Register Allocation for Compressed ISAs in LLVM

Andreas Fried
Karlsruhe Institute of Technology
Karlsruhe, Germany
andreas.fried@kit.edu

Maximilian Stemmer-Grabow
Karlsruhe Institute of Technology
Karlsruhe, Germany

Julian Wachter
Karlsruhe Institute of Technology
Karlsruhe, Germany

Abstract

We present an adaptation to the LLVM greedy register allocator to improve code density for compressed RISC ISAs.

Many RISC architectures have extensions defining smaller encodings for common instructions, typically 16 rather than 32 bits wide. However, these instructions typically cannot access all the processor's registers, and might only have room to specify two registers even for binary operations.

When a register allocator is aware of these restrictions, it can analyze the *compressibility* of instructions and assign registers in such a way that as many instructions as possible can use the smaller encoding.

We adapted four aspects of the LLVM greedy register allocator in order to enable more compressed instructions: 1. Prioritize virtual registers with many potentially compressible instructions for earlier assignment. 2. Select registers so that the number of compressed instructions is maximized. 3. Take compressibility into account when deciding which virtual registers to spill. 4. Weigh more register copies against more opportunity for compression.

We evaluate our techniques using LLVM's RISC-V backend. In the SPEC2000 and SPEC2006 benchmarks, our register allocator produces between 0.42 % and 6.52 % smaller binaries. In the geometric mean, binaries become 1.93 % smaller. We see especially large improvements on some floating-point-heavy benchmarks.

Binaries compiled for better compression show changes in their execution time of at most ±1.5 %. We analyze these against LLVM's spilling metrics, and conclude that the effect is probably not systemic but a random fluctuation in the register allocation heuristic.

CCS Concepts: • **Software and its engineering → Compilers**; • **Computer systems organization → Reduced instruction set computing**.

CC '23, February 25–26, 2023, Montréal, QC, Canada

ACM ISBN 979-8-4007-0088-0/23/02…$15.00
https://doi.org/10.1145/3578360.3580261

Keywords: register allocation, compressed instruction sets, RISC-V, LLVM

ACM Reference Format:
Andreas Fried, Maximilian Stemmer-Grabow, and Julian Wachter. 2023. Register Allocation for Compressed ISAs in LLVM. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction (CC '23), February 25–26, 2023, Montréal, QC, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3578360.3580261

1 Introduction

One of the goals of RISC architectures is to have a simple instruction encoding scheme [7]. They usually have few possible instruction formats, and all instructions have the same size. In many RISC architectures, instructions are 32 bits wide.

However, this leads to poor *code density*, i.e., a given piece of high-level code yields a larger binary in RISC architectures than in CISC architectures [31].

For example, consider reading from an array of 4-byte values (`x = a[i]`). In x86, this can be expressed using the *address mode* feature of the `mov` operation (`a` in `%rax`, `i` in `%rbx`, `x` in `%ecx`):

```
movl (%rax, %rbx, 4), %ecx
```

The same functionality needs three instructions in RISC-V (`a` in `a0`, `i` in `a1`, `x` in `a2`):

```
slli a1, a1, 2      shift i left by 2 bits
add a1, a1, a0      add base pointer
lw a2, 0(a1)        load
```

Where the x86 instruction takes 3 bytes to encode, each of the RISC-V instructions takes 4 bytes, i. e. 12 bytes in total. Of course, the difference is not as pronounced in a full program, but Weaver and McKee show that RISC programs are indeed significantly larger than their CISC counterparts [31].

Lower code density, in turn, leads to more pressure on the memory system, especially instruction cache misses [26].

To improve their code density, some RISC architectures have introduced *compressed instruction sets* such as ARM Thumb [12], MIPS16e [28], and RISC-V's C extension (RVC) [1]. These offer smaller encodings of the most common instructions. Usually, these smaller encodings are only 16 bits wide.

Indeed, in the example above, all instructions are compressible in RVC, so the example only takes up 6 bytes with compressed instructions.

Since compressed instruction sets have less encoding space available to them, they are necessarily more restrictive than their uncompressed counterparts. These restrictions usually are:

- not all kinds of instructions are available in compressed form, e. g., no multiplication or division
- Smaller immediate fields, e. g., for load/store offsets
- Only a subset of registers is accessible, called the *compressible registers*
- Compressed arithmetic instructions are in *two-address* form, i.e., they only have two operands, and encode an operation of the form $x \leftarrow x \circledast y$ rather than $z \leftarrow x \circledast y$.

The first two of these points are of little interest to the compiler: If the instruction called for by the program is not compressible, there is nothing gained by using several compressed instructions in place of one uncompressed one. Likewise, there is no way to use a smaller constant than given in the program outside of some specific circumstances. Nevertheless, Revill and Blackmore were able to exploit some of those circumstances [19].

The latter two points listed above are of a different nature: They show that the attainable compression relies on the decisions of the register allocator. If the register allocator is aware of the properties of the compressed instruction set, it can produce smaller binaries while using the same instructions.

Register allocation is one of the most important compiler optimizations [11], and consequently has been well studied. Modern register allocators stem from either the Chaitin-Briggs *graph-coloring* register allocator [4–6], or the *linear-scan* allocator by Poletto and Sarkar [18].

Both of these allocators have been extended to support *register restrictions* for instructions that cannot access all registers, and two-address instructions. For graph-coloring allocators, both Smith and Holloway [23], and Runeson and Nyström [20] presented such extensions. Mössenböck and Pfeiffer extended linear-scan allocators in the same way [16].

Our goal, however, is different: To maximize the number of compressed instructions we can emit, the register allocator should be able to handle register *preferences*, not restrictions. There is no point in forcing an instruction to use a compressible register if that means adding extra instructions elsewhere.

Existing techniques for compression-aware register allocation are either rather simple, or impracticable for larger programs (see sections 2.4 and 2.5). Our goal is therefore to build a more comprehensive compression-aware register allocator, while keeping the register allocation and performance overhead low.

This paper is organized as follows: In the next section, we present the necessary basics, including the RISC-V compressed ISA, and the LLVM greedy register allocator. In section 3, we first introduce the general principles of our work, followed by a discussion of our adaptations to the register allocator in section 4. We then evaluate our work in section 5, both for code size and for execution time.

2 Basics

In this section, we will introduce the foundations required for our work. We first introduce compressed instruction set architectures (ISAs), and then present the RISC-V ISA with its C extension for compressed instructions. We also briefly examine other compressed ISAs in order to demonstrate that our work is also applicable to other instruction sets. Next, we discuss the state of the art in the LLVM register allocator, and finally we take a look at previous work related to code compression.

Terminology. Throughout this text, we will frequently make reference to both individual instances of instructions, as well as to general types of instructions. For clarity, we will refer to instances of instructions (e. g. `add x1, x2, x3`) as *instructions*, and to types of instructions (e. g. `add`, `sub`, `ld`) as *operations*.

Almost all ISAs we cover have 32-bit and 16-bit wide encodings. We will also call the 32-bit encodings *uncompressed encodings* and the 16-bit wide ones *compressed encodings*. When an operation has one or more compressed encodings, we call it a *compressible operation*. When an instruction can actually use a compressed encoding given its operands, we call that a *compressible instruction*.

2.1 Compressed Instruction Sets

First of all, we need to distinguish two kinds of compressed ISAs: In older *compressed-only* ISAs such as ARM Thumb-1 [12] and MIPS16e [28], all operations only have compressed encodings. Therefore, most operations can only access compressible registers. Only moves (for MIPS16e) or moves and additions (for Thumb-1) can access all registers.

On the other hand, *hybrid* ISAs offer additional non-compressed encodings for compressible operations, or compressed encodings can be interleaved with those from the standard ISA. Most modern general-purpose compressed ISAs are of this kind.

Compression in hybrid ISAs is transparent to the compiler. It does not need to explicitly select compressed encodings. Instead, the assembler checks whether instructions are compressible and uses compressed encodings where possible. In order to achieve good compression, the compiler just has to make sure that as many instructions as possible fulfill the requirements to use their compressed encodings.

2.2 The RISC-V ISA

In this section, we present an overview of the aspects of the RISC-V architecture relevant to our work. For a full description, see the RISC-V specification [1].

The RISC-V base ISA defines a set of 32 general-purpose registers, either 32 or 64 bits wide. These are named `x0` through `x31`.

The instructions are in one of four formats, all 32 bits wide [1, fig. 2.2]. These formats form instructions in a typical RISC style with the following properties:

- Register fields are 5 bits wide, and thus can address all registers.
- Binary arithmetic instructions (format R) are in three-address form.
- Immediates for arithmetic instructions (formats I and S) are 12 bits wide.
- Immediates for jumps (format U) are 20 bits wide.

In contrast, the C extension of the RISC-V ISA (RVC) defines nine 16-bit wide instruction formats [1, tab. 16.1]. Comparing the compressed formats to the uncompressed ones, we can see the following restrictions:

- Register fields in formats CIW, CL, CS, CA, and CB are 3 bits wide, and can therefore address 8 registers.
- Arithmetic instructions (formats CA, CI, CR) are in two-address form.
- Immediates for arithmetic instructions (formats CI or CB) are 6 bits wide.
- Load/Store offsets (formats CL, CS) are 5 bits wide.
- Immediates for jumps (format CJ) are 11 bits wide.

Since we are interested in the register restrictions in particular, we will now discuss them in more detail. The eight registers addressable in compressed instructions are registers `x8` through `x15`. The standard RISC-V ABI assigns these roles to them:

`x8`	`s0`	callee-save/frame pointer
`x9`	`s1`	callee-save
`x10–x15`	`a0–a5`	argument/return values, temporary

These registers are frequently used in many programs anyway. Thus, programs can already profit from compression even if the register allocator is not aware of the conditions for compression.

On systems with floating-point support, the C extension additionally defines compressible load and store operations for floating-point registers `f8` through `f15`. However, floating-point arithmetic operations are not compressible.

2.3 Other Compressed Instruction Sets

Having discussed RVC in some detail, we next briefly introduce other relevant compressed ISAs. We will give a quick overview of them in order to motivate our technique, and to show that its principles are widely applicable. In all ISAs, register-to-register moves are given special treatment, and can access all registers. In addition, many ISAs have special encodings for the most important operations to allow them to access more registers.

Table 1 summarizes the features found in the different ISAs.

2.3.1 ARM Thumb-2.

Compressed instructions can access 8 out of 16 registers. Binary arithmetic instructions are in two-address rather than three-address form. They also cannot use the "flexible second operand" to shift or rotate the second argument. Addition and subtraction use a special encoding, so that they can use three-address form. In addition, there is an encoding for addition in two-address form that can address all registers. Immediate addition can either use two-address form or one-address form with a larger immediate. [2]

2.3.2 microMIPS.

Compressed instructions can access 8 out of 32 registers, with some instructions implicitly accessing the stack pointer (`sp`), global pointer (`gp`), or return address (`ra`) registers. Again, addition is given special treatment by using three-address form, while other arithmetic uses two-address form. In addition, loads and stores accessing the stack or the global variable area can use all registers. [29]

2.3.3 nanoMIPS.

Most compressed instructions can access 8 out of 32 registers, again with binary arithmetic in two-address form. However, some compressed encodings feature 4-bit register fields, which can access 16 out of 32 registers. These are the two-address form of addition, loads and stores with small immediates, and multiplication (which does not even have an encoding with 3-bit register fields). Loads and stores accessing the stack again can use all registers. [30]

2.3.4 PowerPC VLE.

This compressed ISA is less restrictive than the others, with all compressed encodings able to access 16 out of 32 registers in two-address form. On the other hand, it does not afford any special treatment to operations other than moves. [9]

From this overview, we can draw some conclusions concerning the features our technique needs to support:

- All ISAs except nanoMIPS have one size for compressed register fields, and therefore one subset of compressible registers. In the following, we will discount this specialty of nanoMIPS since it complicates the register allocator for the sake of just one architecture.
- It is equally important for the register allocator to fulfill the *two-address requirement*, as well as to assign compressible registers to all operands.
- Given that most compressed ISAs are somewhat non-orthogonal, we need precise information about which operations are compressible under which circumstances. These circumstances might be outside the register allocator's control, such as the size of immediates.

2.4 The LLVM Greedy Register Allocator

The greedy register allocator was introduced with LLVM version 3.0, replacing an earlier linear scan allocator. Being a production-quality register allocator, it is quite complex, and

Table 1. Register fields in 32-bit and 16-bit encodings of hybrid compressed ISAs. "n/b" refers to n register fields, each b bits wide, e. g. 3/5 denotes three 5-bit register fields. See sections 2.2 and 2.3.1 to 2.3.4 for details of the exceptions.

	Encodings		
ISA	32-bit	16-bit	16-bit exceptions
RVC	3/5	2/3	`add`: 2/5 `addi`, `sll`: 1/5 `sp`-relative: 1/5
ARM Thumb-2	3/4	2/3	`addi`, 8-bit imm.: 1/3 `add`: 3/3 or 2/4 `sub`: 3/3
microMIPS	3/5	2/3	`addu`: 3/3 `sp`, `gp`-relative: 1/5
nanoMIPS	3/5	2/3	`addu`: 3/3 or 2/4 `mul`: 2/4 `lw`, `sw`, small offset: 2/4 `sp`-relative: 1/5
PowerPC VLE	3/5	2/4	—

we can only give an overview of the components required for our work here. Braun provides a more comprehensive tutorial [3].

The register allocator works on one function at a time, represented as a list of assembler instructions whose operands are virtual registers known as *live-intervals*.

A live-interval represents the duration for which a value must be kept in a register or in memory. Live-intervals are not necessarily in SSA form, they may have multiple definitions on separate control flow paths. A live-interval does not necessarily appear as a single interval on the list of instructions but may be spread over several *live-ranges* in non-contiguous basic blocks. A live-interval's *size* is the sum of the number of instructions its live-ranges span.

The greedy register allocator uses an iterative technique centered around a priority queue of live-intervals.

The priority of a live-interval in the queue is defined by a number of criteria. For our work, only three of these are relevant:

1. *Global* live-intervals (those spanning multiple basic blocks) have higher priority than local ones.
2. Global intervals are prioritized by size.
3. Local intervals are prioritized top-to-bottom in their basic block.

Initially, all live-intervals of the function are enqueued. Then, in each iteration, the register allocator takes the highest-priority live-interval out of the queue and tries to assign it to a register.

If a register is free for the duration of the live-interval, the register allocator assigns the live-interval to that register. Otherwise, it first tries to *evict* another live-interval (i. e. unassign its register) it considers cheaper to spill. Next, it tries to *split* the live-interval in order to reduce the constraints placed on it. Live-intervals that were evicted or split re-enter the queue, and the register allocator re-assigns them later.

If all else fails, the register allocator *spills* the live-interval, replacing its definitions and uses with spills and reloads respectively.

We will explain these steps in more detail in section 4 as we present our modifications to the register allocator.

2.4.1 Existing Mechanisms for Compressed ISAs.

Normally, the register allocator chooses the first free register for a live-interval, with two exceptions: Firstly, the live-interval might have a *hint* to use a certain register. This is the case if one of the instruction operands represented by the interval is restricted to a certain register, or if the interval is *copy-related*[1] to another live-interval, and that live-interval is already assigned.

Secondly, the register allocator prefers registers with a lower *cost-per-use*. Cost-per-use is a mechanism to mark some registers as generally preferable over others. In practice, cost-per-use is 0 for most registers, 1 for the first use of a callee-save register (CSR), and 1 for every use of certain registers defined by the architecture.

This cost-per-use mechanism is how the register allocator tries to avoid using non-compressible registers: Non-compressible registers have a cost-per-use of 1, so that compressible registers are preferred if they are still free.

In addition, Zhao and Stannard found that preferring compressible CSRs over non-compressible caller-save registers improved compression for ARM Thumb-2 [32]. This change reduced code size by up to 1.48 %, with 0.1 % reduction on average, and a 0.1 % increase in the worst case. However, the performance impacts of this change were not evaluated.

The order in which the greedy register allocator processes live-intervals also has an effect on compression. It allocates large live-intervals first, when more registers are still free. Therefore, large live-intervals are more likely to be assigned to a compressible register. Since they also likely have more uses, assigning a large live-interval to a compressible register likely enables more instructions to be compressed.

Finally, LLVM contains a specialized optimization phase for the RISC-V target aimed at reducing immediate offsets in load/store instructions. Various LLVM contributors reported savings between 0.10 % and 0.76 % using this optimization [19].

[1] Two live-intervals are copy-related if there is a copy instruction which uses one live-interval and defines the other.

2.5 Related Work

Besides the mechanisms built into LLVM we have just seen, there are several more comprehensive approaches to support compressed instruction sets.

One approach is to translate the register allocation problem into a constraint system, and also encode compressibility as part of the objective function. The general method was introduced by Scholz and Eckstein [21]. They encoded register allocation as a PBQP problem but only considered register requirements, not preferences as required for compression.

Later, Lozano et al. applied the principle to compressed ISAs within the Unison project [14, 22]. Their work models register allocation and instruction scheduling as a single optimization problem, which also includes 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, these constraint-based register allocators do not scale to full programs. Lozano et al. report that their system "scales to medium-sized functions of up to 1,000 instructions".

Other work has taken a hardware/software co-design approach, and proposed extensions to the instruction set along with new compiler techniques.

Edler von Koch et al. presented an approach for code generation on the *ARCompact* ISA, a hybrid compressed ISA focusing on embedded use cases [8]. They introduce a method for "feedback-guided code generation", which uses multiple compiler passes to improve code compression. The first compiler pass is used to annotate the IR with information on opportunities for compression, and the following pass uses that information to avoid compressed instructions where this would introduce additional move instructions or spills.

There is also earlier work dealing with compressed-only ISAs. Krishnaswamy and Gupta presented an approach to optimize generation of mixed ARM and Thumb-1 code [13]. Their approach is rather coarse however, generating only Thumb-1 or only ARM instructions in any given function. Selection of whether to use compressed instructions is based on heuristic analysis of the code under consideration. In addition, specific patterns in ARM functions are replaced with Thumb-1 code surrounded by mode-switching jumps.

3 Goals of Compressed Register Allocation

The key idea behind our work is to focus on using compressible registers where they provide the most benefit, rather than generally preferring their use everywhere. We will illustrate this point with two examples using RVC. The question is, which of the symbolic values/live-intervals (denoted `%x`, `%y`, etc.) should be placed in compressible registers.

We assume for both examples that all values are dead at the end of the snippets, and that the load/store offsets are small enough for the compressed encoding (≤ 248 for 64-bit values).

First, consider the following C snippet, where `x`, `y`, `p->x`, and `p->z` are 64-bit integers, and `y` has been defined previously.

```
x = p->x;
p->z = x | ((x - y) & 0xff);
```

This might be compiled as follows:

```
ld %x, x_offset(%p)
sub %t1, %x, %y
andi %t2, %t1, 0xff
or %z, %x, %t2
sd %z, z_offset(%p)
```

Assume that external restrictions have forced `%y` into a non-compressible register. This means that the `sub` instruction (l. 2) is no longer compressible, regardless of the register assigned to `%t1` and `%x`. The immediate of the `andi` instruction (l. 3) does not fit in the compressed encoding, so that will be uncompressed, too.

On the other hand, all other instructions are compressible given a suitable register assignment. In particular, `%x` dies at the `or` instruction (l. 4), so `%z` can be assigned to the same register to fulfill the two-address requirement.

For the register allocation, this means that `%t1` should *not* be assigned to a compressible register, while all other values should be assigned to compressible registers, with `%x` and `%z` being assigned to the same register.

The second example illustrates a particular issue with floating-point code. Consider this C snippet, with `a`, `b`, and all fields of `q` having type `double`:

```
a = q->a; b = q->b;
q->c = (a + b) * (a - b);
```

This might be compiled as follows:

```
fld %a, a_offset(%q)
fld %b, b_offset(%q)
fadd.d %t1, %a, %b
fsub.d %t2, %a, %b
fmul.d %c, %t1, %t2
fsd %c, c_offset(%q)
```

The important point here is that only floating-point loads and stores (`fld`/`fsd`) are compressible, but arithmetic is not. Our goal is therefore to assign the "inputs" and "outputs" of the computation (values `%a`, `%b`, and `%c`) to compressible registers. The intermediate values `%t1` and `%t2` can never profit from compression, so compressible registers would be wasted on them.

Of course we must also assign `%q` to a compressible register, or all three load/store instructions cannot be compressed. If we cannot find a compressible register for `%q`, there is again no point in assigning `%a`, `%b`, or `%c` to compressible registers.

3.1 Basic Concepts

Keeping in mind the examples above, we can now define more concretely what we should care about in a compression-aware register allocator.

Potentially Compressible Instructions. We say that an instruction is *potentially compressible* if the following conditions hold:

1. the operation is compressible,
2. the immediates fit in the compressed encoding or are unknown,
3. all operands requiring compressible registers are either assigned to compressible registers or unassigned,
4. the two-address requirement (if applicable) is fulfilled or one of the relevant operands is unassigned.

Thus, if an instruction is potentially compressible, its unassigned operands can still be assigned in such a way that the instruction is compressible.

We found that these optimistic assumptions (that immediates are small enough, and suitable registers will be found), are frequently justified and lead to better compression. Unknown immediates often refer to stack slots, whose offsets usually fit in the compressed encoding.

Using potentially compressible instructions, we can keep track of which instructions are still worth optimizing. We write $PC(i)$ for the predicate that instruction i is potentially compressible.

Compressibility of Live-Intervals. In the same way that we evaluate the compressibility of instructions, we also want to define a measure of compressibility for live-intervals. For this, we first define $UD(L)$ as the set of instructions that have the live-interval L as one of their operands, both as use and as definition.

We can then define L's compressibility C as the number of potentially compressible uses and definitions of the interval:

$$C(L) = |\{i \mid i \in UD(L) \wedge PC(i)\}|$$

In order to evaluate register choices, we are also interested in a live-interval's compressibility under the assumption that it is assigned to a certain register r.

$$C(L, r) = |\{i \mid i \in UD(L) \wedge PC(i[L \mapsto r])\}|$$

In order to compare the compressibility of live-intervals independent of their size, we also define the *relative compressibility* (C_{rel}) as the share of potentially compressible instructions in a live-interval.

$$C_{rel}(L) = \frac{C(L)}{|UD(L)|}$$

Note that the compressibility of a live-interval can decrease as other live-intervals are assigned to registers, and instructions are no longer potentially compressible.

4 Adaptations to the Register Allocator

Our goal is to achieve better code compression while not sacrificing run-time performance. Therefore, we do not interfere with the register allocator's basic function. We only adjust its heuristics in four key aspects, which we will now examine in turn.

4.1 Live-Interval Priority

We have already pointed out that prioritizing large live-intervals is likely to improve compression. However, this does not take into account the varying relative compressibility of live-intervals.

We therefore want to decrease the priority of less compressible global live-intervals. We choose a simple linear relationship for this: If the live-interval L has no potentially compressible instructions ($C_{rel}(L) = 0$), its priority $P(L)$ should be decreased by a factor of α. If all of L's instructions are potentially compressible ($C_{rel}(L) = 1$), its priority should be its size $S(L)$, as before. This yields the following formula:

$$P(L) = S(L) \cdot ((1 - \alpha) + \alpha C_{rel}(L))$$

We call α the *priority influence*. By varying α between 0 and 1, we can choose the importance of larger versus more compressible live-intervals.

However, we do not modify the priority of local live-intervals. This is because local intervals have a single definition, i. e., they are in SSA form. For live-intervals in SSA form, top-to-bottom allocation is known to produce an optimal register allocation [10], which we do not want to interfere with.

4.2 Register Selection

The original register allocator does not usually choose a specific register since they are all equal in terms of performance. If there are no hints associated with the live-interval, the register allocator simply chooses the first free register with zero cost-per-use.

In contrast, the specific choice of register is important to code compression. On one hand, some of the instructions in the live-interval may need compressible registers as operands to be themselves compressible (this concerns the majority of compressible operations). On the other hand, we have many more preferences for single registers in order to fulfill the two-address requirements of compressible operations.

To allocate a register for a live-interval L, we therefore check each available register and compute L's compressibility if it were assigned to it. We also take into account that we want to avoid using a CSR for the first time. If there are still free registers that are not unused CSRs, we choose among those. Whether or not we need to use a CSR, we then select the register giving the highest compressibility. Algorithm 1 summarizes our register selection technique.

Algorithm 1 Register selection

```
function Select(L: LiveInterval) → Register
    cand ← []
    for each register r do
        if r is free during L then
            c ← C(L, r)
            u ← r is an unused CSR
            cand.Append((r, c, u))
    if any (_, _, false) ∈ cand exists then
        (r*, _, _) ← (r, c, false) ∈ cand with max. c
    else if cand ≠ [] then
        (r*, _, _) ← (r, c, true) ∈ cand with max. c
    else
        r* ← none
    return r*
```

4.3 Live-Interval Eviction

When the register allocator cannot assign a live-interval L to any register, it can choose to *evict* all live intervals interfering with L assigned to a register r. Then, r is free, and L can be assigned to it. The evicted live-intervals re-enter the priority queue.

However, this is only worthwhile if the evicted live-intervals are cheaper to spill than L. The cost of spilling a live-interval is given by its *spill weight* $W(L)$. The spill weight is based on the number of spills and reloads needed for a live-interval, and their estimated execution frequency.

The baseline register allocator finds a suitable r by iterating over all registers and choosing the one which minimizes the maximum spill weight of the live-intervals to be evicted. We extend this heuristic in algorithm 2 by also taking into account the difference in compressibility between the newly assigned and the evicted live-intervals.

Let $E(L, r)$ be the set of live-intervals which need to be evicted to assign L to r. If the register allocator decides to evict, it unassigns the live-intervals in $E(L, r)$ and assigns L to r. This means that the compression enabled by live-intervals in $E(L, r)$ is lost, while L's new assignment may enable new compression.

Since a lower score means a better choice for eviction, the lost compression should count positively towards the score, and L's compression should count negatively. We therefore define the *eviction score* by

$$score(L, r) = (1 - \beta) \cdot \max_{L' \in E(L,r)} W(L') + \beta \cdot \left(\left(\sum_{L' \in E(L,r)} C(L') \right) - C(L, r) \right)$$

Again, we have a parameter, the *eviction influence* β, which allows us to find a balance between cheap spills and good compression.

Algorithm 2 Live-interval eviction

```
function SelectEvict(L: LiveInterval) → Register
    cand ← []
    for each register r do
        m ← 0
        c ← −C(L, r)
        for each L' ∈ E(L, r) do
            if W(L') > W(L) then
                try next r
            m ← max(m, W(L'))
            c ← c + C(L')
        score ← (1 − β) · m + β · c
        cand.Append((r, score))
    if cand ≠ [] then
        (r*, _) ← (r, score) ∈ cand with min. score
    else
        r* ← none
    return r*
```

4.4 Register Hints

When a live-interval has a hint, it is usually copy-related to another live-interval. This means that not fulfilling the hint incurs an extra register copy instruction.

However, register copies are usually very cheap in modern processors. It is therefore sensible to accept an additional copy in exchange for more code compression.

We therefore break hints if doing so yields a large enough additional compression: If r^* is the register selected for a live-interval L by algorithm 1, and r_h is the hinted register, we prefer r^* if $C(L, r^*)$ is greater than $C(L, r_h)$ by at least the *hint breaking limit*.

The hint breaking limit should be at least 2: Then, we get at least 2 additional compressed instructions (saving at least 4 bytes) for one additional copy (costing 2 bytes since copies are always compressed).

5 Evaluation

We integrated our register allocator into a development version of LLVM 16 (commit 4e9dd210).

We evaluate our work using the C and C++ benchmarks from the SPEC CPU2000 and CPU2006 benchmark sets [24, 25]. However, we have to exclude the CPU2000 benchmark "252.eon", as it cannot be compiled with modern LLVM without changing the source code.

We choose the following configuration for the benchmarks: Besides flags needed for compatibility, we compile with the standard `-O3` optimization level. In our register allocator, we set the priority influence (section 4.1) to 0.5, the eviction influence (section 4.3) to 0.6, and the hint breaking limit (section 4.4) to 2.

These values put a relatively high emphasis on compression. We have chosen to do so because any effect the register

allocator may have on run-time performance should be apparent at these settings.

In our evaluation, we will first focus on the text size of the resulting binaries (section 5.1). Afterwards, we will also consider the performance of the compiler (section 5.2), and the performance of the generated code (section 5.3).

5.1 Compression

Figure 1 shows the overall results of our optimization. For each benchmark, we plot the relative change in the binary's text segment size. The geometric mean of the change over all benchmarks (shown in green) is −1.93 %.

We can see that our compression-aware register allocator can reduce the size of most benchmarks (30 of 33) by at least 1 %. Moreover, 10 of 33 benchmarks become at least 2 % smaller.

The floating-point benchmarks (plotted in orange) show better compression than the integer benchmarks. 5 out of 11 are compressed by at least 3 %, and better than any integer benchmark except "401.bzip2". The floating-point benchmark "183.equake" improved most of all, becoming 6.52 % smaller.

5.1.1 Compression by Operation Groups. Next, we analyze how successfully we can compress different kinds of operations. To this end, we have split the operations into the following groups: floating-point load/store, integer load/store, arithmetic, jumps, constant generation, and branches. Each group only includes compressible operations.

Figure 2 shows the results summed over all benchmarks. For each group, we measure the share of instructions that are able to use a compressed encoding, comparing baseline LLVM (orange) and our register allocator (blue).

First of all, we can see that register allocation has almost no effect on jumps and constant generation instructions. For these, the immediate value in the instruction is more important. Our register allocator even compresses fewer branch instructions. This is probably because the associated live-intervals are small (a value is generated and immediately tested), and therefore low-priority.

For integer load/store and arithmetic instructions, we see a clear but modest advantage. With a compression-aware register allocator, 6 % more instructions can be compressed in either group.

The majority of arithmetic instructions that cannot be compressed are not in two-address-form (86 %). Only 14 % are in two-address-form but use one or more non-compressible registers.

However, for floating point load/store instructions, the effect is much more pronounced: With our register allocator, 2.4 times as many of these instructions can be compressed.

5.1.2 Interpretation. The advantage for integer instructions is not as large because most of the operations are compressible. In this case, LLVM's old strategy of always preferring compressible registers already yields good results.

On the other hand, most floating-point operations are not compressible (only loads and stores, see the example in section 3). LLVM's old strategy did not work well here, and a more precise approach was clearly required.

More generally, we can conclude that it is worthwhile to care about the idiosyncrasies of the compressed instruction set, especially when only some operations are compressible. In RISC-V, this pertains to floating-point code but may be different for other architectures.

5.2 Compiler Performance

Although the performance of the compiler itself is not our main concern, we still test whether our register allocator is unduly slow. Our register allocator has to do more work overall, especially in register selection (section 4.2), so we expect it to take longer.

Indeed, our measurements show that the compression-aware register allocator is 64 % slower than the baseline, but this only makes overall compilation 1.3 % slower.

Keeping in mind that our implementation is still a prototype, we do not consider its performance problematic. There are still several points where we can optimize our implementation, e. g. by memoizing compressibility values.

5.3 Run-Time Performance

Finally, we consider the impact of our register allocator on the performance of the compiled programs. Given the reductions in binary size we saw, we can probably not expect performance advantages through better cache utilization. Even for the 6.52 % compression we saw with "183.equake", Steenkiste's model only predicts a speedup of 1 % [26].

On the other hand, even inconsequential changes to a program can have a performance impact of a few percent [15]. In our case, these fluctuations could arise from changing alignment of basic blocks as different instructions use compressed encodings.

So all in all, we expect the performance of the benchmarks compiled with our register allocator to be roughly equal to baseline performance but with some variations.

We execute the CPU2000 benchmarks on a "VisionFive" single-board computer running Ubuntu Linux 22.04. This features a SiFive U74 RISC-V core running at 1.5 GHz, and 8 GiB of RAM.

We cannot use the original SPEC benchmarking tools since they are not compatible with RISC-V. Instead, we have to resort to recreating the benchmarking scripts using the same problem sets. This was not possible for "253.perlbmk", so we are forced to skip this benchmark.

We run each benchmark 10 times using *hyperfine* [17]. This yields relative standard deviations of at most 1.8 %.

Our benchmarking results are shown in table 2. For each benchmark, we list its average execution time and its standard deviation, for both the baseline register allocator, and

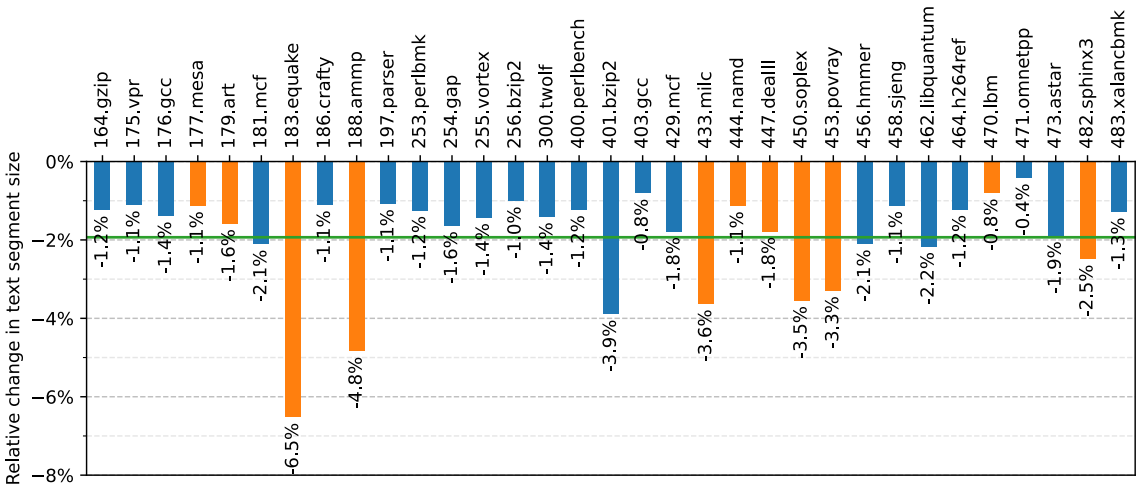

Figure 1. Change in text segment size between baseline LLVM and our compression-aware register allocator. Benchmarks with blue bars are part of the integer-heavy CINT2000 or CINT2006 benchmark sets, those with orange bars are part of the floating-point-heavy CFP2000 or CFP2006. The green line shows the geometric mean of all changes at −1.93 %.

Table 2. Results of the performance benchmarks. The times are given in seconds as "average ± standard deviation". Column "t" gives the result of Welch's t-test between the Baseline and the Result measurements. Change is the relative difference between the averages in percent. Changes marked with "∗" are insignificant according to the t-test with $p = 5\,\%$.

Benchmark	Baseline/s	Result/s	t	Change/%	
164.gzip	652.24 ± 1.81	650.67 ± 2.04	−1.81	−0.2	∗
175.vpr	594.02 ± 2.84	601.77 ± 4.06	4.94	1.3	
176.gcc	364.62 ± 1.00	361.21 ± 0.87	−8.12	−0.9	
181.mcf	933.51 ± 3.32	929.35 ± 3.07	−2.91	−0.4	
186.crafty	255.39 ± 1.78	255.64 ± 4.32	0.17	0.1	∗
197.parser	826.74 ± 7.15	836.62 ± 10.77	2.42	1.2	
254.gap	480.69 ± 3.46	475.36 ± 3.00	−3.68	−1.1	
255.vortex	592.34 ± 5.22	599.51 ± 7.43	2.49	1.2	
256.bzip2	597.69 ± 1.58	596.71 ± 1.84	−1.28	−0.2	∗
300.twolf	750.69 ± 10.87	759.49 ± 10.60	1.83	1.2	∗
177.mesa	459.33 ± 6.66	466.40 ± 8.38	2.09	1.5	∗
179.art	974.59 ± 10.01	974.24 ± 7.35	−0.09	−0.0	∗
183.equake	1086.88 ± 0.91	1086.11 ± 0.71	−2.11	−0.1	
188.ammp	974.37 ± 0.84	977.91 ± 0.70	10.27	0.4	
Geometric mean of changes:				0.3	

our register allocator ("Result"). We also compare both measurements using Welch's t-test with $p = 5\,\%$. Changes judged insignificant by the t-test are marked with "∗".

We can see that the execution times of the benchmarks fluctuate up to −1.1 %/+1.3 % (+1.5 % including insignificant changes). Of the 14 benchmarks, 4 become faster with our register allocator, 4 become slower, and 6 changes are insignificant. The geometric mean over all changes (both significant and insignificant) suggests a slight slowdown of 0.3 % across our benchmark set.

5.3.1 Spill/Reload Analysis.

We now analyze in more detail whether our register allocator systemically introduces worse performance in some cases, or whether the fluctuations we measured are random.

If our register allocator has worsened performance, that will be because it has introduced additional spills, reloads, or register copies.

To investigate these effects, we compare our performance results with performance estimations provided by LLVM. LLVM can estimate the cost of spills, reloads, and register

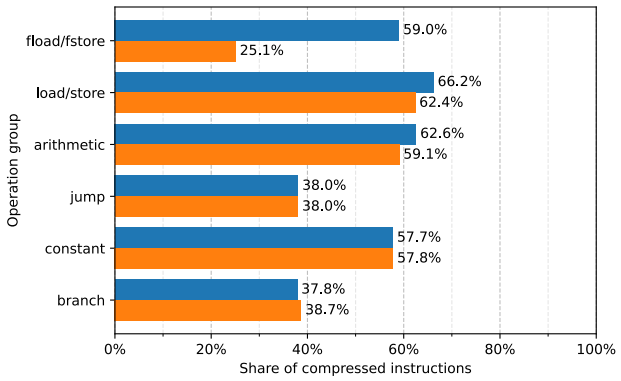

Figure 2. Compressed instructions by category. The bars show the share of compressed instructions over all benchmarks with the compression-aware register allocator (blue) and the baseline (orange).

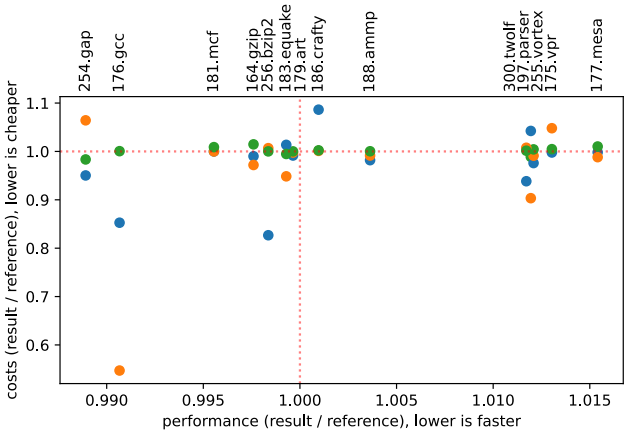

Figure 3. Comparison of changes in execution time (x-axis) and spill/reload/copy costs (y-axis). There are three data points for each benchmark, comparing the change in execution time to the change in spill costs (blue), reload costs (orange), and register copy costs (green) as reported by LLVM.

moves in the program using an execution frequency analysis. These are also the costs used by the baseline register allocator to compute spill weights.

In fig. 3, we plot the change in execution time on the x-axis against the change in reported costs on the y-axis. Spill costs are shown in blue, reload costs in orange, and register copy costs in green. If the change in execution time we saw is caused by increased costs for spilling etc., we expect to see an upward trend, with slower benchmarks also having increased costs.

In fact, all of the cost metrics are relatively weakly correlated to the execution time: The coefficient of determination is $r^2 = 12.2\,\%$ for spills, $r^2 = 9.3\,\%$ for reloads, and $r^2 = 6.0\,\%$ for register copies.

5.3.2 Interpretation. The deviation in performance for the benchmarks is within the range we expected, with an equal number of faster and slower benchmarks. Although the mean shows a slightly worse performance over all benchmarks, the slowdowns we see are not correlated to increased spill, reload, or copy costs. We can therefore conclude that the changes in performance are probably not systemic.

6 Future Work

We have implemented and evaluated our compression-aware register allocator for RVC. However, as we have shown in section 2.3, all compressed ISAs follow similar principles. Our approach, and much of our implementation, is therefore equally applicable to other compressed ISAs. If the ISA does not have different features to RVC, all that is required is a TableGen specification of the compressibility conditions. This is the case for Thumb-2, microMIPS, and PowerPC VLE.

Out of concern for compiler performance, we compute compressibility in a relatively simple way. In particular, we do not consider the context of the instruction in question at all. With more analysis of the context, the register allocator could deduce earlier that an instruction is non-compressible. For example, if neither argument of a binary instruction dies, the instruction cannot be put into two-address form.

Finally, efforts are currently underway to introduce a machine-learning-based register allocator into LLVM [27]. Integrating compressibility information into its objective function would be an all-new challenge, but the basic concepts we introduced in section 3.1 remain applicable.

7 Conclusion

We have presented a compression-aware adaptation to the LLVM greedy register allocator. Our register allocator is still efficient enough to be able to handle real-world benchmarks. By extending the register allocator in four key places, we can achieve a reduction in binary size of up to 6.52 %, with 1.93 % in the mean.

Detailed analysis shows that while our register allocator introduces some changes to performance, these can be explained as random fluctuations in the heuristic.

The improvement in compression is especially pronounced for floating-point-heavy benchmarks. This demonstrates that the register allocator can indeed profit from detailed awareness of the compressed ISA when not many operations are compressible.

Acknowledgments

We thank Florian Schmaus, Sebastian Graf, Sebastian Ullrich, Phillip Raffeck, as well as our reviewers for their illuminating comments.

This work was partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) — Projektnummer 146371743 — TRR 89 "Invasive Computing".

References

[1] Andrew Waterman and Krste Asanovic (Eds.). 2019. *The RISC-V Instruction Set Manual; Volume I: Unprivileged ISA*. https://riscv.org/technical/specifications/ Version 20191213.

[2] ARM Limited. 2005. *ARM Architecture Reference Manual: Thumb-2 Supplement*. https://developer.arm.com/documentation/ddi0308/ Issue D.

[3] Matthias Braun. 2018. Register Allocation: More than Coloring. https://llvmdev18.sched.com/event/H2UP/register-allocation-more-than-coloring

[4] Preston Briggs. 1992. *Register allocation via graph coloring*. Ph. D. Dissertation. Rice University.

[5] G. J. Chaitin. 1982. Register Allocation & Spilling via Graph Coloring. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction* (Boston, Massachusetts, USA) *(SIGPLAN '82)*. Association for Computing Machinery, New York, NY, USA, 98–105. https://doi.org/10.1145/800230.806984

[6] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. 1981. Register Allocation via Coloring. 6, 1 (1981), 47–57. https://doi.org/10.1016/0096-0551(81)90048-5

[7] P. Chow and M. Horowitz. 1987. Architectural Tradeoffs in the Design of MIPS-X. In *Proceedings of the 14th Annual International Symposium on Computer Architecture* (Pittsburgh, Pennsylvania, USA) *(ISCA '87)*. Association for Computing Machinery, New York, NY, USA, 300–308. https://doi.org/10.1145/30350.30384

[8] Tobias J.K. Edler von Koch, Igor Böhm, and Björn Franke. 2010. 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* (New York, NY, USA, 2010-04-24) *(CGO '10)*. Association for Computing Machinery, 180–189. https://doi.org/10.1145/1772954.1772980

[9] Freescale Semiconductor, Inc. 2007. *Variable-Length Encoding (VLE) Programming Environments Manual: A Supplement to the EREF*. https://www.nxp.com/docs/en/reference-manual/VLEPEM.pdf Revision 0.

[10] Sebastian Hack. 2007. *Register allocation for programs in SSA Form*. Universitätsverlag Karlsruhe. https://doi.org/10.5445/KSP/1000007166

[11] John L. Hennessy and David A. Patterson. 2017. *Computer Architecture, Sixth Edition: A Quantitative Approach* (6th ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[12] Dave Jaggar. 1996. *ARM Architecture Reference Manual*. Prentice Hall.

[13] Arvind Krishnaswamy and Rajiv Gupta. 2002. Profile Guided Selection of ARM and Thumb Instructions. *ACM SIGPLAN Notices* 37, 7 (2002), 56–64. https://doi.org/10.1145/566225.513840

[14] Roberto Castañeda Lozano, Mats Carlsson, Gabriel Hjort Blindell, and Christian Schulte. 2019. Combinatorial Register Allocation and Instruction Scheduling. *ACM Transactions on Programming Languages and Systems* 41, 3 (2019), 17:1–17:53. https://doi.org/10.1145/3332373

[15] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2009. Producing Wrong Data without Doing Anything Obviously Wrong!. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (Washington, DC, USA) *(ASPLOS XIV)*. Association for Computing Machinery, New York, NY, USA, 265–276. https://doi.org/10.1145/1508244.1508275

[16] Hanspeter Mössenböck and Michael Pfeiffer. 2002. Linear Scan Register Allocation in the Context of SSA Form and Register Constraints. In *Compiler Construction*, Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and R. Nigel Horspool (Eds.). Vol. 2304. Springer Berlin Heidelberg, Berlin, Heidelberg, 229–246. https://doi.org/10.1007/3-540-45937-5_17 Series Title: Lecture Notes in Computer Science.

[17] David Peter. 2022. *hyperfine*. https://github.com/sharkdp/hyperfine

[18] Massimiliano Poletto and Vivek Sarkar. 1999. Linear Scan Register Allocation. *ACM Trans. Program. Lang. Syst.* 21, 5 (sep 1999), 895–913. https://doi.org/10.1145/330249.330250

[19] Lewis Revill and Craig Blackmore. 2020. [RISCV] Add pre-emit pass to make more instructions compressible. https://reviews.llvm.org/D92105

[20] Johan Runeson and Sven-Olof Nyström. 2003. Retargetable Graph-Coloring Register Allocation for Irregular Architectures. In *Software and Compilers for Embedded Systems* (Berlin, Heidelberg, 2003) *(Lecture Notes in Computer Science)*, Andreas Krall (Ed.). Springer, 240–254. https://doi.org/10.1007/978-3-540-39920-9_17

[21] Bernhard Scholz and Erik Eckstein. 2002. 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* (New York, NY, USA, 2002-06-19) *(LCTES/SCOPES '02)*. Association for Computing Machinery, 139–148. https://doi.org/10.1145/513829.513854

[22] Christian Schulte and Roberto Castañeda Lozano. 2018. Unison: Optimization Technology for Optimizing Compilers. In *Ericsson's Program Analysis Workshop* (2018-04).

[23] Michaela Smith and G. Holloway. 2000. Graph-Coloring Register Allocation for Irregular Architectures. (2000). https://www.semanticscholar.org/paper/Graph-Coloring-Register-Allocation-for-Irregular-Smith-Holloway/4e6196e36941ef40c01a1510752afbbb76d9506d

[24] Standard Performance Evaluation Corporation. 2006. *SPEC CPU2000*. https://www.spec.org/cpu2000/ V1.3.1.

[25] Standard Performance Evaluation Corporation. 2011. *SPEC CPU2006*. https://www.spec.org/cpu2006/ V1.2.

[26] Peter Steenkiste. 1989. The Impact of Code Density on Instruction Cache Performance. In *Proceedings of the 16th Annual International Symposium on Computer Architecture* (New York, NY, USA, 1989-04-01) *(ISCA '89)*. Association for Computing Machinery, 252–259. https://doi.org/10.1145/74925.74954

[27] S. VenkataKeerthy, Siddharth Jain, Rohit Aggarwal, Albert Cohen, and Ramakrishna Upadrasta. 2022. RL4ReAl: Reinforcement Learning for Register Allocation. https://doi.org/10.48550/ARXIV.2204.02013

[28] Wave Computing, Inc. 2013. *MIPS32 Architecture for Programmers Volume IV-a: The MIPS16e Application-Specific Extension to the MIPS32 Architecture*. https://www.mips.com/products/architectures/ase/ase16e/ Revision 2.63.

[29] Wave Computing, Inc. 2016. *MIPS Architecture for Programmers Volume II-B: microMIPS32 Instruction Set*. https://www.mips.com/products/architectures/mips32-2/ Revision 6.05.

[30] Wave Computing, Inc. 2018. *MIPS Architecture Base: nanoMIPS32 Instruction Set Technical Reference Manual*. https://www.mips.com/products/architectures/nanomips/ Revision 01.01.

[31] Vincent M. Weaver and Sally A. McKee. 2009. Code Density Concerns for New Architectures. In *2009 IEEE International Conference on Computer Design* (Lake Tahoe, CA, USA, 2009-10). IEEE, 459–464. https://doi.org/10.1109/ICCD.2009.5413117

[32] Weiming Zhao and Oliver Stannard. 2017. [ARM] Thumb2: favor R4-R7 over R12/LR in allocation order when opt for minsize. https://reviews.llvm.org/D30324

Received 2022-11-10; accepted 2022-12-19