation	45
3.7	Tree Labelling	45

3.8	UTSA Machine Description	46
3.8.1	Solving the Address Problem.	47
3.8.2	Modifying the Machine Description	47
3.9	UFO Machine Description	48
3.9.1	Stores	48
3.10	Variadic Functions for Stack-Machines	49
3.10.1	The Calling Convention for Variadic Functions	49
3.10.2	Variadic Function Preamble	49
3.11	Summary	50
4	Register allocation	51
4.1	Koopman's Algorithm	51
4.1.1	Description	51
4.1.2	Analysis	51
4.1.3	Implementation	52
4.1.4	Analysis of Replacements Required	52
	Maintaining the Depth of the L-stack	52
	Initial Transformations	54
	Transformations After the First Node Has Been Transformed	55
	Transformations After the Second Node Has Been Transformed	55
	Transformations After Both Nodes Have Been Transformed	56
4.2	Bailey's Algorithm	56
4.2.1	Description	56
4.2.2	Analysis of Bailey's Algorithm	57
4.2.3	Implementation	58
4.3	A Global Register Allocator	58
4.3.1	An Optimistic Approach	59
4.3.2	A More Realistic Approach	59
4.3.3	A Global Approach	60
4.3.4	Outline Algorithm	61
4.3.5	Determining X-stacks	61
	Ordering of Variables.	61
	Heuristics	62
	Propagation of Preferred Values	62
4.3.6	Other Global Allocators	63
4.3.7	L-Stack Allocation	64
4.3.8	Chain Allocation	64
4.4	Final Allocation	66
4.5	Peephole Optimisation	68
4.5.1	Example	68
4.5.2	The <i>Any</i> Instruction	69
4.6	Summary	70

5	Results	71
5.1	Compiler Correctness	71
5.2	Benchmarks	71
5.3	The Baseline	72
5.4	The Simulator	72
5.5	Cost Models	73
5.5.1	The Models	73
5.6	Relative Results	74
5.6.1	Comparing the Two Global Allocators	75
5.6.2	Relative program size	77
5.7	Summary	77
6	Conclusions and Future Work	79
6.1	Performance	79
6.2	Limitations of the New Algorithms	79
6.2.1	Advanced Stack Architectures	80
6.3	Improving the Compiler	80
6.3.1	Improving the Global Register Allocator	80
6.3.2	A New Register Allocator	80
6.3.3	Stack Buffering	81
6.4	Advanced Stack Architectures	81
6.4.1	Pipelined Stack Architectures	81
6.4.2	Super-Scalar Stack Machines	83
6.5	Summary	83
A	The Abstract Stack Machine	85
B	Results	87
B.1	Dynamic Cycle Counts	88
B.2	Data memory accesses	89
C	LCC-S Manual	91
C.1	Introduction	91
C.2	Setting up lcc-s	91
C.3	Command line options	92
C.4	The structure of lcc-s	93
C.5	Implementations	94
C.6	The components	94
C.6.1	lcc	94
C.6.2	scc	95
C.6.3	peep	95
C.6.4	as-ufo	96
C.6.5	link-ufo	96

D	LCC Tree Nodes	99
E	Proposed Instruction Set for High Performance Stack Machine	101
E.1	Development path	101
	Out of order execution	101
	Embedded stack manipulations	102
E.2	Instruction Format	102
E.2.1	Categories	102
E.2.2	Arithmetic	102
E.2.3	Arithmetic Immediate	103
E.2.4	Logical	103
E.2.5	Fetch	103
E.2.6	Store	104
E.2.7	Local Fetch	104
E.2.8	Local Store	104
E.2.9	Jump	104
E.2.10	Long Jump	104
E.2.11	BranchT	104
E.2.12	BranchF	104
E.2.13	Test	104
E.2.14	Literal	105
E.2.15	Special	105
F	Quantitative Comparison of Global Allocators	107
G	Source Code	111
	Bibliography	111
	Index	114

List of Tables

1.1	Traditional Forth operations	16
1.2	UFO operations	16
1.3	Possible sets of stack manipulation operators	17
2.1	Stack Classification	22
3.1	Comparison of GCC and lcc	34
3.2	Partial machine description	45
3.3	Extended machine description	46
4.1	Koopman's Algorithm — Initial transformations	54
4.2	Transformations with first node transformed	55
4.3	Transformations with the second node transformed	56
4.4	Transformation with both nodes transformed	57
4.5	Tree transformations — Bailey's Algorithm	59
5.1	The benchmarks	72
5.2	Instruction Costs	73
B.1	Flat model	88
B.2	Harvard model	88
B.3	UFO model	88
B.4	UFO slow-memory model	88
B.5	Pipelined model	88
B.6	Stack mapped model	88
B.7	Data memory accesses	89

List of Figures

2.1	Adding the second and third items on the stack.	22
2.2	Evaluation of expression $y = a * b + 4$	23
2.3	Evaluation of expression $f(x+y)$	24
2.4	Using the l-stack when evaluating $y = a[x] + 4$	25
2.5	C Factorial Function	28
2.6	Determining the edge-sets	29
2.7	Stack profile	30
2.8	Assembly listings	31
3.1	Compiler overview	36
3.2	Translator	37
3.3	Compiler back end	37
3.4	lcc tree for $y = a[x] + 4$	40
3.5	lcc forest for C source in Figure 2.5	41
3.6	<i>drop3</i> as lcc tree	43
3.7	<i>copy3</i> as lcc tree	43
3.8	<i>rot3</i> as lcc tree	44
3.9	<i>rrot3</i> as lcc tree	44
3.10	<i>tuck3</i> as lcc tree	44
3.11	Labelled tree for $y = a[x] + 4$	45
3.12	Relabelled tree for $y = a[x] + 4$	46
4.1	Comparison of allocation by pair and by chain.	67
5.1	Relative performance for the Flat model	74
5.2	Relative performance for the UFO model	75
5.3	Relative performance for the UFO with slower memory model	76
5.4	Relative performance for the Pipelined model	76
5.5	Relative performance for the Mapped model	77
5.6	Relative program size (smaller is better)	78
F.1	doubleloop.c	107
F.2	twister.c	108
F.3	Cycles for UFO model	108
F.4	Relative speed	109

List of Algorithms

1	Koopman's Algorithm	52
2	Implementation of Koopman's Algorithm	53
3	Bailey's Algorithm	57
4	Bailey's Algorithm with x-stack	58
5	Determining the x-stack	63
6	Propagation of x-stack candidates	63
7	L-Stack Allocation	65
8	Chain Allocation	65
9	Matching the L-Stack and X-Stack	66
10	Cost function	68
11	Proposed Algorithm	81

Chapter 1

Introduction

This thesis aims to demonstrate that compilers can be made which produce code well tailored to stack machines, taking advantage of, rather than being confounded by, their peculiar architecture.

This chapter covers the necessary background, explaining stack machines and laying out the problems in implementing quality compilers for them.

1.1 Stack Machines

Stack machines have an elegant architecture which would seem to offer a lot to computer science, yet the stack machine is widely viewed as outdated and slow. Since there have been almost no mainstream stack machines since the 1970s this is hardly surprising, but is unfortunate as the stack machine does have a lot of potential. The usual view is that stack machines are slow because they must move a larger amount of data between the processor and memory than a standard von Neumann architecture[17]. Obviously, this argument only holds true if stack machines do indeed have to move more data to and from memory than other architectures. This dissertation aims to show that this is not necessarily true and that a well designed compiler for stack machines can reduce this surplus data movement significantly, by making full and efficient use of the stack.

1.1.1 The Stack

The defining feature of a stack machine is the arrangement of its registers. These are arranged in a first-in-last-out stack, rather than the conventional arrangement of a fixed array. This first-in-last-out arrangement could be highly restrictive, so all stack machines are provided with the ability to access a number of registers below the first. Usually this is in the form of stack manipulation instructions, such as *swap* or *rotate* which allow values in other registers to be moved to the top of the stack. The number of additional

accessible registers varies. For example the UFO machine[23], which is the target of the compiler discussed in this thesis, has three accessible registers below the top of stack but other machines could have considerably more. In most stack machines, the arithmetic logic unit(ALU) and memory management unit operate strictly at the top of the stack; all ALU operands and memory writes are popped from the top of the stack and all ALU results and memory loads are pushed to the top of the stack. Thus stack machines are also called implicit-addressing machines since no explicit reference to operands are required. This stack is usually implemented in hardware, and although physically finite, it is conceptually infinite, with hardware or the operating system managing the movement of the lower parts of the stack to and from memory.

1.1.2 Advantages of the Stack Machine

Despite their perceived disadvantages, stack machines have enjoyed continued, if not mainstream, interest. This is because stack machines do have a number of advantages: low power requirements; small size and cost; and fast, highly predictable interrupt performance. They also have small program sizes, reducing memory cost and band-width requirements[4]. All such features are desirable for any processor, but do make stack machines of special interest for embedded, real-time and low-power systems.

While it is unlikely that any future generation of super-computers will be stack-based, the vast majority of processors manufactured are used in low-power and embedded systems, and when limited resources are considered the stack machine looks a promising alternative to the norm.

1.1.3 History

Stack machines have been around since the 1960s, the Burroughs B5000 and successors[22] being the first widely used stack-based processors, and were once considered one of the fastest architectures, as it was thought that their design simplified compiler construction. Subsequently improvements in compiler design meant that memory-memory architectures came to the fore. At the time memory-to-memory architectures were often considered superior to register-to-memory or register-to-register architectures, as memories were able to keep up with processors at the time, as suggested by Myers[21]. However it was the development of the IBM 360[3] which sidelined the stack machine. As processor speeds increased, the need for processors to use on-chip registers as their primary data-handling mechanism became apparent and the RISC¹

¹RISC: Reduced Instruction Set Computer. RISC machines have only a few addressing modes and notably cannot move data to or from memory at the same time as performing operations on that data. This simpler design freed silicon resources to be spent on improving performance.

revolution came into being. Despite these changes in technology, resulting in a higher processor to memory performance ratio, that could have brought stack-processors back to the fore, the CISC² vs RISC debate dominated the field of processor design.

The modern, post 1980, hardware stack machine has been traditionally tied to Forth[7], an extremely compact but somewhat idiosyncratic language, most implementations having been hardware implementations of the Forth virtual machine. The Forth virtual machine is a dual stack architecture with two stacks, one for return addresses, to reduce procedure call overhead, and one for data.

1.1.4 Current Stack Machines

As a refinement of the stack machine's set of stack manipulators, which have developed in a haphazard fashion following Forth convention, newer designs of stack machines[24] have a set of registers accessible to a slightly greater depth, typically four, and a fuller set of manipulators. However none of these processor are currently available. All current silicon stack machines are implementations of the Java Virtual Machine, which has a sparse set of stack manipulations chosen to complement the Javac compiler. The set of stack operators provided by the JVM is not suitable for a general purpose stack machine. There are a number of FPGA³ based stack machines available.

1.1.5 Stack Machines, More RISC Than RISC?

The guiding philosophy of the RISC revolution was to have a simple, consistent instruction set that could be executed quickly and facilitate hardware optimisations. Stack machines also have simple instruction sets and the instruction set has more orthogonality, since register addressing and computation are in separate instructions.

Common Features

- Simple instructions - All instructions correspond to a single simple operation within the processor.
- Consistent instruction format - The instructions are usually of a fixed length, with a standard layout.

²CISC: Complex Instruction Set Computer. CISC machines have instructions capable of performing several simple operations at once, but at the expense of complex processor design and performance reductions due to the use of microcoding.

³Field Programmable Gate Array

Differences

The primary difference between RISC and stack machines is in the arrangement of their on-chip registers. RISCs arrange their registers as a fixed size array, whereas stack machines arrange theirs as a stack.

Stack machines make the simple instruction format of RISC even simpler, with fully separate addressing and computation. All RISC ALU instructions, such as ‘add’ include addressing information, but stack machines instruction do not; the stack machine instruction to add two numbers is simply ‘add’.

RISC machine: $r1 \leftarrow r2 + r3$

Stack machine: `add`

The stack architecture evaluates expressions without side effects, that is, it leaves no non-result values in any register. Conventional machine instructions, such as the “add” instruction, have the side effect of leaving a value in a register. Leaving a value in a register is often a useful side effect, but it can be regarded as a side effect nonetheless.

1.1.6 Performance of Stack Machines

Stack machines have a reputation for being slow. This is a sufficiently widely held opinion that it is self-perpetuating. The sole formal comparison of RISC vs stack machine performance was inconclusive[28], and is rather out of date however it is possible to analyse how the differences between stack machines and RISC might affect performance.

Pros

- Higher clock speeds: The stack processor is a simpler architecture and should be able to run with a faster clock, at least no slower.
- Low procedure call overheads: Stack machines have low procedure call penalties, since no registers need saving to memory across procedure calls.
- Fast interrupt handling: Interrupt routines can execute immediately as hardware takes care of the stack management.
- Smaller programs: Stack machine instructions have a much shorter encoding, often one byte per instruction, meaning that stack machine programs are generally smaller. Although the instruction count is usually slightly higher, overall, the code length is significantly shorter.

Cons

- Higher instruction count: The separation of addressing and computation in stack machine instructions, means that more of them are required to do anything.

- Increased data memory traffic: Due to a smaller number of accessible registers⁴, although this can be offset by reduced instruction fetches.
- No instruction level parallelism: RISC machines have made considerable increases through pipelining and super-scalar techniques. Research into whether stack architectures can match these increases is at a very early stage.

In Balance

Whether the pros outweigh the cons or vice versa remains to be seen. Even if it cannot match the RISC machine for pure speed, the stack machine remains a viable architecture, due to its smaller size and lower cost and power requirements. This thesis will demonstrate that good register allocation for stack machines can reduce memory traffic as effectively as register allocation for any other architecture.

1.2 The Stack

1.2.1 An Abstract Stack Machine

In order to discuss stack machines, an ‘ideal’ stack machine and its instruction set needs to be defined. It is only the stack manipulation capabilities of this machine that are of interest in this section. Stack manipulation instructions take the form *nameX* where *name* defines the class of the manipulator, and *X* is the index of the element of the stack to be manipulated. The index of the top of stack is one, so the *drop2* instruction removes the second element on the stack. The abstract stack machine used in this thesis has five classes of stack manipulation instructions, these are:

Drop — Eliminates the X^{th} item from the stack.

Copy — Duplicates the X^{th} item, pushing the duplicate on to the top of the stack.

Rot — Rotates the X^{th} item to the top of the stack, pushing the intermediate items down by one.

Rrot — Rotates the top of the stack to the X^{th} position, pushing the intermediate items up by one.

Tuck — Duplicates the top of the stack, inserting the duplicate into the X^{th} position, pushing the intermediate items up by one. This means that the duplicate actually ends up at the $(X + 1)^{th}$ position.

The abstract machine instruction set is listed more fully in Appendix A.

⁴Typically 4, when stack scheduling is used, whereas small scale RISC machines have considerably more.

Table 1.1: Traditional Forth operations

Depth	Drop	Retrieve	Save	Rotate (up)	Rotate (down)
1	drop	dup	dup	nop	nop
2	nip	over	tuck	swap	swap
3				rot	

1.2.2 Stack Manipulation

As noted earlier, most stack machines provide instructions to manipulate the upper region of the stack. These instructions have typically followed Forth usage and are shown in table 1.1. The greyed out operations are redundant. As can be seen there are gaps in the table for the Forth operators, and access is quite shallow. Traditionally the stack manipulations in Forth have been done by the programmer and too many stack manipulators would merely cause confusion. However, when register allocation is done within a compiler it becomes advantageous to have both more regular stack operators and more available registers. Bailey[4] categorises these stack manipulations into five classes: Drop, Retrieve, Save and two categories of Rotate: Up and Down. The UFO processor (see Section 1.4.1) implements all five categories to a depth of 4; as shown in table 1.2. This regularity allows effective register allocation in the compiler.

Table 1.2: UFO operations

Depth	Drop	Retrieve	Save	Rotate (up)	Rotate (down)
1	drop1	copy1	copy1	nop	nop
2	drop2	copy2	tuck2	swap	swap
3	drop3	copy3	tuck3	rot3	rrot3
4	drop4	copy4	tuck4	rot4	rrot4

With this stack accessing scheme the number of stack manipulation operators required for a depth of N equally accessible registers is $5N - 4$, as shown in the right hand column of table 1.3. This could become a problem if deep access into the stack, say $N = 8, 12$ or even 16 were required. The number of stack manipulators could become prohibitively large for larger values of N , taking up an excessive fraction of the instruction set and potentially complicating the physical design. Not all these operators are required however. The orthogonal set of operators to cover this space consists of *drop1*, *tuckX* $X \leq N$ and *rotX* $X \leq N, X \neq 1$. This only requires $2N$ stack manipulators, as shown in the left hand column of table 1.3. The instruction *dropN* can be emulated with *rotN*

drop1, *rrotN* can be emulated with *tuckN drop* and *copyN* can be emulated with *rotN tuckN*. Although this only requires $2N$ stack manipulators, in order to do register allocation a compiler might have to insert large numbers of stack manipulations in the generated code. As a compromise between these two extremes, an intermediate range of operators is provided by the set of operators, *rotX*, *rrotX*, *copyX* and *drop1*, as shown in the centre column of table 1.3. This would allow most of the convenience provided by the complete set of operators, but with fewer operators.

Table 1.3: Possible sets of stack manipulation operators

Manipulator group	Orthogonal	Intermediate	Complete
Drop	1	1	N
Retrieve	0	N	N
Save	N	0	N-1
Rotate (up)	N-1	N-1	N-1
Rotate (down)	0	N-2	N-2
Total	2N	3N-2	5N-4

1.3 Compilers

1.3.1 Compilers for Stack Machines

In order for any architecture to achieve good performance it needs good compilers. Since the C language is the standard for embedded systems and the standard for measuring the performance of machines, a good C compiler is a necessity for any machine to be taken seriously. Stack machines have lacked good compilers for any language except Forth. As a result of this Forth bias, C compilers seem to have been regarded almost as an afterthought. They have most noticeably lacked good register allocation, which has led to poor performance. Sadly research into compilers for stack machines has been lacking, which given the general perception of stack machines being out of date, or tied to Forth, or both of these, is hardly surprising. This is not to say that compilers for stack machines cannot be as good as for any other architecture, as Koopman says[1]

Stack compilers aren't currently very efficient - but that's because
no-one has tried very hard

This thesis shows that a quality C compiler can be produced that allows stack machines to run C programs efficiently.

1.3.2 Register Allocation for Stack Machines

Register allocation for stack machines, also known as ‘stack scheduling’, is a technique for maintaining commonly used variables on the stack. Instead of loading a value from memory twice or more, it is loaded once and duplicated before use, so that a copy remains for subsequent use. While this somewhat pollutes the conceptual purity of the stack machine model it does improve performance. The simple stack scheduling first proposed by Koopman[18] has subsequently been refined[5, 20], but the refinements have not been included in a compiler.

Register allocation has been the focus of a large amount of research. However the vast majority of it has been directed towards conventional architectures, with only a handful of papers addressing stack machines.

Global register allocation for conventional machines can be loosely categorised as graph-colouring or region based. Graph-colouring attempts to fit as many variables as possible into registers at any given point. Linear programming techniques have been used but the aim is the same. This a standard technique used across a wide range of compilers. Region based allocators select regions most likely to be important, either by profiling or by static analysis, and attempt to minimise some cost, usually the number of memory accesses, in that region. Less important regions are dealt with subsequently. The Trimaran[9] compiler uses this approach.

The problem of register allocation for a stack machine is fundamentally different from the usual one involving a fixed array of registers. The stack’s infinite capacity but finite accessibility, along with its constantly fluctuating depth, require a different approach. Although the stack is effectively infinite, as hardware manages stack overflow to memory, only a finite number of registers near the top of the stack can be accessed directly. The aim of register allocators is to take the variables, both those in the source code and those created by the compiler front-end, and to store as many of them in registers, as possible. Obviously having more registers makes this easier, but it is the job of a good compiler to make the best use of whatever registers are available.

The first work on register allocation for stack machines was done by Philip Koopman[18], although he uses the term ‘stack-scheduling’. His algorithm pairs up uses of variables and, by retaining the value on the stack over the lifetime of those pairs, eliminates the second memory access. This is done repeatedly, but only within basic blocks⁵. This intra-block algorithm was successful at removing a significant number of temporary variables within those blocks, but was unable to handle variables with longer lifetimes. This algorithm was later shown[20] to be near optimal, within the context of a single basic block. Bailey[5] extended this, essentially local, approach to allow

⁵A basic block is a piece of code which has one entry point, at the beginning, and one exit point, at the end. That is, it is a sequence of instructions that must be executed, in order, from start to finish.

allocation across blocks. Bailey's Algorithm tries to find usage pairs that span blocks, and then replaces the memory accesses with stack manipulations. Although this approach spans basic blocks it is still essentially local as it does not directly consider variables across their whole lifetimes, merely in the blocks in which they are used. It is limited to variables whose liveness does not change between blocks, which means that a number of variables, such as loop indices and some variables used in conditional statements, cannot be dealt with. As demonstrated in Chapter 5 this can cause a notable loss of performance for some programs.

In order to produce a better algorithm and understand the issues involved, I have developed a framework which divides the stack into logical partitions and simplifies the analysis and development of stack-based register allocation methods.

Register allocation methods have been developed using this framework that can reduce local variable traffic significantly in most cases and to zero in a few cases.

The compiler developed includes the algorithms covered in this thesis. It also includes a peephole optimiser to minimise stack manipulations generated by the register allocation algorithms. For comparative purposes both Koopman's and Bailey's algorithms were also implemented.

1.4 Context

This work was done as part of the development of a compiler for a new stack machine, called the UFO[23].

1.4.1 The UFO Architecture

The UFO is a dual-stack machine. The data stack has four accessible registers at the top of the stack which can be permuted by various instructions; copy, drop, tuck and rotate. Underneath these is the stack buffer which contains the mid part of the stack. The lower part of the stack resides in memory, with hardware managing the movement of data to and from the buffer. This paper, however, deals with a generalised form of this architecture, with any number of accessible registers from two upwards, and shows that increasing the number of usable registers is as valuable in a stack machine as any other architecture. The UFO design is developed from a research stack architecture, the UTSA (University of Teeside Stack Architecture). As a simulator for the UTSA was readily available, most of the experimentation in register allocation was done with the UTSA version of the compiler.

1.5 Goal

It is the ultimate goal of the work behind this thesis to demonstrate that a stack machine can be as fast as, or even faster than, a RISC machine. Obviously, even if this goal can be achieved, it cannot be achieved in a single Master level thesis. Equally obviously this cannot be done with just software, and it cannot be done overnight, as modern RISC machines have many performance enhancements over the original RISC machines of the 1980s. However with a good compiler it might be possible to demonstrate that a stack machine for a low resource system, such as a soft core processor on an FPGA, can be an equal or better alternative to a RISC machine.

1.6 Contribution

The thesis makes two main contributions:

Stack regions – A framework for analysing and developing register allocation algorithms for stack machines.

Stack-specific register allocation algorithms – Two new register allocation algorithms for stack machines, both of which outperform the previous best.

In addition to this thesis, a fully working and tested optimising compiler for stack machines is provided. This compiler is targeted specifically at stack machines and is easily portable within that class of machines.

1.7 Summary

Stack machines have a number of attractive features, but research on them has been lacking. It has been assumed that stack machines are incapable of high performance, even though analysis shows that this is not necessarily so. A crucial component in the search for higher performance is the compiler and compilers for stack machines have received little research.

Chapter 2

The Stack

In this chapter, the stack is analysed in detail. After looking at the stack from various perspectives, a novel theoretical framework for studying the stack is presented. Finally, the use of this framework to do register allocation is outlined.

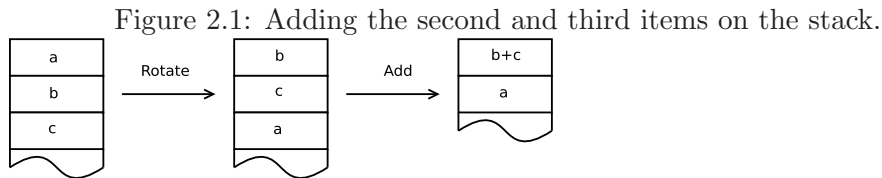
To design a compiler for a stack machine, most of the conventional techniques for compiler design can be reused, with the exception of register allocation and, to a lesser extent, instruction scheduling. Register allocation for stack machines is fundamentally different from that for conventional architectures, due the arrangement of the registers. This chapter looks at the stack from a few different perspectives and describe a way of analysing the stack that is suitable for classifying and designing register allocation methods for stack machines. In its strictest form a stack has only one accessible register and values can only be pushed on to the top of the stack and popped off again, a pure last-in-first-out arrangement, and whilst this form of stack is simple to implement and useful for some applications, it is a very limiting way to organise the registers of a general purpose machine. As a consequence, all real stack machines have stacks that allow more flexible access to the stack.

2.1 The Hardware Stack

The hardware data stack in a real machine is similar to a first-in-last-out stack, but with additional operations. All operands for arithmetic operations are popped off the top of the stack and the result pushed back to the stack. Likewise, all memory reads are pushed to the top of the stack and all the data for all memory write operations are popped from the top of the stack. In order to increase the flexibility of the machine, the stack also has a finite number of locations immediately below the the top, which can be moved or copied, to or from, the top. For example we can add the second and third items on the stack, but to do so we have to rotate the top item down to third position, leaving the items which were second and third accessible for arithmetic operations.

Machine	Depth	Manipulators
B5000	3	4
Forth Virtual Machine	3	9
Java Virtual Machine	4	9
UFO	4	16

They can now be added together, see Figure 2.1. Although this may seem an awkward way of doing things, it saves a lot of memory accesses compared with a pure first-in-last-out stack. Also, there is no reason why the operations cannot be done in parallel with the right hardware[16].



2.2 Classifying Stack Machines by Their Data Stack

Any processor can be classified by the number and purpose of its stacks [24]. The term ‘stack machine’ is taken to mean a machine with two hardware stacks, one for data and one for program control. Stack machines can be further classified by the parameters of its data stack. One way to classify the data stack is by the number of accessible registers and the by number of instructions available for manipulating the stack. The original stack machine, the Burrough B5000 had only a few stack manipulation instructions. A range of stack machines are listed in table 2.1.

2.2.1 Views of the Stack

It is possible to view the stack from a number of different perspectives. For example, when viewed from a hardware perspective the stack consists of a number of discrete registers, a mechanism for moving values between these registers, a buffer, and some logic to control movement of data between the buffer and memory. This perspective is irrelevant to the programmer, who sees a first-in first-out stack, of potentially infinite depth, enhanced with a number of instructions allowing access to a few values directly below the top of stack. In order to do register allocation, neither of these points of view is really that useful and a different, more structured view is required.

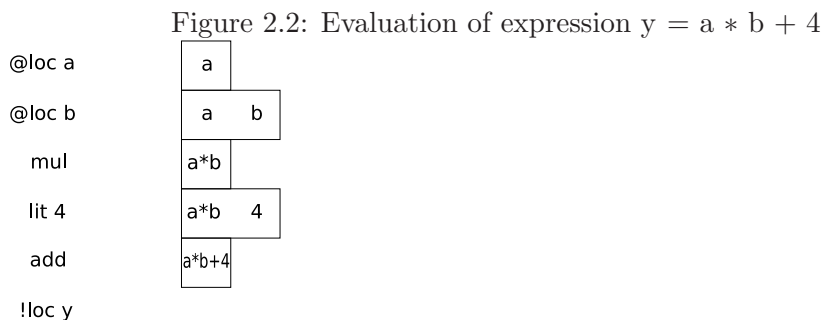
2.2.2 Stack Regions

To aid analysis of the stack with regard to register allocation, the perspective chosen divides the stack into a number of regions. These regions are abstract, having no direct relation to the hardware and exist solely to assist our thinking. The boundaries between these regions can be moved without any real operation taking place, but only at well defined points and in well defined ways. This compiler oriented view of the stack consists of five regions. Starting from the top, these are:

- The evaluation region (e-stack)
- The parameter region (p-stack)
- The local region (l-stack)
- The transfer region (x-stack)
- The remainder of the stack, included for completeness.

The Evaluation Region

The evaluation region, or *e-stack*, is the part of the stack that is used for the evaluation of expressions. It is defined to be empty except during the evaluation of expressions when it will hold any intermediate sub-expressions¹.



The abstract stack assembly language is explained in appendix A.

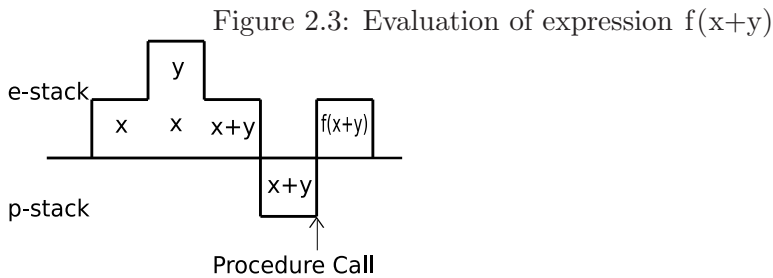
For example, consider the evaluation of the expression $y = a * b + 4$ in Figure 2.2. Initially the e-stack is empty. Then a and b are pushed onto the stack. Then they are multiplied together. The constant 4 is then pushed to the stack and these values are added together to yield $a * b + 4$, the sole value on the e-stack. Finally the value on top of the stack is stored back into y , leaving the e-stack empty. The e-stack can be regarded as immutable from the point

¹This is by definition, any 'expression' that does not fulfil these criteria should be broken down into its constituent parts, possibly creating temporary variables if needed. The conditional expression in C is an example of such a compound expression.

of view of register allocation. Any compiler optimisations which would alter the e-stack, such as common sub-expression elimination, are presumed to have occurred before register allocation.

The Parameter Region

The parameter region, or *p-stack*, is used to store parameters for procedure calls. It may have values in it at any point, both in basic blocks and across the boundaries between blocks. It is emptied by any procedure call². The p-stack is for *outgoing* parameters only; any value returned by a procedure is left on the e-stack and incoming parameters are placed in the x-stack at the point of procedure entry. Although parameters are kept on the p-stack before a procedure call, they are evaluated on the e-stack, like any other expression. Only when evaluation of the parameter is completed is it moved to the p-stack. This is illustrated in Figure 2.3. Note that this movement may be entirely abstract; no physical operation need occur, indeed in this example none is required. The p-stack is, like the e-stack, fixed during register allocation.



The e-stack and p-stack are the parts of the stack that would be used by a compiler that did no register allocation. Indeed the stack use of the JVM[19] code produced by most Java[13] compilers corresponds to the evaluation region.

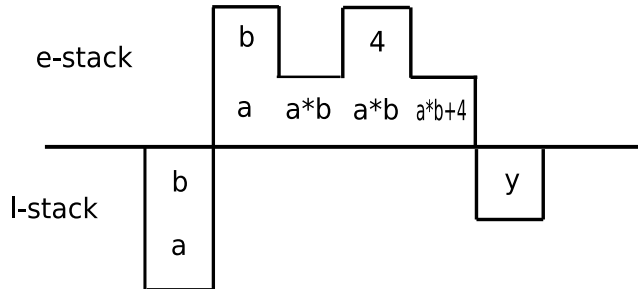
The Local Region

The local region, or *l-stack*, is the region directly below the p-stack. The l-stack is used for register allocation. It is always empty at the beginning and end of any basic block, but may contain values between expressions. In the earlier example, no mention was made of where either **a** or **b** came from or where **y** is stored. They could be stored in memory but it is better to keep values in machine registers whenever possible. So let us assume that in the earlier example, $y = a * b + 4$, **a** and **b** are stored in the l-stack, as shown in Figure 2.4. To move **a** and **b** from the l-stack to the e-stack, we can copy

²This presumes that all procedure calls have been moved to the top level of any expression trees, as they are by most compilers. If this were not the case then procedure calls would remove only their own parameters from the top of the p-stack, leaving the remainder.

them, thus retaining the value on the l-stack, or move them to the e-stack from the l-stack. In this example, *b* might be stored at the top of the l-stack, with *a* directly below it; to move them to the e-stack requires no actual move instruction, merely moving the logical boundary between the e-stack and l-stack. Likewise storing the result, *y*, into the l-stack is a virtual operation.

Figure 2.4: Using the l-stack when evaluating $y = a[x] + 4$



The Transfer Region

The transfer region or *x-stack* is used to store values both in basic blocks and on edges in the flow graph. The x-stack need only be empty at procedure exit. It holds the incoming parameters at procedure entry. Values may only be moved between the x-stack and l-stack at the beginning or end of basic blocks, and they must be moved en bloc and retain their order. Note that values cannot be moved directly between the x-stack and the e-stack, they must go through the l-stack. Since all ‘movement’ between the l-stack and x-stack is virtual it might seem that they are the same, but the distinction between the two is useful; the x-stack must be determined globally, while the l-stack can be determined locally. This separation allows a clear distinction between the different phases of allocation and simplifies the analysis.

The Rest of the Stack

The remainder of the stack or sub-stack, consists of the e-stack, p-stack, l-stack and x-stack of enclosing procedures. It is out-of-bounds for the current procedure.

2.2.3 Using the Regions to do Register Allocation

Register allocation for stack machines is complicated by the moveable nature of the stack. A value may be stored in one register, yet be in a different one when it is retrieved. This complication can be sidestepped by regarding the boundary between the p- and l-stacks as the fixed point of the stack. Values stored in the l-stack do not move relative to this boundary which simplifies

register allocation. The ability of the hardware to reach a point in the l-stack depends on the height of the combined e- and p-stacks above it, but it is fixed during register allocation, meaning it needs to be calculated only the once at the start of register allocation.

The E-stack

The e-stack is unchanged during optimisations. Optimisation changes whether values are moved to the e-stack by reading from memory or by lifting from a lower stack region, but the e-stack itself is unchanged.

The P-stack

For a number of register allocation operations, there is no distinction between the e-stack and p-stack and they can be treated as one region, although the distinction can be useful. For certain optimisations, which are localised and whose scopes do not cross procedure calls, the p-stack and l-stack can be merged increasing the usable part of the stack. For the register allocations method discussed later, which are global in scope and can cross procedure calls, the p-stack is treated essentially the same as the e-stack.

The L-stack

The l-stack is the most important region from a register allocation point of view. All intra-block optimisations operate on this region. Code is improved by retaining variables in the l-stack rather than storing them in memory. Variables must be fetched to the l-stack at the beginning of each basic block and, if they have been altered, restored before the end of the block, since by definition, the l-stack must be empty at the beginning and end of blocks.

The X-stack

The x-stack allows code to be improved across basic block boundaries. The division between the l-stack and x-stack is entirely notional; no actual instructions are inserted to move values from one to the other. Instead the upper portion, or all, of the x-stack forms the l-stack at the beginning of a basic block. Conversely, the l-stack forms the upper portion, or all, of the x-stack at the end of the basic block. Since the e-stack and l-stack are both empty between basic blocks, the p-stack and x-stack represent the complete stack which is legally accessible to the current procedure at those points. This makes the x-stack the critical part of the stack with regards to global register allocation. Code improvements using the x-stack can eliminate local memory accesses entirely by retaining variables on the stack for their entire lifetime.

2.2.4 How the Logical Stack Regions Relate to the Real Stack

The logical stack regions can be of arbitrary depth regardless of the hardware constraints of the real stack. However, the usability of the l-stack and x-stack depends on the capabilities of the hardware. Our real stack machine has a number of stack manipulation instructions which allow it to access values up to a fixed depth below the top of the stack. However, as the e-stack and p-stack vary in depth, the possible reach into the l-stack also varies. Variables that lie below that depth are unreachable at that point, but, as they may have been reachable earlier and become reachable later, they can still be useful. We assume that the hardware allows uniform access to a fixed number of registers, so if we can copy from the n^{th} register we can also store to it and rotate through it.

2.2.5 Edge-Sets

The second part of the analytical framework relates to flow-control. In order that programs behave in a sensible way, the stack must be in some predictable state when program flow moves from one block to another. This means for all the successor edges of any given block, the state of the x-stack must be identical. Likewise, it means that for all the predecessor edges for any given block, the state of the x-stack must be the same. The set of edges for which the stack must contain the same variables is called an *edge-set* and is defined as follows:

For any edge e and edge-set S :

$$\text{if } e \in S \text{ then } \forall S_i \neq S, e \notin S_i$$

For any two edges, $e_1 \in S_1$, $e_2 \in S_2$ and any block b :

$$\text{if } b \in \text{predecessor}(e_1) \text{ and } b \in \text{predecessor}(e_2) \text{ then } S_1 = S_2$$

$$\text{if } b \in \text{successor}(e_1) \text{ and } b \in \text{successor}(e_2) \text{ then } S_1 = S_2$$

In other words, an edge can only belong to one edge-set and if two edges share either a predecessor or successor node (block) they must be in the same edge-set. The state of the x-stack is the same for every edge in an edge-set.

Terminology

The term *child* will be used to refer to the relation between an edge-set and a block, where the block is the successor of at least one edge in that edge-set.

The term *parent* will be used to refer to the relation between an edge-set and a block, where the block is the predecessor of at least one edge in that edge-set.

For any block b , edge-set S , edge e :

If $successor(e) = b, e \in S$ then $b \in children(S)$

If $predecessor(e) = b, e \in S$ then $b \in parents(S)$

2.3 An Example

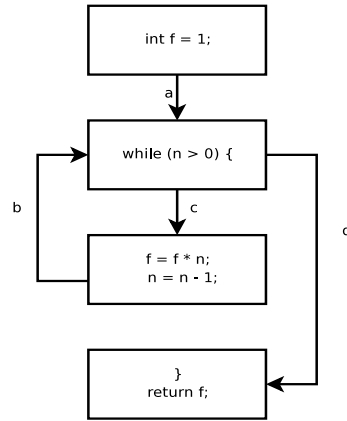
In order to illuminate the process of using the stack regions to perform register allocation we will use an example. The program code in Figure 2.5 is a simple iterative procedure which returns n factorial for any value of n greater than 0, otherwise it returns 1. The C source code is on the left, along side it is the output from the compiler without any register allocation.

Figure 2.5: C Factorial Function

C source	Assembly
	.text
	;function fact
	!loc n
	lit 1
	!loc f
	jump L3
int fact(int n)	L2:
{	@loc f
int f = 1;	@loc n
while (n > 0) {	mul
f = f * n;	!loc f
n = n - 1;	@loc n
}	lit 1
return f;	sub
}	!loc n
	L3:
	@loc n
	lit 0
	brgt L2
	@loc f
	exit

The first part of the stack to be determined is the x-stack, and before that can be done, the edge-sets need to be found; see Figure 2.6. Once the edge-sets are found, the x-stack for each can be determined. Firstly consider the edge-set $\{a, b\}$; both the variables n and f are live on this edge set. Presuming that the hardware can manage this, it makes sense to leave both variables in

Figure 2.6: Determining the edge-sets



The edges a and b share a common child, so form one edge set. The edges c and d share a common parent and form another edge set. So, the two edge-sets are $\{a, b\}$ and $\{c, d\}$

the x-stack. The same considerations apply for $\{c, d\}$, so again both n and f are retained in the x-stack. The order of variables, whether n goes above f , or vice versa, also has to be decided. In this example we choose to place n above f , since n is the most used variable, although in this case it does not make a lot of difference. The algorithms to determine which variables to use in more complex cases are covered in chapter 4.

Once the x-stack has been determined, the l-stack should be generated in a way that minimises memory accesses. This is done by holding those variables which are required by the e-stack in the l-stack, whilst matching the l-stack to the x-stack at the ends of the blocks. Firstly n , as the most used variable, is placed in the l-stack. It is required on the l-stack throughout, except during the evaluation of $n = n+1$, when it is removed, so the value is not duplicated. Secondly f is allocated in the l-stack, directly under n . In the final block the value of n is superfluous and has to be dropped.

The original and final stack profiles are shown in Figure 2.7. Note the extra, seemingly redundant, stack manipulations, such as `rrot2` which is equivalent to `swap`, and `rrot1`, which does nothing at all. These virtual stack manipulations serve to mark the ‘movement’ of variables between the e-stack and l-stack. The final assembly stack code, with redundant operations removed, is shown in Figure 2.8 on the right. Not only is the new code shorter than the original, but the number of memory accesses has been reduced to zero. Although much of the optimisation occurs in the l-stack, the x-stack is vital, since without it variables would have to be stored to memory in each block. Register allocation using only the l-stack can be seen in the centre column of Figure 2.8. This would suggest that the selection of the x-stack

Figure 2.7: Stack profile

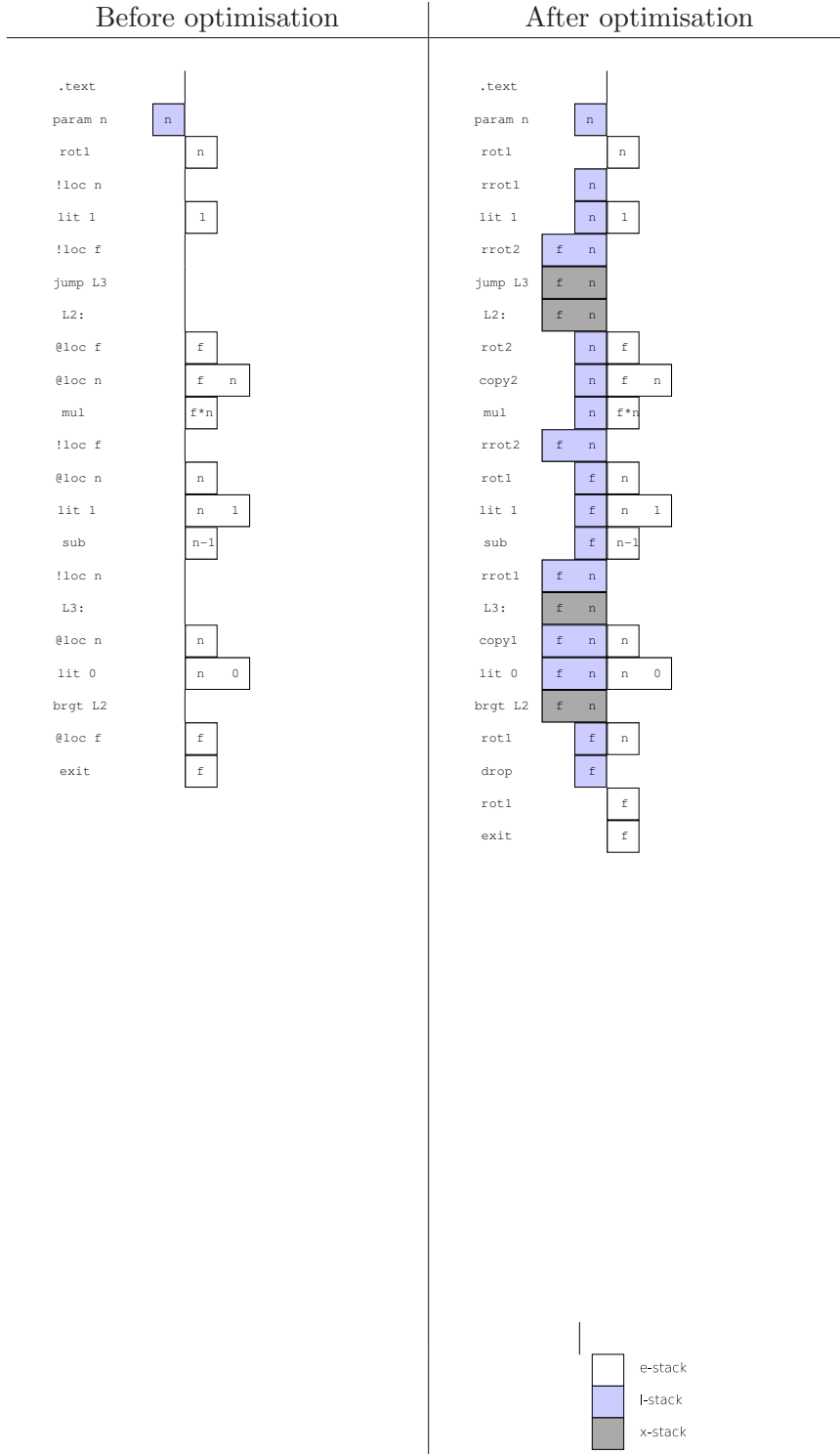


Figure 2.8: Assembly listings

No register allocation	Local register allocation	Global register allocation
<pre>.text ;function fact !loc n lit 1 !loc f jump L3 L2: @loc f @loc n mul !loc f @loc n lit 1 sub !loc n L3: @loc n lit 0 brgt L2 @loc f exit</pre>	<pre>.text ;function fact !loc n lit 1 !loc f jump L3 L2: @loc f @loc n tuck2 mul !loc f lit 1 sub !loc n L3: @loc n lit 0 brgt L2 @loc f exit</pre>	<pre>.text ;function fact lit 1 swap jump L3 L2: tuck2 mul swap lit 1 sub L3: copy1 lit 0 brgt L2 drop exit</pre>

is an important factor in register allocation. Although this is a very simple example, the underlying principles are general and can be applied to much larger programs.

2.4 Summary

Although the stack can be viewed as an arrangement of hardware registers or as a first-in-first-out data structure, it is more useful to view it as a stack of regions. This novel view, the stack-region framework, leads to a new approach to register allocation for stack machines, as outlined in the example above. This new approach will be dealt with more formally in Chapter 4.

Chapter 3

The compiler

This chapter covers the choice of compiler, approaches to porting the compiler to a stack machine and the final structure of the compiler.

3.1 The Choice of Compiler

When developing a compiler for a new architecture it makes sense to develop as little new, and thus untested, code as possible. Using an existing compiler, especially one that is designed to be portable, makes a great deal of sense. There are many C compilers available, but I was only able to find only two that fulfil the following requirements:

- Designed to be portable
- Well documented with freely available source code – In order to modify the compiler, source code needs to be available.
- Supports the C language.

These two are GCC and lcc.

3.1.1 GCC vs LCC

GCC originally stood for the GNU¹ C Compiler it now stands for the GNU Compiler Collection. It is explicitly targeted at fixed register array architectures, but is freely available and well optimised. It is thoroughly, if not very clearly, documented[27]. lcc does not seem to be an acronym, just a name, and is designed to be a fast and highly portable C compiler. Its intermediate form is much more amenable to stack machines. It is documented clearly and in detail in the book[10] and subsequent paper covering the updated interface[11].

¹GNU is a recursive acronym for GNU's Not UNIX. It is the umbrella name for the Free Software Foundation's operating system and tools.

Table 3.1: Comparison of GCC and lcc

	GCC	lcc
C Front end.	yes	yes
Front ends for C++, Java, Fortran and Ada	yes	no
Supports simple optimisations ¹	yes	yes
Supports advanced optimisations ²	yes	no
Detailed control over optimisations	yes	no
Widely used	yes	no
Easy to write machine description	no	yes
Easy to port to stack machines	no	yes
Stack optimisations interfere with other optimisations	yes and no	no

1. Constant folding. Dead code elimination. Common sub-expression elimination.

2. Loop optimisations. Instruction scheduling. Delayed branch scheduling.

A point by point comparison of lcc and gcc is shown in table 3.1. The long list of positives in the GCC column suggests that using GCC would be the best option for a compiler platform for stack machines, at least from the user's point of view. However the two negatives suggest that this porting GCC to a stack machine would be difficult, if not impossible. After all, a working and tested version of lcc is a very useful tool, whereas a non-working version of GCC is useless. This leads to the question: Could a working, efficient version of GCC be produced in the time available? As far as I could discover, there has only been one attempt to port GCC to a stack machine. This port, for the Thor Microprocessor, demonstrated the difficulties involved in such a port.

3.1.2 A Stack Machine Port of GCC – The Thor Microprocessor

The Thor microprocessor was a stack-based, radiation hardened processor with hardware support for Ada tasking, specifically designed for space applications. An attempt was made to port GCC to this architecture by Gunnarsson and Lundqvist[14]. This was highly instructive of the difficulties in porting GCC to a stack machine. Attempts to make GCC stack-friendly were often confounded by the optimisations in GCC which are fundamentally tied to register array machines. In fact it would appear that a large number of optimisations in GCC would be unusable, which removes one of its main advantages over lcc. Interestingly, the Thor processor and the UTSA are both stack-based, word-addressed architectures. Porting a compiler that is implicitly expecting a byte-addressed architecture to a word-addressed architecture is tricky, but it is far from impossible, see Section 3.8.1. However getting a compiler that expects a fixed array of registers to handle a stack is far harder. While the Gunnarsson and Lundqvist were able to generate a working version of GCC for the Thor architecture over the time scale of their Master's project, the end result produced poor code, as the goal of effective stack use was often

defeated by GCC's optimisations. By contrast, it was possible to implement a functioning, if somewhat inefficient, version of lcc for the UTSA architecture within a month. This convinced me to choose lcc over GCC. Using lcc as a base meant that much more time was available for developing new register allocation techniques, resulting in a better end product.

3.2 LCC

lcc is a portable C compiler with a very well defined and documented interface between the front and back ends. The intermediate representation for lcc consists of lists of trees known as forests, each tree representing an expression. The nodes represent arithmetic expressions, parameter passing, and flow control in a machine independent way. The front end of lcc does lexical analysis, parsing, type checking and some optimisations. The back end is responsible for register allocation and code generation.

The standard register allocator attempts to put as many eligible variables² in registers as possible, without any global analysis. Initially all such variables are represented by local address nodes that are replaced with virtual register nodes. The register allocator then determines which of these can be stored in a real register and which have to be returned to memory. The stack allocator uses a similar overall approach, although the algorithm used to do the register allocation is completely different and does include global analysis.

Previous attempts at register allocation for stack machines have been implemented as post-processors to a compiler rather than as part of the compiler itself. The compiler simply allocates all local variables to memory and the post-processor then attempts to eliminate memory accesses. The problem with this approach is that the post-processor needs to be informed which variables can be removed from memory without changing the program semantics, since some of the variables in memory could be aliased, that is, they could be referenced indirectly by pointer. Moving these variables to registers would change the program semantics.

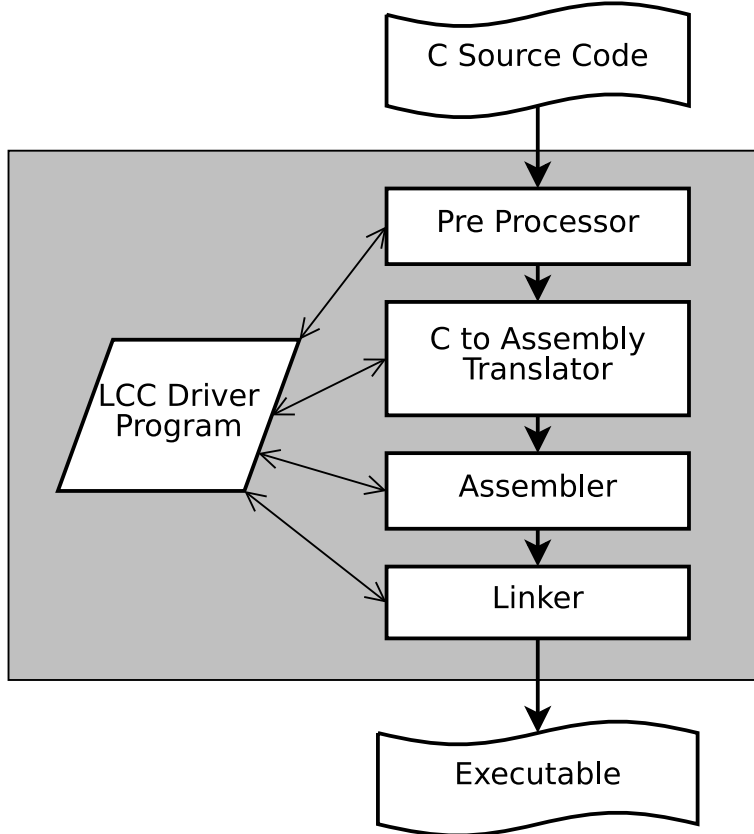
3.2.1 Initial Attempt

lcc is designed to be easily portable, simply by writing a new machine description. This consists of the input to the lburg³ code-generator-generator[12] and a few supporting routines. Since lcc uses trees as its intermediate representation, stack code can be generated very easily from the tree. The initial attempt to port lcc to the UTSA architecture was a naive machine description. Unfortunately problems were encountered due to the implicit assumption within the reusable

²In C local variables may have their addresses used, which renders them ineligible to be stored in a register.

³The lburg code-generator-generator is part of the standard lcc distribution

Figure 3.1: Compiler overview



part of lcc's back end that the target would be a conventional register machine. This was circumvented by claiming the stack machine had a large number of registers⁴ with the intention of cleaning up these virtual-registers later.

Using special pseudo-instructions to mark these virtual-registers in the assembly code ensured that the post-processor was aware of which variables could be legitimately retained on the stack. Many of these virtual registers could then be moved onto the stack and the remaining ones allocated memory slots.

This approach works, that is it is able to produce correct code, but has a number of problems:

- The post-processor must parse its input and output assembly language, and both of these tasks are already done by the compiler.
- It has to be implemented anew for each architecture.

⁴Thirty two, but as unlike a real machine, all of these were available to store variables, none were required for parameter passing or expression evaluation.

Figure 3.2: Translator

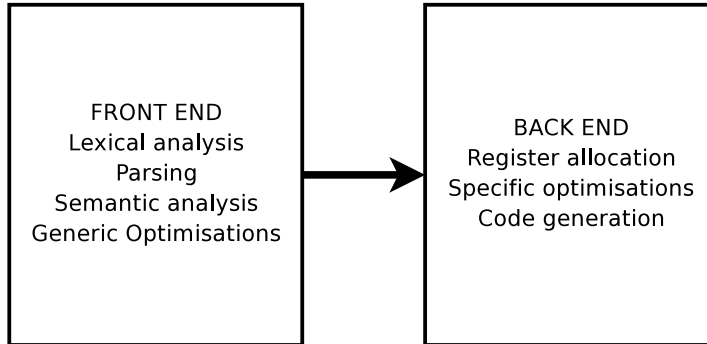
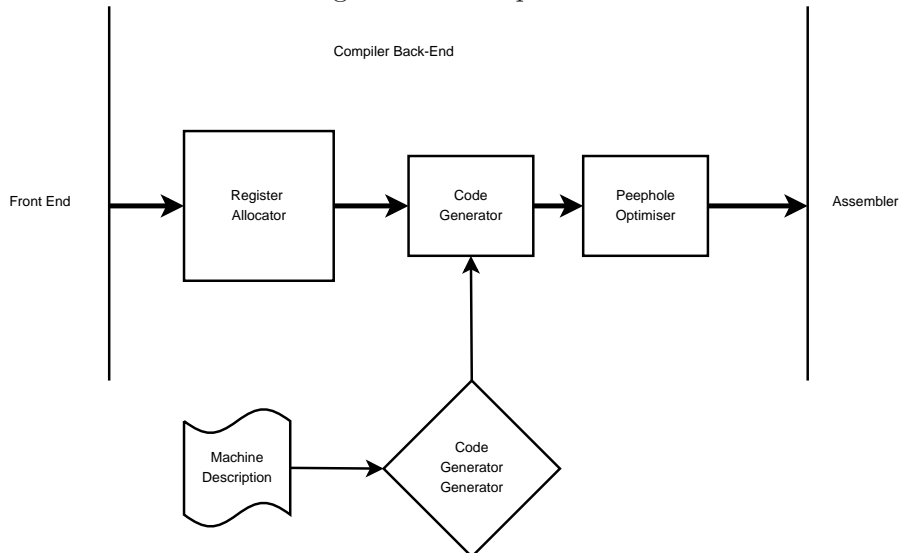


Figure 3.3: Compiler back end



- Flow information, which was readily available within the compiler, has to be extracted from the assembly code. It is extremely difficult, if not impossible, to extract flow control from computed jumps such as the the C 'switch' statement.

Implementing the register allocator within the compiler not only solves the above problems, it also allows some optimisations which work directly on the intermediate code trees[8].

Finally, the code-generator always walked the intermediate code tree bottom up and left to right. This can cause *swap* instructions to be inserted where the architecture expects the left hand side of an expression to be on top of the stack. By modifying the trees directly these back to front trees can be reversed.

3.3 The Improved Version

The standard distribution of `lcc` comes with a number of backends for the Intel x86 and various RISC architectures. These share a large amount of common code. Each back end has a machine specific part of about 1000 lines of tree-matching rules and supporting C code, while the register allocator and code-generator-generator are shared. In order to port `lcc` to a stack architecture a new register allocator was required, but otherwise as much code as possible is reused. Apart from the code-generator-generator, which is reused, the backend is new code. The machine descriptions developed for the initial implementation were reused.

3.3.1 The Register Allocation Phase

Since the register allocation algorithm which was to be used was unknown when the architecture of the back-end was being designed. The optimiser architecture was designed to allow new optimisers to be added later. The flow-graph is built by the back-end and then handed to each optimisation phase in turn. The optimised flow-graph is then passed to the code generator for assembly language output.

3.3.2 Producing the Flow-graph

As the front-end generates forests representing sections of source code these are added to the flow-graph. No novel techniques are used here. When the flow-graph is complete, usage and liveness information is calculated for use in later stages.

3.3.3 Optimisation

`lcc` already supports modification and annotation of the intermediate code forests, all that is required here is an interface for an optimiser which takes a flow-graph and returns the updated version of that flow-graph. The optimisation phase takes a list of optimisations and applies each one in turn. Ensuring that the optimisations are applied in the correct order is up to the programmer providing the list.

Tree Flipping

Some binary operators, assignment and division for example, may require a reversal of the tree to prevent the insertion of extra *swap* instructions. All stack machine descriptions include a structure describing which operators need this reversal performed and which do not.

3.4 Structure of the New Back-End

The source code for the back end is structured in three layers. This is to allow as much modularity as possible. When choosing between data hiding, modularity and reducing errors on one side and speed on the other, safety has been chosen over performance throughout. Performance is noticeably slower than the standard version of lcc, but there is plenty of potential for improving speed, by eliminating runtime checks used during development and replacing some of the smaller functions with macros.

3.4.1 Program flow

The compiler translates the source code from the pre-processor into assembly language in four phases:

1. Intermediate code is passed from the front end and the flow-graph is built
2. Data flow analysis is done. Definitions and uses are calculated, liveness derived from that, and def-use chains created
3. Optimisations are performed
4. Code is generated

3.4.2 Data Structures

This layer is composed of the simple general data structures required: sets, lists and graphs. These are implemented using opaque structures to maximise data hiding. Each data structure consists of a typedef for a pointer to the opaque data structure and a list of functions, all with a name of the form xxx.functionName, where xxx is the data structure name.

For example the set data structure 'BitSet' and some of its functions are defined as:

```
typedef struct bit_set* BitSet;

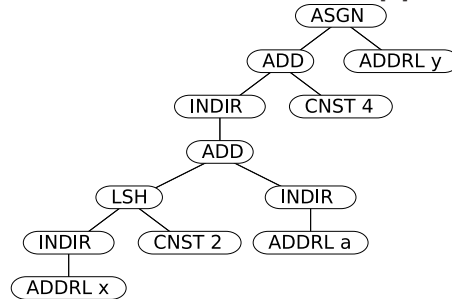
BitSet set_new(int size);

BitSet set_clone(BitSet original);

void set_add(BitSet set, int n);

int set_equals(BitSet set1, BitSet set2);
```

Figure 3.4: lcc tree for $y = a[x] + 4$



3.4.3 Infrastructure

Built upon these classic computer science data structures are the application specific data structures: the flow-graph, basic blocks, liveness information and means of representing the e-, p-, l- and x-stacks. These follow a similar pattern to the fundamental data structures, but with more complex interfaces and imperfect data hiding. For example the ‘Block’ data-structure holds information about a basic block. This includes the depths of the stack at its start and end and which variables are defined or used in that block. It also provides functions for inserting code at the start or end of the block, as well as the ability to join blocks.

3.4.4 Optimisations

This infrastructure allows the optimisers to be implemented clearly and relatively concisely, and allows new optimisations to be added experimentally without unforeseen side effects. The optimisers permute the intermediate code form leaving it in a form that is acceptable to the code generator.

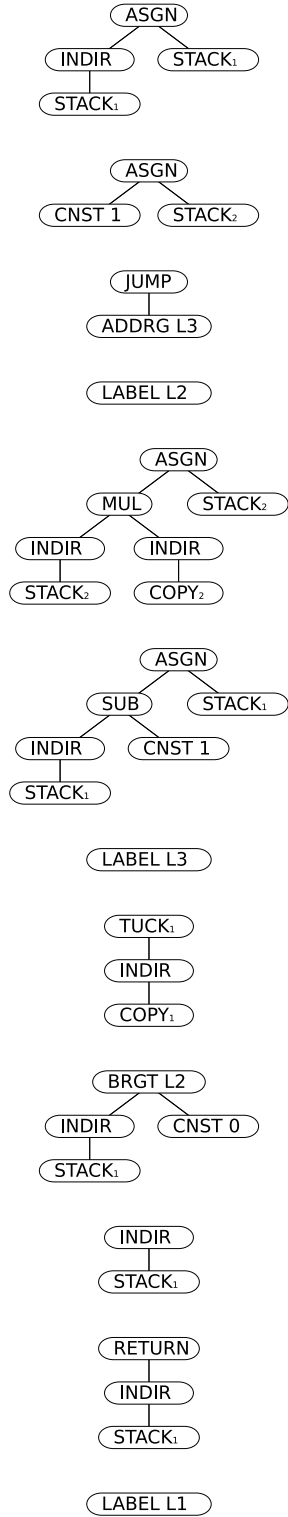
3.5 LCC Trees

The intermediate form in lcc, that is, the data structures that are passed from the front end to the back end, are forests of trees each representing a statement or expression. For example the expression $y = a[x] + 4$, where a is an array of ints, is represented by the tree in Figure 3.4. Trees also represent flow control and procedure calls. For example the C code in Figure 2.5 is represented by the forest in Figure 3.5. A glossary of lcc tree nodes, including the new stack based nodes, is included in appendix D.

3.5.1 Representing Stack Manipulation Operators as LCC Tree-Nodes.

As discussed in Section 1.2, manipulators fall into 5 classes.

Figure 3.5: lcc forest for C source in Figure 2.5



- **Drop** Remove an element from the stack.
- **Copy** Duplicate an element of the stack and place the duplicate on top of the stack.
- **Rotate up** Move an element to the top of the stack, retaining the order of the other elements.
- **Rotate down** Move the top element of the stack to a lower element, retaining the order of the other elements.
- **Tuck** Duplicate the top element of the stack and insert it lower down the stack, displacing the lower elements downwards.

To represent these five classes of operations we introduce three new node types, which can represent all five forms of stack manipulations when incorporated into trees. These are `STACK`, `COPY` and `TUCK`.

3.5.2 The Semantics of the New Nodes

Both `STACK` and `COPY` are virtual address nodes. This means they represent the address of a stack register. They are virtual, since registers cannot have their address taken. This means they can only occur as the l-value in an assignment or as the child of an indirection node.

Stack

The `STACKN` node represents the location of a stack based register. The address of the n^{th} stack register is denoted as `STACKN`. Since registers do not have real addresses, the `STACK` node can only be used indirectly through the `INDIR` node or as the l-value of an `ASGN` node. Since reading the stack is typically destructive, fetching from a `STACK` node will remove the value stored in it from the stack and writing to it will insert a new value, rather than overwriting any value. The `STACK` node will typically be implemented with a rotate instruction, to either pull a value out of the stack for evaluation, or to insert the value currently on top of the stack deeper into the stack.

Copy

The `COPYN` node, like the `STACKN` node, represents the location of a stack based register, but with the side effect of leaving the value stored in n^{th} stack register, when read through the `INDIR` node. The `COPY` node cannot be used as an l-value, as it would be meaningless in this context. The `COPY` node will typically be implemented with a *copy* instruction.

3.5.3 Tuck

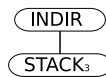
The third new node added is the TUCK node. The TUCK node differs from the other two in that it represents a *value* rather than an address. Its value is that of its only child node, but as a side effect, the TUCK_N node stores that value into the the N^{th} register on the stack. Since, when evaluating an expression on the stack, the value of the current partial expression is held in the top-of-stack register, copying that value into the stack corresponds to the *tuck* instruction. Although this representation is rather inelegant, it is required, both to efficiently represent the semantics of the *tuck* instruction and to allow localised register allocation.

3.5.4 Representing the Stack Operations.

Drop

Since an expression that is evaluated for its side effects only is represented as a complete tree in lcc, a *drop* operation can be represented as an indirection of a stack node. In effect a *drop* instruction can be regarded as discarding the result of the evaluation of the n^{th} register. The semantics of root nodes is explained in Section 3.5.5. See Figure 3.6.

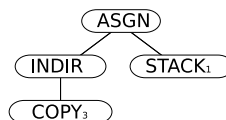
Figure 3.6: *drop3* as lcc tree



Copy

copy is represented by fetching from a COPY node and assigning to the top of stack node, STACK₁. See Figure 3.7

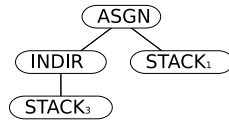
Figure 3.7: *copy3* as lcc tree



Rotate Up

rot is represented by fetching from a STACK node and assigning to the top of stack node, STACK₁. See Figure 3.8

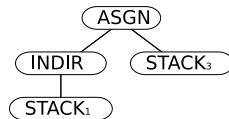
Figure 3.8: *rot3* as lcc tree



Rotate Down

This is a stack node as the l-value of an assignment with the top of stack node, $STACK_1$, as the assignee. See Figure 3.9.

Figure 3.9: *rrot3* as lcc tree



Tuck

This is simply a tuck node, with an arbitrary expression as its child. See Figure 3.10.

Figure 3.10: *tuck3* as lcc tree



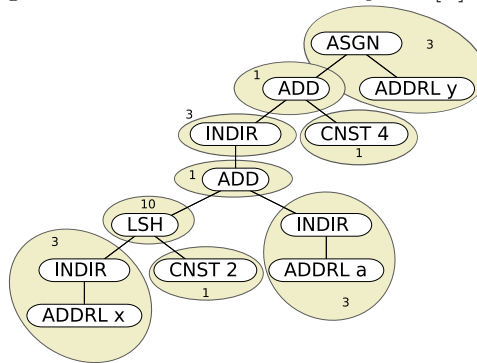
3.5.5 Change of Semantics of Root Nodes in the LCC forest

The other change required for stack based register allocation is a subtle alteration to the semantics of root nodes in the lcc forest. In lcc the roots of trees represent statements in C. As such they have no value. On a stack machine a statement leaves no value on the stack, but an expression does, and so to convert from an expression to a statement the value has to be explicitly dropped from the stack. In lcc CALL nodes may serve as roots, although they may have a value. This is generalised, so that *any* node may be a root node. This means that all the semantics of the stack machine *drop* instruction can easily be represented by using an expression as a statement. This also simplifies the implementation of dead store elimination considerably.

Table 3.2: Partial machine description

stmt:	ASGN(stk, ADDR)	"!loc %1\\n"	3
stk:	INDIR(ADDR)	"@loc %0\\n"	3
stk:	CNST	"lit %a\\n"	1
stk:	LSH(stk, stk)	"call bshl\\n"	10
stk:	INDIR(stk)	"@\\n"	3

Figure 3.11: Labelled tree for $y = a[x] + 4$



3.6 Additional Annotations for Register Allocation

As discussed in chapter 2 the l-stack plays a pivotal role in register allocation. With this in mind it is important that each node that is eligible for placing on the stack, that is the VREG⁵, STACK and COPY nodes, are annotated with the depth of the e-, p- and l-stacks. During register allocation only the depth of the l-stack will change.

3.7 Tree Labelling

In order to select code lcc uses a bottom-up-rewrite labeller, generated by a program called lburg[15][12], which labels nodes or groups of nodes with costs and types to select the least cost match for the tree. The sets of rules it uses is called a machine description. For example the tree in Figure 3.4 could be labelled as shown in Figure 3.11 with the rules in table 3.2, which are from the UFO machine description⁶ for a cost of $3 + 1 + 10 + 3 + 1 + 3 + 1 + 1 + 3 = 26$. The UFO machine has no barrel shifter, so variable shifts have to be done in software, hence the large cost. Note that there is only one possible labelling and it is not very efficient as it uses a function call to left shift a value by

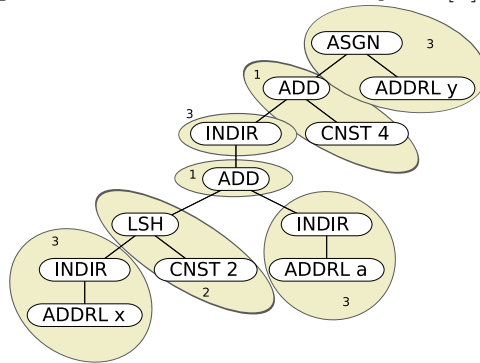
⁵The VREG node is produced by lcc's front end, to represent a virtual register. See Appendix D

⁶These rules are simplified; the real rules include type information, for example the ASGN node is really a CNSTI4 for 4 byte integer constant

Table 3.3: Extended machine description

stmt:	ASGN(stk, ADDRl)	"!loc %1\n"	3
stk:	INDIR(ADDRl)	"@loc %0\n"	3
stk:	CNST	"lit %a\n"	1
stk:	LSH(stk, stk)	"call bshl\n"	10
stk:	INDIR(stk)	"@\n"	3
stk:	LSH(stk, CNST 2)	"shl\nshl\n"	2
stk:	ADD(stk, CNST 4)	"add4\n"	1

Figure 3.12: Relabelled tree for $y = a[x] + 4$



two. The code generated can be improved by adding an extra rule to cover this case, as well as a rule for the UFO's *add4* instruction, to give the machine description in table 3.3, which is labelled in Figure 3.12, for a reduced cost of $3 + 2 + 3 + 1 + 3 + 1 + 3 = 16$. By including sufficient rules and classes to cover the entire instruction set, good quality code can be generated. See Section 3.8 for an example of this in practice.

3.8 UTSA Machine Description

The UTSA⁷ is a stack machine specification and simulator used for researching advanced stack architectures. It was designed with the intent of being implemented with relatively few transistors, yet achieving good performance. It is a standard two stack architecture (computation stack and address stack), with a couple of traditional ‘user’ registers available. All computation is done via the computation stack. The address stack is used primarily for storing return addresses. The two ‘user’ registers are designed with arrays in mind; there are instructions to read and write to the memory locations referred to while incrementing or decrementing the register. The feature of the UTSA which had the most impact on the design of the UTSA machine description was its memory addressing.

⁷University of Teeside Stack Architecture

The UTSA is a word-addressing machine. It cannot access units of memory less than one machine word (32bits). This means that all byte and short (16 bit) read and writes have to be synthesised. Additionally considerable care must be taken in ensuring that spurious conversions to byte-address from word-address do not occur. C in general assumes that individual bytes can be addressed, and we need to ensure that they can. lcc also assumes that all addresses are byte addresses, that is the address N refers to the Nth byte. In UTSA however the address N refers to the Nthword.

3.8.1 Solving the Address Problem.

UTSA uses 24 bit addresses to address 2^{24} words or 64 megabytes of memory. It is a 32 bit machine and thus the top 8 bits are unused for addresses. lcc expects addresses to be byte addresses and UTSA expects addresses to be word addresses.

The naive approach is to store all addresses as byte addresses and to convert to word addresses when required. This creates a lot of extra instructions, since each memory access is preceded by 2 shift instructions. This is clearly not acceptable.

Interestingly, it is nowhere specified what the format of pointers should be. It is often presumed that they are stored as integers, since this simplifies address arithmetic, but this is not necessarily so. Taking advantage of this and the presence of rotate⁸ instructions I decided to use the following format

Bits	31 and 30	29 to 24	23 down to 0
Contents	byte address	zero	word address

Since most addresses will be word aligned, this format is identical to the native word address for the common case. The compiler guarantees that accesses will be appropriately aligned. When using a half word or byte address, a standard library function will need to be called for memory accesses anyway; making these functions aware of the new format is simple.

3.8.2 Modifying the Machine Description

Problems occur however using address arithmetic. Suppose we wish to access the fourth member of an array, each member of the array being one word (4 bytes) wide. The front end of the compiler will generate code to add 16, the offset in bytes, to the address. If we simply add 16 to our new address format, we will have in fact have added 16 words which is 64 bytes. In order to safely add an offset, we will need to roll the address round to bring the

⁸Not stack rotation, but *bit* rotation. The *ror* instruction moves all but the rightmost bit of the top of stack register one place to the right, while moving the rightmost bit to the leftmost position.

byte address down to bits 1 and 0, do the addition, and roll the result back round⁹. In the case just discussed this is incredibly wasteful. Thankfully the machine description has types to allow the description of different addressing modes. To take advantage of this, all values known to be a multiple of 4 are marked as a special type, called ‘addr4’ to distinguish it from the byte-address or just ‘addr’. Where this special type is used, redundant instructions can be removed. Careful consideration of which expressions can be considered to be of this type allows almost all excess instructions to be removed. These expressions are:

- Constant multiples of 4
- Addresses of types of size 4 or greater
- The result of any value left shifted by 2 or more
- The addition or subtraction of two values, both of which of the type ‘addr4’
- The multiplication of two values, either of which is of the type ‘addr4’

Conversion from address to numeric value is achieved by inserting the appropriate rotation instructions. This is relatively rare and the overall effect is a significant saving.

3.9 UFO Machine Description

The UFO[23] machine has been recently developed at the University of York. It is a 32 bit, byte addressed stack machine. It has a simple and compact instruction format. Since the UFO version of lcc supports double word types the machine description is considerably enlarged compared with the UTSA machine description, to cover these extra types. The UFO instruction set includes instructions with three different literal formats; seven, nine and eleven bits. This also adds bulk, but no real complexity, to the machine description.

3.9.1 Stores

The store instruction, where the address is top-of-stack, is unusual in that it is a binary operator but does not consume both its arguments, rather it leaves the address on top of the stack. This means that the store (*stl*, *stw*, *stb*) instructions must be followed immediately by a *drop* instruction. However, when the address argument of the store instruction is the child of a TUCK or

⁹We cannot roll the offset round, since part word addresses will not carry properly when added together.

COPY node, then the copy or tuck is replaced with a rotation, since the copy-store-drop or tuck-store-drop sequence would be too complex for the peephole optimiser to deal with.

3.10 Variadic Functions for Stack-Machines

Stack machine architectures have an ABI¹⁰ which presumes all parameters are passed on the stack and the return result is also returned on the stack. This means that the number of arguments passed to a function must be precisely that which the function expects. C is generally forgiving about formal parameter and actual argument mismatches, and standard register file architectures allow them to be so. Hardware stacks are not. Furthermore variadic functions are superficially similar to other functions and should have the similar calling conventions.

3.10.1 The Calling Convention for Variadic Functions

When calling a variadic function the total number of stack slots used for the variable arguments must be pushed onto the stack immediately before the function is called. Basically we are adding an implicit extra parameter to any variadic function.

3.10.2 Variadic Function Preamble

At the start of any variadic function, the variadic parameters must be stored to memory as follows:

- Pop the top of the stack and store into memory or another register. This is the number of variadic parameters.
- Store each parameter, in turn, to memory.
- Store the address of these parameters to a standard location.
- Proceede as normal.

Provided the architecture has hardware support for local variables, the best place to store the variadic parameters is in the local variable frame, moving the frame pointer to accommodate. The address of the parameters can then be stored at a fixed offset from the frame pointer. Variadic functions are implemented in this way for both the UFO and UTSA architectures.

¹⁰Application Binary Interface.

3.11 Summary

Having compared GCC and lcc, lcc was chosen as a better front-end for a compiler for stack machines. Once the front-end was selected, a new back-end was required, although the code generator generator was reused. Some practical problems involved in the ports to UTSA and UFO were solved. At this stage a working compiler existed, but one that produced poor quality code. The next chapter explains approaches to register allocation, in order to improve the quality of the compiler output.

Chapter 4

Register allocation

In this chapter we look at the pre-existing methods of register allocation for stack machines as well as global allocation techniques developed using the framework laid out in Chapter 2

4.1 Koopman's Algorithm

Koopman's Algorithm, as described in his paper, was implemented as a post processor to the textual output of GCC[27] after partial optimisation. In this case however, it is implemented within the compiler, where it acts directly on the intermediate form.

4.1.1 Description

The algorithm is quite straightforward, see Algorithm 1.

4.1.2 Analysis

In Koopman's Algorithm, when he refers to the bottom of the stack, he is referring to the portion of the stack used by the function being optimised. Since no inter-block allocation is done, the x-stack is empty, and this is clearly the bottom of the l-stack. Therefore points 3a and 3b in Koopman's Algorithm become:

- 3a. Copy the variable at the point of definition, or first use, to the bottom of the l-stack.
- 3b. Replace the second instruction with an instruction to rotate the value to the top of the l-stack.

Algorithm 1 Koopman’s Algorithm

1. Clean up the output using simple peephole optimisation, replacing sequences of stack manipulations with shorter ones if possible.
 2. Locate define–use and use–use pairs of local variables and list them in order of their proximity. That is, in inverse order of the number of instructions separating the pair.
 3. For each pair:
 - (a) Copy the variable at the point of definition, or first use, to the bottom of the stack.
 - (b) Replace the second instruction with an instruction to rotate the value to the top of the stack.
 4. Remove any dead stores.
 5. Reapply the peephole optimisation.
-

4.1.3 Implementation

Each step in Section 4.1.1 can be directly translated to a form suitable for use on the lcc intermediate form, as shown in Algorithm 2.

Koopman’s Algorithm is implemented in approximately 140 lines of C code, not including code shared by all the optimisers and data-flow analysis.

4.1.4 Analysis of Replacements Required

Since each member of the pair can be a definition or a use, there are four possible types of pairs; def-def, def-use, use-def, and use-use. Additionally as the forest is changed during optimisation, either the first or the second member can be changed. This multiplies the number of permutations by four; unchanged-unchanged, unchanged-changed, changed-unchanged, changed-changed. This gives a total of sixteen combinations to consider.

Maintaining the Depth of the L-stack

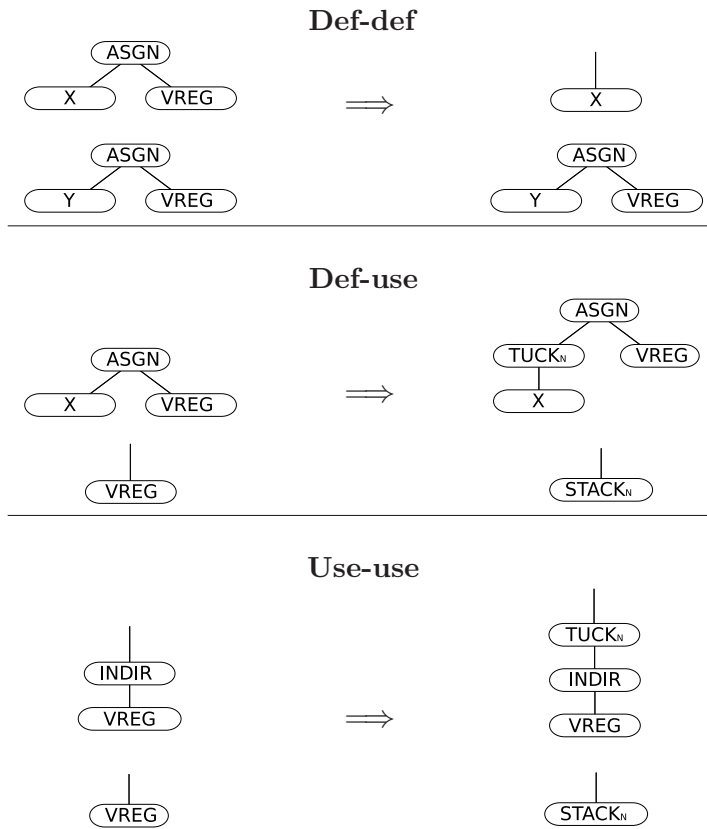
Each time a value is stored on the l-stack, the depth of the l-stack must be increased by one for all nodes between the two nodes forming the pair. This ensures that subsequent pairs are stored at the correct depth in the stack.

Algorithm 2 Implementation of Koopman's Algorithm

1. Cleaning up is not required since it serves mainly to remove some artifacts from the GCC output.
 2. Create an empty table for variable definitions or uses, indexed by variable name.
 3. Create an empty list of definition-use or use-use pairs.
 4. Scan through the forest searching for variable definitions and uses. Each time one is found:
 - (a) If there is a definition or use for this variable in the table, form a pair consisting of it and the definition or use just found and add to a list of pairs.
 - (b) Insert newly found definition or use in table, replacing any previous entry for that variable.
 5. Sort the list, in increasing order of separation.
 6. For each pair:
 - (a) For the first node in the pair:
 - i. If it is a definition, that is an assignment, then the value being assigned is copied to the bottom of the l-stack, before assignment.
 - ii. If it is a usage, then the value is copied to the bottom of the l-stack, directly before the point of use.
 - (b) For the second node in the pair:

The variable read is replaced with a fetch from the base of the l-stack.
 7. Any assignments to dead variables are removed. If the tree does not contain a procedure call it is eliminated, otherwise the procedure call is promoted to the root of the tree, as it needs to be evaluated for side effects.
 8. Since this algorithm applies optimisations piecemeal, a peephole optimiser will be required. This looks for sequences of stack manipulations and replaces them with more efficient equivalents, where possible. This is done after code generation. Since a peephole optimiser is required anyway, this is no additional work, See Section 4.5.
-

Table 4.1: Koopman’s Algorithm — Initial transformations

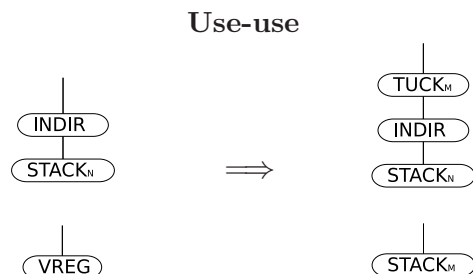


Initial Transformations

Firstly the initial transformations on the four types of pairs are considered. This is before any transformations have been performed. Three different transformations are possible in this case. The transformations are listed below; the modifications to the trees are shown in table 4.1.

- Def-def** The first definition is dead, as it is never used, and can be eliminated. This simplifies subsequent dead store elimination as only the final occurrence of a variable in each block need be considered.
- Def-use** A TUCK node is inserted before the value is stored, and the use replaced with a STACK node.
- Use-def** The two nodes are unrelated, so no change is possible.
- Use-use** As in the def-use case, a TUCK is inserted before the first use, and the second replaced with a STACK node.

Table 4.2: Transformations with first node transformed



Transformations After the First Node Has Been Transformed

When the first node has been transformed only one transformation is possible. The explanations are listed below; see table 4.2 for the trees.

Def-def Since the second item in either def-def, or use-def, is never changed then the first node in this pair cannot have been changed, so this cannot occur.

Def-use Likewise.

Use-def The two nodes are unrelated, so no change is possible.

Use-use Although the first VREG node will have been transformed into a STACK node, a TUCK is inserted before the first use as before, and the second node replaced with a STACK.

Transformations After the Second Node Has Been Transformed

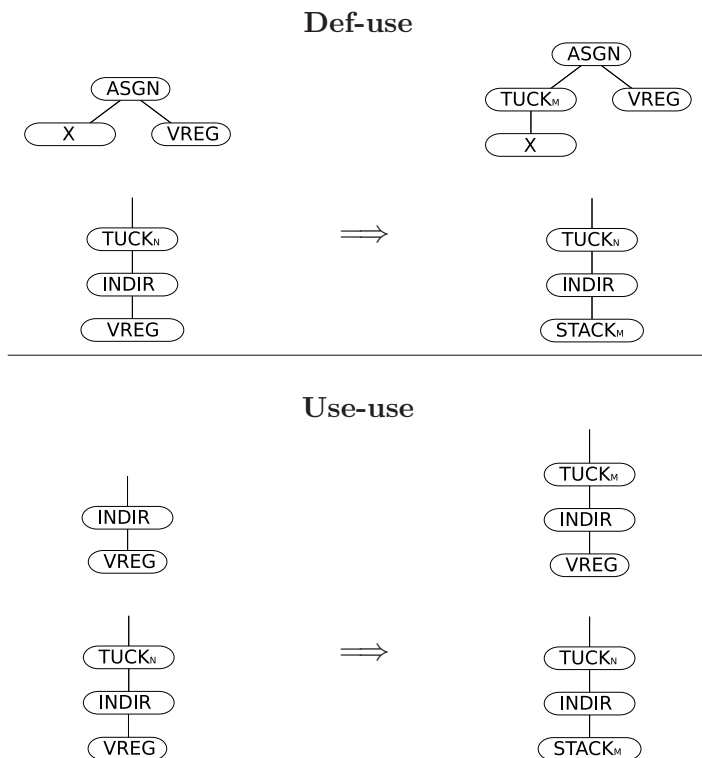
Def-def The first def is dead so should be eliminated. The STACK node, however, must be retained so as not to change the stack state, in other words the *rotate*, will be replaced with a *drop*.

Def-use Although the VREG node will have been transformed to a STACK node, the substitution is identical to the initial case.

Use-def The two nodes are unrelated, so no change is possible.

Use-use Although the second node will have had a TUCK node inserted, the transformations are the same as the initial case.

Table 4.3: Transformations with the second node transformed



Transformations After Both Nodes Have Been Transformed

Def-def Since the second item in either def-def, or use-def, is never changed then the first node in this pair cannot have been changed, so this cannot occur.

Def-use Likewise.

Use-def The two nodes are unrelated, so no change is possible.

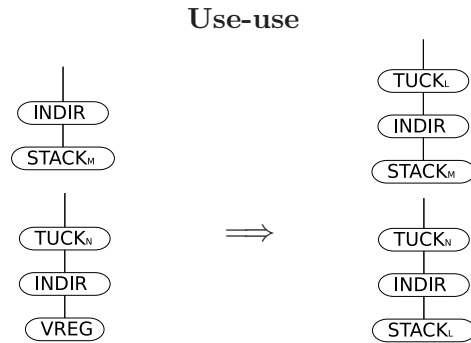
Use-use The first node will have transformed from VREG to STACK and the second node will have had a TUCK inserted, but the transformation required, inserting a TUCK node into the first tree and transforming the VREG node to a STACK node, remains the same.

4.2 Bailey's Algorithm

4.2.1 Description

Bailey's 'inter-boundary' algorithm[5] was the first attempt to utilise the stack across basic block boundaries. This is done by determining edge-sets; although

Table 4.4: Transformation with both nodes transformed



in the paper the algorithm is defined in terms of blocks rather than edges. Then the x-stack, termed ‘sub stack inheritance context’, is determined for the edge-set. See Algorithm 3.

Algorithm 3 Bailey’s Algorithm

1. Find co-parents and co-children for a block (determine the edge-set).
2. Create an empty ‘sub stack inheritance context’.
3. For each variable in a child block, starting with the first to occur:

If that variable is present in all co-parents and co-children, then:

Test to see if it can be added to the base of the x-stack. This test is done for each co-parent and co-child to see whether the variable would be reachable at the closest point of use in that block.

Bailey’s Algorithm is designed to be used as a complement to an intra-block optimiser, such as Koopman’s. It moves variables onto the stack across edges in the flow graph, by pushing the variables onto the stack immediately before the edge and popping them off the stack immediately after the edge. Without an intra-block optimiser this would actually cause a significant performance drop.

4.2.2 Analysis of Bailey’s Algorithm

Bailey’s Algorithm, in terms of the framework laid out in chapter 2, is shown in Algorithm 4

Although Bailey’s Algorithm is an inter-block algorithm, it is not genuinely global, as it makes fairly limited use of the x-stack. No values are left in the

Algorithm 4 Bailey's Algorithm with x-stack

1. Determine edge-sets
2. For each edge-set:
 - (a) Create an empty x-stack.
 - (b) Determine the intersection of the sets of live variables for each edge in the edge-set.
 - (c) Choose a neighbouring block, presumably the first to occur in the source code.
 - (d) For each variable in the selected sets, in increasing order of the distance of usage from the edge in question.

Add the the variable can be added to the x-stack, provided it is reachable by the hardware.

x-stack during blocks. Not only that, but the variables in the x-stack are shunted, via the l-stacks and e-stacks, to and from memory at the ends of blocks. No attempt is made to integrate the allocation of the x-stack and the allocation of the l-stack. In terms of performance, the main failing of Bailey's Algorithm is that it cannot handle variables which are live on some but not all edges of an edge-set.

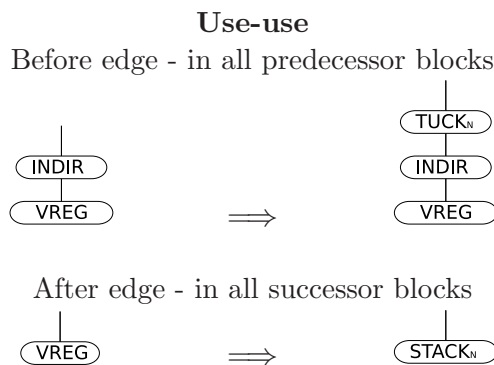
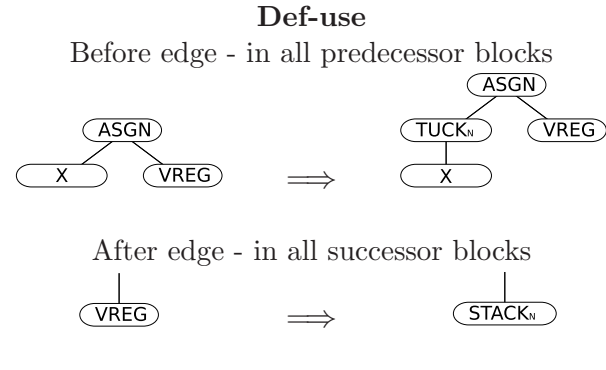
4.2.3 Implementation

As Bailey's Algorithm will be implemented before Koopman's it must leave the internal data structures in a form suitable for that algorithm. As variables are pushed onto the stack across basic blocks, the l-stack depth must be modified to take account of these extra values. The actual substitutions are identical to those done in Koopman's Algorithm, but the substitution may be done in multiple locations. Suppose that a variable x is to be lifted onto the top of the stack on an edge, then the substitutions are shown in table 4.5. The implementation of Bailey's Algorithm takes only about 150 lines of C code, once all the necessary flow information has been gathered.

4.3 A Global Register Allocator

The next step in improving register allocation, is to try to do it in a global (procedure wide) fashion. Once we have gathered full data-flow information including determination of edge-sets, what remains is to determine the x-stack for each edge-set. A policy of successive refinement was used, starting with the maximum possible number of variables and reducing that in an attempt

Table 4.5: Tree transformations — Bailey’s Algorithm



to get close to the optimum.

4.3.1 An Optimistic Approach

The first approach is to use the same framework as Bailey’s; once x-stacks are determined, values are pushed onto the stack before an edge block and popped off afterwards, leaving the local optimiser to clean up, the extra memory accesses. However, in Bailey’s approach the variables chosen quite conservatively. The ‘optimist’ approach uses all plausible variables, the set of all variables that are live on any of the edges in the edge-set. Surprisingly, for a few small programs with functions using only three or four local variables, this approach works very well and produces excellent code. It does, however, produce awful results for larger programs with more variables, as variables are moved from memory to the stack and back again across every edge in the flow graph.

4.3.2 A More Realistic Approach

Obviously a more refined way to select the x-stack is required. The ‘optimist’ algorithm in the previous section chooses the *union* of the sets of variables

which are live on each edge in the edge-set, whereas Bailey's Algorithm chooses the *intersection* of these sets. Experimental modifications to Bailey's Algorithm to use various combinations of unions and intersections of liveness and uses revealed some important limitations in the localised push-on, pop-off approach. These are:

- **Excessive spilling**

There is no attempt to make the x-stack similar across blocks, so variables may have to be saved at the start of a block, and other variables loaded at the end of a block.

- **Excessive reordering**

Even when the x-stack state at the start and end of a block contain similar or the same variables, the order may be different and thus require extra instructions.

- **No ability to use the x-stack across blocks**

The requirement for the entire x-stack to be transferred to the l-stack means that the size of the x-stack is limited. Variables cannot be stored deeper in the stack when they are not required.

4.3.3 A Global Approach

The problems to be solved are:

- **Determination of x-stack member sets**

Although none of the modified versions of Bailey's Algorithm produced better code than the original, some versions did seem to make promising selections of x-stack members. We decided to determine the x-stack set by starting with a large set of variables and reducing it towards an optimum.

- **Ordering of the variables within the x-stack**

If variables are to be kept on the x-stack during blocks then the order of the lower parts of the x-stack is important. Since the ordering of variables on the x-stack cannot be changed without moving variables to the l-stack, the order of the lower parts of the x-stack *must* match across blocks. The simple but effective approach taken was to choose a globally consistent ordering. This also solves the problem of excessive reordering of variables. Determination of this ordering is outlined in Section 4.3.5.

- **Handling the l-stacks to work with the x-stack**

Since allocation of the l-stack depends on the x-stack at both beginning and end of the block, it is necessary to determine the x-stack first.

However, in order to allocate the x-stack in a way that does not impede the subsequent l-stack allocation, the l-stack, must be at least partially determined before the x-stack.

4.3.4 Outline Algorithm

The algorithm chosen runs as follows:

1. Gather all data-flow information.
2. Determine x-stacks.
3. Allocate variables, ensuring l-stacks match x-stacks.

4.3.5 Determining X-stacks

There are two challenges when determining the x-stack. One is correctness, that is, the x-stack must allow register allocation in the l-stacks to be both consistent with the x-stack and legal. The other challenge is the quality of the generated code. For example making the x-stack empty at all points is guaranteed to be correct, but not to give good code. Both of the x-stack finding methods work by first using heuristics to find an x-stack which should give good code, then correcting the x-stack if necessary. The algorithm for ensuring correctness is the same, regardless of heuristic used. For the x-stacks to be correct, two things need to be ensured:

1. **Reachability**

Ensure all variables in the x-stack that are defined or used in successor or predecessor blocks, are accessible at this point.

2. **Cross block matching**

Ensure that all unreachable variables in the x-stack on one edge do not differ from those in the x-stack on an edge with one intervening block.

Ordering of Variables.

As stated earlier, a globally fixed ordering of variables is used. This is done by placing variables with higher ‘estimated dynamic reference count’ nearer the top of the stack. In our implementation, which is part of a port of lcc[15], the ‘estimated dynamic reference count’ is the number of static references to a variable, multiplying those in loops by 10 and dividing those in branches by the number of branches that could be taken. However, any reasonable estimation method should yield similar results. An alternative ordering could be based around ‘density’ of use, which would take into account the lifetime of variables.

Heuristics

By a process of experiment and successive refinement, two different heuristics for x-stack creation have been developed. The first is relatively simple and fast, whereas the second is somewhat more complex, and consequently slower.

Global 1

The first, simpler heuristic is simply to take the *union* of live values as done in the ‘optimist’ approach in Section 4.3.1. So why is this any better? It is better because the subsequent parts of the algorithm remove a lot of variables that would be counter productive, thus ensuring that the x-stack is not overwhelmed with too many variables. The globally consistent ordering reduces unnecessary stack manipulations in short blocks. Finally the l-stack allocation method discussed in Section 4.3.7 is designed to work with x-stacks directly, further reducing mismatched stacks.

Global 2

This heuristic was developed to improve on ‘Global 1’. It considers the ideal l-stack for each block and then attempts to match x-stack as closely to that as possible. Given that the ordering of variables is pre-determined, the x-stack can be treated as a set. In order to find this set, we determine a set of variables which would be counter productive to allocate to the l-stacks. The x-stack is then chosen as the union of live values less this set of rejected values. The set of ‘rejects’ is found by doing ‘mock’ allocation to the l-stack, to see which values can be allocated, then propagating the values to neighbouring blocks in order to reduce local variation in the x-stack. See Algorithm 5. Overall this heuristic out performs ‘Global 1’, but results in slower compilation and can produce worse code for a few programs, appendix F shows an extreme example of this.

Mock L-stack Allocation

Before any real allocation can be done, information needs to be gathered about which variables can be beneficially allocated to the x-stack. In order to do this, it helps to know what variables will be in the l-stack. This ideal l-stack is found by doing a ‘mock’ allocation. This is essentially the same as the proper l-stack allocation, covered in Section 4.3.7, except that is done without regard to the x-stack, by pretending that it is empty. Since this is a ‘mock’ allocation, no change is made to the intermediate form, see Algorithm 7.

Propagation of Preferred Values

Once mock l-stack allocation has taken place, the variables which have been rejected as x-stack candidates for individual blocks are propagated to their

Algorithm 5 Determining the x-stack

1. For each block b :

Determine $b.rejects$ using *mock* l-stack allocation, as described in Algorithm 7
2. For each edge-set, e :
 - (a) Define two sets $rejects = \phi$ and $all = \phi$.
 - (b) For each block $b, b \in e.parents$:
 - i. $reject \leftarrow reject \cup b.reject$
 - ii. $all \leftarrow all \cup b.live_out$
 - (c) For each block $b, b \in e.children$:
 - i. $reject \leftarrow reject \cup b.reject$
 - ii. $all \leftarrow all \cup b.live_in$
 - (d) $e.xstack \leftarrow all - reject$

neighbours. Propagation takes place when the receiving block(s) will be executed less frequently than the sending block, using a simple static estimate of execution frequency. See Algorithm 6.

Algorithm 6 Propagation of x-stack candidates

1. For each basic block, b :
 - (a) If $|b.successors| > 1$ then:

For each block $c, c \in b.successors$:
 $c.reject \leftarrow c.reject \cup b.reject$
 - (b) If $|b.predecessors| > 1$ then:

For each block $p, p \in b.predecessors$:
 $p.reject \leftarrow p.reject \cup b.reject$
 2. Repeat step 1 until no more changes occur.
-

4.3.6 Other Global Allocators

More complex allocation methods were experimented with, largely in an attempt to fix the cases where ‘Global 2’ did not perform well. However these, largely ad hoc, methods were often unreliable as they became more complex. In

order to further improve global register allocation, a regional allocator would be required, see Section 6.3.2.

4.3.7 L-Stack Allocation

The uses of a variable within a block can be divided up into chains. Each chain starts either at the beginning of a block or when the variable is assigned. So if a variable is not assigned during a block then there is only one chain for that variable. If a variable is neither used nor assigned, but is live through the block, it still has a chain. These empty chains are still required because the variable will require stack space, whether it is used or not.

To allocate a chain, the variable represented by the chain is stored at the bottom of the l-stack, extending the l-stack downwards for the length of the chain. See Algorithm 7.

L-stack allocation also needs to ensure that the l-stack matches the x-stack at the block ends. This is done by dropping, storing or reloading variables where the l-stack and x-stack do not match; see Algorithm 9. There is one final complication. If a block has a conditional branch statement at its end, then statements to match up the l-stack and x-stack cannot be inserted after the branch, as they would not necessarily execute. However, if the instructions are inserted before the branch, then the stack would be incorrect, as the branch test would disturb the stack. This can be solved as follows: if either of the values in the comparison are not temporary variables, local to the block, then a temporary is created and assigned the value, before the test. For example, the following intermediate code:

```
if x+4 < y*2 then goto L1
```

becomes

```
t1 = x+4
t2 = y*2
if t1 < t2 then goto L1
```

These temporary variables, $t1$ and $t2$, are ignored during global register allocation. They are left for the local optimiser to clean up. This guarantees correctness and since $t1$ and $t2$ have such short lifetimes, they are easily removed later.

4.3.8 Chain Allocation

Allocating a chain to the l-stack is similar to allocating a def-use pair except that the intermediate nodes are replaced with COPY nodes rather than STACK nodes, as the variable is kept on the stack until the end of the chain. The depth of the l-stack is increased by one for the lifetime of the chain. The advantage of allocating by chain is that stack perturbations are minimised compared with Koopman's Algorithm, see Figure 4.1. Although thanks to

Algorithm 7 L-Stack Allocation

1. For each chain c , in block b :
 - (a) For each node $n, n \in c.nodes$:

If $depth(estack) + depth(lstack) \geq registers$ then:

 - i. If this is *mock* allocation and $c.livein \vee c.liveout$ then:
 $b.reject \leftarrow \{b.reject, c.variable\}$
 - ii. Restart 1. with the next chain.
 - (b) $depth(lstack) \leftarrow depth(lstack) + 1$ for the length of this chain.
 - (c) If this is the *real* allocation phase then:
Allocate chain c at the base of the l-stack.
-

Algorithm 8 Chain Allocation

1. If the first node in c is a definition:

Replace memory write with a stack manipulation, a rotate, to move top-of-stack to the base of the l-stack.
 2. For each other node in c , except the last:

Replace memory read with a stack manipulation, a copy, to place a duplicate of the value at base of l-stack at the top-of-stack
 3. If chain c is live at the end of the block:

Replace memory read with a stack manipulation, a copy, to place a duplicate of the value at base of l-stack at the top-of-stack

Else:

Replace memory read with a stack manipulation, a rotate, to move the value at base of l-stack to the top-of-stack
-

peephole optimisation it often makes little difference which method is used, at least for the UFO machine. The advantages of the chain allocator are that it uses a smaller set of stack manipulators and handles the concept of x-stacks.

Algorithm 9 Matching the L-Stack and X-Stack

X_1 and X_2 are the states of the x-stack at the start and end of the block respectively; b is the current block and L is the l-stack.

1. For each $v, v \in X_1, v \notin L$
 - If $v \in b.live_in$:
 - Save v to memory at start of b
 - Else:
 - Drop v from stack at start of b
 2. For each $v, v \notin X_1, v \in L$
 - Load v from memory at start of b
 3. For each $v, v \in X_2, v \notin L$
 - If $v \in b.live_out$:
 - Load v from memory at end of b
 - Else:
 - Insert dummy value for v into stack at start of b
 4. For each $v, v \notin X_2, v \in L$
 - Save v to memory at end of b
-

4.4 Final Allocation

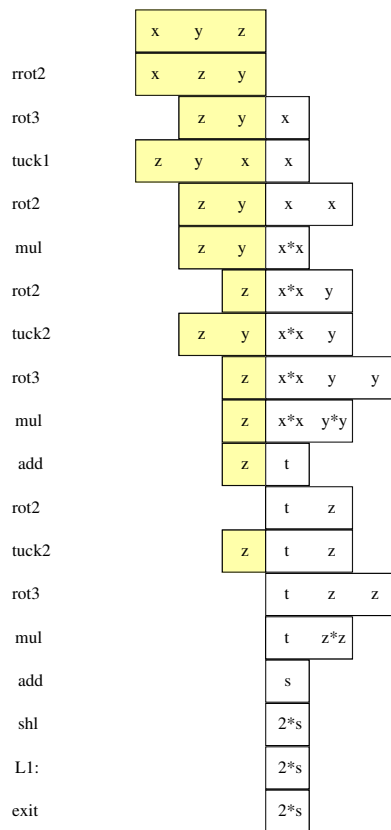
As the global register allocation only allocates variables to the stack when an entire chain can be allocated, it can leave a number of variables unallocated. To further reduce memory accesses, a further allocator is required. The global register allocator will have already used the l-stack to good effect, so Koopman's Algorithm would be unable to handle the output of the that allocator, since the base of the l-stack would be largely unreachable. Consequently a different form of local optimiser is required, one that places values within the l-stack, rather than just at its base. This works by trying to remove definition-use and use-use pairs as well, choosing the depth with a simple cost function, Algorithm 10, based on an estimate of the number of instructions required. This algorithm checks all legal positions in the l-stack for a slot in which to

Figure 4.1: Comparison of allocation by pair and by chain.

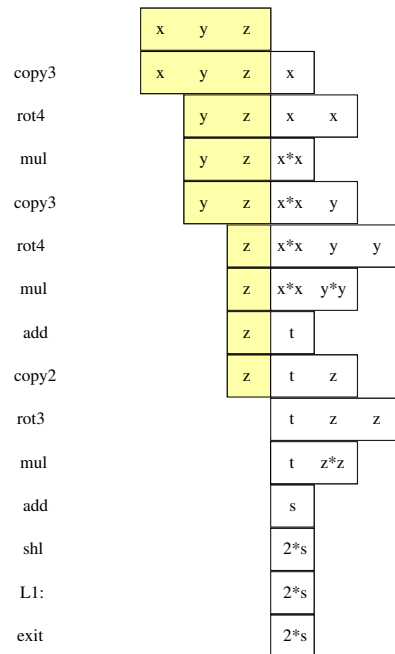
C source code:

```
int cubic_surface(int x, int y, int z) {
    return 2 * (x*x + y*y + z*z);
}
```

By Pair



By chain



insert the value; a position is not legal if it causes a use of another value to become unreachable, or if the value intrudes into the e- or p-stack. If the pair does not span a procedure call then the p-stack may be viewed as part of the l-stack, and only the e-stack preserved. The transformations on the intermediate trees are the same as for Koopman's Algorithm and both share a considerable amount of code.

Algorithm 10 Cost function

For a usage pair with a depth at start of x and a depth at the end of y , then:

1. For first item:

If already a $STACK_N$ node and $N \neq 1$ and $N = x$ then:

$cost1 \leftarrow 0$

Else:

$cost1 \leftarrow 1$

2. For second item:

if $y = 0$ then:

$cost2 \leftarrow 0$

Else:

$cost2 \leftarrow 1$

3. $cost \leftarrow cost1 + cost2$

4.5 Peephole Optimisation

The final stage is to clean up the resulting stack manipulations. For example, in order to move a variable from the l-stack to the e-stack a *swap* instruction might be issued. If two in a row were issued then the resulting sequence, *swap-swap* could be eliminated. The peephole optimiser is table driven. It parses the input accepting stack manipulations and any instruction that puts a simple value on the stack, such as a literal or a memory read. When any other type of instruction is encountered or the increase in stack depth exceeds two, the calculated stack state is looked up in a table to find the optimum sequence of instructions, which are then emitted. Should none be found or a different class of instructions be found, then the input is copied.

4.5.1 Example

Suppose the following input is seen by the optimiser:

```
drop1
copy1
lit 1
rot
add
```

The add instruction halts the sequence of instructions that can be handled with the resulting stack transformation

		B
A		1
B	\implies	B
C		C

The new stack state would be looked up and replaced with the shorter sequence:

```
drop1
lit 1
copy2
add
```

For the UFO architecture, as only the top four values on the stack are in hardware registers, the peephole optimiser is concerned only with those four values plus literals and memory reads; this means that $5^6 + 5^5 + 5^4 + 5^3 + 5^2 + 5 + 1 = 19531$ possible permutations are required. Generation of this table requires an exhaustive search over millions of possible sequences, but can be done at compile time, so is not a problem. The table itself requires about 150k bytes of memory, out of 209k when compiled using GCC for the x86 architecture. Performance is more than adequate at about 1 million lines per second on a typical desktop PC(2006).

4.5.2 The *Any* Instruction

The *any* instruction is a pseudo-instruction used by the compiler and peephole optimiser. Its effect is to push a ‘don’t care’ value onto the stack. This is useful when the compiler needs to push a value onto the stack at the end of a block to ensure that the x-stack is correct, but the actual value pushed does not matter. Note that *any* does not mean an arbitrary instruction, but an instruction that pushes an arbitrary value.

The advantage of the *any* instruction becomes clear with peephole optimisation. Suppose that in the previous example the *copy1* represented the compiler inserting a dummy value. Replacing the *copy1* with an *any* instruction saves an additional instruction:

```
drop1
any
```

```
lit 1
rot
add
```

can be replaced with

```
lit 1
rot
add
```

Since the *any* instruction tends to get inserted in loops, this is a worthwhile saving.

For completeness, and to avoid any surprises when the peephole optimiser is turned off, the assembler recognises the *any* instruction and treats it as a *lit 0* instruction.

4.6 Summary

The analysis of Koopman's and Bailey's algorithms, using the framework presented in Chapter 2, is suggestive of ways to improve upon those algorithms. By exploring these improvements, two new global register allocation algorithms were developed. The relative performance of these algorithms will be considered in the next chapter.

Chapter 5

Results

This chapter discusses the relative effectiveness of the various register allocation algorithms for various stack machines, both real and hypothetical. Before doing that, however, the correctness of both the compiler and the simulator used needs to be verified.

There are two types of test suites for a compiler:

1. To test coverage and correctness of the compiler.
2. To test performance of the compiler and hardware.

When selecting of a suite of test programs for performance testing, it is usual to consider a ready made suite to allow comparisons with other architectures. Since the hardware for UFO was unavailable at the time of writing, comparison with other architectures is impossible. Therefore, in order to minimise effort a significant proportion of the test suite was reused for benchmarking. Some of the benchmarks were also used to verify correctness.

5.1 Compiler Correctness

The test suite consists of the test programs that come with lcc plus the benchmarks listed below. All benchmarks and test programs ran correctly, using the simulator.

5.2 Benchmarks

Twelve benchmarks were selected. The criteria for selected benchmarks were zero-cost and minimal library requirements. Since the UFO compiler is targeting an embedded processor there is no requirement for the standard C library, and so only a very limited library is provided to allow interfacing with the simulator. The characteristics of the benchmarks are listed in table 5.1.

Table 5.1: The benchmarks

Benchmark	Source	Description
bsort	Stanford	Binary sort
image	Stanford	Image smoothing
matmul	Stanford	matrix multiplication
fibfact	—	Recursive implementations of fibonacci and factorial functions
life	Stanford	Conway’s life simulation
quick	Stanford	Integer quicksort
queens	Stanford	Eight queens problem
towers	Stanford	Towers of Hanoi
bitcnts	MiBench	Variety of bit counting methods
dhrystone	dhrystone	The dhrystone synthetic benchmark
wf1	lcc test	Word count program
yacc	lcc test	YACC generated expression parser
Overall	—	The geometric mean result for all the above benchmarks.

5.3 The Baseline

All the results in this chapter are relative to a baseline compiler with no register allocator. This baseline is not simply the compiler with all register allocation turned off, as the front end of lcc generates a number of spurious temporary variables which need to be removed to get a ‘fair’ comparison. Consequently any variable that is defined and never used, or any variable that is used once and only once *immediately*¹ after it is defined, is removed. This gives the output one would expect from a naive compiler.

5.4 The Simulator

The test platform was the simulator for the UFO hardware. This simulator is an instruction level simulator; it does not attempt to mimic the low-level internal architecture of the UFO. It is nonetheless required to be an exact model of the processor at the visible register level. The simulator is also enhanced with a number of features that the hardware will not have. These are debugging aids, monitoring and host-architecture interface. Various cost models, which produce a cost of running any particular program, can be plugged in to simulate the relative effectiveness of the various register allocation algorithms with regards to different hypothetical architectures.

¹With no intervening use or definition of any other variable

5.5 Cost Models

Instructions are classified into groups. The cost model then assigns a cost to each instruction group. The instruction groups and costs for those groups in the different models can be seen in table 5.2. Cost models can have internal state to simulate pipelining, but this simple framework would not allow out-of-order execution to be modelled with any confidence.

Table 5.2: Instruction Costs

Model	Memory access	Predicted branch	Missed branch	Multiply	Divide	Stack	Other
Flat	1	1	1	1	1	1	1
Harvard	2	2	2	2	5	2	1
UFO	2 or 3	2	2	2	5	1	1
UFO (slow)	4 or 5	2	2	2	5	1	1
Pipelined	1 to 3	1	4	2	5	1	1
Stack Mapped	1 to 3	1	4	2	5	0 or 1	1

5.5.1 The Models

These models are not detailed models of real machines, or even realistic models, but serve to illustrate the effectiveness of the different register allocation techniques for different types of architecture.

Flat

All instructions have a cost of one. Not a realistic model, but it gives a dynamic instruction count.

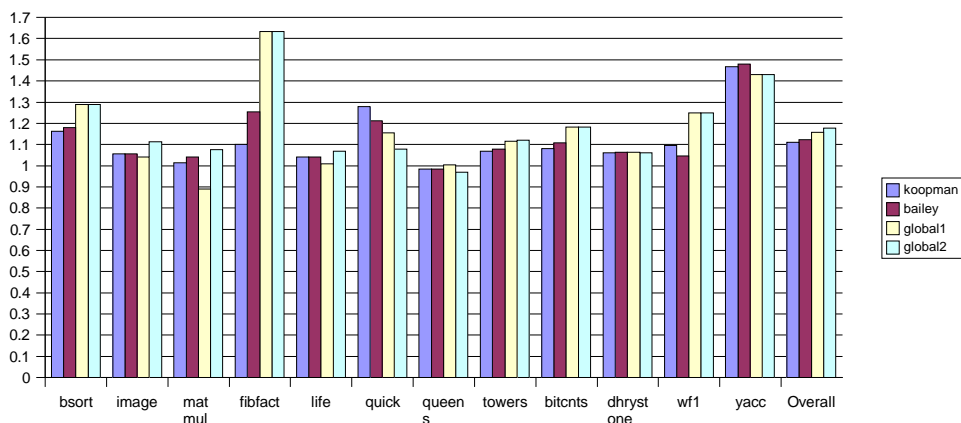
Harvard

This assumes no interference between data memory and instruction fetching. However, the processor is assumed to run twice as fast as the memory, so all simple instructions have a cost of one; memory, multiply and flow control instructions have a cost of two; divide has a cost of five.

UFO

This an attempt to model the UFO architecture. The UFO is a von Neumann architecture, so any memory accesses must stall if an instruction fetch is required. This is modelled by giving each memory access a cost of two, unless it immediately follows another memory access when it has a cost of three. Other instructions are costed the same as the Harvard model.

Figure 5.1: Relative performance for the Flat model



UFO Slow Memory

This is the same as the UFO model, but memory costs are doubled. This models an embedded system without cache.

Pipelined

This models a pipelined processor. All ALU and register-move operations are given a cost of one, except multiply(two) and divide (five). A simple branch predictor is presumed to compensate for increased misprediction penalty of the pipeline, and keep the model simple. Memory accesses have a cost of one, but due to the pipeline, stall the memory unit for another two cycles. This is modelled by increasing the cost of memory access to three immediately after a memory access, then reducing it by one per instruction until either another memory access occurs or the cost drops to one.

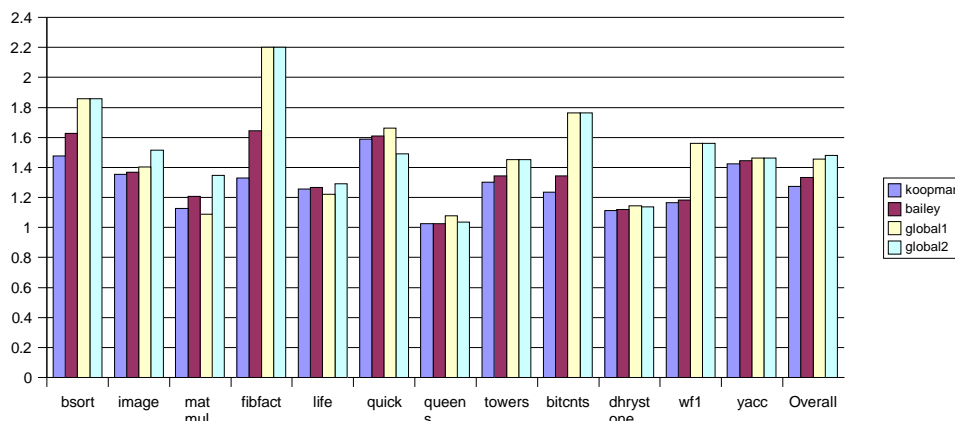
Stack Mapped

The processor decode is presumed to be able to remap registers in the decode so that up to two stack manipulations can be done per cycle in parallel with other operations. In other words up to two stack manipulations in a row have zero cost. Otherwise, it is the same as the pipelined processor.

5.6 Relative Results

The full results are shown in appendix B. Figures 5.1-5.5 show the performance of the different algorithms relative to the baseline. Larger numbers are better.

Figure 5.2: Relative performance for the UFO model



For the flat model, in other words the dynamic instruction count, the global optimisers gain little over Koopman’s and Bailey’s Algorithms. Global2 provides the best performance with a 12% reduction in executed instructions over Bailey’s Algorithm. However, this is not the case for the ‘quick’ benchmark, where the more sophisticated allocators get worse results. In the case of ‘queens’ only Global1 manages an improvement over the baseline at a paltry 0.4%, all the others actually degrade performance relative to the baseline.

In the more realistic UFO model, memory bandwidth becomes important and the performance of programs compiled with register allocators increases relative to the baseline. The relative performance of the best Global allocator relative to Bailey’s allocator remains about the same, with an 11% advantage.

For the UFO with slow memory, the global register allocators have significantly better performance. This is unsurprising as they are designed to reduce memory accesses to a minimum even at the cost of additional instructions. These extra instructions are more than compensated for by the reduced memory access costs in this case. The Global2 allocator outperforms Bailey’s by 20% in this case and is 82% better than the baseline case.

The pipelined and stack mapped models are fairly loose approximations to real architectures so the results should be viewed with caution. Nonetheless the need for register allocation to get good performance out of these sorts of architectures is clear.

5.6.1 Comparing the Two Global Allocators

Regardless of the performance models, the allocator Global2 gives the best performance for the benchmarks ‘image’, ‘matmul’ and ‘life’, whereas for the benchmarks ‘quick’ and ‘queens’ it is Global1 that gives the best performance.

Figure 5.3: Relative performance for the UFO with slower memory model

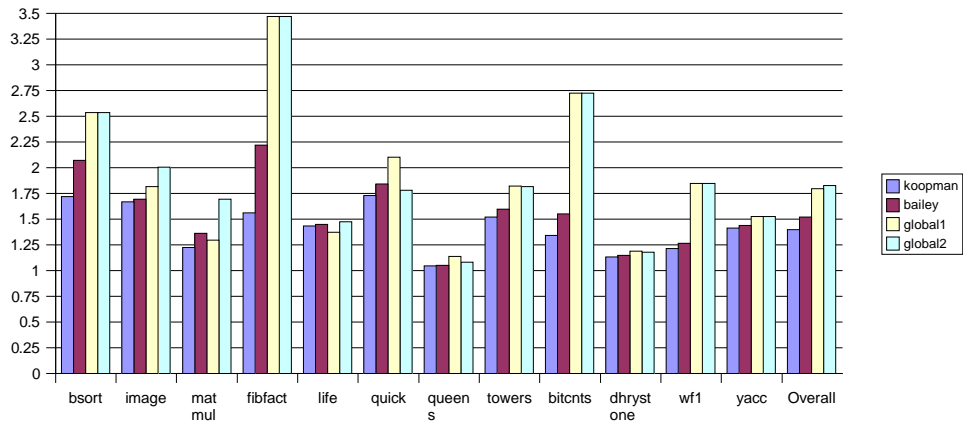


Figure 5.4: Relative performance for the Pipelined model

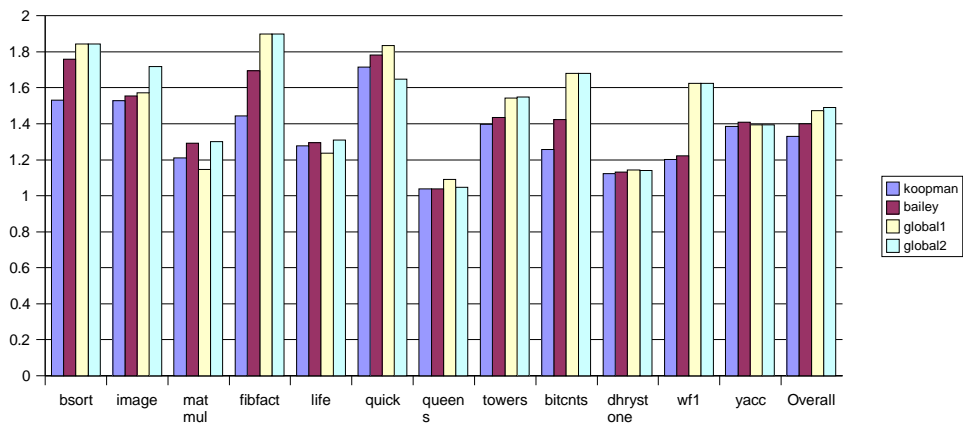
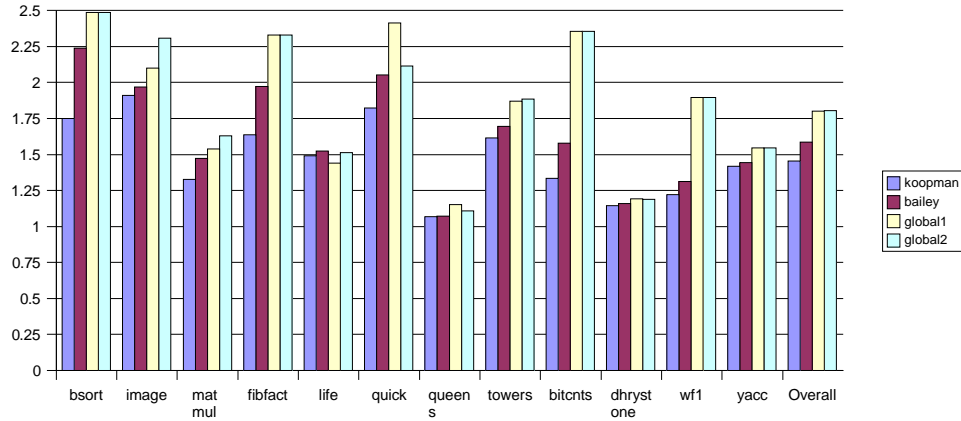


Figure 5.5: Relative performance for the Mapped model



Why is this so? To demonstrate what is going on, consider two programs `doubleloop.c` and `twister.c`. For the source code, assembly code and relative performance of these programs see Appendix F. In the program `doubleloop` two deeply nested loops occur one after the other. The two loops share no variables and the all the variables in the second loop have slightly higher reference counts. Global1 carries the variables for the second loop throughout the first, impeding the variables actually needed. In this highly contrived case, the Global2 version outperforms the Global1 version by 27%. So why does Global1 do better than Global2 sometimes? When there are a lot of very short blocks, each with varying variable use, Global2 can end up causing a lot of spilling, that is, storing one variable to memory, when another is needed in a register. Ways to prevent this and get the best of both algorithms are discussed in Section 6.3.2.

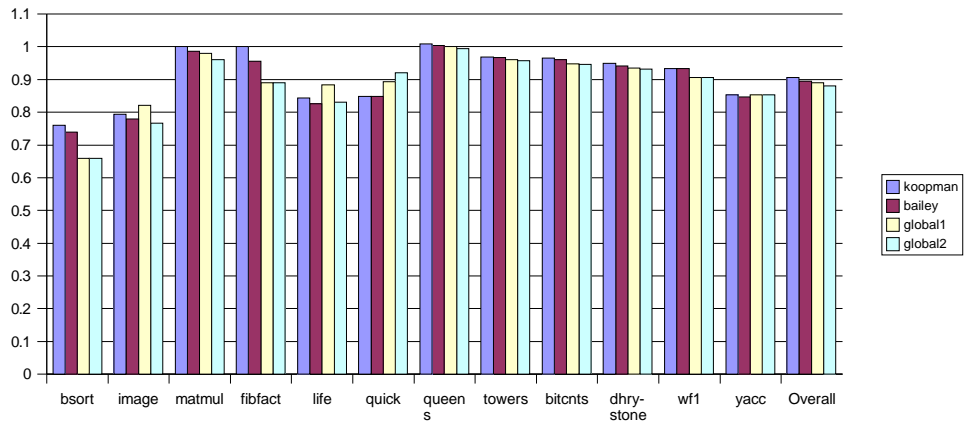
5.6.2 Relative program size

The relative program size shown in Figure 5.6 is the size of the executable less the size of an empty application, so as to remove all linked-in libraries and most symbolic information. As can be seen, register allocation also reduces program size but not greatly, as the difference between global allocation and Koopman's version is only about 4% overall.

5.7 Summary

The results show that for *all* the models the new global optimisers outperform both Bailey's and Koopman's algorithms. The difference was most marked for the models with the greatest difference between register access times and

Figure 5.6: Relative program size (smaller is better)



memory access times.

Chapter 6

Conclusions and Future Work

In this chapter, some conclusions are drawn from the data presented in the previous chapter and the goal of this thesis is reconsidered:

Demonstrating that stack machine for a low resource system, such a soft core processor on an FPGA, can be an equal or better alternative to a RISC machine

Has this goal been met?

6.1 Performance

The results in Chapter 5 demonstrated that the global optimisers improve performance for all the models tested. They achieve this by reducing the number of local variables transferred to and from memory. The compact nature of stack machine code combined with the reduced data traffic means that the overall memory bandwidth of a stack machine could be less than that of a RISC machine. Provided that the extra instructions required for stack manipulation can be done in parallel with ALU or memory operations, it is possible that the stack machine could be even faster than the RISC, at least for serial execution.

6.2 Limitations of the New Algorithms

Although the global register allocators can provide significant improvements to performance, there is currently no way of knowing how close to optimal they might be. They also make a few simplifications for ease of analysis and implementation that are not strictly necessary. The global ordering of the x-stacks is restrictive and the method of determining the x-stacks was largely determined by experiment.

6.2.1 Advanced Stack Architectures

Advanced stack architectures, such as pipelined and super-scalar machines, will need to be designed concurrently with their compilers in order to gain maximum performance. Although global register allocation does a good job of reducing memory accesses, it does increase the number of stack manipulations significantly. Stack manipulations are simple operations and can be done cheaply by hardware, so an advanced stack architecture would need to find ways to do stack manipulations concurrently with other operations. Since global register allocation for stack machines can reduce local memory traffic to low levels, access to local variables in memory should be rare enough that no special hardware need be dedicated to local variable access. This frees up hardware resources to be dedicated to stack manipulations.

6.3 Improving the Compiler

6.3.1 Improving the Global Register Allocator

A number of possible improvements to the allocator worthy of further investigation are:

- Determine the x-stacks around the most frequently executed blocks first, starting with inner loops and then adjusting the x-stacks around less executed blocks to suit.
- In order for a stack machine to access more registers it will need to reduce the types of stack manipulation operators to keep the instruction set to a reasonable size. Develop the allocator so as not to use, or to minimise the usage of *tuck*, and *rotate(down)* instructions, since all reachable values on the stack can be accessed with just *rotate(up)* and *copy* instructions.
- Test whether the allocator still works effectively in the presence of more advanced compiler optimisations, such as global common sub-expression elimination, and other optimisations that might produce a large number of temporary variables.
- In order to implement such advanced compiler optimisations, compilers typically use static single assignment form[2]. The allocator would need to be modified to handle this intermediate form, particularly the ϕ function.

6.3.2 A New Register Allocator

Currently the better of the two global allocators, *global2*, falls down when it introduces too many changes in the x-stack. It does this because it has no concept of loops and therefore cannot make good decisions as to which blocks

to prioritise. It currently does a good job where the loops are regular, but suffers when the flow graph becomes more complex. If the back end were able to find loops in the flow graph, then the register allocator could fix the x-stack for inner loops first, ensuring that spills would occur only outside of loops. A proposed, but unimplemented and thus untested, approach is outlined in Algorithm 11.

Algorithm 11 Proposed Algorithm

1. By using dominators to determine loops, or by analysis of the source code, make a static estimate of the dynamic execution frequency of each block. Note that the dynamic execution frequency could be determined by profiling in a profile driven framework.
 2. For each block, starting with the most frequently executed:
 - (a) Determine the optimum, or near optimum l-stack, for that block given any pre-existing x-stacks. For the first block there will be no x-stack. Note that a pre-existing x-stack can be extended downwards to incorporate more variables.
 - (b) Determine either or both of the upper parts of the x-stacks adjoining that block, if they have not already been determined, leaving the lower parts to be extended by subsequent propagation.
 3. Propagate or eliminate, if necessary, variables in the lower parts of the x-stack, until global consistency is achieved.
-

6.3.3 Stack Buffering

In all the models in chapter 5 it has been assumed that the stack never overflows into memory. Obviously this is not always a reasonable assumption. For all the non-recursive benchmarks run on the simulator, that is all but ‘fibfact’, no stack overflows occur with a reasonable sized (32 entry) stack buffer. For the recursive benchmark, a large number of overflows did occur. A detailed analysis of how the requirements for the stack buffer are changed by register allocation, especially for machines with larger numbers of accessible registers, would be valuable for designing future stack processors.

6.4 Advanced Stack Architectures

6.4.1 Pipelined Stack Architectures

The purpose of pipelining is to reduce the cycle time while maintaining the same number of instructions executed per cycle. Since the lifetimes of instructions

can be overlapped without duplication of the main processor components, pipelining can give significant performance gains at little extra cost.

The archetypal RISC pipeline[17] contains four or five stages, as follows:

1. Instruction Fetch
2. Decode and register fetch
3. Execute
4. Memory¹
5. Writeback

However since traditional stack architectures have had extremely simple instruction formats and the two top of stack registers have traditionally been hardwired to the ALU there is no need for a separate decode or writeback stage and pipelines have scarcely been worthy of the name, as follows

1. Instruction Fetch
2. Execute

To avoid the negative impact of the large number of stack manipulations generated by the compiler, stack manipulations should be done in parallel with other instructions. A simple way to do this is to embed simple stack manipulations into instructions; see appendix E for an example. Since these manipulations do not change the values in registers, merely their locations, it is tempting to do these manipulations in parallel with ALU operations, suggesting the following pipeline, as the stack is decoupled from the ALU:

1. Instruction Fetch
2. Stack manipulation
3. Execute
4. Writeback

This decoupling can be done by maintaining a map of where current register contents will be in future by analysing instructions but not executing them, and this can be done in the decode stage.

Since a stack processor would use conventional memory and buses, there would be no difference to the memory stage. The only difference to the instruction-fetch stage would be reduced bandwidth requirement, thanks to the smaller programs of stack machines.

¹This would depend on the memory architecture and relative processor-memory cycle time.

6.4.2 Super-Scalar Stack Machines

In order to get performance from a stack machine that is comparable with modern processors, super-scalar techniques, where several instructions can be executed simultaneously, will be needed. There is a large amount of instruction level parallelism available in stack machine code that can be exploited[26]. So far only one mechanism for exploiting instruction level parallelism in stack machines has been proposed[6]. Other mechanisms should be possible, but little research has been done in this field. Although the stack would appear to be a serial bottleneck, stack manipulations can be done with simple multiplexers, it would be possible to run the stack-manipulation unit several times faster than an ALU. The processor would then be able to issue sufficient instructions per cycle to exploit the available ILP². In order to minimise the number of stack manipulations, so that maximum ILP can be extracted, the range of stack manipulations for a super-scalar machine must be selected carefully so that the compiler can take full advantage of them.

If hardware can execute more than one instruction per cycle, the dependencies between instructions can also be a limiting factor to machine performance. It is the job of the compiler to generate code which enables the hardware to go as fast as possible. Therefore the compiler must not introduce artificial data dependencies between instructions and it must separate those instructions with real dependencies as much as possible.

Research into instruction scheduling for stack machines, to reduce these data dependencies, would seem to be the next important area of research in compilers for stack machines.

6.5 Summary

Although the performance of C programs compiled for stack machines can be improved using the new algorithms, they have limitations and it is impossible to determine if the goal of this thesis can be met without either real hardware or much more detailed hardware modelling. However, the new framework and algorithms do provide a significant step in the right direction.

²Instruction Level Parallelism

Appendix A

The Abstract Stack Machine

The abstract stack machine assembler is supposed to be largely self explanatory to anyone familiar with stack machines or the Forth language. The main instructions are summarised below. All ALU operations, add, sub, etc. and all test-and-branch instructions breq, brlt, etc. consume the top two items on the stack. ALU operations push the result back to the top of the stack(TOS). Note that *rot1* and *rrot1* are null operations. Also *copy1 = tuck1* and *rrot2 = rot2 = swap*.

<i>rot1</i>	No effect - serves to mark movement from l-stack to e-stack.
<i>rot2</i>	swap - Move from l-stack to e-stack.
<i>rot3</i>	Rotates the 3 rd item on the stack to TOS
<i>rot4</i>	Rotates the 4 th item on the stack to TOS
<i>copy1</i>	Duplicates TOS
<i>copy2</i>	Copies the 2 nd item on the stack to TOS
<i>copy3</i>	Copies the 3 rd item on the stack to TOS
<i>copy4</i>	Copies the 4 th item on the stack to TOS
<i>tuck1</i>	Duplicates TOS - identical to copy1 for the hardware, but this copies from the e-stack to the l-stack.
<i>tuck2</i>	Copies TOS to the 2 nd item on the stack
<i>tuck3</i>	Copies TOS to the 3 rd item on the stack
<i>tuck4</i>	Copies TOS to the 4 th item on the stack
<i>drop1</i>	Eliminates TOS
<i>drop2</i>	Eliminates the 2 nd item on the stack
<i>drop3</i>	Eliminates the 3 rd item on the stack
<i>drop4</i>	Eliminates the 4 th item on the stack
<i>rrot1</i>	No effect - serves to mark movement from e-stack to l-stack.
<i>rrot2</i>	swap - Move from e-stack to l-stack.
<i>rrot3</i>	Rotates TOS to the 3 rd item on the stack
<i>rrot4</i>	Rotates TOS to the 4 th item on the stack
<i>lit X</i>	Pushes the constant X to TOS
<i>!loc N</i>	Stores TOS to local variable N.
<i>@loc N</i>	Pushes the contents of local variable N to TOS.
<i>call f</i>	Calls the procedure f
<i>breq L</i>	Branch to L if TOS equals NOS.

Appendix B

Results

B.1 Dynamic Cycle Counts

Table B.1: Flat model

Optimiser	bsort	image	matmul	fbfact	life	quick	queens	towers	bitcants	dhystone	wfl	yacc	Overall
none	17011726	1433577	8451079	573112	2837827	2629531	268591	3528051	17693249	9429161	4008756	1222415	2828230
koopman	1462326	1355957	8326279	520712	2725660	2050801	272967	32995801	16359801	8889154	3658075	832412	2274299
bailey	1442426	1355957	8115079	456493	2165119	2165119	272967	3275023	15983782	8874154	3834459	825848	2253700
global1	1321425	1347337	9494549	350819	2810731	2273304	267457	3127395	14973834	8874153	3209346	854942	2180554
global2	1321425	1260509	7862549	350819	2656871	2435846	277077	3119204	15435415	8879151	3209346	854942	2148301

Table B.2: Harvard model

Optimiser	bsort	image	matmul	fbfact	life	quick	queens	towers	bitcants	dhystone	wfl	yacc	Overall
none	2522836	231218	12759225	922974	4720577	4410403	429269	5132586	26157154	16172202	6830260	1726075	3979269
koopman	1903736	1856649	11874561	753522	4023616	3131513	425709	4306898	22463747	14842186	6055894	1218675	3320692
bailey	1783935	1846848	11260161	623954	4005531	3140251	425503	4216777	21223808	14747184	6110264	1205546	3212539
global1	1561830	1780698	12529542	467443	4152719	3045400	406002	3881016	16869692	14462175	4833058	1193577	2962415
global2	1561830	1655160	10110390	467443	3915943	3374435	422186	3881017	17685037	14537173	4833058	1193577	2924996

Table B.3: UFO model

Optimiser	bsort	image	matmul	fbfact	life	quick	queens	towers	bitcants	dhystone	wfl	yacc	Overall
none	2902337	2714165	13903329	1028646	5590601	5399992	502863	5729784	29794082	20113884	8110018	1903412	4603792
koopman	1963637	2001300	12326001	774467	4445305	3398804	490917	4397025	24101986	18093862	6960778	1335649	3613779
bailey	1783936	1981698	11519601	625954	4409980	3357982	490663	4265932	22182576	17948860	6857289	1317273	3449838
global1	1561830	1905746	12760182	467443	4583040	3242557	466126	3881234	16896610	17598851	5192526	1301093	3151453
global2	1561830	1760804	10331430	467443	4324344	3618716	485818	3881235	17848631	17688849	5192526	1301093	3115386

Table B.4: UFO slow-memory model

Optimiser	bsort	image	matmul	fbfact	life	quick	queens	towers	bitcants	dhystone	wfl	yacc	Overall
none	4823150	4880196	22184171	162286	9528455	9762024	846062	9077463	49852300	36865911	13975770	2928247	7729966
koopman	2804850	2926048	18133787	1039601	6650146	5644478	807028	5977562	37159592	32480864	11509857	2073102	5523744
bailey	2325246	2876846	16309787	731628	6559715	5302392	806202	5674402	32169752	32105854	11056042	2035770	5079844
global1	1900930	2656920	17080470	467443	6928766	4644333	743772	4848075	18307763	31030827	7572631	1920103	4287930
global2	1900930	2405452	13096614	467443	6449814	5474952	781854	4864460	20243062	31290825	7572631	1920103	4248969

Table B.5: Pipelined model

Optimiser	bsort	image	matmul	fbfact	life	quick	queens	towers	bitcants	dhystone	wfl	yacc	Overall
none	2362434	2512283	11574439	827296	4912094	4987628	439281	4747365	27065830	18873089	7246005	1544516	3996455
koopman	1542734	1645558	957347	573117	3844479	2910228	423619	3396194	21527569	16828070	6036142	1114100	3007023
bailey	1342533	1616155	9868447	488388	3792172	2796924	423457	3308057	19022652	16693064	5935481	1096463	2833808
global1	1282430	1568622	10110702	435551	3974267	2718825	402960	3087168	16105161	16518064	4457299	1106918	2709122
global2	1282430	1433186	8891454	435551	3751039	3029176	420048	3070785	16924286	16543062	4457299	1106918	2690856

Table B.6: Stack mapped model

Optimiser	bsort	image	matmul	fbfact	life	quick	queens	towers	bitcants	dhystone	wfl	yacc	Overall
none	2242834	2464263	10829615	817294	4749213	4907405	427657	4534255	26183796	18462910	7079545	1440414	3888643
koopman	1282834	1290209	8150959	499333	3185319	2690577	389909	2808120	19599378	16122889	5793802	1014871	2650973
bailey	1002632	11251005	7354159	414604	3114623	2391449	399541	2677010	16604452	15902881	5394743	997233	2434217
global1	902326	1173772	7035653	350820	3295607	2033853	371657	2399187	11122810	15477874	3732467	931698	2142389
global2	902326	1067246	6642053	350820	3142827	2320201	385689	2382804	11831953	15537874	3732467	931698	2147379

B.2 Data memory accesses

Table B.7: Data memory accesses

Optimiser	bsort	image	matmul	fibfact	life	quick	queens	towers	bitcnts	dhrystone	wfl	yacc	Overall
none	740313	760500	3221985	253186	1433380	1570142	125407	1301445	7214085	5872496	2103821	406426	1162813
koopman	360613	375551	2462121	156511	848586	865984	117471	704397	4857343	5082487	1680136	289029	761535
bailey	260712	365750	2058921	92726	830499	760402	117265	638790	3991837	5002485	1558122	282464	665658
global1	159608	308220	1948832	39891	892618	557466	103284	450659	646831	4717477	906029	241401	444265
global2	159608	269510	1161680	39891	809702	723891	109848	458851	997636	4787477	906029	241401	445742

Appendix C

LCC-S Manual

C.1 Introduction

The port of lcc for stack machines retains much of the original lcc, but includes a largely new, stack-specific, back-end phase. The front-end has been slightly modified to provide extra data for the back-end. These modifications have been kept as small as possible. lcc-s can be ported to a new stack-machine by writing a new code generator specification and a few ancillary routines, much in the same way as lcc can be ported to a conventional machine.

C.2 Setting up lcc-s

Fixed location

lcc-s is hardwired to run from fixed locations, if installed there, no paths will be needed.

Unix: /usr/local/lib/lcc-s

Windows: C:\Program Files\lcc-s

Using environment variables

If you wish to install lcc-s to a different location, you will need to set up various paths. To set the path named MYPATH to the value MYDIR use the following command:

Unix: MYPATH=MYDIR

Windows: set MYPATH=MYDIR

Do not leave any trailing spaces. To add to a path use the following command.

Unix: MYPATH=\${MYPATH}:MYDIR

Windows: set MYPATH=%MYPATH%;MYDIR

Set up the following paths:

PATH You will need to **add** the location of lcc to your PATH variable.

LCCDIR You will need to **set** LCCDIR to the location of lcc-s's constituent programs.

LCCINCLUDE Setting this allows lcc-s to find the include files. There is no need to set this if the lcc-s header files are stored in LCCDIR/include.

LCC_LIBPATH Setting this allows lcc-s to find the system libraries. There is no need to set this if the libraries are stored in LCCDIR/lib.

LCC_CPP Set this to specify the C pre-processor for lcc to use. If not set then lcc-s will use gnu-cpp on Unix or its own pre-processor on Windows.

Command line

Setting the `-Ldir` option on the command line is equivalent to setting LCCDIR to dir. In fact the `-L` option can be used to override LCCDIR.

C.3 Command line options

Since lcc-s is a modified version of lcc, it supports most of the command line options of lcc, plus some additional options:

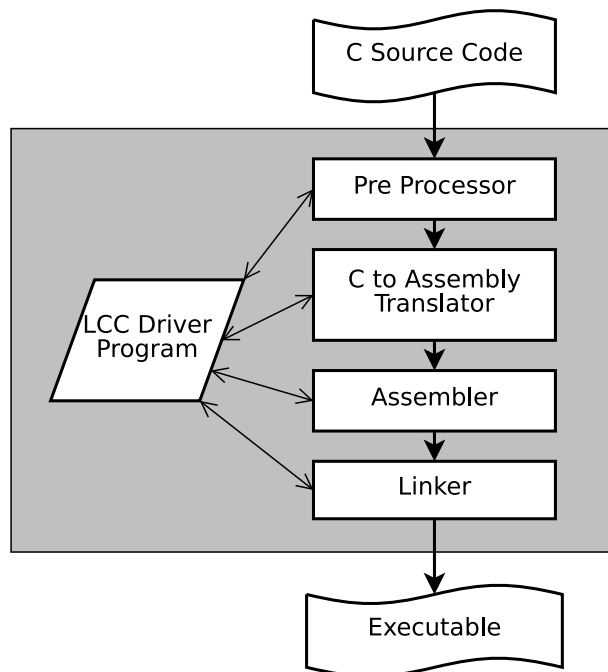
- A warn about nonANSI usage; 2nd -A warns more
- b emit expression -level profiling code; see `bprint(1)`
- Bdir/ use the compiler named 'dir/rcc'
- c compile only
- dn set switch statement density to 'n'
- Dname -Dname=def define the preprocessor symbol 'name'
- E run only the preprocessor on the named C programs and unsuffixed files
- g produce symbol table information for debuggers
- help or -? print help message on standard error
- Idir add 'dir' to the beginning of the list of `#include` directories
- lx search library 'x'

- Ldir Look in dir for components.
- M emit makefile dependencies; implies -E
- N do not search the standard directories for #include files
- n emit code to check for dereferencing zero pointers
- O0 (Oh zero) Turns off optimisation
- O1 Intra-block register allocation and peephole optimisation
- O2 Global register allocation and peephole optimisation
- Oname turns on the named optimisation - used typically with -O0. Mainly for experimental purposes.
 - o file leave the output in 'file'
 - P print ANSI -style declarations for globals on standard error
 - Q Turns off peephole optimisation
 - S compile to assembly language
- t -tname emit function tracing calls to printf or to 'name'
- tempdir=dir place temporary files in 'dir/'; default=/tmp
- Uname undefine the preprocessor symbol 'name'
 - v show commands as they are executed; 2nd -v suppresses execution
 - w suppress warnings
- W[pfal]arg pass 'arg' to the preprocessor, compiler, assembler, or linker

C.4 The structure of lcc-s

lcc-s has a very similar structure to lcc.

- lcc - The driver program
- cpp - The C Pre Processor
- scc - The stack specific translator - The equivalent of rcc in lcc.
- peep - The peephole optimiser - No equivalent in lcc
 - as - The assembler
 - link - The linker



C.5 Implementations

lcc-s has two current implementations lcc-ufo and lcc-utsa.

	lcc-ufo	lcc-utsa
cpp	gnu cpp on linux.	lcc's cpp on windows
scc	scc -target=ufo	scc -target=utsa
peep	peep-ufo	peep-utsa
as	as-ufo	as-utsa
link	link-ufo	link-utsa

C.6 The components

C.6.1 lcc

The driver program has to interface the components with the operating system. Currently four driver programs exist, although they share most source code. They are:

- lcc-ufo for Linux
- lcc-ufo for Windows
- lcc-utsa for Linux (Not yet)
- lcc-utsa for Windows

However, they all share the same interface and will be treated collectively as lcc-s. The command line interface to lcc-s is listed above.

C.6.2 scc

SCC translates pre-processed source code into assembler code. Options.

- target=*name* Targets the named architecture.
- log*name* Writes a log to *name* or lcc.log if *name* is missing.
- graph Writes a file for each function and each optimisation phase showing the complete flow graph and annotations. Written to the default temporary folder.
- verify Does a verification after each optimisation phase to test for corruption.
- O[012] Sets the optimisation level to 0, 1 or 2.
- O*name* Turns on the named optimiser.
- X*name* Turns off the named optimiser.

To access scc options from lcc-s use the -Wf flag, so to turn on logging from the driver use -Wf-log.

Targets

scc currently has four targets. Only two of which are used by lcc-s.

ufo The UFO architecture - invoked by lcc-ufo

utsa The UTSA architecture - invoked by lcc-utsa

abstract A simplified abstract machine architecture for demonstration purposes.

svg Outputs a SVG(Scalable Vector Graphics) representation of the intermediate code forest.

C.6.3 peep

The peephole optimiser is a table driven optimiser which reduces the number of stack manipulations in assembler code. It is highly portable. The UTSA and UFO version differ only in a 20 line instruction description file. The peephole optimiser can be run separately. It takes exactly two arguments and has no options.

peep-ufo source destination

C.6.4 as-ufo

The assembler for ufo is also table driven. It outputs objects files in a linkable a.out format. It takes the input file name as its sole input.

```
as-ufo [option] source
```

Options

-oname Names the output, otherwise 'a.out'

-g Inserts any debugging symbols into the object file.

C.6.5 link-ufo

The UFO linker expects input files in linkable a.out format and produces output in executable a.out format.

```
link-ufo [options] source1, source2, ..., sourceN
```

Options

-oname Names the output, otherwise 'a.out'

-lname Links in the named library named *libname.ufo*

-pathdir Add *dir* to the search path when looking for libraries.

-e Fail if no entry point.

-a Fail if any entry points.

-n Fails if any unresolved symbols.

-strip Remove symbols

-fix Remove relocation data.

-types=none No type checking.

-types=c "C" type checking. Default.

-types=strict Strict type checking.

-boot Insert startup code at zero.

-sp=value Initial SP offset, only valid with *-boot*

-rp=value Initial RP offset, only valid with *-boot*

To access linker options from *lcc-s* use the *-Wl* flag, so to strip symbols from the driver use *-Wl-strip*.

Future Options

`-textaddress` Define start address of text section

`-dataaddress` Define start address of data section

`-bssaddress` Define start address of bss section

Type checking

The compiler emits very simple type information about symbols. This is in the form `.type symbol type_symbol` For example `.type main F21` defines 'main' as a function consuming two stack cells and producing one. The linker can use this to ensure stack consistency. There are three levels of checking

None No checking is done

C Defined types are checked but undefined types, which are allowed in pre-ANSI C, are ignored. This is the default level.

Strict Defined types are checked. Undefined types are not allowed.

Appendix D

LCC Tree Nodes

Description of selected nodes used in this thesis and the new stack nodes.
See the ‘lcc 4.x Code-generation Interface’ paper[11] for more details.

LCC Node	Description
ADDRL	The address of a local variable
CNST	A constant
INDIR	Indirection - Fetch value from address
ADD	Add
MUL	Multiply
ASGN	Assign left child(value) to right child(address)
LSH	Left shift
JUMP	Jump
BRGT	Branch if left node > right node
RETURN	Exits procedure, returns child
VREG	Virtual register node - Replaced during register allocation.

Stack Node	Description
STACK	Address of stack register
COPY	Address of stack register, value retained when fetched.
TUCK	Same value as child, copies into stack as side effect.

Appendix E

Proposed Instruction Set for High Performance Stack Machine

This appendix covers the rationale and outline instruction set for a high performance stack machine.

E.1 Development path

The lcc-s compiler is able to generate code that takes full advantage of stack architecture, yet in order to reduce local variable accesses to RISC levels, more available registers will be required. Increasing the number of registers to eight would appear to be sufficient, but this would need thirty six stack manipulation instructions for the current ‘orthogonal’ stack access. This can be trimmed down to twenty two by removing the tuck and drop instructions, but this is still a lot, so is it possible to reduce it further? By removing the rotate(down) instructions, the number of instructions can be brought down to sixteen, but there is no way to store a value anywhere but the top of the stack. This means that Koopman’s algorithm cannot be implemented, however there is no reason why an effective register allocator cannot be designed to meet this constraint.

Out of order execution

In order to support out of order execution, successive useful instructions¹ must not have dependencies. In order to avoid this the compiler will have to insert *two* stack manipulations between each useful instruction, in fact even with the rotate(down) instruction, most each useful instructions would still need to be separated by a pair of stack manipulations. This means that, not

¹In other words, not stack manipulations, which merely change the arrangement of values and do not change the values themselves

only would the instruction issuing engine have to do a huge number of stack manipulations, the code size would increase unacceptably.

Embedded stack manipulations

Since, for a typical operation such as `add`, the required sequence of instructions would be something like `copy4 rot6 add` it makes sense to embed the stack manipulations within the instruction, so the `add` become `add c4, r6`. The `c` meaning copy, that is leave the value in the stack and use a copy, and `r` meaning rotate, that is pull the value out of the stack for use. Since the copy or rotate packs nicely into four bits, $\frac{3-1}{\text{Register}} \mid \frac{0}{\text{Copy?}}$, a regular instruction format can be used which will assist rapid decoding and issuing of instructions, thus allowing multiple instructions to be issued per cycle. Note that the `rot1` form is not redundant any more, since the addressing is no longer implicit.

E.2 Instruction Format

Fixed 16 bits width. $\frac{15-12}{\text{Category}} \mid \frac{11-8}{\text{Field0}} \mid \frac{7-4}{\text{Field1}} \mid \frac{3-0}{\text{Field2}}$

E.2.1 Categories

Opcode	Name	Stack effect
0	Arithmetic	-1 to +1
1	Arithmetic Immediate	-1 to +1
2	Logical	-1 to +1
3	Fetch	0 or +1
4	Store	-1 or 0
5	Call	0
6	Long Call	0
7	Local Fetch	+1
8	Local Store	-1 or 0
9	Jump	0
10	Long Jump	0
11	BranchT	-1
12	BranchF	-1
13	Test	-1 to + 1
14	Literal	+1
15	Special	?

E.2.2 Arithmetic

15 - 12	11 - 9	8	7 - 5	4	3 - 1	0
Arithmetic	Operation	Carry	Register1	Preserve	Register2	Preserve

If the Carry bit is set, 1 is added to the result. K is the K combinator, that is the result is equal to the first operand, regardless of the second. Shifts are barrel shifts.

Arithmetic operations

1. Add
2. Sub
3. Left shift
4. Logical right shift
5. Arithmetic right shift
6. Multiply
7. Negate
8. K

E.2.3 Arithmetic Immediate

15 - 12	11 - 9	8	7 - 5	4	3 - 0
Arithmetic	Operation	Carry	Register1	Preserve	Constant

Operations are as per Arithmetic

E.2.4 Logical

15 - 12	11	10 - 9	8	7 - 5	4	3 - 1	0
Logical	0	Operation	Invert	Register1	Preserve	Register2	Preserve
Logical	1	Operation	Invert	Register1	Preserve	Constant	

Logical operations

1. And
2. Or
3. Xor
4. K

E.2.5 Fetch

15 - 12	11 - 9	8	7 - 0
Fetch	Address Register	Preserve	Offset (signed)

E.2.6 Store

Address is always TOS and is always preserved

15 - 12	11 - 9	8	7 - 0
Store	Value Register	Preserve	Offset (signed)

E.2.7 Local Fetch

15 - 12	11 - 8	7 - 0
Local Fetch	0000	Offset (signed)

E.2.8 Local Store

15 - 12	11 - 9	8	7 - 0
Local Store	Value Register	Preserve	Offset (signed)

E.2.9 Jump

15 - 12	11	10 - 0
Jump	Link	Offset (signed)

E.2.10 Long Jump

15 - 12	11	10 - 0	Next Packet (16 bits)
Long Jump	Link	Offset (top 11 bits)	Offset (lower 11 bits)

E.2.11 BranchT

15 - 12	11 - 0
BranchT	Offset (signed)

E.2.12 BranchF

15 - 12	11 - 0
BranchF	Offset (signed)

E.2.13 Test

15 - 12	11 - 9	8	7 - 5	4	3 - 1	0
Test	Operation	0	Register1	Preserve	Register2	Preserve
Test	Operation	1	Register1	Preserve	Constant	

Test operations

1. Equals
2. Not equals
3. Less than
4. Greater than or equals

5. Less than or equals

6. Greater than

E.2.14 Literal

15 - 12	11 - 0
Literal	Constant

E.2.15 Special

These operations are not necessarily unusual or rare, they just don't fit into the general framework.

15 - 12	11	10 - 8	7 - 5	4	3 - 0
Special	0	Operation	Register1	Preserve	0000
Special	1	Operation		Constant	

Special (register) operations

1. Icall
2. Ijump
3. Tuck - Move TOS to Register1. Preserve = 1 is Tuck, Preserve = 0 is Rotate down.
4. Read special register
5. Write special register
6. Coprocessor

Special (constant) operations

1. No operation
2. Exit - Constant is ignored
3. Trap
4. TestFlags - Constant is mask.
5. Increment frame pointer

Appendix F

Quantitative Comparison of Global Allocators

Figure F.1: doubleloop.c

```
#include "stdio.h"

int main(int p, int q, int x, int y)
{
    int r = 0;
    int z = 0;
    int i,j,k,l,m,n;
    for (i = 0; i < 10; i++) {
        for (j = 0; j < 10; j++) {
            for (k = 0; k < 10; k++) {
                r += q + q;
            }
        }
    }
    printf("%d\n", r);
    for (l = 0; l < 10; l++) {
        for (m = 0; m < 10; m++) {
            for (n = 0; n < 10; n++) {
                z += x + y + l + m + n;
                z += x + y;
            }
        }
    }
    printf("%d\n", z);
    return 0;
}
```

Figure F.2: twister.c

```
#include "stdio.h"

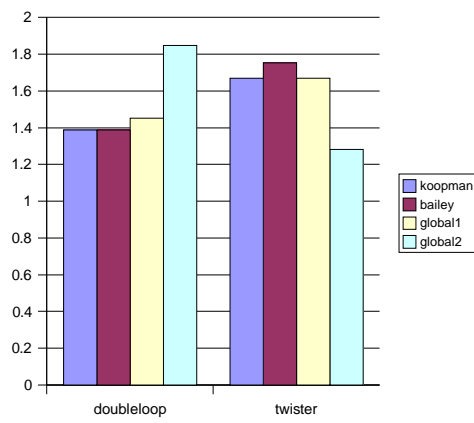
int twister(int a, int b, int c, int d, int e, int f)
{
    int i = a;
    int x = a;
    while (a < b) {
        --b;
        while (c < d) {
            --d;
            if (e < f)
                x = twister(a, b, c, d, a, b);
        }
    }
    return x;
}

int main(void) {
    int i, j = 0;
    for (i = 0; i < 1000; i++) {
        j += twister(1, 2, 3, 4, 0, 0);
    }
    return j;
}
```

Figure F.3: Cycles for UFO model

	doubleloop	twister
none	80241	177014
koopman	57801	106014
bailey	57801	101010
global1	55275	106007
global2	43453	138007

Figure F.4: Relative speed



Appendix G

Source Code

This appendix describes how to build lcc-s from the sources. The source for the back-end of lcc-s is included on the CD. To get the source code for the front-end you will have to download it from <http://www.cs.princeton.edu/software/lcc/>. The front end has been modified slightly to inform the back-end about the jumps in `switch` statements. The following code

```
{
    int i;
    (*IR->swtch)(equated(cp->u.swtch.deflab));
    for (i = 0; i < cp->u.swtch.size; i++)
        (*IR->swtch)(equated(cp->u.swtch.labels[i]));
}
```

should be inserted immediately after:

```
case Switch:
```

and before the following

```
break;
```

in the function

```
void gencode(Symbol caller [], Symbol callee [])
```

In the file "c.h" The structure 'interface' should have the extra member

```
void (*swtch) (Symbol);
```

inserted immediately before

```
Xinterface x;
```


Bibliography

- [1] Usenet nuggets. *SIGARCH Comput. Archit. News*, 21(1):36–38, 1993.
- [2] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–11, New York, NY, USA, 1988. ACM Press.
- [3] G. M. Amdahl, G. A. Blaauw, and F. P. Brooks, Jr. Architecture of the IBM System/360. *IBM Journal of Research and Development*, 44(1/2):21–36, Jan./Mar. 2000. Special issue: reprints on Evolution of information technology 1957–1999.
- [4] C. Bailey. *Optimisation Techniques for Stack-Based Processors*. PhD thesis, jul 1996.
- [5] C. Bailey. Inter-boundary scheduling of stack operands: A preliminary study. *Proceedings of EuroForth 2000*, pages 3–11, 2000.
- [6] C. Bailey. A proposed mechanism for super-pipelined instruction-issue for ILP stack machines. In *DSD*, pages 121–129. IEEE Computer Society, 2004.
- [7] L. Brodie. *Starting Forth: An introduction to the Forth language and operating system for beginners and professionals*. Prentice Hall, second edition, 1987.
- [8] J. L. Bruno and T. Lassagne. The generation of optimal code for stack machines. *J. ACM*, 22(3):382–396, 1975.
- [9] L. N. Chakrapani, J. C. Gyllenhaal, W. mei W. Hwu, S. A. Mahlke, K. V. Palem, and R. M. Rabbah. Trimaran: An infrastructure for research in instruction-level parallelism. In R. Eigenmann, Z. Li, and S. P. Midkiff, editors, *LCPC*, volume 3602 of *Lecture Notes in Computer Science*, pages 32–41. Springer, 2004.
- [10] C. W. Fraser and D. R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
- [11] C. W. Fraser and D. R. Hanson. The lcc 4.x code-generation interface. Technical Report MSR-TR-2001-64, Microsoft Research (MSR), July 2001.
- [12] C. W. Fraser, D. R. Hanson, and T. A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Lett. Program. Lang. Syst.*, 1(3):213–226, 1992.
- [13] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Second Edition*. Addison Wesley, 2000.
- [14] H. Gunnarsson and T. Lundqvist. Porting the gnu c compiler to the thor microprocessor. Master’s thesis, 1995.
- [15] D. R. Hanson and C. W. Fraser. *A Retargetable C Compiler: Design and Implementation*. Addison Wesley, 1995.

- [16] J. R. Hayes and S. C. Lee. The architecture of FRISC 3: A summary. In *Proceedings of the 1988 Rochester Forth Conference on Programming Environments*, Box 1261, Annandale, VA 22003, USA, 1988. The Institute for Applied Forth Research, Inc.
- [17] J. L. Hennessy and D. A. Patterson. *Computer Architecture — A Quantitative Approach*. Morgan Kaufmann Publishers, third edition, 2003.
- [18] P. Koopman, Jr. A preliminary exploration of optimized stack code generation. *Journal of Forth Application and Research*, 6(3):241–251, 1994.
- [19] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
- [20] M. Maierhofer and M. A. Ertl. Local stack allocation. In *CC '98: Proceedings of the 7th International Conference on Compiler Construction*, pages 189–203, London, UK, 1998. Springer-Verlag.
- [21] G. J. Myers. The case against stack-oriented instruction sets. *SIGARCH Comput. Archit. News*, 6(3):7–10, 1977.
- [22] E. I. Organick. *Computer system organization: The B5700/B6700 series (ACM monograph series)*. Academic Press, Inc., Orlando, FL, USA, 1973.
- [23] S. Pelc and C. Bailey. Ubiquitous forth objects. *Proceedings of EuroForth 2004*, 2004.
- [24] J. Philip J. Koopman. *Stack computers: the new wave*. Halsted Press, New York, NY, USA, 1989.
- [25] M. Shannon and C. Bailey. Global stack allocation. In *Proceedings of EuroForth 2006*, 2006.
- [26] H. Shi and C. Bailey. Investigating available instruction level parallelism for stack based machine architectures. In *DSD*, pages 112–120. IEEE Computer Society, 2004.
- [27] R. M. Stallman. *Using and Porting the GNU Compiler Collection, For GCC Version 2.95*. Free Software Foundation, Inc., pub-FSF:adr, 1999.
- [28] J. William F. Keown, J. Philip Koopman, and A. Collins. Performance of the harris rtx 2000 stack architecture versus the sun 4 sparc and the sun 3 m68020 architectures. *SIGARCH Comput. Archit. News*, 20(3):45–52, 1992.

Index

- A global approach, 60
- A Global register allocator, 58
- A new register allocator, 80
- Abstract stack machine, 15
- Additional annotations for register allocation, 45
- Advanced stack architectures, 80
- Advantages of the stack machine, 12
- An example, 28
- Analysis, 51
- Analysis of Bailey's Algorithm, 57

- Back-end infrastructure, 40
- Bailey's Algorithm, 56
- Benchmarks, 71

- Chain allocation, 64
- Change of semantics of root nodes in the lcc forest, 44
- Classifying stack machines by their data stack, 22
- Comparing the two global allocators, 75
- Compiler correctness, 71
- Compilers, 17
- Compilers for stack machines, 17
- Context for This Thesis, 19
- Cost models, 73
- Current Stack Machines, 13

- Data memory accesses, 89
- Data structures, 39
- Description, 56
- Determining x-stacks, 61
- Dynamic Cycle Counts, 88

- Edge-sets, 27

- Final allocation, 66

- GCC vs LCC, 33
- Goal, 20

- Heuristics for determining the x-stacks, 62
- History, 12
- How the logical stack regions relate to the real stack, 27

- Implementation, 58
- Implementation of Koopman's Algorithm, 52
- Implementation of optimisations, 40

- Improving the Compiler, 80
- Improving the global register allocator, 80
- Initial attempt, 35

- Koopman's Algorithm, 51
- Koopman's algorithm – initial transformations, 54

- L-stack allocation, 64
- LCC, 35
- LCC trees, 40
- Limitations of the register allocation algorithms, 79

- Modifying the machine description, 47

- Optimisation, 38
- Ordering of variables., 61
- Outline Algorithm, 61

- Peephole optimisation, 68
- Performance of Stack machines, 14
- Pipelined stack architectures, 81
- Producing the flow-graph, 38
- Program flow, 39
- Propagation of preferred values, 62

- Register allocation for stack machines, 18
- Relative results, 74
- Representing stack manipulation operators as lcc tree-nodes., 40
- Representing the stack operations., 43

- Solving the address problem., 47
- Stack Machines, 11
- Stack machines, more RISC than RISC?, 13
- Stack manipulation, 16
- Stack regions, 23
- Super-scalar stack machines, 83

- The calling convention for variadic functions, 49
- The choice of compiler, 33
- The evaluation region, 23
- The hardware stack, 21
- The improved version, 38
- The local region, 24
- The models, 73
- The parameter region, 24
- The register allocation phase, 38

The semantics of the new nodes, 42
The simulator, 72
The stack, 11, 15
The transfer region, 25
The UFO architecture, 19
Thor microprocessor, 34
Tree flipping, 38
Tree labelling, 45
Tuck, 43

UFO machine description, 48
Using the regions to do register allocation, 25
UTSA machine description, 46

Variadic function preamble, 49
Variadic functions for stack-machines, 49
Views of the stack, 22