Karlsruher Institut für Technologie

Institut für Programmstrukturen
und Datenorganisation (IPD)

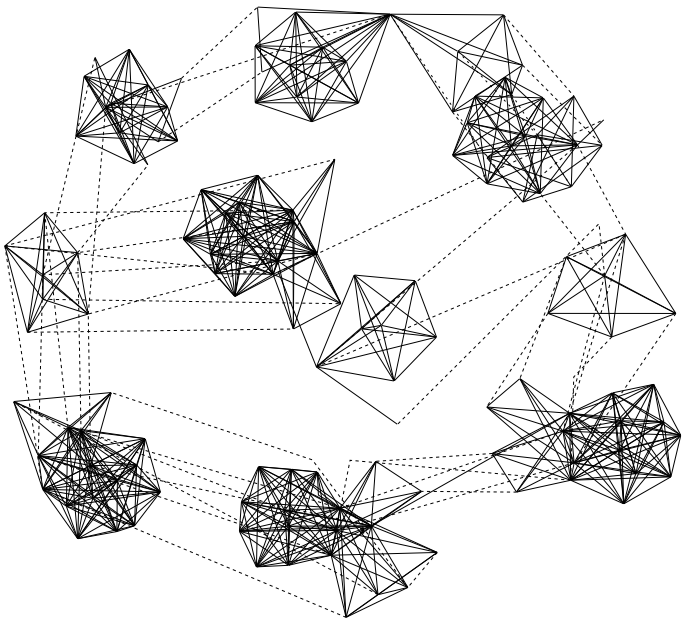
Lehrstuhl Prof. Dr.-Ing. Snelting

SSA-based Register Allocation for Compressed Machine Code

Masterarbeit von

Maximilian Stemmer-Grabow

an der Fakultät für Informatik



Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. rer. nat. Bernhard Beckert
Betreuender Mitarbeiter:	M. Sc. Andreas Fried
Abgabedatum:	20. Dezember 2021

Abstract

Many RISC instruction sets include compressed instruction variants to improve code density. Due to limited encoding space, these compressed variants are restricted, often in terms of the registers they can be used with. This means that explicitly modeling these restrictions during register allocation can be leveraged to improve compression of the resulting code. In this thesis, we present an extension to the SSA-based register allocator used in the LibFirm framework that enables compression-aware register allocation. It is an opt-in optimization that can be enabled for backends by specifying compression requirements for instructions. It supports specifying a limited subset of *compressible* registers to be used in compressed instructions, 2-address requirements for instructions, and combinations of these requirements. We implement the optimization for LibFirm's RISC-V backend and evaluate the resulting code size improvements: Using the optimization, we can observe an overall aggregated reduction of text segment sizes in compiled object files of around 4.2 % or 5.7 % in Embench or SPEC CINT2000 benchmark suites, respectively. At the same time, the overall number of generated instructions does not increase by more than 0.2 % for any of the benchmarks from SPEC CINT2000. This indicates that the optimization does not otherwise negatively affect the performance of the generated code.

Zusammenfassung

Viele RISC-Architekturen enthalten zur Verringerung der Codegröße komprimierte Instruktionsvarianten. Diese sind in ihrer Funktionalität beschränkt, was häufig die Auswahl von Registern für Operanden und Ergebnisse betrifft. Deswegen kann die Codegröße dieser Architekturen verringert werden, indem diese Beschränkungen auch im Compiler beachtet werden. Diese Arbeit enthält eine Erweiterung des auf SSA aufbauenden Registerallokators des LibFirm-Frameworks, die Code-Kompression bei der Registerallokation explizit modelliert. Die Optimierung kann für Backends in LibFirm genutzt werden, indem für alle Instruktionen die Bedingungen spezifiziert werden, unter denen sie komprimierbar sind. Dazu können für komprimierte Instruktionen Beschränkungen auf eine Teilmenge von *komprimierbaren* Registern und 2-Adress-Bedingungen angegeben werden, sowie Kombinationen davon. Die Optimierung wurde für das RISC-V-Backend in LibFirm implementiert und die Verbesserung der Codegröße untersucht: Mit der Optimierung lässt sich die Größe von Textsegmenten in Objektdateien im Mittel um etwa 4,2 % (für Embench) bzw. 5,7 % (für SPEC CINT2000) verringern. Gleichzeitig steigt die Gesamtzahl von generierten Instruktionen bei keinem der SPEC-Benchmarks um mehr als 0,2 % an, die Optimierung verschlechtert also nicht die Performance des generierten Codes.

Contents

1 Introduction

Apart from their semantic correctness, programs generated by optimizing compilers can be evaluated according to different metrics: These include performance when running the code or the overall size of the resulting executables. Optimizing for small code size is especially important when space is limited, for example when dealing with embedded applications, but can also in turn affect code performance. The associated metric is called *code density*, which describes the required code size of a program which performs a specific function – a higher code density implying a smaller footprint for the executable.

Architectures based on the RISC philosophy (short for *reduced instruction set computer*) are successful due to being easier to specify and implement in hardware, but their trade-offs often negatively affect code density of generated code. This difference in code size between traditional fixed-length RISC and CISC code has previously been surveyed and placed at around 25 % on average [1]. Uniform instruction lengths common in RISC instruction sets simplify the architecture, but also reduce flexibility when it comes to the amount of information encoded in specific instructions. RISC architectures are also often designed as load-store architectures, separating instructions accessing memory from those operating on data in registers. This lowers the required number and complexity of different instruction types, but may also increase the number of instructions in a given program when compared to typical CISC architectures.

To counteract problems with lower code density, several architectures have introduced the notion of *compressed instructions*, instruction variants that have shorter encodings than the standard instruction length. Due to the limited encoding space available for these shortened instructions, they can only cover a part of the original instruction set: Not all instructions can be included and there need to be restrictions on the information that is encoded in the instructions themselves. Often, this is achieved by requiring operands or results to be in a certain subset of all available registers, or an operand and the result being placed in the same register (i.e., conforming to a *2-address instruction format*).

The RISC-V architecture this thesis is focused on includes compressed instructions in its C extension (also abbreviated RVC). They are also restricted in this way: For example, RISC-V defines a `sub` instruction which subtracts the contents of two registers: `sub rd, rs1, rs2` places `rs1` − `rs2` into the register `rd`. In the base instruction set, either operand can be chosen from any of the 32 general purpose registers available in RISC-V, and the result can be placed in any register. The compressed `sub` instruction variant requires the result to overwrite the first operand (`rs1` and `rd` have to be identical). Additionally, all involved registers have to be in

the subset of eight *compressible* registers instead of the full register set. Both of these requirements need to hold in order to be able to choose the compressed instruction variant. A more detailed description of the RISC-V Compressed extension will be presented later in section 2.2.

For cases such as RVC where the requirements for compressed instructions are related to the register set, this introduces a dependency on the result of *register allocation* in the compiler: Whether an instruction can be expressed in a compressed format depends on which values in the program are placed in which physical register.

In general, operand placement in registers is often somewhat restricted, especially in CISC architectures. For example, architectures may require that operands are placed in certain registers in order for them to be used with a specific instruction. Of course, these restrictions are *mandatory*: They have to be fulfilled in order to generate correct code for the architecture. For this reason, this kind of restriction is already modeled in compilers and compilers are aware of it during register allocation.

Restrictions related to compression we are concerned with in this thesis are different, however: Compression-related requirements are optional and not required to generate correct code. In cases where the compression restrictions cannot be fulfilled, the uncompressed instruction encoding can be chosen instead, the penalty being the larger code size that comes with using the uncompressed instruction. This makes trying to fulfill compression-related restrictions a trade-off: As many of them as possible should be fulfilled, but without introducing more instructions overall.

However, modern compilers usually do not directly model and optimize for compression requirements when performing register allocation. This thesis aims to improve code density by explicitly making the compiler aware of these requirements. To do this, we focus on the compressed instruction extension for RISC-V and modify the register allocation step which is part of LibFirm, a graph-based intermediate representation library. It includes backends for multiple instruction sets, RISC-V being one of them.

LibFirm's register allocation includes a step called *copy elimination* (or *copy optimization*). It employs a heuristic to try to find a valid register allocation that also minimizes the number of times values need to be copied (or moved) between different registers when running the program. To simultaneously maximize the number of instructions that are compressed in the final compiled program, we extend this step to account for the compression requirements of specific instructions. This also requires a way to express which instructions can be compressed (and under which conditions) in the architecture-specific backends for LibFirm. In the case of the `sub` operation, the backend can provide the information that there is a corresponding compressed instruction which requires the first operand to be in the same register as the result as well as all involved registers to be in the compressible register subset. The copy elimination algorithm can incorporate this into the placement of the operands and result of the instruction. This can then be applied for every instruction that may be compressed if similar requirements hold. The most important design goals for this extension include:

- The quality of the register allocation should not be negatively affected. Register allocation has a large impact on the performance of the generated code: Its result determines the number of times data has to be copied between registers or registers and memory when running the code. This not only affects runtime performance, but also the resulting code size (via the number of move instructions between registers remaining in the program code).

- The extension should also keep the additional complexity introduced into the register allocation algorithm itself to a minimum: Improved compression is desirable, but should not come at the cost of considerably longer running time of the register allocator. It also should not include changes to the allocator that make it much harder to understand and reason about.

We evaluate the results of the optimization by comparing resulting code sizes in benchmarks using cparser, a C compiler which uses LibFirm for intermediate code representation, optimization, and code generation.

The thesis is structured in the following way: Chapter 2 contains an introduction to the context of and the concepts required in the thesis. This includes compressed instruction sets as well the LibFirm infrastructure and its theoretical foundations. Chapter 3 provides an overview over other works relevant in this context: Works dealing with register allocation confined by restrictions present in many computing architectures. It also surveys other works that deal with compiling code for architectures that support compressed machine code. Contributions of the thesis are structured in this order: An overview over the design of the compression optimization as well as how it is implemented can be found in chapter 4. Measurements, comparisons, and results are presented in chapter 5. Chapter 6 outlines starting points for possible future work in the context examined here and concludes the thesis.

2 Foundations

This chapter serves as an introduction to the basic concepts and ideas that are relevant to this thesis. We will first take a look at the rationale and design of compressed instruction sets as well as some specific examples in section 2.1, followed by a more in-depth look at the compressed instruction extension for RISC-V in section 2.2. A brief introduction to the LibFirm approach and its architecture is contained in section 2.3, especially concerning where and how register allocation is integrated there. LibFirm's register allocation approach and especially its copy optimization step is covered in section 2.4.

2.1 Compressed ISAs

As mentioned in the previous chapter, the concept of compressed instructions is related to optimizing *code density*. A higher code density corresponds to a smaller total size of the instructions required for a particular program when compiled for a specific architecture. Especially for RISC architectures with uniform instruction lengths, this is a challenge: Different types of instructions typically also have different requirements for the amount of data that is useful to include in them, e.g. depending on the number of operands or the desirable length of immediate values. Many RISC architectures are also designed as load-store architectures: These isolate instructions for transfer of values between memory and registers from instructions operating on them. As an example, this means that as arithmetic instructions only operate on values in registers, the additional required load and store instructions often lead to overall higher instruction counts. In contrast, CISC architectures are often designed with variable-length instruction encodings.

For example, let us examine the code generated for a simple function that computes the maximum of two operands with one of them being scaled beforehand:

```
int max42(int a, int b) { return (a * 42 > b) ? a : b; }
```

Figure 2.1 shows code generated for RISC-V (with 32-bit instruction length and without compressed instructions) and x86. In this case, even though both code versions contain five instructions, there is a difference in code size: Due to the variable-length encoding of x86 instructions, they only require 11 bytes while the RISC-V instructions take up 20 bytes.

As a matter of course, this difference between code sizes varies a lot across different code samples and architectures. Overall, in a survey of code densities of different

```
max42:
  02a00793  li     a5, 42
  02f507b3  mul    a5, a0, a5
  00f5c463  blt    a1, a5, <max42+0x10>
  00058513  mv     a0, a1
  00008067  ret
```

(a) RISC-V

```
_max42:
  89 d0     mov    %edx, %eax
  6b d1 2a  imul   $0x2a, %ecx, %edx
  39 c2     cmp    %eax, %edx
  0f 4f c1  cmovg  %ecx, %eax
  c3        ret
```

(b) x86

Figure 2.1: Comparison between RISC-V and x86 code generated for the function `int max42(int a, int b) { (return a * 42 > b) ? a : b; }`.

instruction sets, Weaver and McKee found code sizes for fixed-length RISC instruction sets to be an average of about 25% larger than those for CISC instruction sets [1].

Compressed instructions aim to reduce this difference by introducing variable-length encodings: There are multiple RISC instruction sets that include the notion of compressed instructions, including RISC-V. These may be included as variants of or as extensions to the original fixed-length instruction sets. In our example above, using the compression extension for RISC-V, two of the five instructions could be replaced by compressed instructions. These only take up 16 bits each and in this case bring down the code size from 20 to 16 bytes. When evaluating instruction sets that support compression or a specific group of instructions, we use the term *compressed instruction rate* to describe the share of instructions that are compressed (in the example above, it is 40%).

Let us review and compare the approaches for enabling compression in some architectures most relevant in the context of this thesis:

ARM and Thumb Thumb is an instruction set based on the ARM instruction set [2], which was introduced to achieve higher code density when compared to ARM code by utilizing instructions with a length of 16 instead of 32 bits. Most of these shortened instructions are closely related to ARM instructions, but not all of them are directly expandable into uncompressed instructions. Not all instructions in the ARM instruction set are available as shortened instructions in Thumb, and available instructions are constrained: For example, of the 16 general purpose registers available with ARM, only eight are available as general purpose registers with Thumb. Three

additional registers have special functions with certain instructions, but are not arbitrarily accessible. Processors can support both Thumb and ARM instruction sets simultaneously: In this case, explicit mode switch instructions can be used to change between ARM and Thumb execution modes. Other works have investigated how compression can be optimized for this kind of explicitly mode-switching architecture (more details can be found in section 3.2).

Instructions in the original Thumb specification are not designed to provide the full functionality from the ARM instruction set. To alleviate this, Thumb-2 (2003) [2] extends Thumb with new instructions, especially in 32-bit encodings: These aim to reintroduce most functionality from the ARM instruction set missing from Thumb, but are separate instructions not included in ARM. This effectively converts Thumb into a mixed-length instruction set which aims to cover similar functionality as the separate, fixed-length 32-bit ARM instruction set. This also removes the need for changing execution modes in between instructions by allowing 16- and 32-bit encoded instructions to be directly intermixed. To be able to target both Thumb and ARM, the Thumb-2 extension also includes an assembly syntax that can be compiled to both variants, called *Unified Assembly Language* (UAL for short). Note that the Thumb-2 extension replaced the original instruction set and is now only referred to as Thumb in its specification.

MIPS MIPS16e and its successor MIPS16e2 are extensions to the MIPS32 and MIPS64 instruction sets [3] which include 16-bit instructions. Processors implementing them have an execution mode that has to be explicitly set: A mode switch instruction needs to be executed before any of these additional 16-bit instructions. For most instructions in the 16-bit execution mode, only a subset of eight registers is available instead of the full 32 registers available with 32-bit instructions. Only compressed move instructions support access to all registers.

There is also a more recent addition which removes the need for explicit execution mode switching: microMIPS [4] is an extended ISA variant based on MIPS32/64 which includes additional instructions with 16-bit length, similar to those defined in MIPS16e(2). microMIPS is a superset of MIPS32/64 and hence also allows the longer, uncompressed instructions to be used. As it does not require an explicit mode switch, instructions can be freely intermixed between microMIPS' additional 16-bit and regular 32-bit MIPS instructions. Similarly to MIPS16e(2), most instructions are restricted to access a subset of eight registers (registers 2-7, 16, and 17 in MIPS register encoding) [4].

As microMIPS has been introduced to replace MIPS16e(2), it is now the preferred way of implementing compressed instructions for MIPS. MIPS16e has been removed from the MIPS specification in its Release 6 [4, p. 13].

RISC-V RISC-V [5] is designed as a modular instruction set specification: Its full functionality is split across multiple extensions that add instructions to the base instruction set. Compressed instructions are contained in extension C, short for

Table 2.1: RISC-V C instruction formats, taken from the RISC-V specification [5].

Format	Meaning	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CR	Register	funct4				rd/rs1					rs2					op	
CI	Immediate	funct3			imm	rd/rs1					imm					op	
CSS	Stack-relative Store	funct3			imm						rs2					op	
CIW	Wide Immediate	funct3			imm								rd′			op	
CL	Load	funct3			imm			rs1′			imm		rd′			op	
CS	Store	funct3			imm			rs1′			imm		rs2′			op	
CA	Arithmetic	funct6						rd′/rs1′			funct2		rs2′			op	
CB	Branch/Arithmetic	funct3			offset			rd′/rs1′			offset					op	
CJ	Jump	funct3			jump target											op	

Compressed. The extension was suggested by Waterman in 2011, explicitly to improve energy efficiency for execution when compared to the fixed-length RISC-V instruction set [6]. It defines compressed instructions which are freely intermixable with uncompressed instructions from the base instruction set or other extensions. We will now take a more in-depth look at the design of the extension in the following section.

2.2 RISC-V Compressed Instruction Set Extension

The RISC-V specification [5] defines three base instruction sets with different register widths: 32-bit (RV32), 64-bit (RV64), and 128-bit (RV128). The C extension (also called RVC) can be combined with any of these instruction sets. The extension defines instructions with a length of 16 bits that can all be expanded into corresponding uncompressed instructions which are semantically equivalent. Instructions with compressed variants are contained in the base instruction set or in the F and D extensions (which provide hardware floating-point handling). The standard provides nine different instruction layout types in this 16-bit format, shown in table 2.1. The encodings are chosen from these formats according to the required fields in each specific compressed instruction. During execution, compressed instructions can be expanded into their long-format equivalent by the processor in the decoding step. The opcode encoding space for compressed instructions is reserved and not used in uncompressed base instructions sets or other extensions. On machines which support the extension, RVC allows compressed and uncompressed instructions to be freely intermixed without the use of a flag or special instruction to toggle instruction modes. RVC also relaxes the alignment requirements for uncompressed instructions, which precludes the need for padding when switching between compressed and uncompressed instructions (as such padding would negatively affect code size).

The fact that compressed instructions correspond directly to uncompressed instructions and can be intermixed as described make RVC a good candidate for a prototype implementation of explicitly incorporating compression into register allocation. This is why we chose it as the focus for this thesis.

RISC-V provides the following workflow for creating code that includes compressed instructions: Explicit mnemonics for compressed instructions with the prefix `c` do exist as part of RISC-V assembly. However, these are not required to generate compressed code when using the assembler that is part of the RISC-V toolchain. As any compressed instruction has an uncompressed counterpart, the assembler can select compressed variants for those instructions which comply with all compression requirements, even if they are not explicitly indicated as compressible in the assembly code.

This allows a form of *implicit* compiler support: The compiler does not need to be aware of compressed instructions or their requirements if it emits assembly code that is passed to the RISC-V assembler. Instructions which satisfy all requirements are then transformed into their compressed versions. LibFirm uses this fact in its RISC-V backend and emits the same assembly code for platforms with or without support for the C extension, passing the selection of compressed instructions on to the assembler. However, as compared to this "incidental" compression, explicit compiler support can increase the number of overall instructions that satisfy the requirements for compressed instructions, hence reducing code size further.

We will now take a closer look at the specific compression requirements included in RVC. Evaluating whether a single instruction can be compressed means evaluating each compression requirements associated with it. Not all types of instructions have compressed encodings, so the very first criterion is the instruction having an opcode that also has a compressed variant. We will outline below which other types of requirements RVC imposes on compressed instructions apart from this.

Immediate and Offset range Many values that are directly encoded in compressed instructions are limited to a shorter length than in their uncompressed counterparts: This applies to immediate values for register-immediate or constant generation instructions. These have a (signed) immediate field which is limited to 6 bits. Branch and jump target offsets are also limited to 8 and 11 bits, respectively.

Compressible registers Regular RISC-V instructions that can operate on any register include 5-bit register specifiers to select it (making all 32 registers of the RISC-V register set accessible unless specifically forbidden by the specification). Many compressed instructions can only accomodate 3-bit *shortened register specifiers*, which can select from only eight of the available registers. As shown in table 2.1, this is the case for the CIW, CL, CS, CA, and CB instruction formats. The registers which can be specified this way are registers `x8` to `x15` (called `s0-s1` and `a0-a5` in the RISC-V ABI). These registers are most commonly used as they are the ones defined in the RISC-V calling convention to be used to pass the first parameters for function calls, which are passed in registers instead of on the stack. Registers in this subset are referred to as *compressible registers* in this thesis. When combined with the hardware floating-point extensions, there is a similar subset that covers eight

of the 32 floating-point registers which can be addressed using shortened register specifiers.

2-Address Format Instructions with input and output registers may also have a requirement for a 2-address instruction format. This applies for CR, CI, CA, and CB instruction formats. RISC-V specifies uncompressed instructions in 3-address formats where applicable, meaning that operands can be specified separately from the destination. With a 2-address requirement, one operand must be in the same register as the result. This means the first operand cannot be used again in later instructions without moving it to another register prior to executing the 2-address instruction.

Note that for instructions with multiple requirements, every single one of them has to be fulfilled in order for the instruction to be compressible. For example, the `sub` instruction we used as an example in a previous chapter uses the CA format: It combines a 2-address requirement with the need for operands to be in compressible registers. If either of these requirements is not fulfilled, the uncompressed instruction has to be chosen (which has neither of these restrictions). This means that all compression requirements for an instruction are linked: Only fulfilling them partially while others remain unmet does not offer any advantage with respect to compression. However, fulfilling requirements partially may of course affect compressibility of other instructions.

The RISC-V specification [5] defines compressed variants for a number of instruction types, each with a combination of the different limitations described above. The following instructions are compressible:

- **Load** and **store** instructions with a source or target register that is compressible. However, stack-pointer-relative compressed loads and stores can be used with all registers. The offset of the specified address from a register is limited to 5 bits (or 6 bits for stack-pointer-relative loads and stores) and is scaled to increase the range.
- **Jump** instructions with a limited offset. This includes jumps to addresses in registers and jump and link instructions. Compressed **branch** instructions exist for comparisons of a compressible register with zero.
- **Constant generation** instructions with small immediates that fit into 6 bits.
- **Move** instructions from and to all registers.
- Register-based **arithmetic** instructions (`and`, `or`, `xor`, and `sub` instructions) with 2-address format and with all operands and the result being in compressible registers.
- Register-immediate **shift** instructions with 2-address format and partially with compressed register requirements.

- Register-immediate and register-register **add** instructions with 2-address format. Note that for add instructions, operands are not restricted to the compressible register subset. The compression requirements for add instructions are relaxed when compared to other arithmetic instructions because they are usually the most frequently used arithmetic instructions.

  The extension also includes special instructions for operations related to the stack pointer: Addition of scaled immediates onto the stack pointer can be used to adjust the stack pointer, and stack-pointer-relative adds that write to a compressible register are useful to generate pointers to values on the stack.

From this overview, it is already apparent that an improved compression rate cannot be expected for all of these groups when adjusting only the register allocation step in the compiler: All move instructions are already compressible regardless of the result of register allocation, but their overall number is of course affected by the quality of the register allocation result. Whether jumps and constant generation instructions are compressible depends on the offset and immediate sizes. This is not directly related to register allocation. Only a specific subset of all available branch instructions is potentially compressible. Most relevant when optimizing the register allocation step are therefore load/store and arithmetic instructions, as their compressibility requirements are directly related to the register allocation. More details on the relative frequency of the different instruction groups in benchmark code and the share of compressed instructions in them can be found in chapter 5.

2.3 LibFirm Architecture

To understand register allocation in the context of LibFirm, let us first introduce the way it represents programs: LibFirm uses a graph-based intermediate code representation (IR) called Firm [7]. Backend infrastructure to transform this representation into executable code for several architectures is included in LibFirm, as well as a selection of code analyses and optimizations.

LibFirm's intermediate representation is in SSA form (short for *single static assignment*). SSA requires variables in the IR to be assigned only once and then only used afterwards. Reassigning different values to a single variable is disallowed, as is usage of values without a prior definition. These restrictions have many advantages for algorithms and optimizations used in the compiler.

In SSA, dependencies of values on control flow are modeled by using so-called Phi (ϕ) functions: Phi functions choose from a list of previously defined variables based on where control flow originated from and define a new variable with the respective value. Actual hardware architectures in use do not directly support executing Phi functions, so they need to be resolved during compilation: This is done by replacing them with moves between registers (and/or memory) at appropriate places in the program.

Firm being a graph-based representation means that programs are represented as a collection of graphs: A function corresponds to a directed graph. Data and control flow are represented by nodes of different types and edges between these nodes. SSA values such as the results of data flow operations correspond to nodes. Control flow is also encoded in the graph: Instructions related to control flow and basic blocks are also represented as nodes. Their relationships (such as the nodes included in a given basic block or the operands of arithmetic instructions) correspond to directed edges between them.

An example for a Firm graph is shown in figure 2.2. It depicts the `max42` function we introduced as an example in section 2.1. Each node is shown with the type of operation it represents (such as `Mul` for multiplication), the type of the value (also called *mode*, the `Mul` node representing a signed integer or `Is`), and a unique identifier. Basic blocks are depicted as boxes around the nodes they contain. `Proj` nodes are used to extract values for cases where nodes output tuples of multiple values. We can also see a `Phi` node representing the Phi function which selects the correct value for the ternary operator in the function.

Using this representation, LibFirm can perform a multitude of optimizations expected from an optimizing compiler directly on the graphs: This involves transformations of the input graph that do not alter the semantics of the input program, but e.g. improve the performance of the resulting code.

In the backend, the optimized graphs can be converted into code for the target architecture. Even in the backend, the graphs remain in SSA form, but are modified and annotated with additional information. Highly simplified, this includes the following steps:

- During instruction selection, nodes are replaced with architecture specific operations (often representing available instructions on the target architecture).
- Scheduling means arranging the nodes in a way that conforms to the dependencies between values in the program and that can be used as the order instructions are emitted in.
- The register allocator assigns values to physical registers. Before this step, most generic nodes in the graph have already been replaced with architecture specific ones. When determining a register assignment, the register allocator also needs to make sure that enough registers are available to find a valid allocation: To guarantee this, some values may need to be transferred to memory first (which is called *spilling*). The register allocator needs to ensure that after being run, each value (each node in the Firm graph) is assigned a valid register, without conflicts between values that may be used at the same time in the program. Register assignments are annotated as attributes of the Firm nodes. As this is the main phase we are concerned with in this thesis, we will discuss register allocation in LibFirm in more detail in section 2.4 below.
- After handling requirements concerning how functions are called and how the stack is organized, the target architecture code can be generated and emitted.

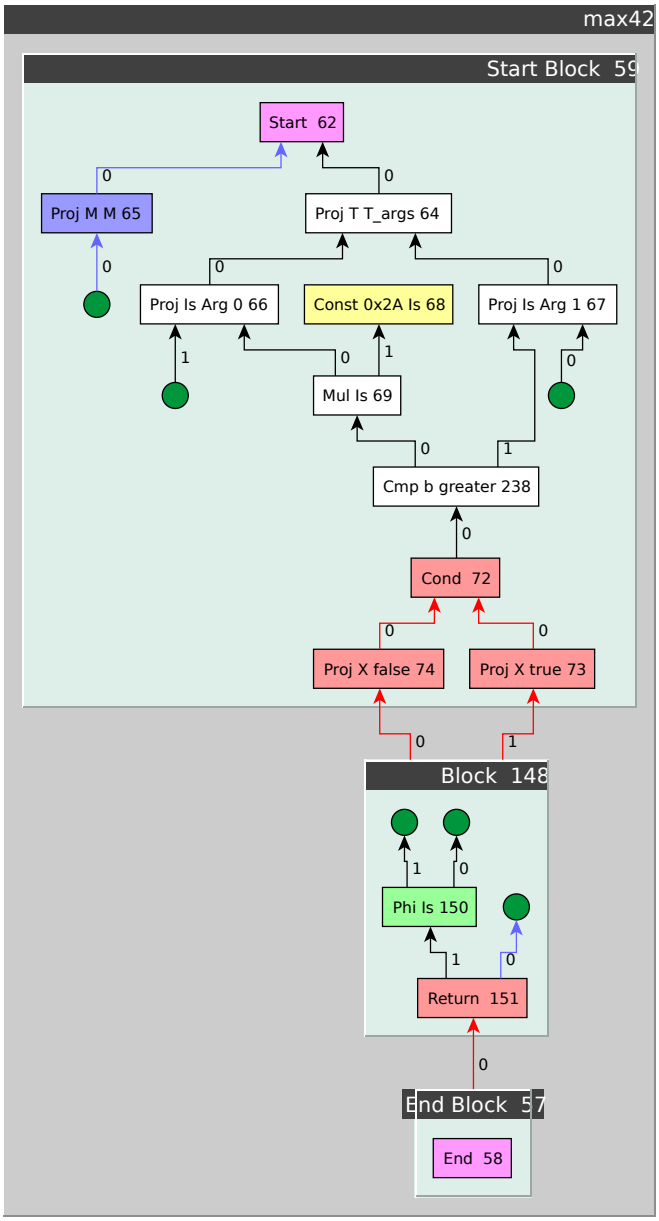


Figure 2.2: Firm graph for the function `int max42(int a, int b) { return (a * 42 > b) ? a : b; }`.

To be able to use this infrastructure to compile code in a high-level programming language, we use cparser, a C compiler built around LibFirm: It is a LibFirm frontend for C, performing the transformation into LibFirm's intermediate representation.

2.4 Register Allocation in LibFirm

LibFirm performs register allocation based on the formalization as a *graph-coloring* problem. A vertex coloring of an undirected graph is an assignment of colors to each node in the graph such that no two nodes that share an edge also share the same color. Register allocation can be expressed as such a problem using the *interference graph*. This is an undirected graph representing the relationship between the variables in the program. Variables or values are nodes in the interference graph, and edges are placed between two nodes if they are *live* at the same time in the execution of the program. Variables are considered live if their value could be needed later in the runtime of the program. This means that variables that are live at the same time cannot be assigned the same register: They are considered *interfering*.

When interpreting register allocation as a coloring problem, available registers correspond to the colors that can be used to color the graph. A valid register allocation then corresponds to a valid coloring for the nodes of the interference graph. This ensures that no two nodes sharing an edge are colored in the same color (i.e., no interfering variables are assigned the same register).

Register allocation is also related to *spilling*: Spilling refers to deciding whether and which values to transfer from registers into memory. This is required in order to produce a valid register allocation in cases where more than the available number of registers would be required without evicting some of them to memory.

This approach to register allocation as a graph-coloring problem was introduced by Chaitin in 1982 [8]. On arbitrary interference graphs, finding a valid coloring or even finding the number of required colors for a coloring (the *chromatic number*) is NP-complete. For this reason, Chaitin's algorithm uses the node degrees in the interference graph as a heuristic for coloring and spilling decisions. Chaitin's work has been extended or built upon in numerous ways [9] [10] [11], probably most notably by Briggs in 1992 [12]. In these algorithms, spilling and coloring are interlocked: During coloring, it may be required to iteratively spill more values in order to find a valid register allocation.

As we discussed above, SSA form is used throughout LibFirm's backend without deconstructing it for further handling. Instead, the transformations performed on the graph by the backend yield a structure that is more and more restricted and contains additional annotations. For example, scheduling or register assignment information is added.

This also means that when register allocation is performed, the program's intermediate representation is still in SSA form. For register allocation, this presents advantages related to the fact that SSA programs' interference graphs are *chordal*.

Chordal graphs are triangulated, meaning that for all cycles of more than three nodes, there are edges not part of the cycle that connect two nodes in the cycle.

The theoretical foundations of the current LibFirm approach to register allocation based on the special characteristics of SSA interference graphs were suggested in works by Hack and Goos. They showed that for a hypothetical fully uniform register set, register allocations for programs can be found in polynomial time [13]. More practically relevant, Hack introduced the current principles of register allocation in LibFirm in his PhD thesis from 2006 (published in 2007) [14]: This includes register allocation on chordal graphs, which allows to decouple spilling from register allocation. After spilling, register allocation on chordal graphs can be performed without possibly having to iteratively spill more values (like it is required when performing register allocation based on the Chaitin/Briggs algorithms).

2.4.1 Copy Optimization Heuristic

Hack also suggested the predecessor of the current *copy coalescing heuristic* used in LibFirm. Copy coalescing, copy elimination, or *copy optimization* attempts to reduce the number of move instructions that are required in the program. This is achieved by intelligently selecting a register allocation which places values which would need to be copied between registers into the same physical register. This heuristic is the component of LibFirm's register allocator which has the largest impact on the quality of the resulting register allocation.

Before running the heuristic, an initial assignment of registers is determined, and the heuristic tries to optimize it without invalidating the coloring. Note that in the descriptions below, assignment of a value to a specific register is referred to as *coloring* its node with the color corresponding to the register.

To aid the explanations below, we use pseudocode formulations of parts of the heuristic. These are shown in algorithms 1, 2, and 3. All of these listings are adapted from section 4.5.2 of Hack's dissertation [14], with changes as they are reflected in the current implementation of the heuristic used in LibFirm.

Affinities and Affinity Graphs The heuristic is based on the concept of the *affinity graph* of the program. This is an extension to the idea of the interference graph. It is an undirected graph that also contains values as nodes, but places edges between nodes that are *copy-related*: This means that finding a register allocation which assigns them the same color (places them in the same physical register) saves a move instruction between the two.

The purpose of the copy optimization heuristic is finding a valid register allocation that satisfies the maximum number of affinities. More precisely, it aims to fulfill the set of affinities which would impose the highest runtime cost if their associated move instructions could not be elided. To be able to model this, affinity edges are also assigned a *weight* representing the cost of not fulfilling the affinity. The cost function is adjustable in LibFirm: The default cost function is based on a simple heuristic estimating instruction execution frequency.

Affinity Chunks The reflexive-transitive closure of the affinity relation yields *affinity components*. These components of the affinity graph which are connected by edges are not always free of internal interferences. If they do have internal interferences, the nodes in the component cannot all be colored with the same color, and not all affinities they include can be fulfilled.

To be able to handle this situation, nodes are put into *affinity chunks*, interference-free subsets of affinity components. Thus they can in principle all be colored with the same color. In practice, this is not always possible. In that case, affinity chunks can be split further.

Algorithm 1 Affinity Chunk Construction

```
1: procedure BuildAffinityChunks(IG G: (nodes V, interference edges E, affinity edges A))
2:     for v ∈ V do
3:         v.chunk ← New chunk
4:     for xy ∈ A sorted by edge weight from high to low do
5:         ▷ Add all nodes in y's chunk to x's chunk
             if there are no interferences between the two chunks
6:         AbsorbChunk(x.chunk, y.chunk)
```

The initial affinity chunks are constructed greedily. Pseudocode for the construction procedure is shown in algorithm 1. Starting out with a single node, affinity chunks are grown by collecting other nodes connected with affinities, going in the order of decreasing affinity edge weights. At each step, this only needs to ensure that newly inserted nodes do not interfere with any of the nodes already in the chunk.

Chunk Weights When executing the heuristic, the weights of all affinity edges contained in a chunk are aggregated for all its nodes. This avoids having to handle each node in the chunk separately. Lists of nodes from other chunks interfering with any node in the chunk are also maintained per chunk for performance reasons. The summed weight of all affinity edges which connect nodes contained in the chunk is referred to as *chunk weight*.

Every chunk has an associated metric which indicates its favorability for each color. It is based on the number of nodes in the chunk that are colorable in the respective color; nodes that are very constrained in their color choice are weighted higher. From this, an order of the colors can be derived, which is called the *color preference* of the chunk.

Chunk Recoloring The chunk recoloring algorithm is a recursive method for heuristically finding the best color for a chunk. A simplified overview of the variant currently used in LibFirm is shown in algorithm 2.

Colors for the chunk are tried in the order that corresponds to the color preference value. When the color *col* is tried, recoloring follows this pattern: Generally, the target is to color all nodes in the given chunk with *col*. If *col* is allowed for a node n,

Algorithm 2 Chunk Recoloring

```
1: procedure ColorAffinityChunk(IG G, Chunk C, Queue Q)
2:     ▷ R is the set of registers in a register class (the set of colors available to color the graph)
3:     Compute color preference, order colors in R by preference
4:     for col ∈ R ordered by color preference do
5:         Unfix colors for all nodes in C
6:         for n ∈ C do
7:             ▷ Try to recolor n with col and choose a color that is not col for interfering neighbors
8:             ChangeNodeColor(n, col)
9:             n.fixed ← true
10:        local_best ← Best subset of C colored to col
11:        if local_best.weight > best_chunk.weight then
12:            best_chunk ← local_best
13:            best_color ← col
14:        if All nodes were colored in col then
15:            ▷ Stop searching if all nodes were recolored
16:            break
17:    for n ∈ C do
18:        ChangeNodeColor(n, best_color)
19:        n.fixed ← true
20:    rest ← C \ best_chunk
21:    if rest ≠ ∅ then
22:        rest_chunk ← New chunk
23:        for v ∈ rest do
24:            v.chunk ← rest_chunk
25:        Add rest_chunk to Q
```

the recoloring decision is then propagated recursively: Interference neighbors of n with the same color *col* are recolored in another color (any allowed color excluding *col*) and so on. This is performed by the ChangeNodeColor procedure on line 8. In case this recursive coloring attempt for the node n is not successful, the process has to be rolled back to the original colors. For this, a list is kept which holds the nodes that were recolored in the current pass as well as their original colors. During this process, the algorithm tracks the "best subset" of the chunk: This is the subset with the largest weight that could be successfully colored in the same color *best_color* (see lines 11 to 13).

It can occur that not all nodes in the chunk could be successfully colored with the same color. If this is the case, the chunk is then *split* into the part that was brought to the color and a new chunk with the remaining nodes for which this failed (shown in lines 20 to 25).

To ensure termination of the heuristic, recoloring is not continued under the following conditions: In the chunk that is currently recolored, a node's color is fixed when it is successfully recolored. Fixed colors are not to be changed again in the same recoloring step. This prevents nodes from being visited multiple times. Additionally, the recursive recoloring of neighbors is aborted if a fixed limit of recursive steps is reached.

The overall steps necessary to execute the heuristic using these components is shown in algorithm 3: This consists of building the affinity chunks as described above, then creating a priority queue that contains the chunks that are yet to be handled, recoloring them one after another, and then applying the resulting coloring.

Algorithm 3 Copy Coalescing Heuristic

```
1: procedure SolveHeuristic(IG G: (nodes V, interference edges E, affinity edges A))
2:     Setup data structures
3:     C ← BuildAffinityChunks(G)
4:     Initialize empty priority queue Q
5:     for c ∈ C do
6:         ▷ Q keeps chunks sorted by decreasing weight
7:         Insert c into Q
8:     while Q ≠ ∅ do
9:         c ← pop(Q)
10:        ▷ c is the chunk from Q with the largest weight
11:        ColorAffinityChunk(G, c, Q)
12:    for v ∈ V do
13:        SetRegister(v, resulting color from heuristic)
```

2.4.2 Mandatory 2-Address Requirements

As indicated in chapter 1, operand placement for some instructions is restricted in many architectures. This means compilers need to be able to model and handle these restrictions in order to generate correct code. For example, certain instructions in the x86 instruction set mandate a 2-address format. We will call these requirements which need to be fulfilled in all cases *mandatory* 2-address requirements. Let us look at how they are implemented in LibFirm:

- Instructions can specify them as a *should-be-same* constraint as part of their definition in the architecture specification. In more complicated cases where it is not yet clear whether such a constraint is required for an instruction, they can also be programmatically added later (during code selection) while building the graphs for the backend.
- While building the affinity graph for the copy optimization heuristic, should-be-same constraints are added into the graph as affinity edges. These are placed between the values that must be contained in the same register. The copy optimization heuristic handles these affinities the same way as any other affinities, trying to fulfill as many of them as possible. Note that the heuristic cannot always resolve all of these 2-address requirements.
- After register allocation has taken place, unfulfilled affinities originating from should-be-same constraints may remain. These are handled in a "fixup" step: In case of unfulfilled affinities, this inserts additional copy instructions before and/or after the offending nodes.

3 Related Work

This section provides an overview over other works relevant in the context of this thesis and how their approaches and applicability are different from the work presented here.

3.1 Irregular Architectures and Register Allocation

Computer architectures differ in the structure of their register set. The simplest model is to allow all data and operations to be used with all available registers. However, many architectures put restrictions on the use of registers or the combination of operations with values in certain registers. This ranges from few restrictions to highly structured register sets which require that different kinds of data be stored in different registers. The latter is sometimes found especially in embedded architectures.

Architectures with these kinds of restrictions are often called *irregular architectures*. While multiple approaches exist for dealing with these irregularities when performing register allocation, many of them focus more specifically on architectures with highly constrained register architectures. As described in section 2.1 in the previous chapter, there are often restrictions for compressed instructions affecting which registers can be used for specific instructions in order for them to be compressed. These differing requirements for compressed and uncompressed instructions also introduce a slight irregularity into a register architecture (even for those which might otherwise be considered fairly uniform). It is informative to take a look at works concerned with the role of irregularities in register allocation in general in order to identify whether their results are also useful for our goal of improving code compression.

Register allocation based on graph-coloring has been extended to explicitly accomodate architectures with irregularities in a multitude of works. This especially includes handling of multiple related register classes, as this introduces irregularities commonly found in many architectures. Smith and Holloway [15] modified the Chaitin-Briggs register allocation approach in 2000 to be able to handle some specific irregularities (especially those present in the x86 architecture) such as more flexible multi-register usage. To handle this, they introduce the notion of a weighted interference graph (WIG), with weights assigned to nodes in the graph, e.g. based on register width. Instead of using the node degree to determine colorability like in the Briggs register allocator, they introduce a more complex heuristic for colorability based on these weights. Runeson and Nyström presented a similar approach to modifying graph-coloring register allocation in 2003 [16], also supplementing additional information to the interference graph, in their case by including a mapping of nodes to their

respective register classes. As compared to Smith and Holloway, they also introduce generic "architecture description models" to be able to adapt their approach to different irregularities.

In 2002, Scholz and Eckstein [17] presented an approach which formalizes register allocation as a partitioned boolean quadratic optimization problem (PBQP) and uses dynamic programming in conjunction with heuristics to obtain register allocation solutions. Their approach focuses on highly irregular architectures which pose strong constraints on which registers can be used with which operations. The hypothetical highly structured register set they use to demonstrate their solution has separate address and index registers. A pairing between those types of registers determines which of them can be used in instructions together (e.g. a specific index register can only be used with a certain set of address registers).

However, these approaches focus specifically on handling irregularities based on requirements which always have to be satisfied, instead of the optional but desirable requirements we are concerned with when optimizing register allocation for compressed instruction sets.

3.2 Register Allocation and Code Generation for Compressed ISAs

Compressed instruction sets and the specific requirements they impose on code generation generally and on register allocation specifically have also been previously investigated. As described in the previous chapter, the original Thumb architecture was based on ARM and processors can support both instruction sets, but switching between Thumb and ARM requires an explicit mode change instruction.

Krishnaswamy and Gupta [18] presented an approach to allow for generation of mixed ARM and Thumb code in 2002 (note that this predates the Thumb-2 extension). It also includes a comparison of Thumb with regular ARM code regarding performance and size of the generated code, but focuses on rather coarse mixing of the instruction sets, meaning it either generates only Thumb or ARM instructions inside of a single function. Selection of whether to use compressed instructions is based on heuristic analysis of the code under consideration. Finer-grained mixing of instruction types inside of single functions is based on matching specific patterns in ARM code which are known to have shorter Thumb representations. These are selectively replaced by compressed instructions and surrounded by mode switch instructions.

Also based on the original Thumb without free instruction length intermixing, in 2005, the same authors Krishnaswamy and Gupta [19] presented a way to use the registers available in ARM by extending the Thumb ISA to allow the visible registers to be changed (of course requiring hardware modifications to support this). Their approach uses a custom extension of ARM/Thumb which adds `SetMask` instructions to choose which registers are visible without extending register specifiers in shortened

instructions. Registers are paired and only one of the paired registers is visible at any one time. To prevent code size penalties due to the `SetMask` instructions that need to be inserted, these instructions are coalesced into the following instructions where possible using their custom "Augmenting eXtensions" (AX) to decrease the number of additional instructions. Their work also includes an algorithm to place `SetMask` instructions in a way that reduces their overall number.

In 2010, Edler von Koch, Böhm, and Franke [20] presented an approach for code generation when allowing mixed 16- and 32-bit instructions, especially focusing on embedded use cases. The custom *ARCompact* ISA they investigate allows short and long instructions to be mixed without needing to modify the processor mode (the same way it is possible with the RVC extension). To leverage this, they introduce a method for "feedback-guided code generation", which is based on multiple compiler passes to improve code compression. The first compiler pass is used to annotate the intermediate representation (IR) with information on compression possibilities, and the following pass can use that information to deactivate generating compressed instructions where this would introduce additional move instructions or spills. They also include a code selection variant for "opportunistic" selection of compact instructions where requirements are incidentally fulfilled. As described in the previous chapter, a similar behavior is already present in the RISC-V toolchain due to the fact that the assembler already selects compressed instructions where all compression requirements are fulfilled. However, in the case of RISC-V, this behavior is transparent to the compiler itself and does not need any special support apart from utilizing the toolchain's assembler.

Lee, Moon, and Park [21] discuss another approach to designing compressed variants for instructions by grouping the full register set into multiple register banks. Their model works similarly to windowed register access concepts: Instead of defining one of the register banks as being available from compressed instructions, all registers can be made available by restricting the available registers to the currently selected bank. Their concept introduces a register allocation method which works with this design by grouping code into regions connected by sections of bank change and inter-bank copy instructions. Switching between banks is accomplished by adding a special register bank change instruction, which of course requires hardware support: For this, they proposed a "banked Thumb" (b-Thumb) instruction set. However, none of the architectures surveyed in the previous chapter use their proposed model to make more registers available for use with compressed instructions.

There are also works that address compressed instruction sets more generally: In 2016, Lopes et al. [22] presented a methodology for finding compression schemes for ISAs. It uses integer linear programming to optimize immediate value sizes in the compressed instructions, which they apply to design a compression extension for SPARC (which they call SPARC16). Their work also includes an analysis of different compressed ISA variants and evaluates the resulting compression ratios.

For our purposes, works that relate to the compression extension for RISC-V (RVC) are especially relevant: RVC has been thoroughly analyzed with regards to performance and code size. Li [23] also tries to further optimize RVC compression

results by reallocating the ISA namespace the RISC-V standard has assigned to compressed instructions. Perotti et al. [24] modify RVC with their *Xpulp* extension, which includes instructions to directly push values onto or pop values from the stack. They also survey compiler options that are useful to reduce code size, but do not introduce new compiler optimizations. These proposed schemes would all require adoption in the RISC-V ISA and hardware changes to be implemented.

Let us also review the current state of the art in mainstream optimizing compilers concerning code generation for compressed instruction sets, especially for RVC. In LLVM [25], this is handled during register allocation by exploiting register allocation order and register usage costs: The *greedy* register allocator included in LLVM includes a facility for specifying costs for using specific registers. These costs can be set in the backend description for each physical register. They are used to resolve allocation order by replacing the chosen register for a value with another, "cheaper" register in cases where there are no other interferences with values which would prohibit that [26]. In the case of Thumb-2 or RISC-V, only the registers which are part of the compressible subset are assigned no cost for using them [27], [28].

The result of the allocation is also dependent on the order in which registers are assigned. In the case of Thumb-2, there were documented changes to LLVM in order to optimize code size via adjusting the allocation order [29], [30]: With Thumb-2, the low registers `R0` to `R7` can be used for all compressed instructions, and these are currently also assigned first in LLVM: Previously, registers `R12` and `LR` were assigned before `R4` to `R7`. The proposed change [29] included an evaluation with a selection of benchmarks from SPEC CPU2000, SPEC CPU2006, and CoreMark, which documented an overall code size reduction of about 0.13 % in these benchmarks (most benchmarks showing a reduction in the range of 0.05 % to 0.4 %). Similar changes to the register allocation order in LibFirm were also implemented for this thesis. More details can be found in chapter 5.

However, to our knowledge, optimizing compilers like LLVM or gcc do not perform explicit modeling of compression requirements which goes beyond preferring compressible registers during allocation in this way, especially no modeling of other requirements such as 2-address requirements or multiple combined requirements.

3.3 Optimal Register Allocation

There are also methods that approach finding a solution for allocating registers in a specific program explicitly as generic optimization problems. These can be modeled e.g. by using integer linear programming, which allows solving them (possibly optimally) with suitable generic solving algorithms. Going this route usually allows requirements and parameters that impact register allocation to be more explicitly modeled in the problem statement than in fundamentally heuristic-driven approaches. It may also allow many other aspects of the register allocation problem to be modeled, such as requirements for instructions to be compressed. Arbitrary optimization metrics can in principle be inserted into the optimization problem and solved for. An example of a

recent comprehensive approach was presented by Lozano et al. in 2019 [31] as part of the Unison project [32]. Their work covers constraint modeling for register allocation combined with instruction scheduling and uses constraint solving to compute optimal register allocations. What makes this relevant in the context of this thesis is the fact that their solution also includes modeling of the costs and benefits of using compressed instructions. The requirements for instructions to be compressed can directly be included in the optimization problem that is built to describe a valid register allocation, e.g. when using code size as the overall cost function that is to be optimized for.

However, optimally solving register allocation in this way is usually only suitable in specific situations due to compilation performance concerns. Lozano et al. demonstrate speedups and code size improvements with their approach finding optimal solutions in cases of up to around 1000 instructions (given a fixed time limit of 15 minutes for finding a solution). They also concede that this approach is only applicable in cases where generated code performance or size is particularly important to warrant the trade-off with compilation times that are considerably longer than with heuristic-based approaches.

4 Compression Optimization

This chapter describes how our optimization is designed and how it is integrated into the copy optimization heuristic in LibFirm to accomodate compression requirements. It also contains an overview of how the implementation is handled in the LibFirm codebase.

The design consists of several parts that have to be in place. We will discuss them in turn: Considerations about the requirements for the proposed solution are outlined in the first section 4.1. To be able to take compression into account during register allocation, we need to be able to tell the register allocator in the backend which instructions in the lowered graph have compressed variants, and which requirements have to be satisfied in order for them to be used. This is discussed in section 4.2. This information can then be used in the copy optimization heuristic to adjust the result in a way that improves compression, but does not negatively affect the quality of the register allocation. This is especially relevant in case no compression information is provided by the backend. This part will be described in more detail in section 4.3. Notes on how the optimization can be implemented for backends can be found in section 4.4.

4.1 Design Requirements

This section serves as a high-level overview over how the extension of LibFirm for compression-aware register allocation is designed, especially focusing on which requirements guide the design. Let us look at the basic requirements the optimization should meet:

- Not all architectures supported by LibFirm make use of compressed instructions. This means that our optimization is not relevant for all target architectures, but only useful for certain backends. Compression only affects the goals of register allocation in detail while the basic target of eliding as many copy instructions as possible remains the main focus of the heuristic. In order to avoid duplicating existing copy optimization code by creating a new compression-aware register allocator, we want to design the optimization as a backwards-compatible extension to the existing heuristic. This enables backends to opt into the optimization where applicable. At the same time, our optimization should not affect the results of register allocation in other cases (e.g. for architectures without compressed instructions).

  Introducing multiple distinct code paths for compression-aware and -unaware register allocation should also be avoided to not impede maintainability: They

may diverge and fixes or performance improvements would have to be duplicated. Introducing an explicit new subsystem that handles register allocation for compressed instructions sets may also complicate testing of the register allocator.

- When including compression handling in the existing register allocation step, the execution time of the register allocator should also not be affected in cases where the backend being used does not support compression or no compression-related information is available. Preferably, this should be handled without needing an explicit flag that disables compression-aware register allocation where it is not applicable in order to preserve performance for this case.

- In the register allocation architecture used in LibFirm, the result of the copy minimization step in general has large performance implications for the generated code. It directly affects the number of copy instructions which need to be executed.

  In contrast, the effects of compression are more specifically related to code size, which is only indirectly linked to code performance (e.g. due to instruction caching; for more considerations about this, see chapter 5). For this reason, compression optimization should in most cases not trade off copy instructions versus compressed instructions, but instead prioritize eliding copy instructions where possible.

- The optimization itself should be independent of a specific backend. As described above, this allows backends to opt into using it: They should only need to specify their requirements for compressed instructions. This information can be provided in their architecture specification and by implementing a clear interface to the register allocator.

- The optimization should not alter the basic approach of register allocation used in LibFirm and be integrated into the concepts already used here: The well-tested copy optimization heuristic based on chunk-recoloring that is operating on affinity graphs.

On account of these goals, the compression optimization described in this thesis uses the metrics and capabilities already present in the existing heuristic and refines them to include added information pertaining to compressibility of the generated code. Specifically, this applies to the *weight* of edges in the affinity graph and *costs* for not fulfilling specific affinities: These are adjusted and extended, but the basic principle of finding a suitable register allocation result remains untouched. That also means that the result for a certain configuration of affinity and interference graphs, edge weights, and associated costs remains unchanged.

4.2 Compressibility Specification

During execution of the copy optimization heuristic, information about compression opportunities and the corresponding register requirements needs to be available. This is highly backend-specific: Different ISAs vary in which compressed instruction variants are available and which requirements they each have. For the purposes of this thesis, the model at least needs to cover the types of requirements that are present for RISC-V, but it should be designed in a way that is sufficient for other similar instruction sets as well. This way, compression-aware register allocation support can be easily added for other instruction sets like Thumb.

In general, LibFirm's register allocation is abstracted from specific backends. It relies on register-specific information that is part of a well-defined interface between the backend and the register allocator. This also covers required information concerning spilling of values (which is performed as the compiler step before). We extend this interface to also cover information related to code compression. Without the compression optimization, it already includes the following information:

- The costs associated with spills and reloads of values along with references to functions that insert the nodes required to perform them into an IR graph are directly included in the interface.

- A description of available registers and instructions as well as associated metadata is part of what we will call *architecture* or *backend specification*. They are implemented in Perl syntax as a way to provide a more compact representation of the specification. Perl scripts take these as input and generate C source code containing information about each register, available instructions, and register requirements for their input and output.

  The register description also contains a list of which registers are included in each available register class. For each register, other associated information such as its name or the index it is encoded with in instructions is also included.

As described in section 2.2, compressed instructions from RVC may have two fundamental types of requirements for the register placement of operands or results: Requirements of values to be placed in the subset of compressible registers and 2-address requirements. We will discuss how each one of them is integrated into the heuristic in more detail in section 4.3, but will first cover which compression information is required to represent this and how it can be specified in the backend. Information about compression requirements exists on two levels:

Compressible register subset On the level of the register set, we need to specify the *compressible register subset*. This means that each register needs information associated with it about whether it is part of this subset. We will also refer to those registers in the subset as *compressible* registers. They are specified by adding a flag to all compressible registers.

This is part of the architecture specification. It already includes the option to include other flags, and we modify the script generating the code representation from Perl to also parse the flag for compressible registers. When handling registers, this flag can then be accessed as a field on LibFirm's `arch_register_t` type which represents all information that is included in the architecture specification for a single register.

Node compression requirements On the instruction level, nodes in the lowered program graph – which represent potentially compressible instructions – need to be mapped to their compression requirements. To cover RVC, it is sufficient to model the two types of requirements outlined above. Compression requirements of one type are also never "mixed" on a single instruction for RVC. This means that if a requirement of one type is present on an instruction, it applies to all involved registers. For example, either all registers specified in a compressed instruction require a compressible register, or they can all be chosen from the full register set.

The option to indicate compression requirements for instructions in principle could be made part of the architecture specification, similarly to how mandatory register requirements are handled. However, compressibility for some instructions also depends on additional information apart from the instruction type and operand and result placement. For example, there are additional restrictions for immediate or offset ranges for compressed instruction variants, which would also need to be covered in the specification.

Hence the more flexible approach we are using in this thesis is to make backends that support compression-aware register allocation implement a predicate function which can inspect nodes in the program graph directly. It assigns compression requirement information to a node that is passed to it: This includes a specifier for the type of compression requirement as well as additional information that is useful to fulfill the requirement. This way, it can evaluate all information associated with the node and return the applicable compression requirement. This approach can also be used to examine a node's context (such as other nodes in the same graph).

Overall, when examining a node n from the lowered program graph, the resulting compression requirement specifier can be one of the following:

- *register subset*: n is compressible if all operands and the result are placed in a compressible register,
- *2-address*: n is compressible if the first operand and the result are placed in the same register,
- *register subset + 2-address*: n is compressible if both conditions above hold, or
- n is *never* or *always* compressible.

The latter are neutral to the behavior of the copy optimization heuristic, but can be included in debug information as part of predicting which instructions will be compressed (see below in section 4.4 for more details).

A reference to a function `get_op_compression_requirements` performing the mapping to compression requirements is added as part of the register allocation interface mentioned above. It can be used to query compression requirements during register allocation. An implementation can be provided by backends to opt into compression-aware register allocation and is provided as an input to LibFirm's register allocator.

Table 4.1 shows the compression requirements which are returned by our implementation of the mapping for the RISC-V backend. Some restrictions in the compressed instructions from RVC are not related to register placement of operands or results: This means that in some cases, attributes of the node have to be examined in order to determine whether it is potentially compressible depending on register placement.

While we can directly check whether immediates can be encoded in the compressed instruction variants, this is not the case for branch and jump offsets: These cannot easily be checked at this stage. Compressible jump instructions do not have an associated register-related compression requirement, which means that our choice for the resulting compression requirement is not essential for the copy optimization. For branch instructions, there is an associated register subset requirement for only a part of available branch conditions: Only equality or inequality comparisons with the `zero` register are potentially compressible. If this is the case, we *optimistically* add a register subset requirement, assuming the range is sufficient in enough cases for the requirement to be useful.

4.3 Compression Requirement Handling

The handling of compression requirements is integrated into the copy optimization heuristic currently used in LibFirm. For the purposes of this section, we will again refer to the description of the heuristic presented in section 2.4. The chunk recoloring algorithm based on Hack's thesis [14] discussed there is reprinted in listing 4. When discussing the heuristic, we will also again refer to registers as colors in the context of interference and affinity graphs.

The compression-aware heuristic uses the compressibility specifications described in the previous section as input for the heuristic: It has access to the program's nodes, register set descriptions, and the mapping from nodes to compression requirements as part of the register allocation interface. We will cover the changes to take compression into account by tracing along the register allocation process followed in LibFirm's register allocator.

4.3.1 Static Register Order

LibFirm's register allocator is somewhat sensitive to register ordering: This is the case because the registers (colors) are in several places evaluated in their order from the architecture specification. Additionally, the chunk recoloring algorithm (listing 4) does not always try all colors. As we can see in lines 14 to 16, recoloring is aborted if

Table 4.1: Overview over register-related compression requirement types that are assigned to operations of the RISC-V backend in our implementation of `get_op_compression_requirements`.

Firm opcode `iro_riscv_*`	Compression requirement			Note
	register	2-addr.	always	
`lw`, `sw`	•			
`j`, `jal`			•	Assumed[1]
`ijmp`, `jalr`, `switch`			•	
`bcc` (branch)	•			Assumed for (in)equality with `zero`[2]
`lui`			•	Depending on immediate[3]
`slli`		•		
`srli`, `srai`	•	•		
`addi`		•		Depending on immediate[4]
`add`		•		
`andi`	•	•		Depending on immediate[4]
`and`, `or`, `xor`, `sub`	•	•		
`SubSP`	•			
`SubSPimm`, `be_incSP`			•	Depending on immediate[5]
Copy (emitted as `mv`)			•	Generic firm node

[1] The sign-extended offset is 11 bits long and the is LSB omitted, yielding a 2 KiB jump range. We cannot check the offset, so we mark all of these jumps as always compressible.

[2] This is only marked as potentially compressible for equality (`eq`) or inequality (`ne`) condition markers. Additionally, the second operand register must be fixed as register `zero`. In this case, the sign-extended offset with LSB omitted is 8 bits long, allowing for a ± 256 B range. We cannot check the offset, so we add a register requirement for all nodes fulfilling the other requirements.

[3] If the upper bits (ignoring the lower 12 bits) fit into 6 bits.

[4] If the (signed) immediate fits into 6 bits.

[5] If the (signed) immediate fits into 6 bits or the lowest 2 bits are zero and the bits above fit into 6 bits (in this case, there is the special `C.ADDI16SP` compressed instruction that scales its immediate by 16 and adds it to the stack pointer).

Algorithm 4 Chunk Recoloring Algorithm

```
1: procedure ColorAffinityChunk(IG G, Chunk C, Queue Q)
2:     ▷ R is the set of registers in a register class (the set of colors available to color the graph)
3:     Compute color preference, order colors in R by preference
4:     for col ∈ R ordered by color preference do
5:         Unfix colors for all nodes in C
6:         for n ∈ C do
7:             ▷ Try to recolor n with col and choose a color that is not col for interfering neighbors
8:             ChangeNodeColor(n, col)
9:             n.fixed ← true
10:        local_best ← Best subset of C colored to col
11:        if local_best.weight > best_chunk.weight then
12:            best_chunk ← local_best
13:            best_color ← col
14:        if All nodes were colored in col then
15:            ▷ Stop searching if all nodes were recolored
16:            break
17:    for n ∈ C do
18:        ChangeNodeColor(n, best_color)
19:        n.fixed ← true
20:    rest ← C \ best_chunk
21:    if rest ≠ ∅ then
22:        rest_chunk ← New chunk
23:        for v ∈ rest do
24:            v.chunk ← rest_chunk
25:        Add rest_chunk to Q
```

a valid color for all nodes in the chunk could be found. This performance optimization is due to the fact that the heuristic views two colors as interchangeable as a valid chunk color if both can be used to color a chunk completely. Without any additional restrictions on the registers, this assumption is valid, but it no longer holds for our case: The compressible register subset should be preferred over the other registers as using registers from it may improve code compression.

In summary, this means that even without any other explicit handling of compression requirements in the compiler, the compression result can be improved by reordering the registers in the architecture specification. As discussed in section 3.2 for the case of LLVM, this kind of optimization has also been shown to be profitable for improving code compression in other register allocators.

The best result is expected if registers that are part of the compressible register subset are provided first as an input to the copy optimization heuristic. We will refer to this configuration where the registers have been reordered in this way, but compression is not considered in other ways when performing register allocation as *static register preference* and will evaluate its effects on code size in chapter 5.

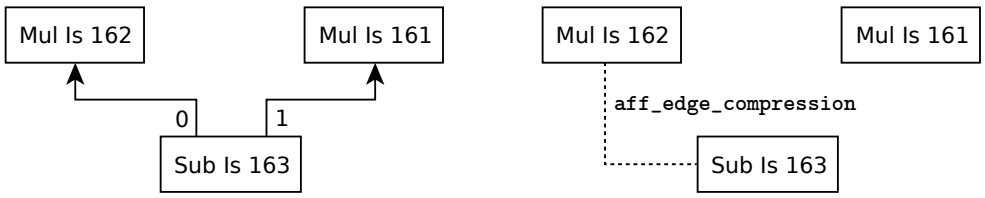
4.3.2 Affinity Graph Structure

Compression-related 2-address requirements can be handled by modifying the affinity graph before the heuristic is being run. In section 2.4.2, we discussed the existing concept of *mandatory 2-address requirements* and how they are handled in LibFirm. To recap, instructions can specify should-be-same constraints which are added as edges to the affinity graph. These are taken into account by the heuristic. In case of unfulfilled affinities, copy instructions are added to ensure all of the mandatory 2-address requirements are satisfied.

This functionality is similar to what is required when optimizing compression, but we are concerned with a different kind of 2-address requirement: Compression-related 2-address requirements are *optional*. As we discussed before, not fulfilling them only results in certain instructions not being compressible. This is why we do not require any fixup in order to handle compression-related requirements which remain unfulfilled after executing the copy optimization heuristic. Instead, our goal is not to fulfill all 2-address requirements, but only those which can be handled without otherwise negatively impacting the register allocation result. For this reason, we do not use the existing should-be-same constraints for compression-related 2-address requirements.

We also map our optional 2-address requirements to additional affinity edges in the graph, but without specifying a should-be-same constraint for the instructions. Additionally, these edges are tagged as a different kind of affinity to be able to distinguish them in the copy optimization heuristic.

Affinities for 2-address requirements always connect the result and an operand of an instruction. In the affinity graph, this yields the following structure: For a node n with 2-address requirement, the affinity edge connects n itself to its predecessor representing the value used as an operand in n. Figure 4.1 shows this for the



(a) Excerpt from a Firm graph **(b)** Resulting affinity

Figure 4.1: Insertion of compression-related affinity edges into the affinity graph.

example of a `sub` instruction (which in RVC has a 2-address as well as a register subset requirement). The affinity edge is inserted between the `Sub` node and its first operand.

By introducing optional 2-address requirements as additional affinities, the affinity graph now contains affinities with different meanings. To distinguish between the types of affinity edges in the graph, edges are annotated and can be handled differently during register allocation depending on which kind of affinity they represent. The available edge types are:

- `aff_edge_noncopy`: An affinity indicating copy-relatedness, meaning that a move instruction can be omitted if it is fulfilled.
- `aff_edge_compression`: A compression-related affinity which indicates that an instruction may be compressed if it is fulfilled.

This approach allows affinities to be handled uniformly in the copy optimization heuristic, while also being able to distinguish the types, e.g. to prioritize them. A way to do this which we will discuss below is adjusting their weight to be able to tune the influence of compression-related affinities versus affinities which may allow move instructions to be elided.

LibFirm's *copy optimization driver* builds the affinity graph before it runs the copy optimization heuristic, and additional affinities are inserted after checking for compression requirements there.

Commutativity Some compressible instructions with a 2-address requirement are commutative, i.e., their operands as well as the requirement can be switched around. This affects some of the arithmetic register-register operations: `add`, `and`, `or`, and `xor` instructions all have a 2-address requirement as well as being commutative.

The RISC-V assembler is also aware of this and uses it to improve code compression: For instructions where the 2-address requirement is fulfilled apart from the ordering of the operands, they are exchanged to allow the compressed encoding to be used. For example, an `xor s0, s1, s0` instruction is converted into the compressible binary representation of the `xor s0, s0, s1` instruction by the assembler (if it targets an architecture with the C extension). Note that for `xor`, there is an additional register

subset requirement which is also fulfilled here as the registers `s0` and `s1` are both compressible.

We also use this during the insertion of affinity edges into the graph: Compression requirements can be marked as being commutative. In this case, if they describe a 2-address requirement, an additional affinity edge for the second operand of an operation is inserted, yielding edges between the result of an operation and both operands. The mandatory 2-address requirements we discussed in section 2.4.2 also use this approach of adding multiple affinities. In most cases, only one of these affinities can be fulfilled at a time, but either is sufficient to satisfy the 2-address requirement for the compressed instruction variant: A compressed instruction is emitted by the assembler regardless of which of the operands is overwritten by the operation's result.

4.3.3 Copy Optimization Heuristic

We can now examine how the compressibility specification described above and the additional information encoded in the affinity graph are used in the copy optimization heuristic to improve code compression.

In the chunk recoloring algorithm from listing 4, we will especially examine and adjust the following parts to incorporate compression:

- The "best" chunk subsets that can be colored with the current color are evaluated according to their *weight* as shown in lines 11 to 13. These chunk weights can be adjusted based on how much the chunk influences the overall compression of the program.
- The *color preference* from line 3 is used to determine the order in which colors are tried when coloring the chunk. This is shown in line 4 of the algorithm. By adjusting the costs associated with certain colors, we can influence the coloring order.

Chunk weights The additional affinity edges used for modeling 2-address requirements described above especially affect chunk construction and chunk *weight*. They affect chunk construction because affinity edge weights determine the order in which chunks are expanded in the affinity chunk construction algorithm discussed in section 2.4 and specified in listing 1. Chunk weight is calculated as the sum of the weights of all affinity edges it contains, so these edges also contribute to the overall weight a chunk has.

When computing the weights of chunks as part of the heuristic, edge weights for affinities arising from compression requirements are scaled down as compared to other affinity edges. This is controlled by a compiler parameter called *ccscale* (short for *compression cost scale*), which is the ratio between weights of compression-related edges and edges between copy-related nodes.

The weight of affinity edges that are due to copy-relatedness is governed by a cost function c_0 that determines the cost of requiring a copy instruction at a particular

node. The standard cost function is based on a heuristic estimating execution frequency. To compute the weight of compression-related edges, we also use the cost function, but scale it down with *ccscale*:

$$c_{\text{compr}}(n) := \mathit{ccscale} \cdot c_0(n) \,.$$

Using the cost function that potentially incorporates execution frequency is done even though code size is not related to execution frequency: Any compressed instruction reduces code size by the same amount, irrespective of its dynamic execution frequency in the program. This is done for two reasons: Firstly, c_0 serves as a reference value for the cost of compression-related edges. Without such a value, we would not know the base cost that is used by the (potentially arbitrary) cost function c_0. Therefore, not aligning it to a reference could potentially lead to compression-related affinity edge weights being completely out of scale from copy-related affinity edge weights, potentially being much higher or lower than any other affinities in the graph. Secondly, not scaling compression-related affinity weights could lead to them being essentially ignored in any context with a higher-than-base estimated execution frequency. We also added a compiler option to the heuristic to be able to disable using the cost function if desired. In this case, all compression-related affinity edge weights are evaluated as the fixed value of *ccscale* in the heuristic.

Chunk weights are used to decide the order in which chunks which remain to be colored are handled. As chunk weights were previously integers, which is not compatible with freely scalable edge weights, they have been extended to be floating-point values. This means that in order to use them in the priority queue implementation employed by the heuristic, the value is truncated. This does sacrifice sub-integer precision in the order represented in the priority queue, which is negligible in practice.

Costs and compressibility In general, the heuristic aggregates metrics across chunks and only recalculates them in case the chunk structure changes, so we also do this for compression-related costs. This removes the need to repeatedly traverse the graphs themselves during register allocation.

As described above, colors are chosen as a potential new color for the chunk in an order that conforms to the chunk's color preference (see line 3 in listing 4). The color preference of a chunk C assigns a cost $\mathit{cost}_0(C, \mathit{col})$ to each color *col*. The colors are then tried in order of descending cost cost_0 in the chunk recoloring step. Note that this means that a color with a *higher* cost value is preferred during chunk recoloring. The color preference cost is based on a combination of how restricted the colors for the nodes in the chunk are and which other chunks it interferes with.

We adjust these costs to incorporate compression into the color preference order: To do this, we adjust them for colors *col* that are part of the compressed register subset. The amount the cost is adjusted by is based on the number of nodes that depend on *col* being in the compressible register subset. Instructions with compressible register subset requirements require both their result as well as all their operands to be in compressible registers: This is why we incorporate both in the calculation.

For this adjustment, we need to track the amount of nodes that depend on the chosen color *col* for the chunk C being in the compressible register subset. To do this, we use a metric we call *compressibility*(C). It is based on two components:

- $|D_{\text{restr}}|$: The number of nodes in the chunk itself which require their own result to be placed in a compressible register in order for them to be compressible. In this case, $D_{\text{restr}} \subseteq C$ denotes the set of nodes in C with a register subset compression requirement.
- $|U_{\text{restr}}|$: The number of nodes that use the value of a node $x \in C$ as an operand and require x to be colored with the color of a compressible register to be compressible themselves. This means that $y \in U_{\text{restr}}$ iff. y has a register subset compression requirement and y uses a value $x \in C$ as an operand. Note that the node y can both be inside or outside the chunk C.

Along with the number of nodes in the chunk denoted by $|C|$, the compressibility metric can be calculated with

$$\mathit{compressibility}(C) := \frac{|D_{\text{restr}}| + |U_{\text{restr}}|}{|C|}.$$

For example, $\mathit{compressibility}(C) = 1$ means that the number of nodes whose compressibility depends on the chunk C being colored with a compressible color is the same as the number of nodes $|C|$ it contains.

The color preference adjustment is also based on an additional parameter: The *compressibility influence* parameter *ci* can be used to specify how much the color preference is adjusted by.

We can now use compressibility to adjust the costs assigned to specific colors: The color preference costs for colors that are part of the compressible register subset $R_c \subseteq R$ are adjusted upwards, proportionally to the prior cost $\mathit{cost}_0(C, \mathit{col})$ assigned to a color by the heuristic:

$$\mathit{cost}_{\text{adj}}(C, \mathit{col}) := \begin{cases} \mathit{cost}_0(C, \mathit{col}) \cdot (1 + \mathit{ci} \cdot \mathit{compressibility}(C)) & \mathit{col} \in R_c \\ \mathit{cost}_0(C, \mathit{col}) & \text{otherwise} \end{cases}$$

Note that this means that the influence parameter *ci* is not a strict cap of the maximum influence on the cost: In cases where a single instruction is used by many others that would themselves benefit from its result being placed in a compressible register, this yields a high *compressibility* value and in turn a larger adjustment to the cost.

Requirement pruning We also experimented with a mechanism we will call *requirement pruning*, but elected not to include it in the final implementation: In our evaluation, it did not appreciably improve code compression, while at the same time introducing some additional complexity to the code.

The mechanism was intended to accomodate the fact that compression requirements for an operation are related in the following sense: The question whether the operation can be compressed depends on multiple coloring decisions. For this reason, we evaluated whether compression could be further improved by disabling (or pruning) requirements that belong to operations that are not compressible for other reasons, regardless of whether these requirements are fulfilled. In the algorithm for coloring chunks, this for example applies when the register for an operation is fixed to a register not in the compressible register subset. In this case, we can disable a related requirement such as an additional 2-address requirement as there is no benefit to fulfill it anyway. Disabled or pruned compression affinity edges can then be excluded from the weight of the chunk they are contained in. We implemented this mechanism in one direction: If an operation cannot be compressed due to the color that has been chosen for its result, we can disable its linked 2-address requirement; the latter is encoded in the affinity edge between its operand and result.

However, as colors are frequently fixed and unfixed during execution of the heuristic, compression requirements also have to be updated frequently to correctly keep track of the current set of active requirements. In our tests, the difference in compression between activated and disabled requirement pruning was very slight and code size improvement or degradation depended on which specific benchmark was used.

Compiler parameters Overall, this yields two continuous parameters that can be used to tune the behavior of register allocation in the presence of compression requirement information:

- The *compression cost scale factor* (*ccscale*) governs the priority of compression-related affinity edges when compared to other affinity edges.

- The *compressibility influence* (*ci*) affects the color preference order based on the number of nodes that are affected by the choice of a compressible or incompressible register for a chunk.

Choosing lower values for these parameters means that the result of the heuristic is affected by the compression parameters to a lesser extend. We would expect no code compression improvements with these parameters set to zero, and improving compression with increasing values. However, especially in the case of *ccscale*, we can also expect the overall result of the heuristic being negatively affected with high values as copy-related affinities are fulfilled in fewer cases and traded off against compression-related affinities.

These parameters are also available to be set externally when invoking cparser to fine-tune the effects of the compression optimization. We will discuss their choice in section 5.3 in the evaluation.

4.4 Backend Integration

Let us also examine how this approach interfaces with backends wanting to support the optimization. The following steps are required to add support for compression-aware register allocation to an additional backend:

- Marking compressible registers in the backend specification.
- Implementing `get_op_compression_requirements` with the compression requirements based on the ISA specification.

  In general, this implementation should be handled conservatively: "Under-specifying" compression opportunities for instructions should be preferred to overspecifying them. This is the case as incorrectly specifying that an instruction is compressible may result in degraded compression rates for other instructions. When not implementing all facets of compression requirements, nodes may then be marked as incompressible instead. In these cases, not specifying potentially non-existant compression opportunities ensures that the quality of the register allocation is not negatively affected regarding the number of move instructions remaining in the program.

  Compression requirements could also be too complex to identify or not all information be available at the time of register allocation. This may for example be the case for jump instructions which are compressible with a limited distance. At the time of register allocation, basic block scheduling information is not yet available. This means it may not be possible or at least not practical to check whether the required offset in a jump is compatible with its compressed variant.

  On the other hand, if jump distances that can be encoded in compressed instruction variants are sufficient to encode a significant portion of jumps, implementors could also decide to mark them as compressible "optimistically" and include the corresponding register-related compression requirements.
- In cases where the generation of compressed instructions cannot be handled transparently by the assembler like for RVC, generation of compressed instruction encodings also needs to be reflected in the code generation step. When implementing a backend which directly generates binary code, selecting compressed or uncompressed instruction variants of course needs to be part of the compiler itself.

For the purposes of RVC, the latter part is not required: As described in section 2.2, the RISC-V assembler (when invoked to generate code for an architecture variant which includes the C extension) generates compressed instructions where applicable, without any of them having to be explicitly selected in assembly code mnemonics.

Still, the overall result of course depends on the accuracy of assigning compression requirements to nodes. To check whether this assignment is accurate, it is useful to

add a *prediction* of whether a specific instruction will be compressed or not after register allocation is completed. This can then be checked against the actual output of the RISC-V (or other) assembler. To do this, the prediction needs to be included in generated assembly code.

LibFirm already includes a facility to include debug information in the compiled code (such as the origin of instructions in the source program and internal graph node labels). To cover compression predictions, we extend this with a way to include additional arbitrary debug *notes* in the generated assembly. When emitting code, the predictions are then inserted based on assigned compression requirements and the result of register allocation. This is also useful when adding compression support for additional backends to check the consistency of the compression requirement assignment with the final generated code.

5 Evaluation

After discussing how to model and integrate compression-awareness into LibFirm's SSA-based register allocator, this chapter examines the applicability and effectiveness of the described optimization.

We will begin by describing some general considerations and the metrics that are affected by our optimization in section 5.1. An overview over the benchmarking data set we chose for the evaluation can be found in section 5.2. The benchmarking setup is discussed in section 5.3, including how benchmark data was collected and the compiler configurations we use to compare the results. Results of the compression optimization are presented in section 5.4.

5.1 Scope and Affected Metrics

We will first take a look at the ways in which the compression-aware register allocation optimization proposed in this thesis affects the generated code. The overall structure of the generated code is unaffected as no changes to code selection, spilling, or scheduling were made in the optimization. However, the heuristic of course does affect the number and placement of remaining copy instructions in the code. As described in previous chapters, move instructions remain in the resulting code when affinities are not satisfied. These affinities are provided as an input for the heuristic in the form of the affinity graph.

The parts of LibFirm which are used for estimating dynamic code behavior were also not changed for the optimization: Most notably, this includes the heuristic for estimating dynamic execution frequency of operations in the code. It is used to assess the priority of particular affinities when performing copy optimization.

An improved compression result can be directly measured by comparing static code size between the code generated with and without our optimization. This is related to the number of compressed and overall instructions that are generated: We will refer to their ratio in a certain program or group of instructions as *compressed instruction share*. Note that not all types of instructions have compressed variants: This means that not all instructions can be considered for compression if code selection is unaltered. However, in this evaluation, the compressed instruction share is always measured relative to the overall number of instructions, not relative to the number of instructions that are potentially compressible given the right registers and immediates. Due to the fact that the scope of our optimization is restricted to the register allocation phase (or more specifically, to the copy optimization), the number

of overall instructions that are generated can only be affected by a lower or higher number of remaining move instructions.

Generated Code Performance As indicated in chapter 4, an important design goal of the optimization was to not negatively affect the overall number of move instructions. This does have a direct impact on the runtime performance of the generated code. Unfortunately, we could not evaluate this in a test setup with real RISC-V hardware in the context of this thesis. However, the limited scope of the changes in the generated code we described above also limits the runtime performance impact of the optimization: An overall performance degradation can only be expected in case the overall number of move instructions remaining in the compiled program is increased with the compression optimization. We will discuss this in section 5.4.

There is also another potential factor that may influence runtime performance of compressed code in a more indirect way. On machines using instruction caching, code compression may have a performance impact apart from the number of (e.g. move) instructions that need to be executed: For code with higher compression rate, more instructions may fit into instruction caches. This potentially leads to fewer instruction cache misses. Therefore, code with optimized compression could also be executed faster. However, this may be subtle and hard to measure without hardware with effective instruction caching. Benchmarking the effect of this is beyond the scope of this evaluation. We instead focus on the direct performance impact of remaining move instructions as described above.

5.2 Benchmarking Data Set

To be able to judge the results both in terms of ensuring that the optimization does not affect correctness of generated code as well as its effects on code density in real-world contexts, we use a collection of tests and benchmarks. Using the latter, we can compare resulting code sizes with and without the optimization.

LibFirm Test Suite During development, cparser and LibFirm are tested with an extensive test suite to ensure correctness of the compiler and its optimizations. We use this test suite to make sure that changes to the compiler do not break previously working compilation test cases. We could not observe any regressions in tests from the test suite when comparing between our modified compiler and the mainline version, especially for code compiled for RISC-V.

The test suite contains a variety of test types and could also be used to measure compression performance. However, most test cases included in it are not designed to simulate real-world code. Instead, many minimized, specific test cases have artificial constraints such as only covering very small functions, including unstructured code, and exhibiting low register pressure. For this reason, we did not use the test suite to measure the results of our optimization, but only to validate its correctness.

Embench Code size and code density are especially important for embedded systems. Embench [33] is a benchmark set especially curated to benchmark these kinds of systems. It is focused on benchmarks that represent smaller applications found on embedded systems, such as "Internet of Things" devices. The benchmark suite which was introduced in 2019 [34] is based on benchmarks from the Bristol/Embecosm Embedded Benchmark Suite (BEEBS), and also based on previous benchmark suites such as MiBench and DSPStone.

It is a collection of 22 benchmarks, all designed to only require small amounts of memory (intended to not exceed 64 KiB of program space and RAM during execution) and restricted in their execution time to make it feasible to use them for benchmarking embedded hardware. From the set, we used 21 benchmarks: We excluded `cubic` from our tests as it uses the `long double` data type which is not yet supported by LibFirm's RISC-V backend.

SPEC CINT2000 The SPEC CPU packages are standard benchmark suites that are especially used for performance testing (both of computing systems as well as for compilers). SPEC CPU2000 [35], the version we are using for the purposes of this evaluation, is split into benchmarks focused on integer (CINT2000) and on floating-point performance (CFP2000). The benchmarks are much larger in code size and more demanding in resource usage when compared to the benchmarks from Embench.

The RISC-V LibFirm backend is not yet fully-featured and does not support hardware floating-point instruction handling, but does include software floating-point translation. However, this limits the type of benchmarks which are useful to test the quality of the code generated with this backend: All programs were compiled with software floating-point handling, which would bias the results in cases where a substantial part of the program consists of floating-point calculations. For this reason, only benchmarks from the CINT package were included.

The package includes 12 benchmarks, most of them written in C, with one using C++. As cparser's support for C++ is very limited, the benchmark `252.eon` was excluded from our benchmarking data set. We also excluded `253.perlbmk` due to lacking file system support in the libraries included with the RISC-V toolchain. Due to problems in the RISC-V backend that are unrelated to our optimization, compilation of the `176.gcc` benchmark could be completed, but was not successful and produced miscompiles: These issues are related to the handling of the RISC-V calling convention and jump ranges. However, as complete object files for all parts of the program could be fully generated, we chose to still include the benchmark in our analyses as these issues likely do not affect compression results.

5.3 Setup

When compiling code for RISC-V, cparser emits assembly code which is then passed to the assembler and linker which are part of the RISC-V toolchain. The assembler

evaluates which of the instructions fulfill all compression requirements and emits instructions in compressed encodings for those.

To measure and compare static code size, there are several methods which are available, the simplest being a comparison between resulting executable sizes. However, (statically) linked executables may contain substantial amounts of code not under our control: The size of precompiled system library code in the executable is of course unaffected by the use of any optimizations in the compiler used for user code. Thus comparing overall executable size may reduce comparability across different benchmarks.[1] The same is true for code sections in the executable other than the text segment. Therefore, we compare the results of our compression optimization by using a "net" code size: For code size comparisons, we compare only *text segment* sizes for unlinked object files. All of the code included in these is generated by cparser, which increases comparability between benchmarks. However, keep in mind that the relative code size reduction expected for the overall executable may be lower than the values given in this evaluation, depending on the size of code sections other than the text segment and the amount of foreign code linked into the executable. For the purposes of this evaluation, the term *code size* is used to describe this net text segment size.

To evaluate text segment sizes, we use the output from the `size` command included in the RISC-V toolchain. For more detailed analyses of instruction types and their compression properties in the generated code, we use the decompiled output from `objdump` (executed on the aforementioned object files).

Compiler configurations To put the results of our optimization into context, code size and compressed instruction share are compared across different configurations of the compiler:

- *Uncompressed* refers to not using any compressed instructions at all. This corresponds to compiling code for a RISC-V target configuration that does not include the C extension. In our case, this is also handled by the assembler, which does not emit any compressed instruction variants.

- *Reference* is the current latest state of cparser and LibFirm without any of the changes proposed in this thesis.

- *Static* refers to only statically reordering the registers in the architecture specification as described in section 4.3.1 without any other changes to the register allocator and copy optimization heuristic.

[1] The suggested solution for the Embench benchmark suite when evaluating static code size is to use "dummy" libraries. These contain empty declarations for all required libraries which do not contain any implementations. When comparing static code size, they can be linked into the compiled binaries without adding to the code size footprint. This however requires creating special dummy versions of all libraries that are used in the benchmarks and is hence not practical for our use case.

- *Result* is the configuration when applying our compression optimization including all mechanisms discussed in this thesis.

Comparisons that are given as a relative change in this chapter are given in relation the *reference* configuration. It represents the current mainline implementation of the compiler without any explicit handling of compression in its register allocator.

Instruction groups To be able to investigate the effects of compression optimization in different subsets of the instruction set, instructions were grouped based on their functionality. The groups we identified are:

- loads and stores,
- move instructions,
- arithmetic instructions,
- instructions used for constant generation,
- multiplication and division instructions,
- (unconditional) jump instructions,
- conditional branch instructions, and
- all other instructions.

A detailed list of which instructions were included in which of these groups can be found in appendix A.1.

The share of instructions which are from each of these instruction groups in code generated for the benchmarks from SPEC CINT2000 is shown in figure 5.1. Note that the share of move instructions in this graph is likely higher than may be expected in real-world generated code. This is due to the fact that as we described above, floating-point calculations were performed as soft-float operations without hardware floating-point instructions. This results in more calls into the corresponding library functions and hence more moves to satisfy the calling convention. Also, LibFirm's handling of callee-saved registers is not yet fully optimized, which may also contribute to an elevated share of move instructions.

Compression Cost Scale and Compressibility Influence As we described in section 4.3.3, we introduced two new tunable parameters into the copy optimization heuristic that are related to compression optimization. We will now take a closer look at how their values influence the results. As they affect the overall compression of the code in conjunction with one another, we will also look at them that way here. Our tests indicated that a relationship of $ci = 2 \cdot ccscale$ produced the overall best results when comparing text segment sizes, so this is the setting we will look at here. Figure 5.2 shows the relative changes in text segment sizes, instruction count, and compressed instruction share for different values of the parameters with the above ratio for the benchmark `186.crafty` from SPEC CINT2000. It was chosen as a medium-sized

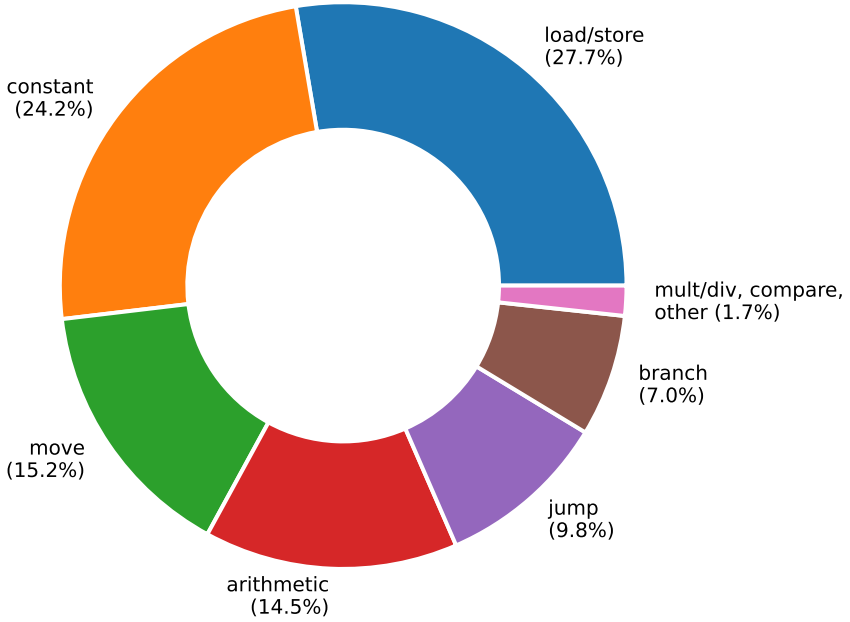


Figure 5.1: Shares of instruction groups in code generated for the benchmarks from SPEC CINT2000. The instruction group assignments can be found in appendix A.1.

benchmark from our data set that exhibited very consistent code sizes when compiled with the *reference* configuration repeatedly.

The results are indexed against the result of compilation with both parameters set to zero, which yields results that are close to the *static* configuration both in code size as well as in compressed instruction share.

For this benchmark, even very low nonzero values for the parameters yield a sizable improvement in code size, with the maximum improvement of around 2.8 % being reached at a *ccscale* value of about 0.2. If we aggregate across all benchmarks from our data set, the resulting overall code size is smaller with *ccscale* values in the range of 0.1 to 0.3 than with values that are very close to zero.

For larger values of the parameters, we can observe the trend we anticipated in section 4.3.3, if only slightly: Increasing the parameters further yields an increased number of overall instructions, which in turn increases code size. This is likely the case because compression-related affinities compete against affinities that are due to copy-relatedness: With a high enough priority, the former are fulfilled at the expense of the latter.

It is also visible that the result of the heuristic is “unstable” in the sense that small changes in the input parameters can yield large jumps in the register allocation result: Single coloring decisions may have a profound effect on the further course of

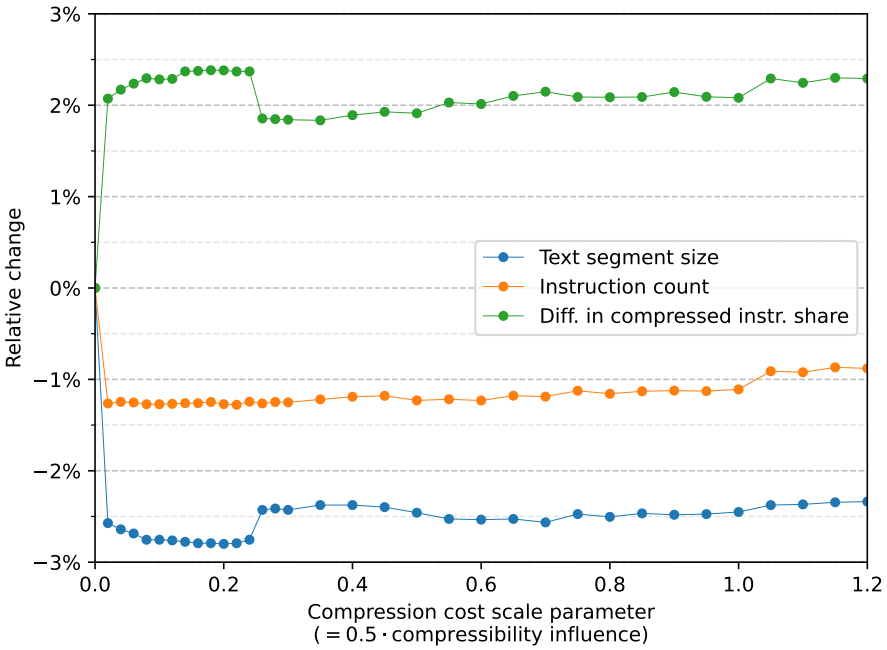


Figure 5.2: Relative changes in text segment size, overall instruction count, and the difference in compressed instruction share for different values of *ccscale* (compression cost scale) and *ci* (compressibility influence) for the `186.crafty` benchmark from SPEC CINT2000. Changes are referenced to code generated with both values set to zero.

the algorithm. As default values for the parameters, we chose values of 0.2 for *ccscale* and 0.4 for *ci*. These values were used to compile all benchmarks examined below.

Compiler Flags All code was compiled with the RISC-V backend for the `RV32IMAC` architecture variant and the `ILP32` ABI specification. Apart from the C extension, this architecture includes the extensions for multiplication and atomic instructions. Notably, it does not contain the F or D extensions as floating-point handling was performed in software.

Unless otherwise indicated, we used LibFirm's highest optimization level `O3` to generate the code for the evaluation. Even though our primary focus for this evaluation is code size, we opted against only using the optimization level `Os` which optimizes for code size: With `Os`, LibFirm currently does not perform any special optimizations to reduce code size apart from disabling inlining while `O3` optimizes the code more extensively.

5.4 Results

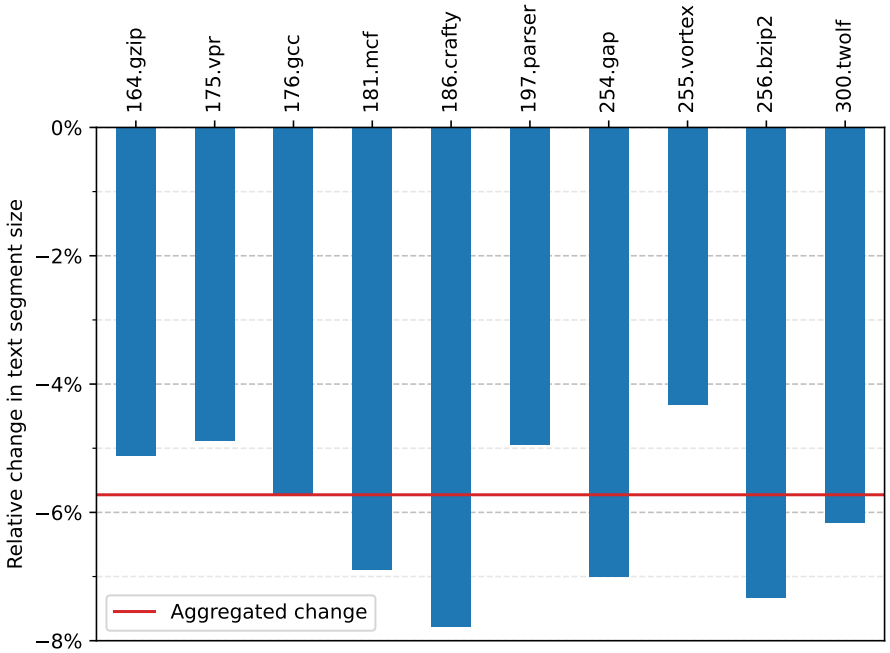
An overall comparison of text segment sizes between the *result*, *static*, and *reference* compiler configurations is shown in table 5.1. Benchmarks from the benchmark sets are aggregated by summing over their text segment sizes. This means that benchmarks in the set are weighted according to their code size instead of every benchmark being assigned equal weight. This avoids a distortion that would occur when a benchmark that is considerably smaller than others has an outlier result. When aggregating all benchmarks from the respective benchmark sets according to this method, our optimization yields a code size reduction of about 5.7 % for the benchmarks from SPEC CINT2000 and 4.2 % for Embench. The relative changes in code size between the *result* and *reference* configuration broken down by benchmark are also shown in figure 5.3. Code size decreases across the single benchmarks range from 4.3 % to 7.8 % for SPEC CINT2000 and from 0 % to 9.6 % for Embench. None of the benchmarks we tested showed an increased total text segment size when compared to the *reference* configuration. The decrease in text segment size when compared to the *static* configuration was about 2.2 % for Embench and 1.6 % for SPEC benchmarks.

The differences in these results for SPEC and Embench benchmarks may be related to the differing code structure across the benchmark sets: As it is focused on software for embedded devices, Embench contains much smaller benchmarks. On average, functions in Embench code are also much shorter: The benchmarks included in SPEC CINT2000 have an average function length of 1042 instructions, compared to only 542 average instructions per function for Embench.

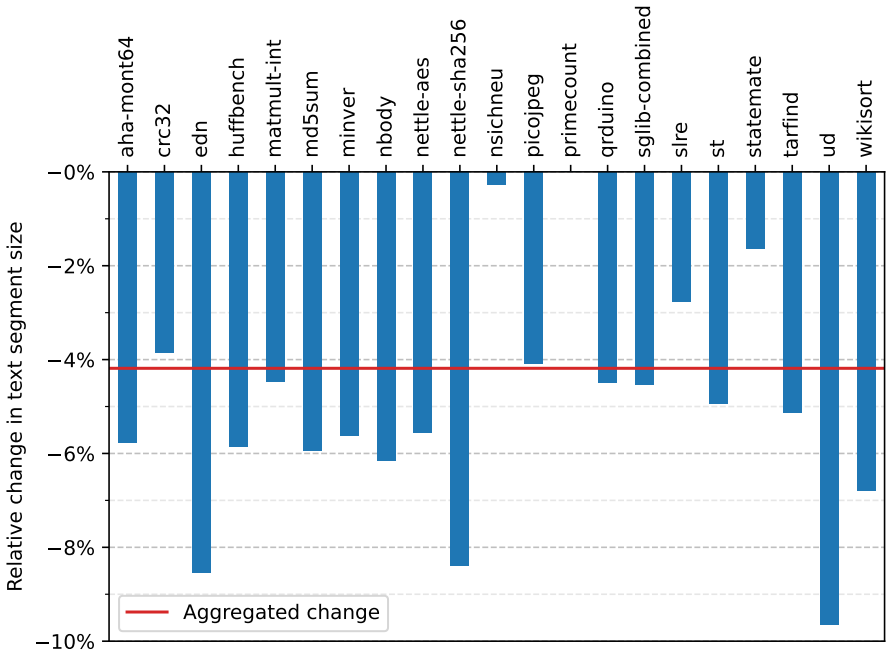
We could not observe an appreciable increase in the number of overall instructions in the compiled programs for either benchmark set. For benchmarks from the SPEC suite, the range for the change in overall instructions was between −1.4 % to 0.2 %, but the aggregated instruction count was still lower than the reference. For the

Table 5.1: Resulting text segment sizes for *result*, *static*, and *reference* compiler configurations for the SPEC CINT2000 and Embench benchmark suites.

	Benchmark	Text segment size			% change vs. ref.	
		reference	static	result	static	result
Embench	aha-mont64	4992	4912	4704	−1.60	−5.77
	crc32	416	416	400	0.00	−3.85
	edn	3376	3264	3088	−3.32	−8.53
	huffbench	3280	3152	3088	−3.90	−5.85
	matmult-int	1072	1056	1024	−1.49	−4.48
	md5sum	1616	1568	1520	−2.97	−5.94
	minver	2560	2480	2416	−3.13	−5.63
	nbody	2080	2016	1952	−3.08	−6.15
	nettle-aes	7776	7712	7344	−0.82	−5.56
	nettle-sha256	6288	6080	5760	−3.31	−8.40
	nsichneu	17 936	17 408	17 888	−2.94	−0.27
	picojpeg	30 864	30 656	29 600	−0.67	−4.10
	primecount	448	448	448	0.00	0.00
	qrduino	11 776	11 584	11 248	−1.63	−4.48
	sglib-combined	25 072	24 320	23 936	−3.00	−4.53
	slre	7520	7424	7312	−1.28	−2.77
	st	2592	2544	2464	−1.85	−4.94
	statemate	11 776	11 664	11 584	−0.95	−1.63
	tarfind	624	608	592	−2.56	−5.13
	ud	1328	1280	1200	−3.61	−9.64
	wikisort	6832	6624	6368	−3.04	−6.79
	Aggregated	150 224	147 216	143 936	−2.00	−4.19
SPEC CINT2000	164.gzip	56 880	55 184	53 968	−2.98	−5.12
	175.vpr	147 520	141 904	140 320	−3.81	−4.88
	176.gcc	2 095 488	2 009 072	1 975 584	−4.12	−5.72
	181.mcf	9280	8784	8640	−5.34	−6.90
	186.crafty	222 288	210 656	204 976	−5.23	−7.79
	197.parser	182 240	180 208	173 232	−1.12	−4.94
	254.gap	502 752	475 552	467 520	−5.41	−7.01
	255.vortex	687 712	664 720	658 000	−3.34	−4.32
	256.bzip2	45 344	42 864	42 016	−5.47	−7.34
	300.twolf	203 184	193 808	190 672	−4.61	−6.16
	Aggregated	4 152 688	3 982 752	3 914 928	−4.09	−5.73



(a) SPEC CINT2000



(b) Embench

Figure 5.3: Relative text segment size change between the compression-aware register allocation (*result* configuration) compared with the previous register allocation implementation (*reference* configuration).

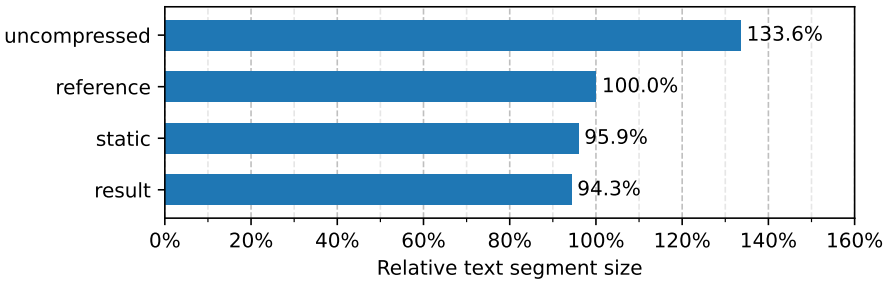


Figure 5.4: Text segment size for benchmarks from SPEC CINT2000 across different cparser configurations: not using any compressed instructions (uncompressed), without compiler adjustments for compression (reference), with static register preference (static), and with the optimization proposed in this thesis (result).

smaller benchmarks from Embench, the span was larger with changes from $-2.7\,\%$ to $1.3\,\%$, still yielding an overall instruction count with our optimization that was about $0.2\,\%$ lower than the reference.

A comparison between all of the compiler configurations from section 5.3 is shown in figure 5.4. It shows the overall aggregated relative text size of all benchmarks from the SPEC CINT2000 suite when compared to the *reference* configuration. Even only using the incidental compression of the *reference* configuration, we can observe a large code size reduction of over $30\,\%$ versus not using any compressed instructions. The reasons for this include move instructions which are always compressible and load/store instructions which often access compressible registers. RISC-V's compression extension is designed in a way that allows many of the compression requirements to be fulfilled by virtue of the calling convention: It prescribes passing the parameters that are passed in registers in the compressible subset. An evaluation of this effect was done in 2019 by Li [23], who performed a usage frequency analysis of the RISC-V register set.

We can also see that only using the static register preference also affords large code size reductions of about $4.1\,\%$, while the full optimization achieves an additional $1.6\,\%$ on top of this. This shows how much the heuristic depends on the register order as we discussed in section 4.3.1, especially without any other information related to compressible instructions.

We also evaluated our optimization with the cparser optimization level `Os` instead of `O3`, yielding a reduction in overall text segment size of about $4.6\,\%$ for SPEC benchmarks and about $3.1\,\%$ for Embench. As expected due to disabled inlining, the overall text segment size in all configurations was considerably lower than with optimization level `O3`: In the *reference* configuration, the code size for the Embench benchmarks was about $22\,\%$ smaller than with `O3`. For SPEC, the difference was around $18\,\%$. The worse result of our compression optimization with this setting is expected: Inlined functions afford more opportunities for finding register assignments

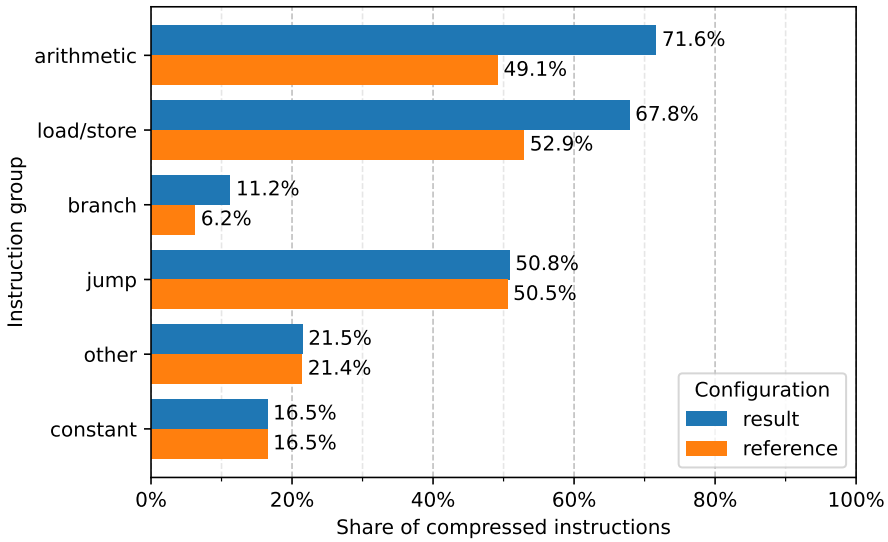


Figure 5.5: Share of compressed instructions among instructions groups, compared for *result* and *reference* configurations, aggregated across all benchmarks from SPEC CINT2000. Move instructions, comparison, and multiplication or division instructions are omitted here.

which make additional instructions compressible that were previously not compressible. If calls are not replaced with inlined functions, many register assignments are already fixed due to the calling convention (which in itself promotes instruction compression).

Instruction groups The share of compressed instructions among the instruction groups we introduced in section 5.3 is shown in figure 5.5 for the code generated for the SPEC benchmarks. Note that groups whose compression ratio did not change were excluded in this plot: This applies to move instructions which are compressed for all registers. Comparison, multiplication, and division instructions which do not have compressed variants are also not shown.

The largest difference in the share of compressed instructions can be seen for arithmetic instructions, followed by load/store and branch instructions. As expected, arithmetic instructions especially exhibit potential for improved compression. A large subset of the instructions in this group are `add` or `addi` instructions, which only have a 2-address requirement, but no register subset requirement. For arithmetic instructions, the improvement in compression ratio for our full compression optimization versus simple static register preference was also highest among the evaluated instruction groups.

Load and store instructions have the advantage of having only register subset requirements that are simple to fulfill. This means they are also more affected by the static register reordering: We can observe that their compressed instruction share

already increases in the *static* configuration, and does not increase further with our full compression optimization.

For branch instructions, only a limited subset is suitable for compression, which also limits the potential for improved compression in this group: As we described in chapter 4, only conditional branches which test for equality or inequality with zero have compressed variants. When examining the frequency of these instructions in our data set for the benchmarks from SPEC, only around 20 % of all branch instructions are `beqz` or `bnez`. This means that the increase of about 5 % in the group of all branch instructions corresponds to an increase of 25 % among potentially compressible branch instructions.

6 Conclusion and Future Work

In this work, we extended LibFirm's SSA-based register allocator to improve code density in architectures with compressed instructions. This is achieved by explicitly taking into account register requirements for compressible instructions in LibFirm's copy optimization heuristic. We also implemented our optimization for the corresponding RISC-V backend.

Evaluation shows the optimization to be successful in improving the code density of code compiled for RISC-V: Aggregating across text segment sizes of all benchmarks we examined, we can observe reductions of around 5.7 % compared to the reference LibFirm implementation or about 1.7 % compared to a simple static preference of compressible registers. Overall, our evaluation shows that relevant improvements in code compression can be achieved by using a compression-aware register allocator. It also shows that the approach we use is suitable for modeling compression requirements of the type used in RISC-V. At the same time, comparing the overall remaining move instructions, we could not observe any indication of appreciable performance degradation of the copy optimization itself. The optimization is specifically designed to not affect register allocation for target architectures without compressed instructions. We believe that by adapting the existing mode of operation of the copy optimization heuristic, our optimization does not substantially increase its code complexity. Overall, we can recommend the optimization also be included in the mainline LibFirm register allocator.

Note that our optimization specifically interfaces with and extends the copy optimization heuristic used in the SSA-based register allocator in LibFirm. This may limit applicability of our approach and implementation in compilers which use a different strategy for register allocation.

Of course, the topic also provides many other opportunities to expand on the approach shown in this thesis. Some ideas which may be of interest in the future are outlined below.

Backend Support Extending the number of backends with support for the optimization is desirable. As we described, it is currently only used with LibFirm's RISC-V backend. However, as discussed in section 2.1, there are other architectures with a similar approach for compressed instructions. The optimization presented in this thesis could also be beneficial for those.

For example, LibFirm does support targeting ARM, but not Thumb (neither the original nor the extension Thumb-2). Thumb-2 is not a straightforward extension to ARM the way the C extension is for RISC-V (where all incompressible instructions

can remain unchanged when targeting RVC). This means supporting Thumb-2 would entail larger changes, e.g. by implementing a backend separate from ARM for it. This is currently not included in LibFirm and means that evaluating compression for this architecture was out of scope for this thesis.

Compressed instructions from the Thumb-2 instruction set also use register restrictions that are covered by the compressibility specifications presented in this thesis. Therefore, our approach is also suitable for implementing compression-aware register allocation for this architecture. We outlined the process for supporting additional backends in section 4.4: This comprises specifying the compressible register set and applicable compression requirements for compressible instructions.

Compression and Instruction Caching Smaller static code size is not the only benefit of improved code compression. For processors which use instruction caching, code size is also related to cache effects: This means that improved compression may result in improved runtime performance of the code. Several other works have remarked on the possibility of this effect [36], [37]. The effects of adding a compressed instruction extension on instruction cache misses have also been previously investigated [22].

The impact of this may be subtle and highly dependent on the concrete benchmarks and architecture parameters chosen to evaluate it. Therefore, evaluating the difference in runtime performance between code generated with compression-unaware and compression-aware register allocation was not in scope for this thesis.

For many current architectures following the RISC philosophy, compressed instructions are key to achieving code densities that are competitive with other architectures. However, we believe their handling in compiler backends still provides plenty of potential for improved compression. Our work shows that including dynamic handling of compression requirements that goes beyond statically preferring registers is effective in improving code compresssion in a compiler that previously did not consider code compression. This suggests that register allocators in other compilers may also benefit from more explicit modeling of compression requirements.

Bibliography

[1] V. M. Weaver and S. A. McKee, "Code density concerns for new architectures", in *2009 IEEE International Conference on Computer Design*, Lake Tahoe, CA, USA: IEEE, Oct. 2009, pp. 459–464. DOI: 10.1109/ICCD.2009.5413117.

[2] "ARM architecture reference manual: Thumb-2 supplement". Issue D. (Dec. 2005), [Online]. Available: https://developer.arm.com/documentation/ddi0308/ (visited on 12/15/2021).

[3] "MIPS32 architecture for programmers: MIPS16e2 application-specific extension technical reference manual". Revision 01.00. (Apr. 26, 2016), [Online]. Available: https://www.mips.com/products/architectures/ase/ase16e/ (visited on 12/15/2021).

[4] "MIPS architecture for programmers, Volume II-B: microMIPS32 instruction set". Revision 6.05. (Jun. 6, 2016), [Online]. Available: https://www.mips.com/downloads/the-micromips32-instruction-set-v6-05/ (visited on 12/15/2021).

[5] A. Waterman and K. Asanovic, Eds. "The RISC-V instruction set manual; Volume I: Unprivileged ISA". Version 20191213. (Dec. 13, 2019), [Online]. Available: https://riscv.org/technical/specifications/ (visited on 12/15/2021).

[6] A. Waterman, "Improving energy efficiency and reducing code size with RISC-V Compressed", M.S. thesis, EECS Department, University of California, Berkeley, May 2011.

[7] M. Braun, S. Buchwald, and A. Zwinkau, "Firm – a graph-based intermediate representation", Karlsruhe Institute of Technology, Tech. Rep. 35, 2011.

[8] G. J. Chaitin, "Register allocation & spilling via graph coloring", *ACM SIGPLAN Notices*, vol. 17, no. 6, pp. 98–101, Apr. 1982. DOI: 10.1145/872726.806984.

[9] B. R. Nickerson, "Graph coloring register allocation for processors with multi-register operands", *ACM SIGPLAN Notices*, vol. 25, no. 6, pp. 40–52, Jun. 1, 1990. DOI: 10.1145/93548.93552.

[10] L. George and A. W. Appel, "Iterated register coalescing", *ACM Transactions on Programming Languages and Systems*, vol. 18, no. 3, pp. 300–324, May 1, 1996. DOI: 10.1145/229542.229546.

[11] J. Park and S.-M. Moon, "Optimistic register coalescing", *ACM Transactions on Programming Languages and Systems*, vol. 26, no. 4, pp. 735–765, Jul. 1, 2004. DOI: 10.1145/1011508.1011512.

[12] P. Briggs, "Register allocation via graph coloring", Ph.D. dissertation, Rice University, 1992.

[13] S. Hack and G. Goos, "Optimal register allocation for SSA-form programs in polynomial time", *Information Processing Letters*, vol. 98, no. 4, pp. 150–155, May 31, 2006. DOI: 10.1016/j.ipl.2006.01.008.

[14] S. Hack, *Register allocation for programs in SSA Form*. Universitätsverlag Karlsruhe, 2007, 123 pp. DOI: 10.5445/KSP/1000007166.

[15] M. Smith and G. Holloway, "Graph-coloring register allocation for irregular architectures", 2000.

[16] J. Runeson and S.-O. Nyström, "Retargetable graph-coloring register allocation for irregular architectures", in *Software and Compilers for Embedded Systems*, A. Krall, Ed., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2003, pp. 240–254. DOI: 10.1007/978-3-540-39920-9_17.

[17] B. Scholz and E. Eckstein, "Register allocation for irregular architectures", in *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems*, ser. LCTES/SCOPES '02, New York, NY, USA: Association for Computing Machinery, Jun. 19, 2002, pp. 139–148. DOI: 10.1145/513829.513854.

[18] A. Krishnaswamy and R. Gupta, "Profile guided selection of ARM and Thumb instructions", *ACM SIGPLAN Notices*, vol. 37, no. 7, pp. 56–64, Jun. 19, 2002. DOI: 10.1145/566225.513840.

[19] ——, "Efficient use of invisible registers in Thumb code", in *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*, Nov. 2005, pp. 30–42. DOI: 10.1109/MICRO.2005.19.

[20] T. J. Edler von Koch, I. Böhm, and B. Franke, "Integrated instruction selection and register allocation for compact code generation exploiting freeform mixing of 16- and 32-bit instructions", in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '10, New York, NY, USA: Association for Computing Machinery, Apr. 24, 2010, pp. 180–189. DOI: 10.1145/1772954.1772980.

[21] J.-H. Lee, S.-M. Moon, and J. Park, "Region-based dual bank register allocation for reduced instruction encoding architectures", *Microprocessors and Microsystems*, vol. 55, pp. 26–43, Nov. 1, 2017. DOI: 10.1016/j.micpro.2017.09.005.

[22] B. C. Lopes, L. Ecco, E. C. Xavier, and R. Azevedo, "Design and evaluation of compact ISA extensions", *Microprocessors and Microsystems*, vol. 40, pp. 1–15, Feb. 1, 2016. DOI: 10.1016/j.micpro.2015.09.010.

[23] P. Li, "Reduce static code size and improve RISC-V compression", M.S. thesis, EECS Department, University of California, Berkeley, Jun. 2019.

[24] M. Perotti, P. D. Schiavone, G. Tagliavini, *et al.*, "HW/SW approaches for RISC-V code size reduction", 2020. DOI: 10.3929/ETHZ-B-000461404.

[25] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation", in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar. 2004.

[26] *LLVM greedy register allocator source code*, lines 769–779, (Dec. 13, 2021), [Online]. Available: https://github.com/llvm/llvm-project/blob/657adcb0779d2d119737f2ca557a87456024a5ac/llvm/lib/CodeGen/RegAllocGreedy.cpp (visited on 12/15/2021).

[27] *LLVM ARM register definitions*, lines 78–99, (Dec. 7, 2021), [Online]. Available: https://github.com/llvm/llvm-project/blob/63eb7ff47de5df48b6bc0cf0a6d3d17022634151/llvm/lib/Target/ARM/ARMRegisterInfo.td (visited on 12/15/2021).

[28] *LLVM RISC-V register definitions*, lines 71–116, (Dec. 10, 2021), [Online]. Available: https://github.com/llvm/llvm-project/blob/5861cf77da4f7d235d435dd8fb89b100d1698112/llvm/lib/Target/RISCV/RISCVRegisterInfo.td (visited on 12/15/2021).

[29] O. Stannard. "Thumb2: Favor R4-R7 over R12/LR in allocation order when opt for minsize". Implemented in https://reviews.llvm.org/rG830b20344bdd3f8790bb913b9e699a3fa5f446f6. (Jul. 3, 2019), [Online]. Available: https://reviews.llvm.org/D30324 (visited on 12/15/2021).

[30] D. Green. "Alter the register allocation order for optsize on Thumb2". Implemented in https://reviews.llvm.org/rG6a858a94250ecf96f849d2b9108ebacc86be98a9. (Jan. 23, 2019), [Online]. Available: https://reviews.llvm.org/D56008 (visited on 12/15/2021).

[31] R. C. Lozano, M. Carlsson, G. H. Blindell, and C. Schulte, "Combinatorial register allocation and instruction scheduling", *ACM Transactions on Programming Languages and Systems*, vol. 41, no. 3, 17:1–17:53, Jul. 2, 2019. DOI: 10.1145/3332373.

[32] C. Schulte and R. C. Lozano, "Unison: Optimization technology for optimizing compilers", in *Ericsson's Program Analysis Workshop*, (Kista, Sweden), Apr. 2018.

[33] "Embench: Open benchmarks for embedded platforms". (2021), [Online]. Available: https://github.com/embench/embench-iot (visited on 12/15/2021).

[34] J. Bennett, C. Garlati, G. Madhusudan, T. Mudge, and D. Patterson, "Embench: A free benchmark suite for embedded computing from an academic-industry cooperative (towards the long overdue and deserved demise of dhrystone)", in *RISC-V Workshop Zurich*, Zurich, Switzerland, Jun. 12, 2019.

[35] Standard Performance Evaluation Corporation. "SPEC CPU2000". V1.3.1. (Nov. 20, 2006), [Online]. Available: `https://www.spec.org/cpu2000/` (visited on 12/15/2021).

[36] H. Lozano and M. Ito, "Increasing the code density of embedded RISC applications", in *2016 IEEE 19th International Symposium on Real-Time Distributed Computing (ISORC)*, May 2016, pp. 182–189. DOI: `10.1109/ISORC.2016.33`.

[37] P. Steenkiste, "The impact of code density on instruction cache performance", in *Proceedings of the 16th Annual International Symposium on Computer Architecture*, ser. ISCA '89, New York, NY, USA: Association for Computing Machinery, Apr. 1, 1989, pp. 252–259. DOI: `10.1145/74925.74954`.

Erklärung

Hiermit erkläre ich, Maximilian Stemmer-Grabow, dass ich die vorliegende Masterarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Ort, Datum

Unterschrift

A Appendix

A.1 RISC-V Instruction Groups

The instruction groups used in the evaluation and the instructions included in them are listed in table A.1.

Table A.1: Instruction groups and included instructions.

Instruction group	Included instructions
Load/store	`lw`, `sw`, `flw`, `fld`, `fsd`, `fsw`, `lb`, `lbu`, `lh`, `lhu`, `sb`
Arithmetic	`add`, `addi`, `and`, `andi`, `neg`, `not`, `or`, `ori`, `sll`, `slli`, `sra`, `srai`, `srl`, `srli`, `sub`, `xor`, `xori`
Compare	`slt`, `slti`, `sltiu`, `sltu`, `snez`, `seqz`
Mult/div	`div`, `divu`, `mul`, `mulh`, `mulhu`, `rem`, `remu`
Branch	`beq`, `beqz`, `bge`, `bgeu`, `bgez`, `bgtz`, `blez`, `blt`, `bltu`, `bltz`, `bne`, `bnez`
Jump	`j`, `jal`, `jalr`, `jr`, `ret`
Move	`mv`
Constant generation	`li`, `lui`, constant generation instructions that include unlinked references in object files (decompiled to `mv` instructions)
Others	All other instructions