FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Implementing a Global Register Allocator for TCC

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Sebastian Falbesoner
Matrikelnummer 0725433

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Martin Ertl

Wien, 22.08.2014

(Unterschrift Verfasser) (Unterschrift Betreuer)

Erklärung zur Verfassung der Arbeit

Sebastian Falbesoner
Pachmüllergasse 1/14, 1120 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

In Dankbarkeit gewidmet
meinen Eltern Lotte und Josef

Abstract

Register allocation is a long-standing research topic of computer science that has been studied extensively over the last decades. Its goal is to map a theoretically infinite number of program variables onto a finite, small set of CPU registers during compilation. Even though caches try to bridge the gap between register and memory access time, keeping as many values in registers as long as possible is crucial for good performance. Hence register allocation is still considered to be one of the most important compiler optimizations.

The present diploma thesis describes the process of implementing a register allocator for TCC, a small single-pass C compiler written in C[1]. While TCC is very fast (up to a magnitude faster than `gcc -O0` for the x86 architecture), it produces code that is quite inefficient – some naive kind of register allocation is done only on the basis of statements. Our goal of implementing register allocation done on the basis of whole functions, that is, *global register allocation*, should hence result in a notable performance increase.

As prerequesite for register allocation in TCC, a proper IR (*Intermediate Representation*) needs to be generated in a first pass. On top of this internal representation, live variable analysis, register allocation and finally the code generation is then performed. We determine the variable live intervals with an extraordinarily simple approach that avoids the usual costly data-flow analysis techniques – it trades off the accuracy of the calculated intervals for higher execution speed. For the register allocation we use *Linear Scan*, a strategy that doesn't abstract the problem to graph coloring, but rather takes a simple approach involving the linear traversal of variable live intervals. While linear scan is very fast, it is stated that the generated code is almost as efficient as with graph coloring, making it a popular choice for JIT compilers.

As target machine, ARM, a classical RISC architecture, now widespread especially in mobile phones and in many other embedded systems, is chosen. Since data processing operands must not reside in memory in this *Load/Store architecture*, register allocation is even more important.

We determine the performance gain with various test applications, predominantly from the benchmark suite *MiBench*. Additionally, we compare compile-time as well as run-time performance with the widespread compiler `gcc`.

The execution time speedup of code generated by our implementation is about 32% on average (compared to the original TCC), while the compile-time is increased only marginally and is still an order of magnitude lower than for `gcc` without any optimization. Consequently, our register allocator implementation is an attractive trade-off for dynamic code generation with TCC.

[1] of course, TCC itself can also be compiled with TCC

Kurzfassung

Registerzuteilung (engl. *register allocation*) ist eines der ältesten Forschungsthemen in der Informatik, welches im Laufe der letzten Jahrzehnte ausgiebig untersucht wurde. Das Ziel besteht darin, bei der Übersetzung von Programmen eine theoretisch unendliche Anzahl an Programmvariablen einer endlichen, kleinen Menge an verfügbaren Prozessor-Registern zuzuordnen. Auch wenn mit Hilfe von Caches die Kluft zwischen Register- und Speicherzugriffszeit überbrückt wird, so ist es doch wichtig, so viele Werte wie möglich so lange wie möglich in Registern zu halten, um eine gute Leistung zu erzielen. Somit wird Registerzuteilung nach wie vor als eine der wichtigsten Compileroptimierungen angesehen.

Die vorliegende Diplomarbeit beschreibt den Vorgang der Implementierung eines Registerallokators für TCC, einem kleinen Ein-Pass C-Compiler, der selbst auch in C geschrieben ist[2]. TCC ist sehr schnell (bis zu zehnmal schneller als `gcc -O0` für die x86-Architektur), erzeugt jedoch sehr ineffizienten Code – eine sehr einfache Art der Registerzuteilung wird nur auf Basis von Anweisungen gemacht. Unser Ziel, Registerzuweisung auf Basis von ganzen Funktionen zu implementieren (d.h. einen *globalen Registerallokator*), sollte somit einen deutlichen Geschwindigkeitszuwachs bringen.

Als Voraussetzung für einen Registerallokator in TCC muss ein passender Zwischencode (engl. IR, *Intermediate Representation*) in einem ersten Durchlauf erzeugt werden. Auf Basis dieser internen Repräsentation können dann die Lebensdaueranalyse der Variablen, Registerzuteilung und letztendlich die Codeerzeugung durchgeführt werden. Wir bestimmen die Lebensdauerintervalle der Variablen mit einem außergewöhnlich einfachen Algorithmus, der die üblichen Techniken der Datenflussanalyse vermeidet – die Genauigkeit der ermittelten Intervalle wird zugunsten einer höheren Ausführungsgeschwindigkeit geopfert. Für die Registerzuteilung benutzen wir *Linear Scan*, eine Strategie welche das Problem nicht auf Graphenfärbung reduziert, sondern mit einer sehr einfachen Vorgangsweise arbeitet, welche lediglich einen sequentiellen Durchlauf von den aktiven Intervallen (engl. *live intervals*) der Variablen benötigt. Obwohl die Strategie sehr schnell ist, wird angegeben dass der erzeugte Code beinahe so effizient ist wie jener von Graphenfärbungs-Algorithmen. Aufgrund dieser Tatsache ist Linear Scan sehr beliebt für JIT Compiler.

Als Zielarchitektur wird ARM gewählt, eine klassische RISC-Architektur, die mittlerweile weitverbreitet in Mobiltelefonen als auch in vielen anderen eingebetteten Systemen ist. Da in dieser *Load/Store-Architektur* die Operanden bei Datenverarbeitungsoperationen nicht im Speicher liegen dürfen, ist Registerzuteilung hier besonders wichtig.

Wir ermitteln den Geschwindigkeitszuwachs der Implementierung mit verschiedensten Testprogrammen, die vorwiegend von der Benchmark-Sammlung *MiBench* stammen. Zusätzlich vergleichen wir sowohl die Übersetzungs-Zeit- als auch die Laufzeit-Leistung mit dem weitverbreiteten Compiler `gcc`.

Unsere Implementierung erzeugt Code der im Durchschnitt um ca. 32% schneller ist (verglichen mit dem originalen TCC), wobei sich die Übersetzungs-Zeit nur marginal erhöht und immer noch über zehn Mal niedriger ist als für `gcc` ohne jegliche Optimierung. Somit ist unsere Implementierung des Registerallokators ein guter Kompromiss für dynamische Codeerzeugung mit TCC.

[2] natürlich lässt sich TCC auch mit TCC kompilieren

Contents

CHAPTER 1

Introduction

Compilers usually have the two main goals of generating *correct* and *fast* code. For dynamic code generators however, compile-time speed can be even an more important factor than run-time speed. *TCC* is a C-compiler which is very fast and well-suited for dynamic code generation, but it creates fairly inefficient code. By extending TCC with a *register allocation* phase, the code quality can be increased significantly with only a marginal loss of the high compilation speed.

1.1 Register Allocation

Due to the gap between register and memory access time in computers, one of the fundamental, obvious rules for writing efficient programs is to take advantage of the register set as much as possible. Processing data in main memory can be up to three magnitudes slower than directly on the CPU-near register set. The advent of data caches sped up operations on memory on average, but the rule still holds and bad register utilization can make drastic differences in program performance.

However, nowadays programs are rarely written in assembly language anymore, but higher-level languages are used where the programmer doesn't have control about the register set. Hence this is now the task of the compiler, called *register allocation*. That optimization phase is responsible for assigning a theoretically unbounded number of program variables to a bounded number of physical registers. If the number of active variables exceeds the available registers, registers need to be *spilled*, i.e. stored to a dedicated memory location, usually the stack.

Register allocation is considered to be a very central optimization, and virtually every widespread compiler nowadays implements it. Hennessy and Patterson state in their famous standard work about computer architecture the following:

> "*Because of the central role that register allocation plays, both in speeding up the code and in making other optimizations useful, it is one of the most important – if not the most important – optimizations.*"[1]

[1] [HP90]

Register allocation strategies are divided into *local* and *global*, where local algorithms see basic blocks as the basic unit, and global ones aim at performing register allocation over whole functions.

The traditional solution of globally approaching the register allocation problem is to reduce it to a graph coloring problem, but alternative strategies also exist. This is described in more detail in the next chapter.

1.2 TCC – The Tiny C Compiler

TCC, short for *Tiny C Compiler* (also called *TinyCC*) is a small C compiler written by Fabrice Bellard[2]. While the first versions targeted only the x86 architecture, support for x86-64, ARM and TMS320C67 was added over the years by various contributors.

TCC has a number of interesting properties that distinguish it from other C compilers:

- **Speed**: TCC is a single-pass compiler that directly generates binary code "on the fly", without relying on any external tools, not even an assembler. This makes it obviously very fast – a compilation speed test for the *Links Browser* project[3] found that TCC is about nine times faster than `gcc` on the x86 architecture, where the time includes assembly and linking. For the ARM architecture, we observed a speedup factor of even 12 and above (see chapter 9).

- **Size**: TCC is very small. For example, the x86 TCC executable needs about 100kB. This, together with the speed factor, makes it an interesting option to use it on systems with limited resources, e.g. rescue disks or embedded systems.

- **Scripting support**: Provided with the `-run` switch on the command line, TCC directly compiles, assembles and links the given C files into memory and executes it. This enables the creation of command line scripts in C.

- **Dynamic code generation**: With `libtcc`, the core library of the compiler, TCC can be used as backend for dynamic code generation. By simply passing C code strings to a library function, it can be compiled and then run at runtime. There also exist bindings to enable scripting languages to access `libtcc`, for example Luatcc for Lua[4].

TCC implements all of the ANSI C (C89/C90) standard and supports, apart from complex and imaginary numbers and variable length arrays, also the newer C99 standard[5]. Additionally, it also supports some GNU extensions and provides inline assembly support for x86.

The project is distributed under the GNU Lesser General Public License (LGPL). Currently it is maintained with the version-control system `git`; the repository can be received via the command `git clone git://repo.or.cz/tinycc.git`. An online version of it, including instructions on how to supply patches, is found on `http://repo.or.cz/w/tinycc.git`.

[2] see `http://bellard.org/tcc/`

[3] see `http://links.twibright.com/`

[4] see `http://piratery.net/luatcc/`

[5] see the TCC reference documentation `http://bellard.org/tcc/tcc-doc.html`

1.3 ARM Architecture

ARM is a popular 32-bit RISC architecture that is mostly known for its use in mobile phones. The simple instruction set with fixed-size encoding, the relatively large uniform register set (15 general purpose registers) and simple addressing modes (Load/Store architecture) make it a pleasant choice for code generation.

Since ARM is especially widespread in use in embedded domains, our hope is to make TCC an attractive option for those systems. We develop the implementation on the Raspberry Pi (see Appendix B), a credit-card sized home computer that surely could profit from having an improved TCC, as fast alternative to `gcc` that still generates efficient code.

1.4 Goal

The obvious drawback of TCC is that with the current code generation method, it is unable to generate optimized code. Hence an additional pass must be generated that allows the manipulation of *intermediate code* for global register allocation. On top of this IR, data-flow analysis, register allocation and finally code generation can then be implemented.

We have two conflicting goals on the implementation: While the quality of the generated code in terms of speed should clearly increase, TCC should still retain its notable property of high-speed compilation.

1.5 Source Code

The source code of the implementation is version-controlled with Git and can be found on BitBucket (⇒ `https://bitbucket.org/theStack/tccls_poc.git`). It was started with the latest official version 0.9.26 of TCC as basis, and we also applied all changes from the "mob" branch [6] up to July 24th, 2013.

1.6 Outline

Chapter 2 gives an overview of register allocation strategies, predominantly graph coloring algorithms and linear scan. New research work in this field that has been done within the last decade, like register allocation for SSA-programs is also covered shortly.

Chapter 3 describes how the implementation of TCC works, with a focus on those parts that we need to modify for our register allocator. Several important decisions (choice of register allocation algorithm and target architecture) and simplifying restrictions are established in chapter 4. It also contains a short introduction on the development environment, the Raspberry Pi, and gives an outline on the implementation plan.

The base of our implementation, the generation of an IR (intermediate representation) that abstracts variable accesses with virtual registers, is described in chapter 5. The next step, the

[6] mob branches have the special property that everyone can push changes to it without authentication, similar to the concept of Wikipedia; see `http://repo.or.cz/h/mob.html` for more informations

variable live interval calculation, is covered in chapter 6. Chapter 7 describes the mapping of virtual registers to physical registers, that is, the global register allocation with linear scan. Finally, chapter 8 covers the important aspect of generating the target code out of the register-allocated IR. This includes generating spill code.

Measurements of compile-time and run-time efficiency, compared to other compilers, are presented in chapter 9.

Chapter 10 concludes the thesis and gives some ideas on how the implementation could be further improved in the future.

Appendix A covers the ARM architecture, including the *Procedure Call Standard* (ARM EABI) that plays an important role for the register allocation. Appendix B lists details about our target machine, the Raspberry Pi. Appendix C contains relevant code snippets of our implementation.

CHAPTER 2

Register Allocation and Related Work

2.1 Motivation

An important criterion for the code generating part of an optimizing compiler is *where* the instruction operands (this is, predominantly, program variables and temporary values) should be stored: in memory or in registers? Obviously, due to the nature of the memory hierarchy [HP90], register access is much faster than memory access.

Even if we assumed that all instructions take the same amount of time, keeping values in registers can be beneficial in terms of instruction count. On typical Load/Store architectures like ARM, all instruction operands for ALU operations must be registers. Hence for every modification of a variable that is stored in memory (that is, *spilled*), an extra load and store instruction pair is needed.[1] If we take as example a simple increment of a variable on a RISC machine, this needs three instruction if the variable in memory versus one if it is kept in a register, as shown in Table 2.1.

operand in memory location n	operand in register `r`
`t` $\leftarrow$ `mem[`n`]` (load)	
`t` $\leftarrow$ `t` + 1	`r` $\leftarrow$ `r` + 1
`t` $\rightarrow$ `mem[`n`]` (store)	

Table 2.1: Instructions for the simple assignment `x := x + 1` depending on the storage

The register file on common architectures is still quite limited: ARM has 15 general purpose registers available (see Appendix A), while the dominating architecture for desktop computers, x86, even only has as few as eight register available. A subset of the register set is usually reserved for special purposes (e.g. stack pointer, frame pointer, return address) and can't be used for holding values of variables or temporaries. Hence the decision on which program value

[1] note that there exist architectures like x86 where memory operands are possible at least in some instructions

resides in a register is even more crucial for improving the performance by minimizing the spilling instructions.

2.2 Subproblems of Register Allocation

The problem of register allocation actually implies three subproblems:

- What values in a program should reside in registers, what values should be stored in memory? (**spilling**)[2]
- In which register should each value reside? (**register assignment**)
- Given two non-interfering variables A and B that are related by an assignment (`A = B`), should we map them to the same register? (**coalescing**)

For optimal register allocation all of those subproblems have to be taken into account efficiently, which is still an open problem. For example, most early approaches were concentrated too much on assigning and spilling, ignoring the influence of a good coalescing strategy.

2.3 Register Allocation Scope

There are generally different approaches to register allocation, depending on the scope of the variables it is applied on. The original TCC uses some simple register allocation only based on *expressions*, which is very ineffective, as we will show in section 3.8. Usually the allocation is performed on a larger variable scope to minimize the amount of spilling instructions, with the following two main approaches:

- **Local register allocation**: based on the scope of basic blocks (that is, instruction sequences with only one entry and exit points each)
- **Global register allocation**: based on the scope of whole procedures, therefore also called *intraprocedural register allocation*

As third category there also exists *interprocedural register allocation*, which aims to find a register mapping over the "whole program" scope. To the knowledge of the author, though various research papers have appeared about this within the last two decades [WG92, KF96], no common compiler exists that implements this strategy.

This thesis is focussed solely on global register allocation, which is nowadays the most widespread approach in industrial compilers.

[2] in early research prior to graph coloring approaches, the term *register allocation* addressed only this subproblem [ASU86]

2.4 Liveness Analysis – Preliminary

As a preliminary step for any register allocation algorithm, the compiler needs to find out which variables are live at the same time. This process called *liveness analysis* is essential, because then the values are not allowed to be assigned to the same register. Research papers about register allocation usually don't discuss liveness analysis, but rather assume that this information is already given. A simplified liveness analysis for our implementation, suitable as input for the linear scan algorithm, is described in chapter 6.

The following algorithms described are all independent of any platform and only assume that we have K uniform physical registers available that can be used.

2.5 Graph Coloring

One fundamental discovery for an efficient solution to global register allocation was its connection to the mathematical problem of graph coloring. This has first been recognized by John Cocke[3] in the early 70s [AC71], but the first implementation was not until ten years later by Chaitin (see below).

With the information gathered by liveness analysis, the compiler can construct an undirected graph in which every vertex represents a unique variable, and edges between vertices mean that the lifespan of the two variables overlap.

Consider the intermediate code skeleton in Figure 2.1, involving five variables A to E.

```
1  A = ...
2  B = ...
3  C = ...
4  ... = A
5  ... = B
6  D = ...
7  E = ...
8  ... = D
9  ... = E
10 ... = C
```

Figure 2.1: IR code

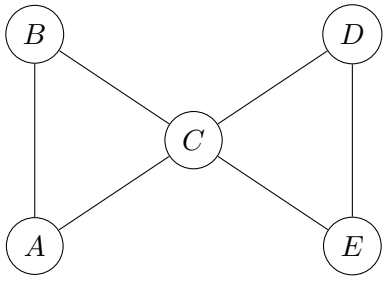


Figure 2.2: Interference Graph

Figure 2.2 shows the so-called *interference graph* corresponding to the IR code to its left. The problem of register allocation can now be approached by K-coloring this graph, where each color represents one physical register. Two vertices sharing an edge may not not be assigned the same color. Figure 2.3 shows a possible coloring for our example, assuming we have three registers available ($K = 3$). The substitued IR code is shown in Figure 2.4.

There are situations where a coloring is not possible, because the number of variables that are live at the same time is larger than K (theoretically it can be unbounded). In this case, some registers need to be spilled, that is, moved to memory. As the intermediate code changes by the

[3] often called "the father of RISC architecture"

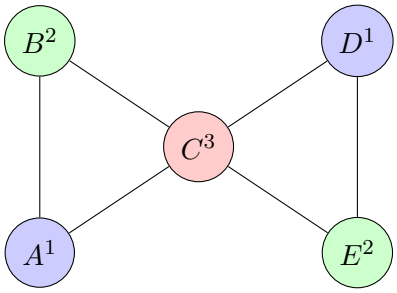


Figure 2.3: Colored Interference Graph ($K = 3$)

```
1  R1 = ...
2  R2 = ...
3  R3 = ...
4  ... = R1
5  ... = R2
6  R1 = ...
7  R2 = ...
8  ... = R1
9  ... = R2
10 ... = R3
```

Figure 2.4: IR code with virtual registers substituted by physical registers

insertion of spill code, the graph needs to be rebuilt and the process repeats, until the graph is K-colorable (in other words: its chromatic number is $\leq K$) and no more spills are needed.

The general scheme for graph-coloring-based register allocation is shown in Figure 2.5.

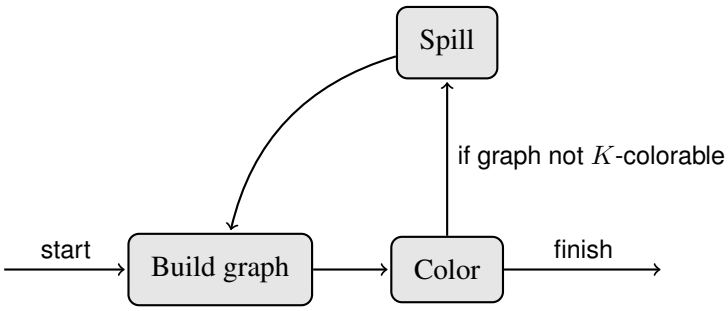


Figure 2.5: Steps in register allocation based on graph coloring

Since the problem of determining whether an arbitrary graph G is n-colorable is NP-complete (for $n > 2$), so is register allocation[4], and heuristic techniques have to be used to search for a coloring.

Chaitin

The first global register allocator based on graph coloring was designed by G.J. Chaitin [CAC+81] in the course of implementing a PL/I compiler for the IBM System/370. It is commonly called the "*Yorktown allocator*", as the working place was in Yorktown Heights.

In Chaitin's first approach, the spilling was done randomly when the graph was not K-colorable. In a later paper from 1982 [Cha82], smarter heuristics were introduced (a *spill cost*

[4]Proof sketch:
Chaitin showed that given an arbitrary graph, a program can be created with this exact interference graph [CAC+81]. Hence, if we could solve register allocation, we could also color any graph (polynomial reduction). As a result, register allocation is NP-complete.

phase), when the allocator was implemented in the PL.8 compiler for the IBM 801 RISC system. The register allocation scheme is shown in figure 2.6.

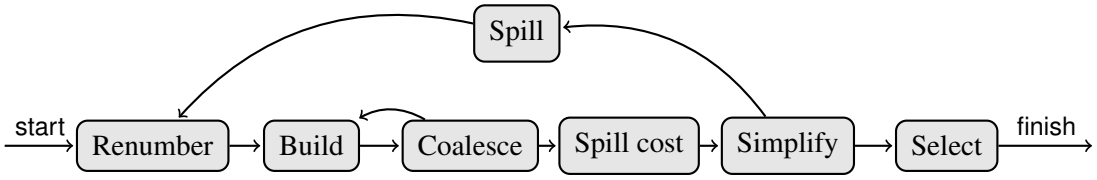


Figure 2.6: Register allocation scheme of Chaitin's Yorktown allocator

The coloring procedure was based on *Kempe's algorithm* (from 1879). The idea is that given a graph G that contains a vertex v with degree $< K$, the graph is K-colorable if G without the node v is K-colorable. Knowing that, we can iterately remove nodes with a degree $< K$ and push them on a stack (this is the *simplify* stage). If this doesn't work (e.g. there are no nodes with degree $< K$), a node has to be removed, actual spill code is inserted and we start again over with the graph building stage. If the empty graph is reached in the *simplify* stage we know that the graph can be K-colored and are finished: each vertex is popped again from the stack and colored (this is the *select* stage).

Between building and coloring the graph, nodes are coalesced whenever possible. This aggessive coalescing policy turned out to be a problem later, when the SSA-form arrived and the large number of temporaries that got coalesced led to huge live ranges. Those would often interfere with too many variables, introducing many spills.

Chaitin/Briggs

Preston Briggs improved on the Yorktown allocator in 1992 in the course of his PhD thesis [Bri92]. The main contribution was a better graph coloring heuristic that could produce more efficient code by saving many spills.

The register allocation scheme for the optimistic "*Chaitin/Briggs*" approach is shown in figure 2.7. This looks quite similar to the original approach, with the only difference that the

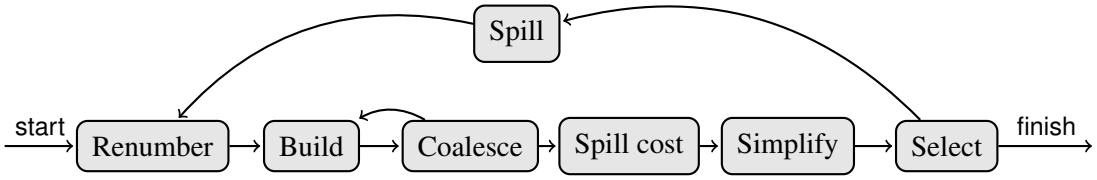


Figure 2.7: Register allocation scheme of Brigg's improved allocator

back-edge representing the spill code insertion occurs one step later, namely at the *select* phase

instead of the *simplify* phase. Instead of failing as soon as a node has degree $\geq K$, it is just left and the allocator continues with the hope to color it later, when it is popped back from the stack. Then only in the *select* phase, if a node can't be colored, it is spilled and the graph is rebuilt.

It turned out that the method can color more graphs. Briggs also introduced the *conservative coalescing* strategy which should solve the problem of the too aggressive coalescing used in the original allocator. However, this was again too conservative.

Iterated Register Coalescing

While earlier papers on register allocation algorithms based on graph coloring were primarily focused on coloring heuristics, the research community started to notice that good coalescing strategies are equally important. Too aggressive coalescing strategies (like Chaitin's) can make graphs uncolorable, introducing spills which could be avoided, while too conservative strategies (like Brigg's) leave too many move instructions in the code.

Appel and George's "Iterated Register Coalescing" algorithm [GA96] published in 1996 extends Brigg's allocation scheme with a coalescing strategy that is safe but still aggressive. By interleaving Chaitin's simplification steps and Brigg's conservatice coalescing, much more move instructions can be eliminated, without introducing too many spills.

It is a standard algorithm for implementing new register allocations, especially in the research community, and every new register allocation scheme is compared to it.

2.6 Recent Research involving SSA and Chordal Graphs

Recent research on graph-coloring-based register allocation has been published that targets on programs internally stored in SSA (*static single assignment*) form, which is the standard representation for modern compilers.

In SSA form, each variable is assigned only once. For variables that depend on program flow, special so called ϕ-nodes are created. Before register allocation, those abstract nodes are substituted by concrete move instructions. Note that this leads to a large number of temporary variables, making a good coalescing strategy even more important.

Around 2005, different research groups showed that interference graphs of programs in SSA form are always *chordal* [HG06]. A graph is said to be chordal if every cycle of a length at least 4 has a cycle chord, which is an edge that connects two non-adjacent vertices of the cycle.[5]

Chordal graphs have many interesting properties, the most important one concerning register allocation being that chordal graphs can be colored optimally in linear time. An obvious idea is to transform programs into SSA form (in polynomial time), applying register allocation on it, and then applying the classical SSA elimination to do register allocation in overall polynomial time. However, this approach doesn't work since this elimination breaks the coloring and there is no possibility to make use of the SSA form to get polynomial coloring, as shown by Pereira [PP06]. Hence, register allocation after classical SSA elimination is also NP-complete.

Pereira and Palsberg proposed a simple algorithm which takes advantage of the useful properties of chordal graphs [PP05]. The technique is not relying on SSA form, but is based on the

[5] see `http://mathworld.wolfram.com/ChordalGraph.html` for examples

observation that a very high percentage of functions have chordal interference graphs (95% of the methods for the Java 1.5 standard library). The idea is to perform optimally if the function has a chordal graph with a greedy coloring, and perform well otherwise. Contrary to other graph coloring algorithms, it is non-iterative. The authors claim that the algorithm is competitive to iterated register coalescing, and can even outperform it for settings with few registers.

2.7 Linear Scan

A completely different approach to register allocation called *Linear Scan* was presented in the late nineties by Poletto and Sarkar [PS99].[6] It doesn't use graph coloring, but rather tackles the problem by using a greedy algorithm for assigning colors to an ordered sequence of intervals, in linear-time.

This approach was actually not entirely new, but is closely related to the *bin packing* register allocation that evolved during the work of the production quality compiler-compiler project (PQCC) at Carnegie Mellon University [LCH+80], led by William Wulf. His Bliss/11 compiler implements the register allocation phase in the `TNBIND` module which treats registers as bins with one valid value at any point [WJW+75]. The constraint is that overlapping live ranges can not be assigned to the same bin. If there is no bin left for a register allocation candidate, the one with the lowest cost (determined by a special heuristic based on lifetime – candidates distributed over a large span are generally less important) is decided to be spilled. This method was also used later in the GEM optimizing compiler (see the "second-chance binpacking" refinement below).

Linear Scan got very popular within the last decades and is used many real world compilers such as LLVM[7] or the Java HotSpot client compiler.

Description with Example

Prerequisites: instruction numbering and live intervals

Linear scan assumes all instruction in the intermediate representation to be numbered according to some order. In the original paper they assume depth-first ordering, while the possibility of numbering the instruction simply as how they appear in the IR is also mentioned[8].

A central notion for linear scan is the *live interval*: $[i, j]$ is said to be a live interval for variable/value v if there is no instruction with number $j' > j$ such that v is live at j', and there is no instruction with number $i' < i$ such that v is live at i'. Note that this is a conservative approximation of the usual live ranges, since there could be intervals within $[i, j]$ in which v is not live.

[6] The term "linear scan" was originally coined in an earlier paper about a dynamic code generation called *tcc* [PEK97] which implements the "Tick-C" programming language. Note that while also named "tcc", this a different compiler than the one covered in this thesis.

[7] since the release of version 3.0 of LLVM in 2011, the default allocator is a new greedy register allocator (see `http://blog.llvm.org/2011/09/greedy-register-allocation-in-llvm-30.html`), but linear scan can still be enabled

[8] this approach is used in our implementation, see chapter 5

Algorithm 1 Linear Scan Register Allocation

```
1:  procedure LSRA(Intervals)  ▷ Intervals is sorted in order of increasing starting points
2:      Active ← {}
3:      for LI ∈ Intervals do                                  ▷ traverse list of live intervals
4:          EXPIREOLDINTERVALS(LI)
5:          if |Active| = K then                    ▷ all registers are used, we have to spill
6:              SPILLATINTERVAL(LI)
7:          else                          ▷ still registers available, register can be assigned
8:              LI_register ← REGISTERPOOLGET()
9:              Active ← Active ∪ LI
10:         end if
11:     end for
12: end procedure

13: procedure EXPIREOLDINTERVALS(CurrentInterval)
14:     for LI ∈ Active do                                 ▷ traverse list of active intervals
15:         if LI_endpoint ≥ CurrentInterval_startpoint then
16:             return                                          ▷ no old intervals anymore
17:         end if
18:         Active ← Active \ LI                  ▷ expire old interval and release register
19:         REGISTERPOOLADD(LI_register)
20:     end for
21: end procedure

22: procedure SPILLATINTERVAL(Interval)
23:     Spill ← Active_last                ▷ spilling heuristic: widest distance of endpoint
24:     if Spill_endpoint > Interval_endpoint then                ▷ interval Spill is spilled
25:         Interval_register ← Spill_register
26:         Spill_location ←NEWSTACKLOCATION()
27:         Active ← (Active \ Spill) ∪ Interval
28:     else
29:         Interval_location ←NEWSTACKLOCATION()                 ▷ passed interval is spilled
30:     end if
31: end procedure
```

Algorithm

In a first step, with the information from liveness analysis, live intervals are computed. Those are stored in a list that is sorted in order of increasing starting points. The linear traversal of this list is the outer skeleton of the linear scan method, as shown in Algorithm 1.

The central data structure is the *active list*, which contains the live intervals that overlap at the current point. The list is kept sorted in order of increasing end points and is empty at the beginning of the algorithm.

At each iteration of the live intervals, the following simple steps happen (lines 4–10):

1. Expiring of old live intervals (`ExpireOldIntervals()`):
   The list of active intervals is scanned. All active intervals where the endpoints precede the current interval's starting point are not overlapping and thus not relevant anymore, they are "expired". This leads to their removal from the active list and the corresponding register is marked as free again. Note that the traversal of the active list can be aborted as soon as an endpoint is behind the current interval's starting point, since the active list is sorted in order of increasing end points.

2. Register assignment (`else`-branch):
   If no spilling is needed (see step below), we can choose a register from the pool and assign it to the current interval. It is added to the active list as well.

3. Spilling (`if`-branch, `SpillAtInterval()`):
   The length of the active list shows how many registers are already used. If it is already as large as the number of available registers K, we don't have a register left for the current interval, and one interval must be spilled. While there are several heuristics for choosing a live interval to spill, the one in the paper is simple: it takes the interval that ends last, furthest away from the current point.

Example

Consider again the IR code example from Figure 2.1. Liveness analysis (which is trivial for this straight-code example) yields the following live intervals:

$$A \leftarrow [1, 4], B \leftarrow [2, 5], C \leftarrow [3, 10], D \leftarrow [6, 8], E \leftarrow [7, 9]$$

To show how linear scan handles spilling, we assume that the number of available physical registers $K = 2$. Note that the intervals are already listed in order of increasing starting points. The following steps happen in each iteration of the linear scan traversal loop:

1. Interval A: no intervals expired, $\boldsymbol{A_{reg}} \leftarrow \boldsymbol{R1}$, $Active \leftarrow \{A\}$

2. Interval B: no intervals expired, $\boldsymbol{B_{reg}} \leftarrow \boldsymbol{R2}$, $Active \leftarrow \{A, B\}$

3. Interval C: no intervals expired, spilling needed since $|Active| = |K|$
   $\boldsymbol{C}$ **is spilled** since its endpoint is larger than the endpoints of A and B; $Active$ unchanged

4. Interval D: A and B are expired and get removed from the $Active$ set,
   registers $R1$ and $R2$ are available again, $\boldsymbol{D_{reg}} \leftarrow \boldsymbol{R1}$, $Active \leftarrow \{D\}$

5. Interval E: no intervals expired, $\boldsymbol{E_{reg}} \leftarrow \boldsymbol{R2}$, $Active \leftarrow \{D, E\}$

Conclusion

If V denotes the number of canditates for register allocation (that is, variables or temporary values, respectively live intervals), the algorithm takes $O(V)$ time – K is bounded and assumed to be constant and hence doesn't influence the complexity.

There are two obvious advantages of linear scan over graph coloring approaches that make it an attractive choice: its simplicity, leading to comparatively low implementation effort, and the high compile-time speed (see also section 4.1). The latter is especially important for JIT compilers and dynamic compilation systems.

The drawback of linear scan is the quality of the generated code. Since live ranges are approximated and lifetime holes are not recognized, often many spills are unnecessarily generated.

The original paper argues that while linear scan is significantly faster than graph coloring (which even holds for fast graph coloring allocators that don't perform coalescing), the generated code is still efficient and only about 10% worse [PS99]. To our knowledge, there is no up to date research study that compares linear scan to other register allocation algorithms to verify this for state-of-the-art implementations and architectures.

Refinements

Second-chance binpacking

An improved, more complex variant of linear scan is called *second-chance binpacking* [THS98]. It is based on the bin packing approach used in the GEM optimizing compiler by the Digital Equipment Corporation [BCD+92]. By keeping track of the register allocation candidate's *live range holes* (e.g. intervals during which no useful values are maintained) the register file can be better used and hence less spills are needed. Another notable difference to linear scan is that the algorithm performs register allocation and instruction rewriting in a single pass. This enables spilled allocation candidates to have multiple chance to reside in registers during their lifetimes, hence the name.

The improvements result in a notable increase of compile-time. The original paper on linear scan compares to second-chance binpacking and suggests that there is only a marginal increase in performance of the produced code, while the allocation time almost doubles up for the presented benchmarks [PS99].

Extended Linear Scan

In 2007, Sarkar and Barik proposed two alternative algorithms based on linear scan called "*Extended Linear Scan*" [SB07]. As motivation, the authors show that graph coloring, the de facto standard foundation for global register allocation, introduces unnecessary constraints and has some theoretical limitations. The basic extended linear scan algorithm (called ELS_0) overcomes those limitations and based on that, a further extension of the basic algorithm (called ELS_1) is presented. The former solves the problem for spill free register allocation (SFRA), while the latter covers register allocation with total spills (RATS), meaning that whole lifetime intervals are spilled.

Experimental results for a number of SPECint2000 benchmarks compare graph coloring with ELS_1 and show that the compile-time speedups were significant (like for the original linear scan), while the resulting execution time also improved by 2.3% on average.

Linear scan for SSA form

Using linear scan on an IR which is in SSA form has several benefits, as pointed out by Wimmer and Franz [WF10]: first, for the calculation of the liveness intervals, no iterative data flow analysis is needed, since SSA form implies that there is only one definition for each variable. Secondly, tests on intersecting intervals during compilation can be omitted because non-intersection is already guaranteed by SSA form as well. Thirdly, the obligatory SSA deconstruction after register allocation can be integrated into the resolution phase.

Those advantages lead to both a lower compilation time as well as to a simpler implementation. The algorithm was implemented for the Java HotSpot client compiler, with the result that the compile-time time was decreased by 4% to 8% while the run-time was about equal, in one case even slightly better.

2.8 Other Approaches

Apart from the two major register allocation strategies, that is, graph coloring and linear scan, there have been proposed various other interesting and promising approaches within the last two decades [Pro09, Per08]:

- **Integer Linear Programming**: At this approach, the register allocation problem is modeled as constraints in a system of integer linear equations. Those equations, representing the interactions between registers and variables, can then be solved with well-known, optimized ILP solvers[9].

  The idea was first brought up in 1996 by Goodwin and Wilken [GW96], where they used 0-1-integer programming (a subclass of the integer linear programming model) for the formulation of the problem. While this solution produced very efficient code, it was too slow for practical use – it could take hours to find an optimal solution. The reason for this is the fact that integer linear programming is NP-complete, leading to a worst-case exponential running time.

  To improve the ILP-based register allocation, Appel and George separated the phases between spilling and register assignment [AG01]. In a first step, the ILP solver finds the optimal solution to the question which variables should be kept in registers and which variables are spilled. For the remaining subproblems of coloring and coalescing, various solutions have been proposed so far, since the "*optimal coalescing challenge*"[10] has been published in 2000.

[9] for example, `lp_solve`, see `http://sourceforge.net/projects/lpsolve/`

[10] see `http://www.cs.princeton.edu/~appel/coalesce`

Shortly after this, George and Appel themselves had found a solution called *optimal coalescing*, where they again use integer linear programming to model coloring and coalescing. The reasearch was heavily focussed on architectures with few registers, predominantly the x86 architecture, where only six registers can be used for variables and temporaries. In practice the solution is, despite its name, not optimal and still too slow.

In 2006, Hack and Grund found a solution giving optimal results using a cutting-plane algorithm [GH07], which in turn again uses integer linear programming. It is faster than the approach from George and Appel – in cases where register allocation takes too long, the authors suggest to halt the computation and fall back to a more usual solver that doesn't take forever.

Though ILP-based register allocation is a very interesting approach with some advantages, it is still much slower than traditional approaches and is not implemented in industrial compilers yet.

- **PBQP**: The Partitioned Boolean Quadratic Problem (PBQP) belongs to the class of Quadratic Assignment Problems (QAP) and is a generalization of the graph coloring problem. PBQP is NP-complete, but a subclass of these problems can be solved in polynomial time.

  The goal is to find a function of minimal cost, which is controlled by two sets of terms: the cost of asssigning one variable x to another variable y (measured by a *local* cost function $l(x, y)$), and the cost given by the interactions between two variables, that is, assigning x to y and z to a (measured by a *related* cost function $r(x, z, y, a)$).

  The algorithm by Scholz and Eckstein [SE02] solves in $O(nm^3)$, where n is the number of variables and m is the maximum size of any domain. The complexity of PBQP in regards of register allocation is $O(|V|K^3)$, where $|V|$ is the number of variables in the source program.

- **Puzzle-solving**: The problem of register assignment is analogous to solving a collection of puzzles, as shown by Pereira in 2008 [QaPP08]. The register file is modeled as a puzzle board and and the program variables as puzzle pieces. This approach takes linear time for a large number of architectures (including x86, PowerPC and StrongARM) and can match with widespread register allocators (e.g. the extended version of linear scan used by LLVM) both in regards of compile-time and run-time of the generated code.

CHAPTER 3

TCC Internals

The following sections describes the inner workings of TCC, with emphasis on the parts that are relevant for our extension. The informations apply to TCC version 0.9.26, the latest stable release (released in February 2013) at the time of writing.

3.1 Overview and Features

Most modern compilers are constructed with the aid of generators for certain well-defined stages that can be described with appropriate formal languages, e.g. regular expressions for lexical analysis, context-free grammars for syntax analysis (examples of such generators include the prominent `lex`/`yacc`[1] duet). TCC goes the classical way in form of a single-pass compiler and implements all the stages "by hand".

TCC also doesn't rely on any external tools during run-time. Preprocessing is interweaved with lexical analysis (whereas gcc relies on `cpp`, the C preprocessor tool), an assembler is not needed since target code is output directly in binary format (whereas gcc relies on `as`, the GNU assembler), and even linking (whereas gcc relies on `ld`, the GNU linker).

Hence, TCC is purely self-contained. This property as well as the obvious speed gain makes it an interesting option for interpreting C code. [2]

Scripting support

When started with the `-run` switch, TCC compiles the program into memory and directly starts it from there. In this way, C scripts can be easily provided in Linux systems similar to shell scripts by simply preceding the file with the she-bang "`#!/usr/local/bin/tcc -run`" and setting the executable bit.

[1] called `flex`/`bison` by now

[2] since the whole code snippet is compiled at once before execution, it is in a strict sense not an interpreter, but rather a *load-run* compiler

With this mechanism, simple tasks can even be formulated as one-liners in the shell. Sometimes it would be comfortable to quickly try out how some small code snippets, e.g. calls to library functions, behave without the obligatory "save, compile and execute" cycle. This is possible with TCC by providing the `-run` switch and reading from *stdin* by putting the dash as "input file", for example:

```
$ echo 'int main() { printf("%d", time(0)); }' | tcc -run -
1397324291
```

Dynamic code generation

With the library `libtcc`, C code can be dynamically compiled and executed at runtime. The idea is to pass a single string containing the code to the library which compiles the code and stores it to the desired location, either memory or file system.

The following listing shows a minimum, pretty self-explanatory example of how to achieve this for a single trivial function adding two numbers (passed in line 13):

```
1  #include <stdlib.h>
2  #include "libtcc.h"
3
4  int main(int argc, char *argv[])
5  {
6      TCCState *s;
7      int size;
8      void* mem;
9      int (*func)(int, int);
10
11     s = tcc_new();
12     tcc_set_output_type(s, TCC_OUTPUT_MEMORY);
13     tcc_compile_string(s, "int foo(int x, int y) { return x+y; }");
14     size = tcc_relocate(s, NULL);
15     mem = malloc(size);
16     tcc_relocate(s, mem);
17     func = tcc_get_symbol(s, "foo");
18
19     printf("23 + 42 = %d\n", func(23, 42));
20 }
```

3.2 Overview of Modules and Phases

TCC consists of the following modules:

- `tcc.c`: entry point, evaluates command-line arguments
- `libtcc.c`: directs the compilation process, provides interface for dynamic code generation

category	lexeme	token	notes
literal	1423	TOK_CINT	literal value in tokc.i
operator	++	TOK_INC	
keyword	for	TOK_FOR	
operator	(	'('	
identifier	foo	$n \geq$ TOK_UIDENT	every identifier has unique token number
keyword	if	TOK_IF	
literal	"bar"	TOK_STR	literal value in tokc.cstr
operator	->	TOK_ARROW	

Table 3.1: Examples of some tokens in TCC

- tccgen.c: the core of the compiler, implements the top-down parser which directs the translation process
- tccpp.c: implements preprocessor and lexer
- ***[Arch.]***-gen.c: low-level code generator for the specific architecture (arm-gen.c for ARM architecture)
- tccelf.c: routines for generating ELF files (object files, executable files or dynamic libraries)
- tccrun.c: support for the -run switch

TCC is a single-pass compiler, meaning that it doesn't create any form of intermediate representation of the code. In most compilers nowadays the parser usually emits either an AST (abstract syntax tree) or some intermediate language that is passed as input to the next phase. The TCC parser however directly emits object code when parsing expressions by calling the low level code generations functions, hence there is no separate code generation phase. This is the main reason why TCC compiles so extraordinarily fast.

3.3 Lexical Analysis

The lexer is implemented in tccpp.c. It is accessed repeatedly by the parser through the procedure next(), which places the next token into the integer variable tok. For some tokens, tokc contains additional infos, e.g. values for literals.

Basic token values are defined in the header file tcc.h. Table 3.1 lists some examples of tokens returned by the lexer for certain lexemes. Note that in case of single character lexems (such as parantheses, commas or basic math operations), the token value equals this exact character value. Identifiers get unique token numbers greater or equal than TOK_UIDENT.

3.4 Syntax Analysis

Syntax analysis in TCC is performed with *recursive-descent* parsing, a special form of top-down parsing where no backtracking is needed [ASU86]. Every non-terminal is associated with a recursive procedure that recognizes exactly that non-terminal in the input by implementing the corresponding production rule. Terminal symbols are directly matched and for non-terminals, the associated procedure is called.

This very simple approach is surprisingly effective and has been proposed and consistently used by compiler construction pioneer Niklaus Wirth [Wir96, BGP00], e.g. for the languages Pascal and Oberon.

The parser is started by fetching the first token and calling the procedure that is associated with the start symbol. In TCC, this is done in the function `libtcc.c:tcc_compile()` via the `next()` and `decl(VT_CONST)` calls. All the non-terminal procedures are implemented in `tccgen.c`.

Table 3.2 lists the EBNF-grammar non-terminals for C expressions (as stated in Appendix A13 of "The C Programming Language" (2nd edition) [KR88]) and the associated TCC parser procedures.

EBNF non-terminal	TCC parser function
expression:	`gexpr()`
assignment-expression:	`expr_eq()`
conditional-expression:	`expr_cond()`
logical-OR-expression:	`expr_lor()`
logical-AND-expression:	`expr_land()`
inclusive-OR-expression:	`expr_or()`
exclusive-OR-expression:	`expr_xor()`
AND-expression:	`expr_and()`
equality-expression:	`expr_cmpeq()`
relational-expression:	`expr_cmp()`
shift-expression:	`expr_shift()`
additive-expression:	`expr_sum()`
multiplicative-expression:	`expr_prod()`
cast-expression:	`unary()`
unary-expression:	`unary()`

Table 3.2: C expression non-terminals and the corresponding parser functions in TCC

As an illustrative example, the implementation of `expr_or()` is shown in Listing 3.1. Note that left-recursion can cause top-down parser to go into an infinite loop, hence the rules were not directly applied, but the list was rather implemented through a loop.

Listing 3.1 Implementation of the *inclusive-OR-expression* non-terminal

```
static void expr_or(void)
{
    expr_xor();
    while (tok == '|') {
        next();
        expr_xor();
        gen_op('|');
    }
}
```

The non-terminal procedures also contain the code generation calls (⇒ `gen_op('|')` in this example) – the concept where source language translation is completely driven by the parser is called *syntax-directed translation*.

3.5 Syntax-Directed Translation

The central data structure for the syntax-directed code generation is the *value stack*. Whenever operands of expressions are recognized by the parser, they are put on this stack, as elements of the structure type `SValue` (defined in `tcc.h`). The value stack primarily contains information on *where* the operands are located from the perspective of the generated code. It's field `SValue.r` contains the location value, together with some special flags in the upper bits.

Table 3.3 shows a subset of possible values and flags on elements on the value stack (shortly called *svalues*).

Location values of `r`	**Meaning**
`REG_R`***X***	value is stored in register R***X***
`VT_CONST`	value is a constant, stored in `SValue.c`
`VT_LOCAL`	value is stored on the stack (i.e. a local variable) with the offset `SValue.c.i`
`VT_CMP`	value is stored in the processor's flags (meaning it is the consequence of a test)
`VT_JMP`/`VT_JMPI`	value is the consequence of a conditional jump
Flags in `r`	**Meaning**
`VT_LVAL`	value is an *lvalue*, meaning that it is a pointer to the wanted value
`VT_SYM`	value is a global variable – it's symbol `SValue.sym` must be added to the constant

Table 3.3: Some values and flags for svalues and their meanings

Whenever an *operator* is parsed, values are popped from the stack as operands, code is generated and the result is again pushed on the stack.

Consider the following assignment expression: `sum = 23 + x` where `sum` and `x` are both local variables. Figure 3.1 shows how the code generating functions for binary addition and for assignment change the value stack for this expression.

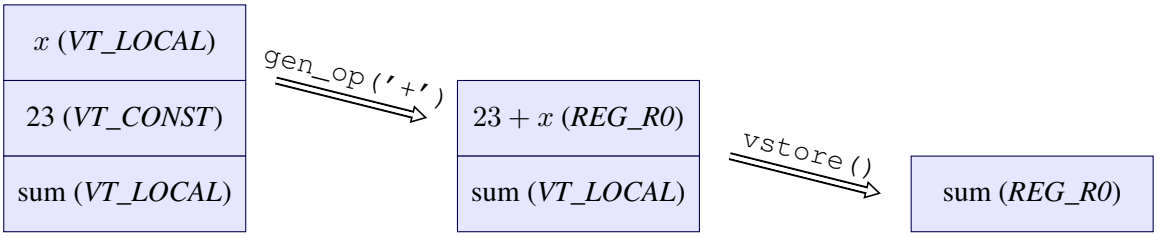


Figure 3.1: Value stack example for expression `sum = 23 + x`

An important part for the code generation is the function `gv()` which evaluates the top element on the value stack (`vtop`) into a register.

3.6 Simple Optimizations

Though most sophisticated types of optimizations like loop optimizations (e.g. loop unrolling, loop interchange) or data-flow optimizations (e.g. common subexpression elimination, constant propagation) are not possible in TCC as they require the presence of some form of intermediate representation, it still performs some basic optimizations:

- Constant Folding within single expressions: constant expressions are evaluated at compile-time rather than computed at run-time; for example, the expression `a + (8*9+5) + b*0` is folded into `a + 77`. Note that the TCC documentation wrongly states that *constant propagation* is performed[3], which would also check whether the variables involved are constant and needs data-flow analysis techniques. Rather, constant propagation is only done for literals in single expressions.
- Multiplications/divisions by powers of two are substituted by shifts
- Optimization of comparison operators by maintaining a special cache for the processor flags

3.7 Backpatching

TCC has to solve a classical problem that one-pass compilers have to tackle: whenever control-flow statements (that is, conditionals or loops) are translated, the target address of forward branches is not known yet at the point of time when they are generated.

With the technique of *backpatching*, the instruction is generated with the address operand left empty (put to zero). The address of the instruction is saved (into a *symbol* variable) and is patched later when the code generator knows the target address.

[3] see `http://bellard.org/tcc/tcc-doc.html#SEC35`

In some cases, there can be multiple forward branches with the same target address. For example, in a loop there can be an arbitrary number of `break;` statements that all target the address of the first instruction after the loop body. To avoid saving a list of symbols, TCC creates a linked list with the address operators of the jump instructions. The symbol always points to latest jump instruction, and this instruction has as operand the address of the preceeding jump instruction, and so on. The first jump instruction always has zero as operand – this is the recognition that the linked list ends. The backpatching process then simply involves the traversal of the linked list, substituting every element with the actual target address. The backpatching is implemented in `arm-gen.c:gsym_addr()`.

We later adapt this technique to be used for the IR generation as well, see section 3.7.

3.8 Register Usage and Code Quality

The code generator only uses the subset of caller-saved registers (e.g. R0–R3 and R12 for the ARM architecture; EAX, ECX and EDX for x86) for evaluating expressions, the others remain completely unused. If there are more registers needed, one registers is chosen to be "spilled" onto the stack as temporary variable.

As TCC lacks local or global register allocation, the generated code never saves variable values in registers for the long term. Hence every assignment statements involves *load* instructions for all source operands in memory and a *store* instruction for the assignment. This naturally leads to fairly inefficient code, similar to that generated by `gcc -O0`.

Figure 3.3 shows an example for code generated by TCC. The input, shown in Figure 3.2, is bit counting function which primarily consists of short assignment statements (this function is later used as benchmark, see `bitcount1` in section 9.3). Note the large amount of instructions which only have the purpose of reloading the registers with values from memory and vice-versa. From the 12 loop instructions (spanning from addresses `c8` to `f4`), 5 instruction belong to the load/store category, this is more than 40%. With global register allocation, all those expensive instructions can be eliminiated, yielding a significant run-time speedup.

```
int bit_count(long x)
{
  int n = 0;

  if (x) do
    n++;
  while (0 != (x = x&(x-1)));

  return(n);
}
```

Figure 3.2: A simple bit counting function in C

```
000000a0 <bit_count>:
... [ Prologue ]
  b4:  mov r0, #0
  b8:  str r0, [fp, #-4]
  bc:  ldr r0, [fp, #12]
  c0:  teq r0, #0
  c4:  beq f8 <bit_count+0x58>
  c8:  ldr r0, [fp, #-4]
  cc:  mov r1, r0
  d0:  add r0, r0, #1
  d4:  str r0, [fp, #-4]
  d8:  ldr r0, [fp, #12]
  dc:  sub r0, r0, #1
  e0:  ldr r1, [fp, #12]
  e4:  and r0, r1, r0
  e8:  str r0, [fp, #12]
  ec:  mov r1, #0
  f0:  cmp r1, r0
  f4:  bne c8 <bit_count+0x28>
  f8:  ldr r0, [fp, #-4]
... [ Epilogue ]
```

Figure 3.3: TCC-generated ARM object code for the function `bit_count()`

CHAPTER 4

Implementation Considerations for the Proof-of-Concept

This chapter substantiates the rough idea of "implementing a global register allocator for TCC" by specifying several important choices about the proof-of-concept, predominantly the register allocation strategy used and the supported target architecture.

4.1 Choice of the Global Register Allocation Strategy

By thinking about how we should either use some form of classical graph coloring algorithm or linear scan as allocation strategy, the answer is quite obvious if we keep the main feature of TCC in mind: *speed*. Since TCC is so fast in compilation – which is especially pleasant for the unique feature of interpreting C code – and we don't want to sacrifice too much of that for global register allocation, $\Rightarrow$ **Linear Scan** seems to be the perfect choice, with similar reasons as for JIT compilers. It only traverses once linearly over the live ranges and doesn't take multiple rounds like graph coloring approaches, if spill code is inserted.

Another decisive advantage of linear scan over graph coloring is *simplicity*. This is, apart from the very short and straightforward algorithm, also reflected in the data structures needed: linear scan only works with arrays and lists, and the latter can be easily represented as fixed-sized arrays if we are willing to reserve the space statically large enough and don't extend memory on demand. Hence no dynamic data structures are needed, as explained more in detail in chapter 7. Whereas for graph coloring approaches, usually the graph is represented both as matrix and adjacency list format. We also would have to support the insertion of spill code into the IR – in our solution, operands are simply marked and spill code insertion is done by code generator in the final phase. Our guess is that even a simple graph coloring implementation would take several times the effort than for linear scan.

With its two main advantages, linear scan matches perfectly the spirit of TCC: simple and fast.

The drawback of this choice is the lower quality of the generated object code. This is the tradeoff for the speed. Compared to the original TCC, the runtime efficiency will still increase noticably.

4.2 Choice of the Target Architecture

For the proof-of-concept it would be way too costly to implement the register allocator for all architectures supported by the original TCC (that is: x86, x86_64, ARMv4, TMS320C67). While the target architecture is a matter of taste, it is advantageous to choose an architecture where code generation is simple.

We saw several benefits in choosing ⇒ **ARM** over Intel x86(_64) as target architecture:

- RISC machine: few instructions, few addressing modes, simple instruction encoding
- fixed-size instructions (32 bits)
- uniform register file – x86 would be register-constrained and had only 8 registers

The drawback of this choice is that special hardware is needed, as ARM is not common for desktop environments. With the *Raspberry Pi* (see Appendix B we found an easy and reasonable way to develop the TCC register allocation extension for the ARM architecture.

4.3 Language support restrictions

For the proof-of-concept, we restrict the supported input language for TCC to the following C subset:

- no `float`/`double` data type support: the support for more than one *register class* would increase the complexity of the implementation drastically; a fairly large part of the TCC code generation for the ARM architecture deals with floating points (type conversion, special calling conventions for so-called float aggregates etc.)
- no `long long` data type support: this data type would be stored in register pairs and hence complicates the register allocation process
- no GNU extension support
- no support for bitfields
- no support for passing/returning `struct` data types by value: the ARM calling conventions are quite complex for data types larger than four bytes – this is reflected in `arm-gen.c:gfunc_call()`, the function generating code for function calls, contains more than 300 lines; for comparison, `i386-gen.c:gfunc_call()` is less than 100 lines.

In short, it supports ANSI C (C89) without floating points, bitfields and `struct` parameter/return value passing.

4.4 The Big Picture

Figure 4.1 shows the steps of the implementation that are traversed for every function of the input source:

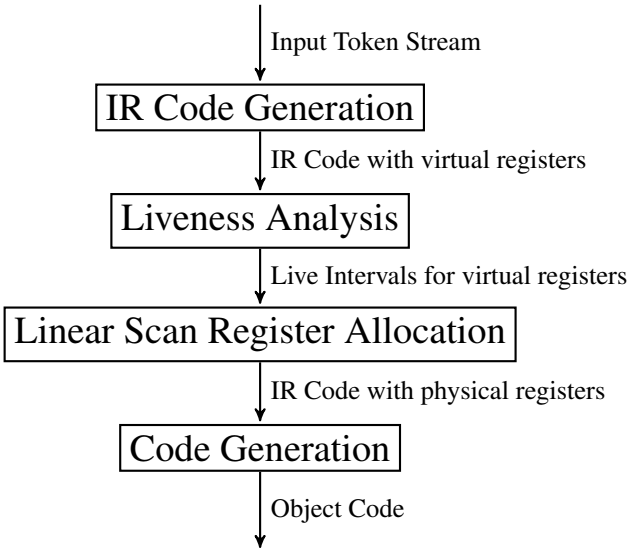


Figure 4.1: Overview of the Linear Scan extension for TCC

1. **IR Generation**: The first phase is driven by the top-down parser of TCC. It gets a token stream of a C source code function as input and outputs a corresponding IR listing in quadruple format. For all register allocation candidates (local and temporary variables) virtual register (*vreg*) numbers are assigned which are used as operands for the IR instructions, in order to abstract the concrete target destination. [⇒ see chapter 5]

2. **Liveness Analysis**: This phase determines the live intervals for all occuring virtual registers in the IR listing. It uses a simple approach that avoids data-flow analysis techniques but iteratively searches for backward jumps and extends the live interval if necessary. [⇒ see chapter 6]

3. **Linear Scan Register Allocation**: The actual register allocation with linear scan is performed in this phase: with the list of virtual registers and its corresponding live intervals as input, each *vreg* is mapped to either a concrete physical register or a spilling destination on the stack. [⇒ see chapter 7]

4. **Code Generation**: Finally, this phase generates the target code out of the IR listing and the mapping of virtual registers to concrete target locations which was the result of the previous phase. [⇒ see chapter 8]

CHAPTER 5

Step 1 – Implementing an Intermediate Representation

5.1 Implementing the Additional Pass

Since the basic translation unit for global register allocation is a function, we take the function parsing procedure `gen_function()` in `tccgen.c` as starting point for our implementation. Listing 5.1 shows the essential parts of the original TCC.

Listing 5.1 Function parsing / code generation procedure in original TCC

```
1  static void gen_function(Sym *sym)
2  {
3      [ ... ]
4      gfunc_prolog(&sym->type);
5      rsym = 0;
6      block(NULL, NULL, NULL, NULL, 0, 0);
7      gsym(rsym);
8      gfunc_epilog();
9      [ ... ]
10 }
```

Our implementation is shown in Listing 5.2. In a first step, the IR code listing is cleared. In the original implementation, the parsing of the function parameters and the prolog code generation are tied together in `gfunc_prolog()`. This doesn't work for our approach, since code generation is only done in the last step. For this reason we have an own function `IR_AddFuncParams()` which parses the parameters without generating code. The `block()` call, representing the top-down parsing non-terminal symbol for the whole function block, remains unchanged. Within this call, the IR code is generated by invoking functions to the `tccir.c` module. The remaining calls are pretty self-explanatory.

Listing 5.2 Function parsing / code generation procedure in our extended TCC

```
1  static void gen_function(Sym *sym)
2  {
3      [ ... ]
4      IR_Clear();                              // Phase 1: IR code generation
5      IR_AddFuncParams(&sym->type);
6      block(NULL, NULL, NULL, NULL, 0, 0);
7      IR_LivenessAnalysis();                   // Phase 2: Liveness Analysis
8      LS_RegisterAllocation();                 // Phase 3: LSRA
9      IR_RegisterAllocationParams();
10     IR_GenCode();                            // Phase 4: Code generation
11     [ ... ]
12 }
```

5.2 Abstracting Variables with Virtual Registers

As the original TCC doesn't know any intermediate representation and hence also provides no abstraction for values that could possibly be kept in registers, we have to implement the concept of *virtual registers* first. We use only two types of register allocation candidates: local variables and (compiler-generated) temporaries.

Local variables

As soon as a local variable is declared, a unique virtual register identifier should be assigned to it. First, the data structure for the symbol table (`struct Sym` in `tcc.h`) is extended by the `vreg` variable:

```
typedef struct Sym {
    ...
    unsigned int vreg; /* associated virtual register */
}
```

The assignment of the virtual register takes place in `tccgen.c:sym_push()`, where we first check if the symbol fits to our criteria: the symbol must be a local variable which is an *lvalue* (arrays, for example, are no lvalues) and must fit into a single register, that's why we don't include variables of the basic type `VT_STRUCT`[1]. Then we call `tccir.c:IR_GetVRegLocal()` to get a new uniqe virtual register identifier and assign it to the `.vreg` field. The default value for this field is -1, meaning that the symbol is not associated to a virtual register.

The extension of the function is shown in Listing C.1. When identifiers are recognized by the parser and put on the value stack, the new `vreg` field has to be assigned as well.

[1] floating point data types and 64-bit values are not supported in our implementation and will throw a compiler error when declared, so we don't need to exclude them explicitely at this point

Compiler-generated temporary variables

Temporary variables, generated during the evaluation of expressions, are the other register candidates. For them, the extension on the `Sym` data structure above can't be used, since they are nameless and hence are not stored in the symbol table. Rather they only appear in the value stack as `SValues`, so we need to extend this structure as well:

```
typedef struct SValue {
    ...
    unsigned int vreg; /* associated virtual register */
}
```

This field is then assigned with a call to `tccir.c:IR_GetVregTemp()` when needed, usually for the top element:

```
   vtop->vreg = IR_GetVregTemp();
```

In the IR, we always access the virtual register identifier directly through this field and don't differentiate between local variables. For this reason we copy the `vreg` value from the symbol table to the value stack as soon as a local variable is referenced.

5.3 IR Instruction Set

For the design of an intermediate representation, it is useful to keep two conflicting goals in mind, considering the level of abstraction: on the one hand, the IR should be fairly easy to generate out of the source language. On the other hand, it should also resemble the target assembler code to ease the final code generation.

For compilers supporting multiple architectures, the latter goal can't be achieved if the target architectures are too different. However, our proof-of-concept implementation supports only ARM, and for the register allocation it is practical to have a representation that is basically the same than the target instruction set, with the possibility to abstract physical registers with *virtual registers*.

Encoding

We use the most common way of representing intermediate instructions, namely *three-address-code* (TAC) [ASU86]. In this form, each instruction has three operands (at most), one for destination and two for source, e.g.:

```
ADD x, y, z
```

meaning `x := y + z`. This fits perfectly well to the ARM target architecture which is a three-address machine, also consisting mostly of instructions with three operands.

The enconding takes place in Quadruple form. As type for the operands, we simply take the `SValue` structure (defined in `tcc.h`) used during code generation in the original TCC,

since it contains all the original information needed. This results in the following structure `Quadruple`, defined in `tccir.h`:

```
typedef struct {
    int op;
    SValue src1;
    SValue src2;
    SValue dest;
} Quadruple;
```

The data structure used for the IR instruction listing is a fixed-size array together with a counter of how many quadruples are currently in use: [2]

```
#define MAX_QUADRUPLES 10000
Quadruple IR[MAX_QUADRUPLES];
...
int irpc;                    /* location of the next free quadruple */
```

Description of Instructions

Figure 5.1 shows an overview of the IR instruction set. It does not abstract the whole ARM instruction set, but rather a subset that is used in the code generator of the original TCC. Note that we don't need a separate load operation, since the source operands already implicitely encode the location through the `r` field (see also section 3.5), including direct and indirect addressing modes.

The following locations types are possible:

- **VR**: operand is a *virtual register* (location is not known yet)
- **LV**: operand is *located on the stack* (i.e. local variables that are not allocation candidates)
- **GV**: operand is *located on a fixed memory address* (i.e. global variables)
- **IM**: operand is an *immediate value*

Data-Processing Instructions

IR operations belonging to this category are the simplest as they map directly to the corresponding target architecture instruction. Table 5.1 shows the mapping between IR and ARM instructions.

All instructions have three operands, except `IR_ASS` which only has one source operand. The following operand types are allowed for each of those instructions:

- Dest: VR/LV/GV

[2] Note that if the number of IR instructions for a function exceeds `MAX_QUADRUPLES`, the compiler aborts with an error.

IR operation	dest	src1	src2	Code generation semantics
Data-Processing Instructions				
IR_ADD	X	X	X	dest ← src1 + src2
IR_SUB	X	X	X	dest ← src1 - src2
IR_RSUB	X	X	X	dest ← src2 - src1
IR_MUL	X	X	X	dest ← src1 * src2
IR_AND	X	X	X	dest ← src1 & src2
IR_OR	X	X	X	dest ← src1 | src2
IR_XOR	X	X	X	dest ← src1 ^ src2
IR_SHL	X	X	X	dest ← src1 « src2
IR_SHR	X	X	X	dest ← src1 » src2
IR_SAR	X	X	X	dest ← src1 / (2 ^ src2)
IR_ASS	X	X	-	dest ← src1
Conditional Instructions and Branches				
IR_CMP	-	X	X	compare(src1, src2)
IR_TSTZ	-	X	-	test_for_zero(src1)
IR_SETIF	X	X	-	dest ← (CC == src1) ? 1 : 0
IR_BRANCHIF	X	X	-	if (CC == src1) jump to dest
IR_JMP	X	-	-	jump to dest
Function Call and Return Instructions				
IR_RETVOID	-	-	-	return without value
IR_RETVAL	-	X	-	return with value src1
IR_PARAM	-	X	X	set src2 as parameter
IR_CALLVOID	-	X	-	call function src1
IR_CALLVAL	X	X	-	call function src1, store return value in dest
IR_PARAMVOID	-	-	-	void parameter
Special Instructions				
IR_STORE	X	X	-	mem[dest] ← src1

Figure 5.1: Overview of IR operations

IR instruction	ARM instruction
IR_ADD	ADD
IR_SUB	SUB
IR_RSUB	RSB
IR_MUL	MUL
IR_AND	AND
IR_OR	OR
IR_XOR	EOR
IR_SHL	MOV with "Logical Shift Left"
IR_SHR	MOV with "Logical Shift Right"
IR_SAR	MOV with "Arithmetical Shift Right"
IR_ASS	MOV

Table 5.1: Mapping from IR instructions to ARM instructions

- Src1: VR/LV/GV
- Src2: VR/LV/GV/IMM (not used for IR_ASS)

The concrete target instructions can only have physical registers and small immediate values as operands. The code generator automatically creates instructions to fetch the values into appropriate registers before emiting the actual abstracted instruction. Hence each IR instruction is mapped to multiple ARM instructions, if at least one operand is located in memory. This mapping is described more in detail in section 8.3.

Conditional Instructions and Branches

- IR_CMP: compare the two operands (→ ARM CMP instruction)
  - Dest: – (unused), Src1: VR/LV/GV, Src2: VR/LV/GV/IMM
- IR_TSTZ: test operand for zero (→ ARM TEQ instruction)
  - Dest: – (unused), Src1: VR/LV/GV, Src2: – (unused)
- IR_SETIF: set destination operand to 0/1 dependend on condition code
  - Dest: VR/LV/GV, Src1: IMM (condition code to test for), Src2: – (unused)
- IR_BRANCHIF: jump to IR destination if condition is met
  - Dest: IMM, Src1: IMM (condition code to test for), Src2: – (unused)
- IR_JMP: jump to IR destination
  - Dest: IMM, Src1: – (unused), Src2: – (unused)

Function Call and Return Instructions

- `IR_RETURNVOID`: return from function without value
  - Dest: – (unused), Src1: – (unused), Src2: – (unused)
- `IR_RETURNVAL`: return from function with value
  - Dest: – (unused), Src1: VR/LV/GV/IMM, Src2: – (unused)
- `IR_PARAM`: set parameter (before function calls)
  - Dest: – (unused), Src1: IMM (argument number), Src2: VR/LV/GV/IMM
- `IR_CALLVOID`: call void function
  - Dest: – (unused), Src1: GV, Src2: – (unused)
- `IR_CALLVAL`: call function with return value
  - Dest: VR/LV/GV, Src1: GV, Src2: – (unused)
- `IR_CALLVOID`: void parameter (need to be passed for functions without parameters)
  - Dest: – (unused), Src1: – (unused), Src2: – (unused)

Special Instructions

- `IR_STORE`: encodes indirect addressing mode for assignments (the value contained at "Dest" is the address of the target)
  - Dest: VR/LV/GV (unused), Src1: VR/LV/GV/IMM, Src2: – (unused)

5.4 Simple Expressions

This section describes the generation of IR code for all expressions which lead to the generation of straight-line code without function calls – differently put, expressions that only emit IR instructions that we assigned to the category of *data-processing instructions*. Boolean expressions (consisting of the operators for logical AND, OR and NOT, as well as the relational operators and the ternary operator) involve branches due to short-circuit evaluation and are covered in the next section 5.5. The division and modulo operators generate IR code for function calls, as those operations have to be simulated in software in the ARMv4 architecture. This will be covered in section 5.6.

Binary ALU operations

Listing C.2 shows the replacement to the low-level code generating function `gen_opi()`.

We simply pass the two top elements of the value stack to the new IR instruction, together with a destination operand that refers to the new temporary register. This is then again the new top value for the value stack, which is decreased by one.

Assignment

The assignment operator is handled with `tccgen.c:expr_eq()`, where `vstore()` is called to invoke the low-level code generator call `store()`. The replacement is simple (it is safe to ignore the parameters since there is only occurence of `store()` in our code and it does the right thing):

Listing 5.3 Assignment operator implementation

```
1 static void ir_store()
2 {
3     if ((vtop[-1].r & VT_VALMASK) == VT_LOCAL)
4         IR_PutOp(IR_OP_ASS, &vtop[0], NULL, &vtop[-1]);
5     else
6         IR_PutOp(IR_OP_STORE, &vtop[0], NULL, &vtop[-1]);
7 }
```

Redundant Move Optimization

With the introduction of compiler-generated temporaries, many IR instruction blocks with the following scheme are created for assignment expressions:

```
IR_[*]   VRx,    VRa,    VRb
IR_ASS   VRy,    VRx
```

where `IR_[*]` can be any data-processing IR instruction (note that for `IR_ASS`, there is no second source operand *VRb*). Given that the virtual register X is not used elsewhere, the second instruction is obviously redundant, as the result could be immediately stored in register Y in the first expression:

```
IR_[*]   VRy,    VRa,    VRb
```

We have called the collapsing of those instructions *redundant move optimization*. It is performed directly when a new instruction is appended to the current IR listing. The relevant code for the optimization in `IR_PutOp()` is shown in Listing C.9.

Example

Figures 5.2 and 5.3 show an example of a small C code snippet and its corresponding IR listing. Note that we have only virtual registers and immediate values as operands in this case, where local variables are starting with *VReg0* and compiler-generated temporaries start with *VReg100*.

```
void straight_example()
{
  int a, b, c;
  a = 200;
  b = (a*4) | b;
  a = (a+b)*(5+c+8*9);
  b += a;
  c += ((b*3-b)<<2) & 0xf0;
  --b;
}
```

Figure 5.2: Straight-line C code snippet

```
// a <= VR0, b <= VR1, c <= VR2
01: IR_ASS VR0,   #200
02: IR_SHL VR100, VR0,   #2
03: IR_OR  VR1,   VR100, VR1
04: IR_ADD VR102, VR0,   VR1
05: IR_ADD VR103, VR2,   #5
06: IR_ADD VR104, VR103, #72
07: IR_MUL VR0,   VR102, VR104
08: IR_ADD VR1,   VR1,   VR0
09: IR_MUL VR107, VR1,   #3
10: IR_SUB VR108, VR107, VR1
11: IR_SHL VR109, VR108, #2
12: IR_AND VR110, VR109, #240
13: IR_ADD VR2,   VR2,   VR110
14: IR_ADD VR1,   VR1,   #-1
```

Figure 5.3: The corresponding IR listing for function `straight_example`

5.5 Boolean Expressions and Control-Flow Statements

Backpatching

The technique of backpatching, introduced in section 3.7, is also applied for the IR code generation. This makes sense since this syntax-directed generator is driven by the same top-down parser than the original code generator, emitting the (abstracted) instructions in basically the same order – the problem of not knowing the target address of forward branches at the point of time when they are generated occurs here as well.

Instead of concrete code (=text segment) addresses for the backpatching symbol values, we use the IR listing addresses of the corresponding quadruples. The backpatching functions `IR_Backpatch()` and `IR_BackpatchToHere()` are shown in Listing C.8.

Relational and Logical operators

For the support of relational operations (those are ==, !=, <, >, <=, >=) we first extend our `ir_gen_opi()` function (see Listing C.3). The code generator just emits a CMP instruction and saves the operation in `vtop->c.i`.

The decision on how to generate code for a relational expression (and for every other boolean expression as well) is context-dependend: if the expression is part of another boolean statement or as condition for a control-flow statement (e.g. `if`), then a branch has to be taken. In the other cases, the expression has to be evaluated to yield a numerical value representing *true* (anything but zero) or *false* (zero).

Control Flow Statements

Having implemented boolean expressions as well, the IR code generation of control-flow statements is quite easy. It basically involves again only a substitution for the backpatching of branches, which occur in the following statements/operators:

- `if/else` statement
- `while` and `do/while` statement
- `for` statement
- `switch/case` statement
- `goto` statement
- ternary `?`, `:` operator

The code generation of all the above statements is implemented in `tccgen.c:block()`, except for the ternary operator which is trated in `tccgen.c:expr_cond()`.

As an example, the required modifications needed for the `while` statement is shown in Listing C.7.

5.6 Function Calls

A function call is recognized by the parsing function `tccgen.c:unary()`, in the `while`-loop for post operations, after the check whether the current token is `'('`. Instead of leaving the result of the parameter code generation on the value stack, we generate `IR_PARAM` instructions and pop it off the stack again immediately.

Besides of the parameter itself which is passed in the first operand, also the parameter index is stored as second operand. This is important for *nested function calls* (e.g. `f(x, g(y, z)...)`) when it comes to correctly associating the parameters to the calls in the IR code (this will be explained in section 8.5).

Due to the calling conventions of ARM, we can only handle the first four parameters this way (which are finally passed in registers) – the remaining parameters have to be passed in *reverse order*! Hence we have to keep them on the value stack and add another loop before the call itself will be generated (an `IR_CALLVOID` or `IR_CALLVAL` instruction, depending on whether there is a return value).

Both the extension to the parser are listed in Listings C.5 and C.6.

Divison and Modulo operations

The ARMv4 instruction set doesn't support division or multiplication, hence those instructions have to be emulated in software. The IR generator creates function call instructions to `__aeabi_idivmod()` or `__aeabi_udivmod()` (dependend on whether the calculation is signed or unsigned) in this case, as shown in Listing C.4.

```
int test(int a, int b);

int full_example(int n)
{
  int x=0, y=1;
  while (n--) {
    x += y;
    y += test(x+5, y-10);
  }
  return y;
}
```

Figure 5.4: C code snippet involving branches and function calls

```
// n <= VR200, x <= VR0, y <= VR1
01: IR_ASS      VR0,    #0
02: IR_ASS      VR1,    #1
03: IR_ASS      VR100,  VR200
04: IR_ADD      VR200,  VR100,  #-1
05: IR_TSTZ     VR100
06: IR_BRANCHIF #15,    [==]
07: IR_ADD      VR0,    VR0,    VR1
08: IR_ADD      VR103,  VR0,    #5
09: IR_PARAM    --,     #1,     VR103
10: IR_SUB      VR104,  VR1,    #10
11: IR_PARAM    --,     #2,     VR104
12: IR_CALLVAL  VR105,  [test()]
13: IR_ADD      VR1,    VR1,    VR105
14: IR_JMP      #3
15: IR_RETVAL   --,     VR1
```

Figure 5.5: The corresponding IR listing for function `full_example`

Example

Figures 5.4 and 5.5 show another IR code generation example, this time involving the full instruction set, with branches and function calls.

CHAPTER 6

Step 2 – Calculating Variable Live Intervals

As prerequisite for register allocation, liveness information about all variables and temporary values need to be known.

Usually this is achieved with Data-Flow-Analysis techniques, which target the problem with iterative solutions over the control flow graph of the intermediate representation.

For linear scan, only contiguous *live intervals* are needed, which are conservative approximations of the live ranges (see section 2.7).

6.1 Simple Approach

We have developed an easy approximative approach to calculate the liveness intervals which fits in well to the *simple and fast* paradigm of TCC and linear scan. Consider how a liveness range would be calculated for straight-line code: we simply search on the IR forward from the start to find the *first definition* of a certain virtual register and then have the starting point of the interval. Accordingly, we search backwards for the *last usage* of this virtual register and have its interval's endpoint. This is illustrated in Figure 6.1. We label the intervals yielded by this trivial technique *basic live intervals*.

This method does not only work for straight-line code, but also for IR code only containing forward branches: if the last usage of a variable has been found at instruction n, any instructions executed after n won't branch to a location smaller than n and with this also never need the relevant virtual register again. No instruction is executed more than once. Hence it is guaranteed that the found start- and endpoint of the interval are correct.

However, as soon as we have backwards jumps (that is, by loops), the virtual register could be needed again in the future after instruction n has passed. We now simply *extend* the end of the simple live interval if there is any backwards jump on a later instruction that has a destination within the basic interval. This technique is depicted in Figure 6.2.

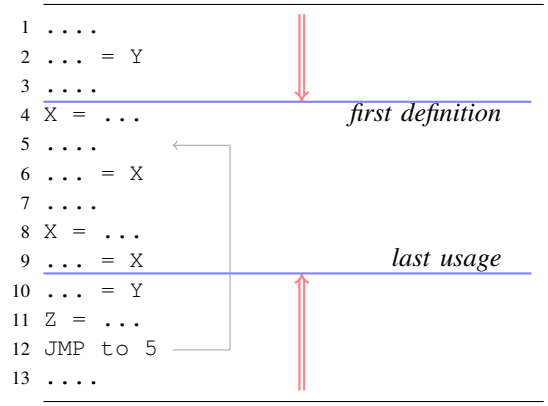


Figure 6.1: Basic live interval determination for virtual register $X \Rightarrow L = [4, 9]$

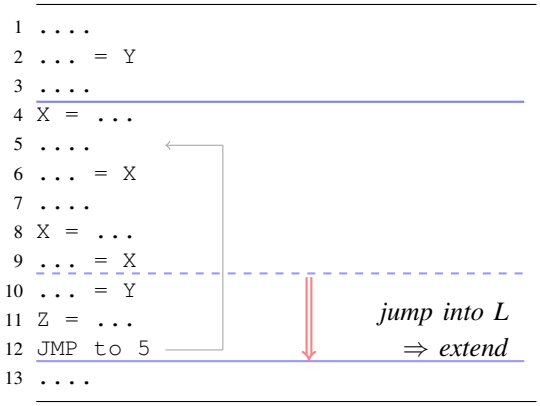


Figure 6.2: Live interval extension for virtual register $X \Rightarrow L' = [4, 12]$

Expressed more formally, the method works as the following:

1. find the basic live interval $L = [start_L, end_L]$ by linearly searching for first definition and last usage

2. repeatedly search downwards starting from instruction $end_L + 1$ for any branching instructions (`OP_JMP` or `OP_JMPIF`);
   if any particular instruction I with address I_{addr} and destination I_{dest} targets to within the basic live interval ($I_{dest} \in]start_L, end_L]$), extend the live interval by $L' = [start_L, I_{addr}]$.

The basic live intervals can already be determined efficiently in the IR generation phase: whenever an IR instruction is add to the listing, the basic interval of the occuring operands is updated accordingly. The relevant parts of the `IR_PutOp()` function is shown in Listing C.10. This way we only have to perform the linear search for the second step.

Listing C.11 shows the the core implementation for the simple liveness determination through live interval extension, based on the basic live intervals. The function sets the live interval for a given virtual register. Note that compiler-generated temporaries are never live across loop boundaries, hence step 2 is only needed for the *vreg* type of local variables. For temporaries, the function is called with `check_for_backwards_jumps` set to zero.

While our method is simple and obvious much faster than any data flow analysis which needs a CFG, it is very approximative and can lead to fairly large live ranges, especially for deeply nested loops. We sacrifice this imprecision in favor of fast compile-time and low implementation effort. As we show in chapter 9, the generated code is still efficient and the trade-off makes sense, keeping the original goal of TCC in mind.

6.2 Restriction

The method has one known restriction in terms of variable initializations: whenever a variable is live across loops, it has to be initialized *before* entering the loop. Usually this is not a problem, since the critical case can only occur if an initialization is bound to a conditional statement within the loop – if the initialization is unconditional, then the variable is not live across the loop, since the value is guaranteed to be written before it is read.

Consider the minimal code example in Listing 6.1. The first step of our algorithm yields the live interval $[7, 9]$ (this is now applied to the source listing line numbers, in reality it would be the corresponding numbers in the IR, of course). Since the later backwards jump isn't branching into this interval, it is not extended, and the final determined live interval is still $[7, 9]$. This is obviously wrong because the content of `var` can be still needed in a new interation of the loop.

Listing 6.1 Example of a code snippet where our simple live interval calculation fails

```
1  void foo(...)
2  {
3      int var;
4
5      for (...) {
6          if (...) {
7              var = ...;
8          } else {
9              ... = var;
10         }
11     }
12 }
```

Solution

We overcome the restriction by a special treatment for intervals that have their *first definition* within an if/else-block within a loop in a third step. In this case the live interval is extended to the most outer loop. For the example in Listing 6.1 the algorithm would yield the interval $[5, 11]$.

The source code snippet for this solution is shown in Listing C.12.

6.3 Interface to next phase

For each liveness interval calculated in this step, the linear scan module is informed through a call to `LS_AddLiveInterval()` which takes three arguments: the virtual register, the start point and the end point of the interval each.

CHAPTER 7

Step 3 – Performing Global Register Allocation with Linear Scan

Having the intermediate representation and its corresponding liveness information about the virtual registers ready, the register allocation itself is quite simple to implement. This is due to the nature of the linear scan algorithm: no graph data structure is built at all, but the central information we work on are simply the *liveness intervals* of the virtual registers, which are linearly iterated. Unlike in most graph-based approaches, the algorithm doesn't need to run multiple times after spill code has been introduced.

Note that the only goal of this phase is to map virtual registers in the IR to either concrete physical registers or, in case spilling is needed, with the target address on the stack (as offset from the frame pointer). Code generation, especially the critical part of generating spill code, out of the register-allocated IR is much more complex and will be treated in the next chapter.

The implementation of this phase is found in the file `tccls.c`.

7.1 Register Set Mapping

We use the register set R0–R14 in the following way (remember that R15 is the program counter and hence is not free):

register	**usage**
R0–R3	argument and return registers
R4–R10	**registers used by LSRA for virtual register substitution**
R11	frame pointer (`fp`)
R12	→ spilling register 1 (`ip`)
R13	stack pointer (`sp`)
R14	→ spilling register 2 (`lr`)

Out of 15 general-purpose registers, we use only seven (R4–R10) for the assignment to virtual registers ($K = 7$), that is, local variables or compiler-generated temporaries. This is due to some decisions that keep the implementation simple: by never using the argument and return registers for other purposes, we avoid the creation of extra spill instructions before and after function calls (the exception to this is if the calls are nested, see section 8.5). By reserving two registers for spilling (it's two because there are two source operands in the IR and in the final machine instructions, see section 8.3), we don't need to rewrite the IR code. We also use a frame pointer besides of the obligatory stack pointer, even though all local variables could be referenced from the stack pointer as well[1], but this is way more complicated.

7.2 Relevant Data Structures

While in the paper the data structures needed are described as *lists* we don't need any dynamic data structures like linked lists at all, but can easily implement everything with simple statically-sized arrays.

The List of Live Intervals

This list is generated from the previous phase, the data-flow analysis, and serves as input to the register allocator. The main loop of the linear scan traverses this list linearly after sorting it in order of increasing start point. We define a live interval with the following structure:

```
typedef struct {
    unsigned int vreg; /* virtual register */
    unsigned int reg;  /* assigned physical register */
    unsigned int loc;  /* spilling destination */
    unsigned int start;
    unsigned int end;
} LiveInterval;
```

Its the task of the register allocation algorithm to assign to every live interval (that is, to every virtual register in the IR) the fields `loc` and `reg`. Having the value `loc` set to non-zero implies that the virtual register is spilled and the `reg` field is ignored in this case. Conversely, if `loc` is set to zero, we know that a physical register `reg` has been assigned to the virtual register by the allocator.

The *list* of live intervals is simply created by a fairly large fixed-size array `LiveIntervals` and a variable `nIntervals` which indicates how many entries of the array are used:

```
#define MAX_LIVEINTERVALS 100000
LiveInterval LiveIntervals[MAX_LIVEINTERVALS];
int nIntervals;
```

With this data structure, sorting entries to some criteria is as easy as using `qsort()` provided by the C standard library, together with a proper compare function.

[1] in `gcc`, this optimization is turned on with the `-fomit-frame-pointer` option

The Pool of Free Registers

The *register pool* defines which of the physically available registers are currently free for allocation. It can be simply represented by an array of size K (K being the total number of registers available for allocation) that represents the status of the registers R_0 to R_{K-1} each: if entry n is non-zero, register R_n is free and vice versa. Initially, all registers are free, hence we set all entries of the `register_pool[]` array to one.

We need two functions operating on this data structure: one that returns the next free register from the pool and mark it as used (`RegisterPool_Get()`), and a second ones that frees a given register again (`RegisterPool_Pool()`). Both are shown in Listing C.13.

The Active List

The active list contains all live intervals that are currently overlapping. The size of this list never exceeds the number of available registers K; again we store the list in a linear array `ActiveSet` together with a counter variable `nActive`, but this time with pointers to `LiveInterval` entries.

```
LiveInterval* ActiveSet[K];
int nActive;
```

Adding a new element to the list can be done by indexing it with `nActive` which is incremented afterwards. To keep the list sorted by end intervals, we also have to call `qsort()` with the proper comparison function.

```
ActiveSet[nActive++] = &someliveinterval;
qsort(&ActiveSet[0], nActive, sizeof(ActiveSet[0]), licomp_endpoint);
```

7.3 Algorithm

Having prepared all necessary data structures and its manipulating functions, implementing the algorithm with the pseudocode from [PS99] is straight-forward. The algorithm is so simple that its implementation was basically completed within a few hours. The module consists only of about 200 lines of code, including debugging code.

The core functions `ExpireOldIntervals()`, `SpillAtInterval()` and `LSRA()` (see section 2.7 for a formal description) can be found in the Appendix Listings C.14, C.15 and C.16 each.[2]

[2]The interval compare functions `licomp_startpoint()` and `licomp_endpoint()` which are passed to the sorting function are trivial and hence not listed explicitely.

7.4 Interface to next phase

Now that all live intervals were assigned physical registers respective memory locations in case of spilling, we can create a table that maps virtual registers to destinations with fast access. The array `RegMapping[]` has the following structure:

```
typedef struct {
    int preg;
    int offset;
} VRegReplacement;
VRegReplacement RegMapping[MAX_VARS+MAX_TEMPS+MAX_PARAMS];
```

The array is defined in `tccir.c` and is filled out by the linear scan module for every live interval with the following function:

```
void IR_ReplaceVReg(int vreg, int offset, int preg);
```

To code generator can simply access the concrete target location of a virtual register $vreg$ by accessing `RegMapping[`$vreg$`]`.

CHAPTER 8

Step 4 – Generate Target Code

The last crucial step of our implementation is the generation of the object code. Through the decisions made by the register allocation algorithm in the previous step, we now have a IR code listing ready where all virtual registers have been mapped to either physical registers or spilling destinations.

8.1 Prologue and Epilogue

Prologue

The function prolog has the following tasks: save the link register to give the control back to the caller later, [1] save all callee-saved registers that are modified (at least `fp`, `sp`, optionally R4–R10), provide memory on the stack for the local variables. In our implementation, we additionally push the arguments on the stack and access them there, unless the function is a leaf-function (e.g. it contains no other function calls and hence doesn't need the registers R0–R3 for parameter passing respective receiving of return values). All this work is done by code performing the following steps:

1. save copy of stack pointer
2. if the function has arguments: push argument registers on the stack, adjust stack pointer
3. push callee-saved registers (including the stack pointer copy) and link register on the stack
4. set frame pointer
5. adjust stack pointer to the local variable space used (aligned to a multiple of 4)

Figure 8.1 shows the stack layout after the prolog is executed.

[1] Note that this step could theoretically be omitted, if we are guaranteed to know that the link register is never modified in this function. However, in our case where the `lr` is used as spilling and constant register, chances are very low for this – except for small leaf functions – and it doesn't pay off to implement this optimization.

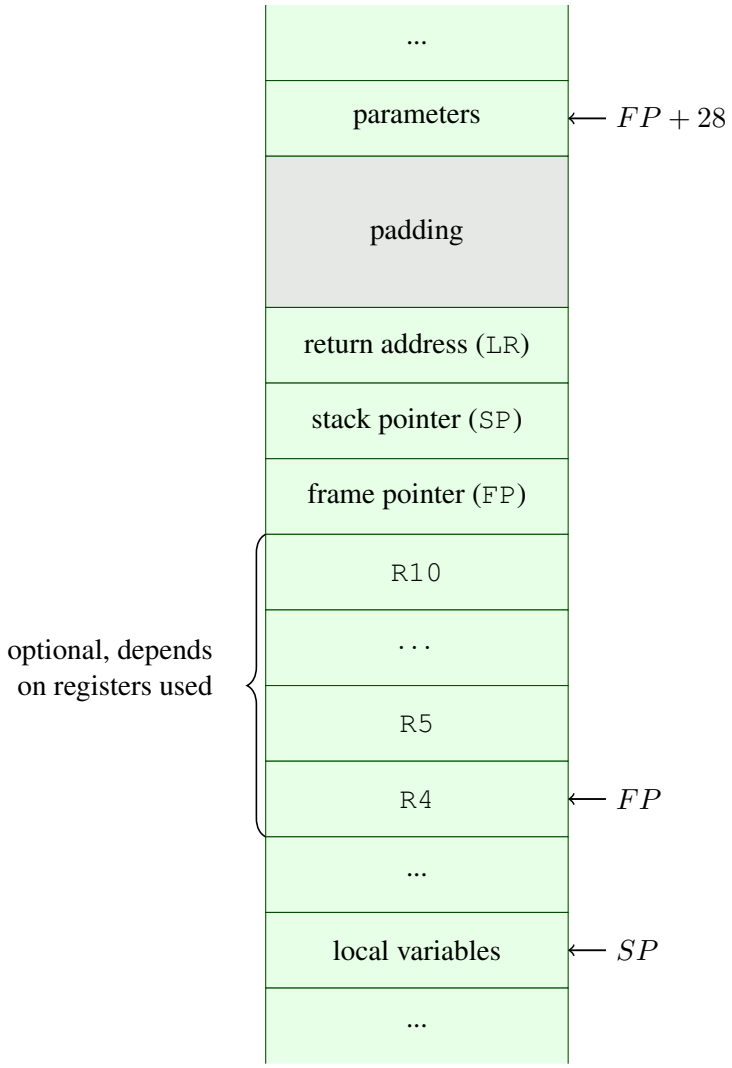


Figure 8.1: Stack layout after execution of prolog

Note that the gray padding area is needed to simplify the code generation by always access the function parameters at the same offset relative to the frame pointer. In case that the function is a leaf-function and there is no local variable space needed at the stack, the saving of stack and frame pointer is omitted.

The implementation of the prolog is located in `tccir.c:GenProlog()` (see Listing C.17).

Example

Suppose we have a function with two arguments that uses 10 bytes of stack space and three callee-saved registers (R4–R6) have been used by the register allocator. Then the prolog looks like the following:

```
00000000 <prolog_demo>:
   0:   e1a0c00d    mov     ip, sp                      ; step 1
   4:   e92d0003    push    {r0, r1}                    ; step 2
   8:   e24dd010    sub     sp, sp, #16                 ;
   c:   e92d5870    push    {r4, r5, r6, fp, ip, lr}    ; step 3
  10:   e1a0b00d    mov     fp, sp                      ; step 4
  14:   e24dd00c    sub     sp, sp, #12                 ; step 5
        ......
```

Epilogue

The function epilog has the purpose of restoring all the callee-saved registers from the stack and jumping back to the caller code (that is, the next instruction after the "branch-and-link" instruction). Since the return address is also saved in a register, we can perform this with a single block data load instruction:

```
        .....
  30:   e89ba870    ldm     fp, {r4, r5, r6, fp, sp, pc}
        .....
```

With the frame pointer as base, all saved registers R4–R10 are restored, the saved link register is transferred into the program counter. While the original TCC only puts the epilog once at the end and performs a jump to this location whenever desired, we save this jump and put this single instruction on every place where a return is performed.

The implementation of the prolog is located in `tccir.c:GenEpilog()` (see Listing C.17).

8.2 Data-Processing Instructions

The most elementary part of the code generator is the generation of data-processing instructions. Those are the instructions that were abstracted by the intermediate representation and thus represent the actual logic of the translated program.

Since ARM is a load/store architecture, those instructions must have either physical registers or constant values as operands. The function `GenDataProcessingOp()` is shown in Listing C.18.

8.3 Generation of Spilling Code

The category of *spilling registers* serves several purposes:

- access of variables in memory (global variables, variables on the stack)
- spill code temporary memory
- loading of large constants that don't fit into the immediate field of instructions

Suppose we have the following general IR three-address code instruction

$$Dest \rightarrow Src1\ (op)\ Src2$$

and R_{Src1}, R_{Src2} and R_{Dest} describe the physical register of the operands. If an operand R_X is not stored in a register, R_X is set to NIL. Then the IR instruction is translated with the following scheme:

- if $R_{Src1} = NIL$, then
  - generate code to evaluate $Src1$ into R12 (= spilling register 1)
  - $R_{Src1} \leftarrow 12$
- if $R_{Src2} = NIL$, then
  - generate code to evaluate $Src2$ into R14 (= spilling register 2)
  - $R_{Src2} \leftarrow 14$
- if $R_{Dest} = NIL$, then
  - $R_{Dest} \leftarrow 14$
- generate code for the data-processing instruction $R_{Dest} \leftarrow R_{Src1}\ (op)\ R_{Src2}$
- if $R_{Dest} = 14$, then

  generate code to store R14 into the memory location that is specified in $Dest$

This scheme is implemented with the following simplified code (see `tccir.c:IR_GenCode()`):

```
dest_reg = OperandInMemory(dest) ?
               LoadIntoReg(dest, 14) : RegMapping[dest->vreg].preg;
src1_reg = OperandInMemory(src1) ?
               LoadIntoReg(src1, 12) : RegMapping[src1->vreg].preg;
src2_reg = OperandInMemory(src2) ?
               LoadIntoReg(src2, 14) : RegMapping[src2->vreg].preg;

// generate ARM instruction with dest_reg, src1_reg and src2_reg

if (dest_reg == 14)
      StoreFromReg(dest, 14, 12);
```

Example

In the worst case, all three operands of a data-processing operations are in the memory, with large stack offsets. Consider the following example:[2]

```
    IR_ADD   StackMem[-40000], StackMem[-42000], StackMem[-44000]
```

This IR instruction is translated to object code by the code generator with the following steps:

- Prepare first source operand:
  1. → Load constant -42000 into R12
  2. → Indirect load of [FP-R12] into R12
- Prepare second source operand:
  1. → Load constant -44000 into R14
  2. → Indirect load of [FP-R14] into R14
- Generate actual ARM instruction:
  - → generate `ADD R12, R12, R14`
- Prepare destination operand and store result:
  1. → Load constant -40000 into R14
  2. → Store of R12 into [FP-R14]

[2]Note that this IR can not only emerge as result of a large number of spilled variables, but as well when high-indexed elements of very large fixed-size arrays on the stack are accessed

8.4 Jumps and Branches

For generating branches, we need a mapping between IR addresses and concrete target code addresses (that it, addresses in the text section). This is represented as a simple array that is filled during the first pass of the code generation phase, as the text section pointer proceeds.

The branch instructions (`IR_OP_JMP` and `IR_OP_BRANCHIF`) generate instructions with empty operands and are all patched in a second pass. This is shown in Listing C.19.

8.5 Function Calls

In the ARM architecture the first four arguments for functions are passed in dedicated registers. This reduces the calling overhead, but has also a drawback for our implementation: when having nested function calls, we need to save those registers in-between. Consider the following example, a simple expression involving nested calls and the according IR code:

```
Expression:
  f(10, g(20));

Corresponding IR listing:
================================
  IR_PARAM    --,    #1, #10
  IR_PARAM    --,    #1, #20
  IR_CALLVAL  VR100, [g()]
  IR_PARAM    --,    #2, VR100
  IR_CALLVOID --,    [f()]
================================
```

Both *10* and *20* are first parameters to function calls and hence have to be stored in register R0. But obviously, the straight approach would overwrite the *10* multiple times; first by the value *20*, then by the return value of function `g()`. The translation of `f(10, g(20));` would hence wrongly behave as `x = g(20); f(x, x);`. Even if there appears a function call without return value, R0 could still be modified within `g()`. [3]

For this reason, we have to keep track of the nesting level of function calls as well as the current parameter count of all the nesting levels. The affected registers have to be saved on the stack prior to first parameters (the start of a call phase, nesting level increases) and restored immediately after function calls (the end of a call phase, nesting level decreases). The implementation handling this parameter and return value passing is found in Listing C.20.

The correct translation of the example above is shown in Listing 8.1.

[3] Remember that register R0-R3 are caller-saved, hence we have to assume that any function call modifies them!

Listing 8.1 correct code for the expression `f(10, g(10))`

```
      ......
44:   e3a0000a    mov     r0, #10
48:   e92d0001    push    {r0}
4c:   e3a00014    mov     r0, #20
50:   ebffffe     bl      0 <g>
54:   e1a04000    mov     r4, r0
58:   e8bd0001    pop     {r0}
5c:   e1a01004    mov     r1, r4
60:   ebffffe     bl      20 <f>
      .....
```

CHAPTER 9

Results

The following sections describe how our implementation (abbreviated as "tccls" in this chapter) affects the performance of TCC both in run-time and compile-time. We compare the results with the popular GNU compiler gcc. All benchmarks and their results, including object dump listings for each compiler configuration (intended for people who don't have a proper target available but still want to see the generated code) can be found in the benchmarks folder of our repository.

9.1 Test Environment and Methods

We evaluate the performance on a Raspberry Pi Model B (see Appendix B for hardware details) running with the standard clock rate of 700 MHz. The board uses as operating system *Raspbian* Linux, the officially recommended distribution based on Debian "wheezy" 7.0[1], running kernel 3.10.25+.

The used compilers versions are tcc 0.9.26 and gcc 4.6.3. For a fair comparison, we tell gcc to compile for ARMv4 (by passing the parameter -march=armv4), as this is the architecture that is targeted by tcc and tccls. By default gcc would compile for architecture ARMv6, complying to the CPU (an ARM1176) of the Raspberry Pi.

All measurements are performed with the standard Linux tool time[2]. Due to some deviations in the returned execution times, the median of 15 samples is taken.

Much more exact values and other metrics like memory loads/store, cache misses/hits could be achieved with the performance counter subsystem in Linux, which is accessed by the tool perf. Unfortunately, though hardware counters are available for the Raspberry Pi, the kernel used doesn't support them yet.[3]

[1] see http://www.raspbian.org

[2] note that we use the seperate tool called time, found in /usr/bin/time, not the bash built-in time

[3] see http://web.eece.maine.edu/~vweaver/projects/perf_events/faq.html#q4b

9.2 Simple Showcase Example: The Collatz Conjecture

The collatz conjecture is an unsolved mathematical problem first proposed by Lothar Collatz in 1937.[4] It states that starting with an arbitrary integer a_0 and repeatedly applying the iteration

$$a_{n+1} = \begin{cases} \frac{a_n}{2}, & \text{if } a_n \text{ is even} \\ 3a_n + 1, & \text{if } a_n \text{ is odd} \end{cases}$$

finally always returns to 1. For example, for $a_0 = 20$ the iteration is $20 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$.

The benchmark is a simple program which accepts an unsigned number n as command line argument and checks the conjecture for all integers in the range $[1, n]$ (see Listing 9.1 for the checking routine). Note that this program is extremely simple and fulfills the optimal prerequisites for a significant speedup through register allocation: due to the small number of local variables no spills are introduced, most of the time is spent in the iteration loop minimizing the function call overhead. The expressions are all very short, leading to a high load/store overhead in the original TCC.

Listing 9.1 Collatz Conjecture Check, C-Code

```
1  int collatz(int an)
2  {
3      int iters = 1;
4
5      while (an != 1) {
6          /* odd number, a[n] = a[n-1]*3 + 1 */
7          if (an & 1) {
8              an *= 3;
9              an += 1;
10         }
11         else /* even number, a[n] = a[n-1]/2 */
12             an >>= 1;
13
14         iters++;
15     }
16
17     return iters;
18 }
```

Test case: We call the benchmark with $n = 2000000$ (two million.)

Result: The comparison of the generated functions for original TCC and our improved version is shown in listings 9.1 and 9.2 each. The register allocated code executes in 5.18s compared to 10.58s with the original code, meaning a speed-up of more than 100%. Examining the loop

[4] see `http://mathworld.wolfram.com/CollatzProblem.html`

```
00000000 <collatz>:
   0:   mov ip, sp
   4:   push    {r0, r1}
   8:   push    {fp, ip, lr}
   c:   mov fp, sp
  10:   sub sp, fp, #4
  14:   mov r0, #1
  18:   str r0, [fp, #-4]
  1c:   ldr r0, [fp, #12]
  20:   cmp r0, #1
  24:   beq 78 <collatz+0x78>
  28:   ldr r0, [fp, #12]
  2c:   and r0, r0, #1
  30:   teq r0, #0
  34:   beq 58 <collatz+0x58>
  38:   ldr r0, [fp, #12]
  3c:   mov r1, #3
  40:   mul r0, r1, r0
  44:   str r0, [fp, #12]
  48:   ldr r0, [fp, #12]
  4c:   add r0, r0, #1
  50:   str r0, [fp, #12]
  54:   b   64 <collatz+0x64>
  58:   ldr r0, [fp, #12]
  5c:   lsr r0, r0, #1
  60:   str r0, [fp, #12]
  64:   ldr r0, [fp, #-4]
  68:   mov r1, r0
  6c:   add r0, r0, #1
  70:   str r0, [fp, #-4]
  74:   b   1c <collatz+0x1c>
  78:   ldr r0, [fp, #-4]
  7c:   nop
  80:   ldm fp, {fp, sp, pc}
```

Figure 9.1: `collatz(),tcc`

```
00000000 <collatz>:
   0:   push    {r4, r5, lr}
   4:   mov r4, #1
   8:   cmp r0, #1
   c:   beq 3c <collatz+0x3c>
  10:   and r5, r0, #1
  14:   teq r5, #0
  18:   beq 2c <collatz+0x2c>
  1c:   mov lr, #3
  20:   mul r0, lr, r0
  24:   add r0, r0, #1
  28:   b   30 <collatz+0x30>
  2c:   lsr r0, r0, #1
  30:   mov r5, r4
  34:   add r4, r5, #1
  38:   b   8 <collatz+0x8>
  3c:   mov r0, r4
  40:   pop {r4, r5, pc}
```

Figure 9.2: `collatz(),tccls`

instr. type	`tcc`	`tccls`
#loads	4	0
#stores	2	0
#branches	3	3
#data-proc.	6	6
#total	15	9 (-40%)

Table 9.1: instruction count for loop iteration, "even" case

instr. type	`tcc`	`tccls`
#loads	5	0
#stores	3	0
#branches	4	4
#data-proc.	8	8
#total	20	12 (-40%)

Table 9.2: instruction count for loop iteration, "odd" case

body, we see that for example the if-block is translated to 7 instructions (address range 38–50), including two load/store-pairs, where the extended TCC yields only 3 instructions (address range 1C–24), without any loads or stores.

The instruction count for one loop iteration for both cases (even number and odd number) is shown in tables 9.1 and 9.2, respectively. In both cases, the instruction count is reduced by 40%.

Assuming that each instruction type takes the same amount of time and the time spent outside this loop is negligible, this would mean a speedup of $\frac{1}{1-0.4} \approx 67\%$. We come close to the actual measured speedup by assuming the execution time of load/store instructions corresponds to 1.5 data-processing instructions. Note that we have to assume to always have data cache hits here – on scenarios with lots of memory accesses in-between, load/store instructions may take much longer! Nevertheless, we can safely say that we save at least 1.5 instructions for each variable access that is register allocated on this concrete target.

9.3 MiBench Benchmarks

MiBench is a freely available embedded benchmark suite made up by the EECS Department of the University of Michigan.[GRE+01] It contains 35 embedded applications which are claimed by the authors to be commercially representative. The application set is divided up into the following six suites, where each targets a specific area of the embedded market: *Automotive and Industrial Control*, *Consumer Devices*, *Office Automation*, *Networking*, *Security* and *Telecommunications*.

Due to the lack of floating point support and other language restrictions for our proof-of-concept (see section 4.3) the set of possible applications to choose for benchmarking was very limited. We picked five benchmarks, one of each suite except of the category "consumer devices", where every application contains heavy floating-point calculations needed for image or audio processing.

BitCount

This simple benchmark is from the "Automotive" suite and counts the number of bits in an integer with four different algorithms:

- (⇒ bitcount1) "Optimized 1 bit/loop counter": a loop which executes exactly the number of bit times, working through bit-manipulations
- (⇒ bitcount2) "Ratko's mystery algorithm": computes without a loop, but successive bit-mask and shift operations
- (⇒ bitcount3) "Recursive bit count by nibbles": recursive version of bitcount4
- (⇒ bitcount4) "Non-recursive bit count by nibbles": computes by repeatedly accessing a prestored "bits per nibble" lookup table

Adaptions in source code: None.

Test case: We use the test module (bitcnts.c) that is shipped with the benchmark. It takes a number n as command line argument and measures the execution time for counting bits for n numbers of a fixed sequence. This is done seperately for each of the four bit counting methods. We execute the program with $n = 20000000$ (twenty million).

Note that for a fair comparison, only the bitcount modules (bitcnt_1.c, bitcnt_2.c, bitcnt_3.c and bitcnt_4.c) are compiled with different compilers/options, but the test module is always compiled with the same compiler. This way we ensure that the execution time differences are solely caused by the bit counting routines.

SHA1

Belonging to the "Security" suite, this benchmark calculates the SHA1[5] hash for a given input file.

Adaptions in source code: None.

Test case: We execute the application with the source code archive of gcc 4.5.0 (.tar.gz format) [6] which is ≈ 82 MB in size.

String Search

This application is part of the "Office" suite and contains "string searching" algorithms, i.e. routines that find the place of a given string (the *pattern*) within another given, larger string. The search is done with the classical algorithms Pratt-Boyer-Moore (⇒ strsearch1) and Boyer-Moore-Horspool (⇒ strsearch2), both searching case-sensitive.

Adaptions in source code: The original source code encodes the test strings directly within the file, which isn't very flexible. The test data was too small to yield meaningful execution time result, as the larger string was never longer than 80 characters. We extended the test program to

[5] SHA1 is for "Secure Hash Algorithm 1" and is described in RFC 3174, see http://tools.ietf.org/html/rfc3174

[6] available at ftp://gd.tuwien.ac.at/gnu/gcc/releases/gcc-4.5.0/gcc-4.5.0.tar.gz

take a patterns file and a search text file as command line arguments. Each line of the patterns file is stored into an array of strings, and the content of the search text file is stored into a single large string. Each pattern is then searched within the large string.

Test case: We search the text from James Joyce's *Ullyses*[7], having about 1.5 MB in size, for a list of 458 "bad words"[8].

ADPCM

ADPCM is for "Adaptive differential pulse-code modulation" and denotes a audio signal encoding method. The benchmark belongs to the "Telecommunications" suite and comes with an encoding (⇒ `adpcm encode`) and a decoding (⇒ `adpcm decode`) application, each taking a file as input that is given on the command line and writing the result to the standard output.

Adaptions in source code: None.

Test case: The applications are executed with the supplied test input files, which are about 25 MB (for encoding) and 6 MB (for decoding) in size each.

Dijkstra

As the name suggests, this benchmark from the "Network" suite implements the graph search algorithm by Dijkstra, which solves the shortest path problem for a given graph with non-negative edge path costs. The graph is provided as distance matrix in a text file which is supplied as command line argument. The program calculates the shortest path for 100 vertex pairs.

Adaptions in source code: In the core function `dijkstra()`, all accessed variables (except the parameters) are declared as global variables, though they are not used in any other function. This means that we don't have any register allocation candidates (besides of temporary values) and the speedup is moderate. We declared the variables (`i`, `ch`, `iPrev`, `iNode`, `iCost`, `iDist`) as local and split up this benchmark in two subtests, the supplied code (⇒ `dijkstra orig`) and the modified code with more register allocation candidates (⇒ `dijkstra mod`).

Test case: The test is performed with the supplied test graph, consisting of 100 nodes.

[7] available from Project Gutenberg: `http://www.gutenberg.org/ebooks/4300.txt.utf-8`
[8] available at `http://urbanoalvarez.es/blog/2008/04/04/bad-words-list/`

9.4 Complex Benchmark

bzip2

As largest and most complex benchmark we use an early version of the open-source lossless data compressing tool `bzip2`, written by Julian Seward.[9] It comes as single source file with more than 4000 lines of code involving operations like CRC calculation, Huffman coding routines, sorting algorithms etc. The tool can compress (⇒ `bzip2 inflate`) and decompress (⇒ `bzip2 deflate`) files as well as test the integrity of compressed files (⇒ `bzip2 test`).

Adaptions in source code: In some verbose (`-v`) outputs, floating point numbers are used for the calculation of statistical values. As the proof-of-concept can't handle floating points, we commented those out – the benchmarked functionality of the tool is not affected by that. We also commented out the abortion if an output file already exists. This enables us to use our benchmark script which performs the same command repeatedly.

Test case: The compression is applied on the source code archive of busybox 0.60.5 (`.tar` format) [10] which is $\approx$ 2900 KB in size. The decompression and test sub-benchmarks are applied on the result from the compression.

9.5 Run-Time Evaluation

Table 9.3 lists the execution-time results for each benchmark.

The performance of code generated by `tcc` and `gcc -O0` is about equal. In three out of our 15 test cases, `tcc`-code performs even slightly better.

The speedup for each benchmark of `tccls` compared to the original `tcc` is shown as bar chart in Figure 9.5 (p. 62). On average, we have a performance improvement of $\approx 32.6\%$. The largest speedups are achieved on the benchmarks `bitcount1` (+132.27%), `collatz` (+104.64%) and `adpcm code` (+62.10%). Looking at those test cases, we find some common properties that advantage a code quality improvement through our global register allocator:

- generally: a high usage rate of register allocation candidates; that is, in our implementation, local variables and parameters in leaf functions
- a large amount of short expressions (e.g. `x++;` or `y += n;`) that would lead to a high density of additional load/store pairs without register allocation
- if the number of register allocation candidates exceeds R: many local variables with short live ranges (not spanning across loops) to minimize the need for spilling

[9] see `http://www.bzip.org/`; the version used for benchmarking is available at `http://gd.tuwien.ac.at/gnu/sourceware/bzip2/v01pl2/bzip2-0.1pl2.tar.gz`

[10] available at `http://www.busybox.net/downloads/busybox-0.60.5.tar.bz2` (compressed)

Benchmark	`tcc`	`tccls`	speedup[1]	`gcc -O0`	`gcc -O1`	speedup[2]
collatz	10.58	5.17	⇒ +104.64%	10.44	2.87	⇒ +80.14%
bitcount1	10.15	4.37	⇒ +132.27%	10.52	3.10	⇒ +40.97%
bitcount2	3.59	2.77	⇒+29.60%	3.44	2.14	⇒ +29.44%
bitcount3	11.66	11.20	⇒ +4.11%	11.91	6.43	⇒ +74.18%
bitcount4	4.75	4.29	⇒ +10.72%	4.48	2.57	⇒ +66.93%
sha1	26.00	20.59	⇒ +26.27%	24.79	6.65	⇒ +209.62%
strsearch PBM	15.22	13.30	⇒ +14.43%	13.27	11.36	⇒ +17.08%
strsearch BMH	13.25	12.20	⇒ +8.61%	13.46	10.73	⇒ +13.70%
adpcm code	4.62	2.85	⇒ +62.10%	4.41	1.63	⇒ +74.85%
adpcm decode	3.58	2.81	⇒ +27.40%	3.48	1.18	⇒ +138.14%
dijkstra orig.	1.72	1.69	⇒ +1.77%	1.50	0.75	⇒ +125.33%
dijkstra mod.	1.26	0.95	⇒ +32.63%	1.09	0.57	⇒ +66.67%
bzip2 inflate	18.51	16.66	⇒ +11.10%	17.62	10.36	⇒ +60.81%
bzip2 deflate	5.04	4.84	⇒ +4.13%	4.76	3.47	⇒ +39.48%
bzip2 test	7.18	6.04	⇒ +18.87%	7.09	4.50	⇒ +34.22%
average MiBench			⇒ **+31.81%**			⇒ **+77.90%**
average all			⇒ **+32.58%**			⇒ **+71.44%**

[1] speedup of `tccls` ⇒ `tcc`
[2] speedup of `gcc -O1` ⇒ `tccls`

Table 9.3: Summary of run-time benchmarks

The worst speedup is achieved with the test case `dijkstra orig.` (+1.77%). In this code, all variables accessed in the core function (`dijkstra()`) are declared as global, hence we simply don't have any register allocation candidates available. We modified the benchmark (`dijkstra mod.`) and moved declared all those variables (which wouldn't be used in other functions anyway) as local, leading to a significant speedup of +32.63%.

The bitcount algorithm `bitcount3` uses a lookup-table for the calculations, has only one local variable and even works with recursion, meaning that we don't have a leaf function. Hence the speedup is also rather decent with +4.11%. All other MiBench test cases yield a speedup of at least +8.6%, the average speedup of all MiBench applications is +31.81%.

The average speedup of the complex benchmark test cases (`bzip2 inflate`, `deflate` and `test`) is +11.37%, which is still significant, considering that this application is very memory-intensive and primarily works with lookup-tables and consists of complex expressions mostly with operands in memory.

`tccls` vs. `gcc -O1`

There's a significant difference in the quality of the code generated by `tccls` and `gcc -O1`. As Table 9.3 shows, `gcc` with optimization performs clearly better than our implementation with a speedup of almost 80% in average – some benchmark execute in less than half the time than `tccls` (+ >100%), in the one extreme case of `sha1` the speedup is even more than 200%.

The reason for this is the fact that gcc performs by far more optimizations than global register allocation. The following differences occur in `gcc`-optimized code compared to our `tccls`-code:

- usage of *combined shift instructions* for data-processing instructions
- usage of *conditional instructions* for small if/else-blocks to save branch instructions
- loading of large 32-bit constants: `gcc`-code stores constants at the end of a function and loads them with a single load instruction; `tcc(ls)`-code stores constants in-between the code and hence needs an extra branch instruction to continue the execution flow after loading

Taking again the simple Collatz conjecture example, Figures 9.3 and 9.4 show the listings the code for the function `collatz()`, generated by `tccls` and `gcc -O1` each.

```
00000000 <collatz>:
  0:  push {r4, r5, lr}
  4:  mov  r4, #1
  8:  cmp  r0, #1
  c:  beq  3c <collatz+0x3c>
 10:  and  r5, r0, #1
 14:  teq  r5, #0
 18:  beq  2c <collatz+0x2c>
 1c:  mov  lr, #3
 20:  mul  r0, lr, r0
 24:  add  r0, r0, #1
 28:  b    30 <collatz+0x30>
 2c:  lsr  r0, r0, #1
 30:  mov  r5, r4
 34:  add  r4, r5, #1
 38:  b    8 <collatz+0x8>
 3c:  mov  r0, r4
 40:  pop  {r4, r5, pc}
```

Figure 9.3: `collatz()`, `tccls`

```
00000000 <collatz>:
  0:  mov    r3, r0
  4:  cmp    r0, #1
  8:  beq    30 <collatz+0x30>
  c:  mov    r0, #1
 10:  tst    r3, #1
 14:  addne  r3, r3, r3, lsl #1
 18:  addne  r3, r3, #1
 1c:  lsreq  r3, r3, #1
 20:  add    r0, r0, #1
 24:  cmp    r3, #1
 28:  bne    10 <collatz+0x10>
 2c:  mov    pc, lr
 30:  mov    r0, #1
 34:  mov    pc, lr
```

Figure 9.4: `collatz()`, `gcc -O1`

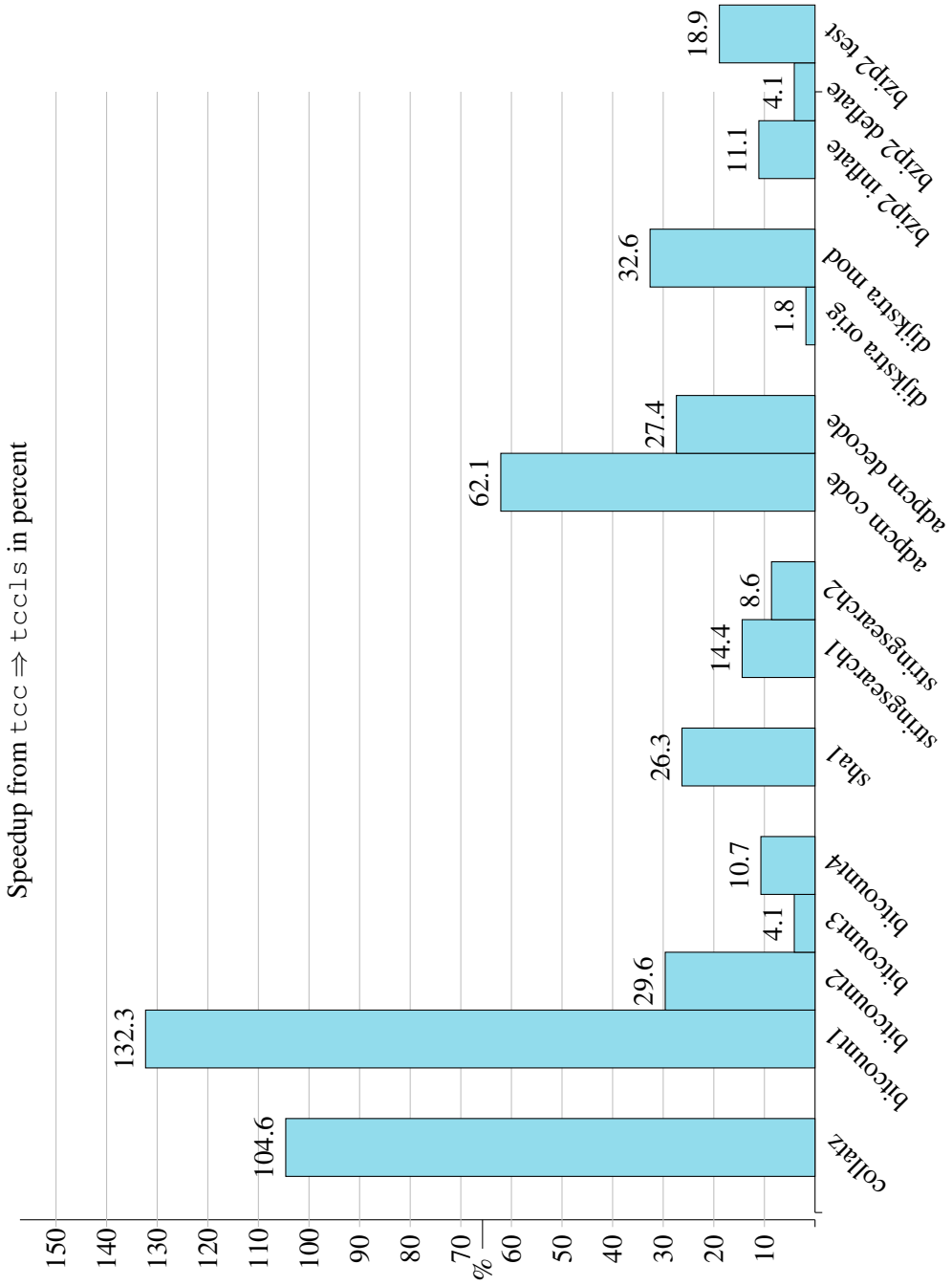


Figure 9.5: Run-time speedup of tested applications yielded by our linear scan implementation

9.6 Compile-Time Evaluation

Table 9.4 lists the compilation times for each benchmark. We intentionally didn't include the "collatz" and "bitcount" as those, each consisting of only a single small function, were compiled too fast to yield meaningful compilation time results.

Benchmark	`tcc`	`tccls`	slowdown	`gcc -O0`	slowdown	`gcc -O1`	slowdown
sha1	0.11	0.12	$\Rightarrow \times 1.09$	1.41	$\Rightarrow \times 12.82$	1.97	$\Rightarrow \times 17.91$
strsearch	0.09	0.09	$\Rightarrow \times 1.00$	1.18	$\Rightarrow \times 13.11$	1.75	$\Rightarrow \times 19.44$
adpcm	0.09	0.10	$\Rightarrow \times 1.11$	1.15	$\Rightarrow \times 12.78$	1.38	$\Rightarrow \times 15.33$
dijkstra	0.07	0.07	$\Rightarrow \times 1.00$	0.89	$\Rightarrow \times 12.71$	1.22	$\Rightarrow \times 17.43$
bzip2	0.26	0.30	$\Rightarrow \times 1.15$	7.03	$\Rightarrow \times 27.04$	20.50	$\Rightarrow \times 78.85$
average			$\Rightarrow \times \mathbf{1.07}$		$\Rightarrow \times \mathbf{15.69}$		$\Rightarrow \times \mathbf{31.59}$

Table 9.4: Summary of compile-time benchmarks (The values in the "slowdown" columns are all related to the original `tcc`)

The results show that `tccls` is almost as fast as `tcc` – even with the complex benchmark of `bzip2`, the slowdown is only 1.15. Consequently, the goal of keeping the property of high compilation speed of `tcc` with the implementation of the global register allocator is clearly achieved. Our optimized implementation is still at least twelve times faster than `gcc` without any optimization.

9.7 Implementation Effort

Table 9.5 shows the implementation effort quantified by lines of code and the estimated relative time exposure for each of the four phases.

Phase	LOC	rel. time exposure
IR code generation	400	45%
Liveness analysis	400	15%
Linear Scan register allocation	200	5%
Code generation	500	35%
total	1500	100%

Table 9.5: Rough implementation effort of the `tccls` extension

Designing and implementing an intermediate representation for TCC and generating correct and fast target code out of the register-allocated listing were clearly the most time-consuming implementation tasks. In contrast, register allocation with linear scan was surprisingly simple and easy to integrate, and for the liveness analysis we intentionally developed a method that was simple and thus straight-forward to implement too.

CHAPTER 10

Conclusions and Future Work

Within this thesis we have implemented a proof-of-concept global register allocator for TCC using the simple but yet very promising linear scan approach. We designed an intermediate representation instruction set close to that of the target architecture (ARM) which abstracts the concrete destination for register allocation candidates with virtual registers. For keeping the high execution speed of TCC, we developed a simple and fast live variable analysis algorithm which yields conservative intervals but performs much faster than usual data-flow analysis techniques. The code generator implements some optimizations regarding prolog/epilog and faster argument access for leaf functions.

The quality of the produced code ($\Rightarrow$ *run-time speed*) has improved by more than 32% on average, while the featuring property of the compiler of being fast ($\Rightarrow$ *compile-time speed*) is still met – our optimizing compiler executes still more than ten times faster than `gcc` without any optimization.

The results support the original claim that linear scan is a very attractive option for systems where manpower lacks and compile-time is an important goal.

Though we have no comparison to the efficiency other global register allocation strategies like graph coloring for this compiler, we are convinced that the trade-off between implementation effort/compile-time speed and run-time speed is best met by linear scan.

10.1 Further Implementation

For keeping the implementation effort low, especially considering the code generation part, we have decided to use simple approaches in many cases, sacrificing target code quality. The run-time speed could be further enhanced by the following improvements:

- Also treat function parameters as register allocation candidates – currently all parameters are accessed on the stack, except for leaf functions, where the first four arguments are accessed in R0–R3

- Also use R0–R3 for register allocation, on sections where no function calls occur – currently only R4–R10 are used for substitution to virtual registers, limiting the number of physical registers to 7
- Use a more sophisticated liveness analysis – currently the liveness calculation is very naive, leading to overly conservative live ranges and thus to many spills
- Implement other optimizations for TCC that are now possible with the existence of a IR generation pass, for example: constant propagation, common subexpression elimination, function inlining

10.2 Future Projects

Based on this thesis, several projects would be interesting to do to support further research about global register allocation:

- For this concrete target: implement a kernel module which accesses the hardware counters, to perform a more detailled analysis of the generated code with the `perf` tool
- Port the implementation to other target architectures (e.g. x86, x86-64) and analyze how different architectural properties (e.g. RISC or CISC architecture, number of available registers, cache etc.) and code generation influence the run-time performance of our implementation
- Implement a global register allocator based on graph coloring (also needs data-flow analysis) and compare the results to linear scan

APPENDIX A

The ARM Architecture

A.1 History and Overview

The ARM architecture originated within *Acorn Computers Ltd.*, a British computer company founded in 1978. It was best known for the *BBC Micro*, an 8-bit home computer similar to the Commodore 64 that dominated the UK educational computer market in the 1980s and early 1990s.[1]

When the company planned the successor to this series, the designers were looking for a replacement for the 6502 processor. Since none of the 16-bit architectures available at that time were satisfying, they decided to design their own 32-bit processor, inspired by the Berkeley RISC [SP82] project. The *Acorn RISC Machine* [2], released in 1987 as ARM2[3], was one of the world's first commercial RISC processors.

Later, when other companies became interested in the processor, Acorn's processor design team split off and together with Apple Computer and VLSI Technology, founded *Advanced RISC Machines Ltd.*[4] in 1990. It was later renamed to *ARM Holdings*.

The company is designing the famous ARM family of RISC processor cores, which is now extremely popular in embedded mobile systems – 95% of the world's smart phones contain at least one ARM processor. This success is due to the reduced cost, heat and power usage compared to more complex designs. Other uses include digital cameras, automotive systems and gaming consoles.

The processors are not manifactured by ARM Holding itself, but rather is based on an IP (Intellectual Property) business model: silicon is produced only by companies who license the ARM processor designs.

Basic properties of ARM processors are as follows:

[1] see `http://www.telegraph.co.uk/finance/newsbysector/epic/arm/8243162/History-of-ARM-from-Acorn-to-Apple.html`

[2] the original meaning of ARM

[3] ARM1 was a prototype which was never released

[4] the meaning of the acronym ARM was changed accordingly

- Load/Store architecture
- uniform register set (32-bit)
- fixed size instruction set (32-bit)

A.2 Register File

In total, 37 registers physically exist in ARM, all 32 bits long. At one time, at most 16 of these registers are accessible (R0-R15), plus at most two program status registers (CPSR and SPSR, **c**urrent and **s**aved), depending on the mode the processor is in. We assume to only operate in user and system modes here, both having the same register set visible. Other modes are used for interrupts, exception handling and memory access violations.

Though all of the 16 registers can be used as operands in most instructions (encoded in a 4-bit field), two of them have special purposes defined by hardware:

- **R15** = PC: *Program Counter*. It can also be used in most instructions to get the address of the current instruction plus eight, which is two instructions after[5]. All instructions must be word-aligned, hence the PC resides in bits 32–2 and bits 1–0 are undefined.
- **R14** = LR: *Link Register*. After a Branch and Link (`BL`) instruction, this register holds the address of the next instruction, that is the return address for a subroutine call. If the register is saved/restored on entry/return of functions, it can be used as general-purpose register in-between.

Since one of the 16 registers, R15, can not be used as a data store, it is often stated that ARM has 15 GPRs available in user/system mode. The uses of those registers are entirely defined by software. The standard uses and the rules regarding procedure calls (the *calling conventions*), defined by ARM's EABI, are covered in section A.4.

The CPSR contains four condition code flags (**N**egative, **Z**ero, **C**arry and o**V**erflow) and the current processor mode. The flags can be set as results of arithmetical and logical operations.

A.3 Instruction Set

In the following, we will describe the general structure of the ARM instruction set and give an overview of all instructions that are used by TCC's code generator.

While the newest ISA version available is ARMv8[6], TCC generates ARMv4 code[7]. Modern ARM processors also include other instruction sets, like *THUMB* for a compact 16-bit wide alternative instruction encoding, *NEON* for SIMD (Single Instruction Multiple Data) processing or *VFP* for floating-point processing. We only use the core 32-bit ARM instruction set.

[5] this offset is caused by instruction prefetching

[6] ARMv8-A, adding a 64-bit architecture, has been first implemented by Apple in the *Apple A7* chip, part of the iPhone 5s, in the end of 2013

[7] note that all architectures up to ARMv3 are now obsolete

Code	Flags	Suffix	Description
0000	Z set	EQ	Equal
0001	Z not set	NE	Not equal
1010	N == V	GE	Greater than or equal (signed)
1011	N != V	LT	Less than (signed)
1100	Z not set && N == V	GT	Greater than (signed)
1111	Z set || N != V	LE	Less than or equal (signed)
1110	any	AL	Always (default)

Table A.1: Some condition codes for ARM instructions

More details about the instruction set for various versions can be found in the ARM-ARM (ARM Architecture Reference Manual). [Sea00]

General Organisation

ARM instructions have a three-address format, consisting of two source registers and one destination register. The commonly used assembler syntax is as follows:

```
<op>{cond}{flags} Rd, Rn, Operand2
```

- <op>: Three-letter mnemonic representing the opcode, e.g. SUB, MOV
- {cond}: Optional two-letter condition code (see next subsection below), e.g. EQ, NE
- {flags}: Optional additional flags, e.g. S
- Rd: Destination register
- Rn: First source register
- Operand2: Flexible second operand; may be register or immediate value

Note that this is the general format and some instructions deviate from it, for example the MOV instruction omits the first source registern Rn.

Condition Code

All instructions contain a 4-bit wide *condition field* (bits 31–28) that defines the condition under which an instruction is to be executed. Table A.1 lists some conditions with the corresponding assembler mnemonics and meanings. TCC uses the condition code primarily for implementing conditional branches.

Code	Mnemonic	Description
1101	MOV	Rd := Op2
1111	MVN	Rd := NOT Op2
0000	AND	Rd := Op1 AND Op2
0001	EOR	Rd := Op1 EOR Op2
1100	ORR	Rd := Op1 OR Op2
0010	SUB	Rd := Op1 - Op2
0011	RSB	Rd := Op2 - Op1
0100	ADD	Rd := Op1 + Op2
1000	TST	set condition codes on Op1 AND Op2
1001	TEQ	set condition codes on Op1 EOR Op2
1010	CMP	set condition codes on Op1 - Op2
1011	CMN	set condition codes on Op1 + Op2

Table A.2: ARM Data-processing operations

Data-Processing Instructions

The majority of the operations we use fall into the category of data processing instructions. Since we have a Load/Store-architecture, operands can only be registers or immediate values. The instruction encoding is as follows:

- Cond: condition code, see above
- I: 0 $\rightarrow$ Operand2 is a register, 1 $\rightarrow$ Operand2 is an immediate value
- OpCode: defines the operation, see below
- S: 0 $\rightarrow$ do not modify condition codes, 1 $\rightarrow$ set condition codes
- Rd: destination register
- Rn: first operand register
- Operand2: depending on I, contains a register together with shifting properties, or an immediate value with shifting properties

Table A.2 lists all opcodes that are used in TCC's code generator together with its semantics.

A special instruction that deviates from the data-processing instruction encoding, but logically also belongs to the same category, is the `MUL` operation. It multiplies two signed or unsigned 32-bit values and writes the least significant 32 bits of the result to the destination register.

There is no division instruction for the ARMv4 architecture, thus it has to be emulated by software.[8]

[8] ARM EABI defines the functions `__aeabi_idivmod` and `__aeabi_udivmod` (for signed and unsigned division/modulo calculation) for this purpose, see chapter 5

Load and Store Instructions

The ARM data sheet defines two types of instructions that can read or write to memory: *Single Data Transfer* instructions (LDR, STR) and *Block Data Transfer* instructions (LDM, STM). For practical reasons, we only associate the former group to "load/store instructions" in this thesis, since the latter group is mostly used to save/restore registers on the stack – it is described below in the subsection "stack operations".

The LDR instruction is used to to load a single word or byte of data (in case of a byte, the mnemonic is LDRB). Accordingly, the STR instruction is used to store a single word or byte of data (in the latter case, the mnemonic is STRB).

The memory address is calculated by adding or subtracting (depending on the Up/Down bit) a *base register* and an offset. The offset can again be a register or a 12-bit immediate value. The offset register can optionally be shifted by an 8-bit value, however this is not used in tcc.

There are also special instructions for halfword and signed data transfer. These are used to load/store 16-bit values or to load sign-extended bytes or halfwords of data (corresponding to the `short` data type and the `signed` modifier in C).

Branch Instructions

There are only two types of branch instructions in ARMv4: Branch (B) and Branch with Link (BL). Both contain a 24-bit offset (signed two's complement) that will be shifted left two bits, signed extended two 32-bits and add to the PC. This way, relative branches in the range of +/- 32MB ($= 2^{26-1}$) are possible.

The *Branch with Link* instruction additionally saves the address of the instruction following in R14, (the link register). This is obviously used to be for function calls – the function can then simply return to the caller by moving R14 into R15, the program counter[9] (given that the link register isn't modified in-between).

Note that for conditional branches there is no need for seperate instructions, as simply the condition code is used, yielding the mnemonics BEQ, BNE, BGT, BLT, BGE, BLE.

Stack Operations

Unlike other architectures, ARM doesn't have dedicated stack operations (like PUSH and POP for Intel x86 or 8051), but rather has more general and flexible instructions that can be used for that purpose. Those so-called *Block Data Transfer* instructions load (LDM) or store (STM) any subset of the currently visible registers. Though these instructions are also convenient for moving large data blocks around in main memory, TCC uses them solely for saving and restoring context on the stack.

The syntax is as follows:

```
LDM{cond}<FD|ED|FA|EA|IA|IB|DA|DB> Rn{!},<Rlist>
STM{cond}<FD|ED|FA|EA|IA|IB|DA|DB> Rn{!},<Rlist>
```

[9] there are no special instructions for indirect branches in ARM, but simply ordinary load or data-processing instructions with R15 as destination operand are used

The two-letter mnemonic following the condition specifies the addressing mode.

The ARM stack according to the Procedure Call Standard (see section A.4 below) is *full-descending*, meaning that the pointer points to the location in which the last item was stored (full) and the stack grows from higher to lower memory addresses (descending), that is, downwards.[10] Hence we will only need the `FD` addressing mode. Rn is the base registers, it contains the starting address of the block transfer. If the **!** is present (that is, the Write-back bit W is set), the base registers is updated after the transfer and points to the next relevant address. The register list Rlist specifies the subset of all registers (R0-R15) which should be transferred.

Most ARM assemblers as well as the disassembly listing received by `objdump -d` are supporting the following pseudo-instructions for better readability.

pseudo-instruction	actual instruction
PUSH [reg. list]	STMDB SP!, [reg. list]
POP [reg. list]	LDMIA SP!, [reg. list]

A.4 Procedure Call Standard (EABI)

The currently used ABI (Application Binary Interface) for ARM, *EABI2*[11,12] (published in 2005) defines standard conventions that object files respectively executable files have to follow. This section covers the most important part of ARM EABI2 in regards of register allocation, which is the calling conventions, named *Procedure Call Standard*[13](AAPCS[14]) in the ARM documents.

The base standard defines fixed roles for some of the core registers, as listed in table A.3. The pre-assigned registers R12 and R13 have the following meaning:

- **R12** = IP: may be used by a linker as a scratch register
- **R13** = SP: Stack Pointer

Note that the purpose of registers R14 and R15 are defined by hardware and were already discussed in section A.2 above. The definitive register assignment used for our code generator is described in chapter 7.

Registers R0-R3, R12 and R14 are *caller-saved* registers, i.e. they need to be saved by the caller prior to calling subroutines.

Registers R4-R11 and R13 are *callee-saved* registers, i.e. they need to be preserved by the subroutine. R4-R11 are also sometimes called *variable registers*.

Function parameters are passed in registers R0-R3. If there are more than four parameters, those are stored on the stack. Note that the stack must be at all times aligned to a word boundary ($SP \bmod 4 = 0$). On all external intefaces (that is basically, before functions are called) even

[10] other supported stack types are *full-ascending*, *empty-descending* and *empty-ascending*

[11] the 'E' is for *embedded*

[12] see `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0036b/index.html`

[13] see `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042e/IHI0042E_aapcs.pdf`

[14] note that *APCS* is a different standard which is obsolete

Register	Alternative Name	Description
R0		Argument 1 / Return value
R1		Argument 2 / Return value
R2		Argument 3
R3		Argument 4
R4-R11		Variables
R12	IP (Intra-Procedure-call scratch register)	
R13	SP (Stack Pointer)	
R14	LR (Link Register)	
R15	PC (Program Counter)	

Table A.3: AAPCS core register mapping

a double-word alignment of the stack pointer ($SP \bmod 8 = 0$) is required for EABI compliant code.

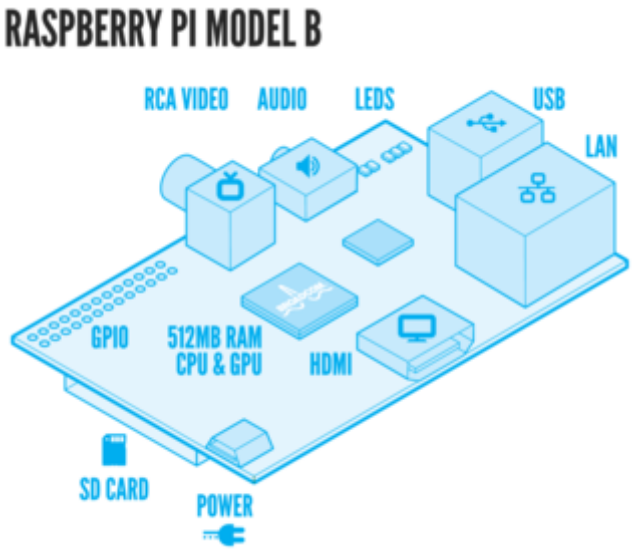
Return values of functions are stored in registers R0 and R1. Our code generator will only return values in register R0 – R1 would be used for data types larger than 32-bit which we don't allow (see limitations in chapter 4).

APPENDIX B

The Raspberry Pi

The *Raspberry Pi*[1] is a credit-card-sized single-board computer developed in the UK by the Raspberry Pi Foundation, a charity which was founded in 2009. Its goal was to promote the teaching of basic computer science in schools, as the declining skills of numbers and skills of applying students became a concern for the team of the University of Cambridge Computer Laboratory. With the existence of a tiny and affordable computer, children should get motivated for programming in an early age.

The following figure shows the layout of the Rasperry Pi with its components[2]:



[1] see `http://www.raspberrypi.org`

[2] Image source: `http://www.raspberrypi.org/help/faqs/`

The Raspberry Pi has the following specifications for the *Modell B*, [3] which was used for this thesis:

- CPU: ARM1176JZF-S (ARMv6 architecture), 700 MHz
- GPU: Broadcom VideoCore IV, 250 MHz
- RAM: 512 MB (shared with GPU)
- Onboard storage: SD/MMC/SDIO card slot
- 2 USB ports, 10/100 Ethernet port
- Composite RCA (PAL and NTSC) video output, HDMI output (resolutions up to 1920x1200)
- 3.5mm jack audio output
- Low-level peripherals: 8xGPIO, UART, I^2C bus, SPI bus
- Power ratings: 700 mA (3.5 W)

CPU, GPU and RAM are all part of the Broadcom BCM2835 system on a chip. As operating system, linux distributions are used, predominantly a Debian variant called “Raspbian”.

[3] *Modell A* is cheaper, but has only 256 MB RAM, no ethernet port and only one USB port

APPENDIX C

Implementation Code Snippets

C.1 IR Generation (`tccgen.c` / `tccir.c`)

Listing C.1 Assignment of virtual register to local variables

```
ST_FUNC Sym *sym_push(int v, CType *type, int r, int c)
{
    Sym *s;
    ...
    int vreg = -1;

    if (((r & VT_VALMASK) == VT_LOCAL) && (r & VT_LVAL) &&
        ((type->t & VT_BTYPE) != VT_STRUCT))
        vreg = IR_GetVregVar();

    ... /* after the other assignments to the "s" fields */
    s->vreg = vreg;
    ...
}
```

Listing C.2 IR code generation for binary operations

```
static void ir_gen_opi(int op)
{
    int irop;

    switch (op) { /* map token operation code to IR operation code */
    case '+':       irop = IR_OP_ADD; break;
    case '-':       irop = IR_OP_SUB; break;
    ...
    case TOK_SAR:   irop = IR_OP_SAR; break;

    /* ... */
    }

    SValue dest;
    memset(&dest, 0, sizeof(dest));
    dest.vreg = IR_GetVregTemp();
    dest.type.t = vtop[-1].type.t;
    dest.r = 0;

    IR_PutOp(irop, &vtop[-1], &vtop[0]);
    vtop[-1].vreg = dest.vreg;
    vtop[-1].r = 0;
    vtop--;
}
```

Listing C.3 IR code generation for relational operators

```
static void ir_gen_opi(int op)
{
    int irop;
    ...

    switch (op) { /* map token operation code to IR operation code */
    ...
    case TOK_EQ:  case TOK_NE:
    case TOK_LT:  case TOK_GT:  case TOK_LE:  case TOK_GE:
    case TOK_ULT: case TOK_UGT: case TOK_ULE: case TOK_UGE:
        IR_PutOp(IR_OP_CMP, &vtop[-1], &vtop[0], NULL);
        vtop--;
        vtop->r = VT_CMP;
        vtop->c.i = op;
        return;

    /* ... */
    }
    ...
}
```

Listing C.4 IR code generation for division and modulo operators

```
static void ir_gen_opi(int op)
{
    int irop;
    ...

    switch (op) { /* map token operation code to IR operation code */
    ...
    /* ARM platform specific: substitute / and % ops with function calls */
    case '/':
    case TOK_PDIV:
    case TOK_UDIV:
    case '%':
    case TOK_UMOD:
        memset(&num, 0, sizeof(num));
        num.vreg = -1;
        num.c.i = 1;
        IR_PutOp(IR_OP_FUNCPARAM, &vtop[-1], &num, NULL);
        num.c.i = 2;
        IR_PutOp(IR_OP_FUNCPARAM, &vtop[0], &num, NULL);
        if ((op == '/') || (op == TOK_PDIV))
            func = TOK___divsi3;
        else if (op == TOK_UDIV)
            func = TOK___udivsi3;
        else if (op == '%')
            func = TOK___aeabi_idivmod;
        else if (op == TOK_UMOD)
            func = TOK___aeabi_uidivmod;
        vpush_global_sym(&func_old_type, func);

        SValue dest;
        memset(&dest, 0, sizeof(dest));
        dest.vreg = IR_GetVregTemp();
        dest.type.t = vtop[-2].type.t;
        dest.r = 0;
        IR_PutOp(IR_OP_FUNCCALLVAL, &vtop[0], NULL, &dest);
        vtop--; vtop--; vtop--;
        vpushi(0);
        vtop->r = 0;
        vtop[0].vreg = dest.vreg;
        return;

    /* ... */
    }
    ...
}
```

Listing C.5 IR code generation for first four parameters in function calls

```
ST_FUNC void unary(void)
{
    ...
    /* post operations */
    while (1) {
        if (tok == TOK_INC || tok == TOK_DEC) {
            ....
        } else ...
        ...
        } else if (tok == '(') {
            ...
            if (tok != ')') {
                SValue num;
                memset(&num, 0, sizeof(num));
                num.vreg = -1;
                for(;;) {
                    expr_eq();
                    gfunc_param_typed(s, sa);
                    if (nb_args < 4) {
                        IR_PutOp(IR_OP_FUNCPARAM, &vtop[0], &num, NULL);
                        vtop--;
                    }
                    nb_args++;
                    if (sa)
                        sa = sa->next;
                    if (tok == ')')
                        break;
                    skip(',');
                }
            }
            if (sa)
                tcc_error("too few arguments to function");
            skip(')');
            if (!nocode_wanted) {
            }
        } else ...
    }
}
```

Listing C.6 IR code generation for parameters 5+ and function calls

```
ST_FUNC void unary(void)
{
    ...
    /* post operations */
    while (1) {
        if (tok == TOK_INC || tok == TOK_DEC) {
            ....
        } else ...
        ...
        } else if (tok == '(') {
            int ret_vreg = -1;
            ...
            if (!nocode_wanted) {
                int j;
                SValue num;
                memset(&num, 0, sizeof(num));
                num.vreg = -1;
                if (nb_args > 4) {
                    for (j=0; j<nb_args-4; j++) {
                        num.c.i = nb_args-j;
                        IR_PutOp(IR_OP_FUNCPARAM, &vtop[0], &num, NULL);
                        vtop--;
                    }
                }
                if (vtop[0].type.t == VT_VOID) {
                    IR_PutOp(IR_OP_FUNCCALLVOID, &vtop[0], NULL, NULL);
                } else {
                    SValue dest;
                    memset(&dest, 0, sizeof(dest));

                    dest.type.t = VT_INT;
                    dest.vreg = ret_vreg = IR_GetVregTemp();
                    IR_PutOp(IR_OP_FUNCCALLVAL, &vtop[0], NULL, &dest);
                }
                vtop--;
            } else {
                vtop -= (nb_args + 1);
            }
            /* return value */
            vsetc(&ret.type, ret.r, &ret.c);
            vtop->vreg = ret_vreg;
            }
        } else ...
    }
}
```

Listing C.7 Backpatching IR adaption example for the `while` statement (original code is in comments, replacement code is indented)

```
static void block(int *bsym, int *csym, int *case_sym, int *def_sym,
                  int case_reg, int is_expr)
{
    int a, b, c, d;
    ...
    if (tok == TOK_WHILE) {
        next();
        // d = ind; /* original */
            d = IR_GetPC();
        skip('(');
        gexpr();
        skip(')');
        // a = gtst(1, 0); /* original */
            a = ir_gtst(1, 0);
        b = 0;
        block(&a, &b, case_sym, def_sym, case_reg, 0);
        // gjmp_addr(d); /* original */
            SValue dest;
            memset(&dest, 0, sizeof(dest));
            dest.vreg = -1;
            dest.c.i = d;
            d = IR_PutOp(IR_OP_JMP, NULL, NULL, &dest);
        // gsym(a); /* original */
            IR_BackpatchToHere(a);
        // gsym_addr(b, d); /* original */
            IR_Backpatch(b, d);
    } else if (...)
    ...
}
```

Listing C.8 Backpatching functions for IR listing

```
void IR_Backpatch(int t, int target_addr) /* replacement for gsym_addr() */
{
    SValue *cur;
    while (t) {
      cur = &(IR[t].dest);
      t = cur->c.i; /* select next element in linked list */
      cur->c.i = target_addr; /* patch entry */
    }
}

void IR_BackpatchToHere(int t) /* replacement for gsym() */
{
    IR_Backpatch(t, irpc);
}
```

Listing C.9 Redundant move optimization

```
1  int IR_PutOp(int op, SValue *src1, SValue *src2, SValue *dest)
2  {
3      ...
4      /* optimization: eliminate redundant moves */
5      if ((op == IR_OP_ASS) && (IR_GetVregType(src1->vreg) == VREG_TEMPORARY) &&
6          (src1->vreg == IR[irpc-1].dest.vreg) &&
7          (src1->r & VT_LVAL) == 0) && (IR[irpc-1].op != IR_OP_FUNCCALLVAL))
8          IR[irpc-1].dest = IR[irpc].dest;
9      ....
10 }
```

C.2 Liveness Analysis (`tccir.c`)

Listing C.10 Determination of basic liveness intervals during IR code generation

```
typedef struct {
    int start;
    int end;
} VRegInterval;

/* the basic interval for vreg X can be accessed with BaseIntervals[X] */
VRegInterval BaseIntervals[MAX_VARS+MAX_TEMPS+MAX_PARAMS];

int irpc;

int IR_PutOp(int op, SValue *src1, SValue *src2, SValue *dest)
{
    ...
    /* destination operand */
    if (OP_USES_DEST(op)) {
        ...
        /* set basic interval start (first usage) */
        if (VREG_VALID(dest->vreg) && BaseIntervals[dest->vreg].start == 0)
            BaseIntervals[dest->vreg].start = irpc;
    }
    ...
    /* first source operand */
    if (OP_USES_SRC1(op)) {
        ...
        /* extend basic interval end (last usage) */
        if (VREG_VALID(src1->vreg))
            BaseIntervals[src1->vreg].end = irpc;
    }
    ...
    /* second source operand */
    if (OP_USES_SRC2(op)) {
        ...
        /* extend basic interval end (last usage) */
        if (VREG_VALID(src2->vreg))
            BaseIntervals[src2->vreg].end = irpc;
    }
    ...
}
```

Listing C.11 Implementation of the simple liveness analysis, sets interval for given *vreg*

```
1  static int FindLiveInterval(int vreg, unsigned int *start,
2      unsigned int *end, int check_for_backwards_jumps)
3  {
4      int i, retval=0;
5      Quadruple* quad;
6
7      /* lookup basic intervals calculated during IR code generation */
8      *start = BaseIntervals[vreg].start;
9      *end = BaseIntervals[vreg].end;
10     if (*start > 0 && *end > 0)
11         retval = 1;
12
13     if (!check_for_backwards_jumps)
14         return retval;
15
16     /* extend interval end if there is a later backwards jump
17        into the the base liveness interval */
18     i = *end;
19     quad = &IR[i];
20     for (;i<irpc;i++) {
21         if ((quad->op == IR_OP_JMP || quad->op == IR_OP_BRANCHIF) &&
22             (quad->dest.c.i <= *end) && (quad->dest.c.i > *start)) {
23             *end = i;
24         }
25         quad++;
26     }
27     ...
28     return retval;
29 }
```

Listing C.12 Solution for overcoming the simple liveness analysis restriction

```
1  static int FindLiveInterval(int vreg, unsigned int *start,
2      unsigned int *end, int check_for_backwards_jumps)
3  {
4      ...
5      /* special treatment for first definitions within conditional:
6         extend interval to outermost loop */
7      if (BaseIntervals[vreg].start_within_if) {
8          quad = &IR[irpc-1];
9          for (i=irpc-1; i>*start; i--) {
10             if ((quad->op == IR_OP_JMP || quad->op == IR_OP_BRANCHIF) &&
11                 (quad->dest.c.i < *start)) {
12                 *start = quad->dest.c.i;
13                 if (i > *end) *end = i;
14                 break;
15             }
16             quad--;
17         }
18         for (i=*start; i<*end; i++) {
19             quad = &IR[i];
20             if ((quad->op == IR_OP_JMP || quad->op == IR_OP_BRANCHIF) &&
21                 (quad->dest.c.i < *start)) {
22                 *start = quad->dest.c.i;
23                 break;
24             }
25         }
26     }
27     ...
28 }
```

C.3 Linear Scan Register Allocation (`tccls.c`)

Listing C.13 Register pool functions

```
1  int register_pool[K];
2
3  int RegisterPool_Get()
4  {
5      int i;
6      for (i=0; i<K; i++) {
7          if (register_pool[i]) {
8              register_pool = 0; /* remove register from pool */
9              return i;
10         }
11     }
12     tcc_error("no register available for allocator, should never happen!");
13 }
14
15 void RegisterPool_Add(int reg)
16 {
17     register_pool[reg] = 1;
18 }
```

Listing C.14 Expiring intervals that are not relevant any more (part of the LSRA algorithm)

```
1  static void ExpireOldIntervals(int i)
2  {
3      int j, removed_intervals=0;
4      static LiveInterval dirty = {0, 0, 0, 0, ~0};
5
6      /* "foreach interval j in active, in order of increasing end point" */
7      for (j=0; j<nActive; j++) {
8          if (ActiveSet[j]->end >= LiveIntervals[i].start)
9              break;
10         RegisterPool_Add(ActiveSet[j]->reg);
11         ActiveSet[j] = &dirty; /* mark as dirty for deletion through sort */
12         removed_intervals++;
13     }
14
15     /* remove entries from active set */
16     qsort(&ActiveSet[0], nActive, sizeof(ActiveSet[0]), licomp_endpoint);
17     nActive -= removed_intervals;
18 }
```

Listing C.15 Spilling a certain interval (part of the LSRA algorithm)

```
static void SpillAtInterval(int i)
{
    LiveInterval* Spill = ActiveSet[nActive-1]; /* "last interval in active" */
    if (Spill->end > LiveIntervals[i].end) {
        LiveIntervals[i].reg = Spill->reg;
        Spill->loc = NewStackLocation();

        ActiveSet[nActive-1] = &LiveIntervals[i];
        /* sort by increasing end point */
        qsort(&ActiveSet[0], nActive, sizeof(ActiveSet[0]), licomp_endpoint);
    } else {
        LiveIntervals[i].loc = NewStackLocation();
    }
}
```

Listing C.16 The core function of the LSRA algorithm

```
void LS_RegisterAllocation()
{
    int i;
    for (i=0; i<R; i++)
        register_pool[i] = 1; /* initially, all registers are free */

    nActive = 0; /* empty active set */

    /* sort live intervals in order of increasing starting point */
    qsort(&LiveIntervals[0], nIntervals,
          sizeof(LiveIntervals[0]), licomp_startpoint);

    /* "foreach live interval, in order of increasing start point" */
    for (i=0; i<nIntervals; i++) {
        ExpireOldIntervals(i);
        if (nActive == R) {
            SpillAtInterval(i);
        } else {
            LiveIntervals[i].reg = RegisterPool_Get();
            ActiveSet[nActive++] = &LiveIntervals[i];
            /* sort by increasing end point */
            qsort(&ActiveSet[0], nActive,
                  sizeof(ActiveSet[0]), licomp_endpoint);
        }
    }

    ...
}
```

C.4 Code Generation (`tccir.c`)

Listing C.17 Generation of Prologue and Epilogue code

```
1  static void GenProlog() {
2    int save_reg_list; /* criterion for optimizing fp and sp away */
3    int ignore_fp_sp = (leaffunc && num_params <= 4 && loc == 0);
4    save_reg_list = ((1 << regs_used)-1) << 4; /* save regs r4-r10 if needed */
5    save_reg_list |= 0x5800; /* save r11=fp, r12=ip (sp), r14=lr (ret. addr.) */
6    if (ignore_fp_sp) {
7       save_reg_list &= ~((1 << 11) | (1 << 12));
8    } else
9      o(0xE1A0C00D); /* mov ip, sp */
10   if (num_params > 0 && (!leaffunc || num_params > 4)) {
11       o(0xE92D0000 | param_reg_list); /* push arguments */
12       o(0xE24DD000 | ((7-regs_used)*4)); /* adapt stack pointer */
13   }
14   o(0xE92D0000 | save_reg_list); /* context save */
15   if (!ignore_fp_sp)
16     o(0xE1A0B00D); /* mov fp, sp */
17   if (loc) {
18       int diff = (-loc + 3) & -4; /* align to multiple of 4 */
19       if (diff > 255) {
20           GenConstIntoReg(12, diff);
21           o(0xE04DD00C); /* substract with ip */
22       } else
23           o(0xE24DD000 | diff); /* stack adjustment; */
24   }
25 }
26
27 static void GenEpilog() {
28   int restore_reg_list; /* criterion for optimizing fp and sp away */
29   int ignore_fp_sp = (leaffunc && num_params <= 4 && loc == 0);
30   restore_reg_list = ((1 << regs_used)-1) << 4; /* rest. regs r4-r10 if needed */
31   restore_reg_list |= 0xA800; /* restore r11=fp, r13=sp, r15=pc */
32   if (ignore_fp_sp) {
33     restore_reg_list &= ~((1 << 11) | (1 << 13));
34   }
35
36   if (!ignore_fp_sp)
37     o(0xE89B0000 | restore_reg_list); /* context restore */
38   else
39     o(0xE8BD0000 | restore_reg_list); /* if fp and sp ignored, directly pop! */
40 }
```

Listing C.18 Generation of Data-Processing Instructions

```
static void GenDataProcessingOp(int op, int dest_reg, int src1_reg,
    int src2_reg, int second_const, int cval)
{
    static const int DataProcessingOpMap[] =
      /* add, sub, and, or, xor, [shl, shr, sar], mov, rsb, cmp, teq */
      { 4, 2, 0, 12, 1, 13,13,13,13, 3, 10, 9 };
    uint32_t opc = 0xE0000000; /* unconditional execution */
    int dpopc = DataProcessingOpMap[op]; /* data-processing opcode */

    if ((op == IR_OP_CMP) || (op == IR_OP_TSTZ))
      opc |= (1<<20); /* set condition codes */
    if (op == IR_OP_ASS) { /* source operand for mov must be in op2 field */
      src2_reg = src1_reg;
      src1_reg = 0;
    }

    opc |= (dpopc << 21); /* set data-processing opcode [I=0, S=0] */
    opc |= (dest_reg << 12); /* set destination register */

    if (op == IR_OP_SHL || op == IR_OP_SHR || op == IR_OP_SAR) { /* shifts */
        opc |= src1_reg; /* set source register */
        if (op == IR_OP_SHR) opc |= 1<<5;
        if (op == IR_OP_SAR) opc |= 1<<6;
        if (!second_const)
          opc |= 1<<4 | (src2_reg << 8);
        else
          opc |= (cval & 0x1f) << 7;
    } else {
      opc |= (src1_reg << 16); /* set first source register (0 for mov) */
      if (!second_const)
        opc |= src2_reg; /* set second source register */
      else {
        if (cval < 0) { /* negative constants */
            if (op == IR_OP_ADD || op == IR_OP_SUB) {
              opc ^= 0x00C00000; cval = -cval;
            } else if (op == IR_OP_ASS) {
              opc ^= 0x00400000; cval = ~cval;
            }
        }
        if (cval > 255 || cval < 0) {
          GenConstIntoReg(14, cval);
          opc |= 14;
        } else {
          opc |= 1<<25; /* set I bit */
          opc |= cval & 0xff;
        }
      }
   }
   o(opc);
}
```

Listing C.19 Branch instruction patching

```
...
#define MAX_QUADRUPLES 10000
...
uint32_t IR_addrs[MAX_QUADRUPLES];
...
void IR_GenCode()
{
  /* first pass: generate code */
  for (i=0; i<irpc; i++) {
    IR_addrs[i] = ind;
    ...
  }

  ...
  /* second pass: patch branch instructions */
  for (i=0; i<irpc; i++) {
    op = IR[i].op;
    dest = &(IR[i].dest);
    if ((op == IR_OP_BRANCHIF) || (op == IR_OP_JMP)) {
      int instr_addr = IR_addrs[i];
      int target_addr = IR_addrs[dest->c.i];
      int offset = (target_addr-instr_addr-8)/4;
      uint32_t *instr = &(cur_text_section->data[instr_addr]);
      if ((offset >= 0x1000000) || (offset < -0x1000000))
        tcc_error("branch offset too large (>=16MB)");
      offset &= 0x00ffffff;
      *instr |= offset;
    }
  }
}
```

Listing C.20 Function calls, parameter handling

```
...
static int call_level; /* -1=not in any function call phase */
int num_params_per_level[50];
...

static void FuncCallParam(int param_num, int src_reg, int isconst, int cval)
{
    /* parameters 5+ are supplied on the stack */
    if (param_num > 4) {
        if (isconst) {
            GenConstIntoReg(src_reg=14, cval);
        }
        o(0xE52D0004|(src_reg<<12)); /* str r,[sp,#-4]! */
        return;
    }

    /* new function call phase starts */
    if (param_num == 1) {
        call_level++;
        /* for nested function calls, save arg regs (r0-r3) in-between */
        if (call_level >= 1) {
            int args_to_save = num_params_per_level[call_level-1];
            if (args_to_save > 4)
                args_to_save = 4;
            o(0xE92D0000 | ((1<<args_to_save)-1)); /*P=1, U=0, W=1, L=0*/
        }
    /* special case: void function calls with return value, save r0 */
    } else if (param_num == 0) {
        call_level++;
        if (call_level >= 1)
            o(0xE92D0001);
        num_params_per_level[call_level] = 1;
        return;
    }

    GenDataProcessingOp(IR_OP_ASS, param_num-1, src_reg, -1, isconst, cval);
    num_params_per_level[call_level] = param_num;
}

static void FuncCallAfter()
{
    /* restore arg regs (r0-r3) if call was nested */
    if (call_level >= 1) {
        int args_to_restore = num_params_per_level[call_level-1];
        o(0xE8BD0000 | ((1<<args_to_restore)-1)); /*P=0, U=1, W=1, L=1*/
    }

    call_level--;
}
```

Bibliography

[AC71] F.E. Allen and J. Cocke. A Catalogue of Optimizing Transformations. In R. Rustin, editor, *Design and Optimization of Compilers*, pages 1–30. Prentice-Hall, 1971.

[AG01] Andrew W. Appel and Lal George. Optimal Spilling for CISC Machines with Few Registers. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 243–253. ACM, New York, NY, USA, 2001. ISBN 1-58113-414-2.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[BCD+92] D. S. Blickstein, P. W. Craig, C. S. Davidson, R. N. Faiman, K. D. Glossop, R. P. Grove, S. O. Hobbs, and W. B. Noyce. The GEM Optimizing Compiler System. *Digital Equipment Corporation Technical Journal*, 4(4):121–135, 1992.

[BGP00] L. Böszörményi, J. Gutknecht, and G. Pomberger. *The School of Niklaus Wirth: The Art of Simplicity*. Elsevier Science & Technology Books, 2000. ISBN 9781558607231.

[Bri92] Preston Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, Houston, TX, USA, 1992. UMI Order No. GAX92-34388.

[CAC+81] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register Allocation via Coloring. *Comput. Lang.*, 6(1):47–57, 1981.

[Cha82] Gregory J. Chaitin. Register Allocation & Spilling via Graph Coloring. In *SIGPLAN Symposium on Compiler Construction*, pages 98–105, 1982.

[GA96] Lal George and Andrew W. Appel. Iterated Register Coalescing. *ACM Transactions on Programming Languages and Systems*, 18:300–324, 1996.

[GH07] Daniel Grund and Sebastian Hack. A Fast Cutting-Plane Algorithm for Optimal Coalescing. In Shriram Krishnamurthi and Martin Odersky, editors, *Compiler Construction 2007*, volume 4420 of *Lecture Notes In Computer Science*, pages 111–125. Springer, March 2007.

[GRE+01] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, pages 3–14. IEEE Computer Society, Washington, DC, USA, 2001.

[GW96] David W. Goodwin and Kent D. Wilken. Optimal and Near-Optimal Global Register Allocations Using 0–1 Integer Programming. *Softw. Pract. Exper.*, 26(8):929–965, 1996. ISSN 0038-0644.

[HG06] Sebastian Hack and Gerhard Goos. Optimal Register Allocation for SSA-form Programs in polynomial Time. *Information Processing Letters*, 98(4):150–155, May 2006.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, 1990.

[KF96] Steven M. Kurlander and Charles N. Fischer. Minimum Cost Interprocedural Register Allocation. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 230–241. ACM Press, 1996.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988. ISBN 0131103709.

[LCH+80] B. W. Leverett, R. G. Cattell, S. O. Hobbs, J. M. Newcomer, A. H. Reiner, B. R. Schatz, and W. A. Wulf. An Overview of the Production-Quality Compiler-Compiler Project. *IEEE Computer*, 13(8):38–49, 1980.

[PEK97] Massimiliano Poletto, Dawson R. Engler, and M. Frans Kaashoek. tcc: A System for Fast, Flexible, and High-level Dynamic Code Generation. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, pages 109–121. ACM, New York, NY, USA, 1997. ISBN 0-89791-907-6.

[Per08] Fernando Magno Quintão Pereira. A Survey on Register Allocation. `http://compilers.cs.ucla.edu/fernando/publications/drafts/survey.pdf`, 2008.

[PP05] Fernando Magno Quintão Pereira and Jens Palsberg. Register Allocation via Coloring of Chordal Graphs. In *Asian Symposium on Programming Languages and Systems (APLAS'05)*, pages 315–329, 2005.

[PP06] Fernando Magno Quintão Pereira and Jens Palsberg. Register Allocation After Classical SSA Elimination is NP-Complete. In *Foundations of Software Science and Computation Structures (FoSSaCS 2006)*, volume 3921 of *Lecture Notes in Computer Science*, pages 79–93. Springer, 2006. ISBN 3-540-33045-3.

[Pro09] Jonathan Protzenko. A Survey of Register Allocation. `http://xulforum.org/files/reportra.pdf`, 2009.

[PS99] Massimiliano Poletto and Vivek Sarkar. Linear Scan Register Allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, 1999.

[QaPP08] Fernando Magno Quintão Pereira and Jens Palsberg. Register Allocation by Puzzle Solving. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 216–226. ACM, New York, NY, USA, 2008. ISBN 978-1-59593-860-2.

[SB07] Vivek Sarkar and Rajkishore Barik. Extended Linear Scan: An Alternate Foundation for Global Register Allocation. In Shriram Krishnamurthi and Martin Odersky, editors, *CC*, volume 4420 of *Lecture Notes in Computer Science*, pages 141–155. Springer, 2007. ISBN 978-3-540-71228-2.

[SE02] Bernhard Scholz and Erik Eckstein. Register Allocation for Irregular Architectures. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES/SCOPES '02, pages 139–148. ACM, New York, NY, USA, 2002. ISBN 1-58113-527-0.

[Sea00] David Seal. *ARM Architecture Reference Manual*. Addison-Wesley, London, UK, 2000.

[SP82] Carlo H. Séquin and David A. Patterson. Design and Implementation of RISC i. Technical Report UCB/CSD-82-106, EECS Department, University of California, Berkeley, Oct 1982. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/1982/5449.html`.

[THS98] Omri Traub, Glenn Holloway, and Michael D. Smith. Quality and Speed in Linear-Scan Register Allocation. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 142–151. ACM, 1998.

[WF10] Christian Wimmer and Michael Franz. Linear Scan Register Allocation on SSA Form. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 170–179. ACM, 2010.

[WG92] John Wood and Harold C. Grossman. Interprocedural Register Allocation for RISC Machines. In *Proceedings of the 30th Annual Southeast Regional Conference*, pages 188–195. ACM, New York, NY, USA, 1992. ISBN 0-89791-506-2.

[Wir96] Niklaus Wirth. *Compiler Construction*. International Computer Science Series. Addison-Wesley Publishing Company, 1996. ISBN 9780201403534.

[WJW+75] W. A. Wulf, R. K. Johnsson, C. B. Weinstock, S. O. Hobbs, and C. M. Geschke. *The Design of an Optimizing Compiler*. Elsevier Science Inc., New York, NY, USA, 1975. ISBN 0444001646.