Bringing GNU Emacs to Native Code

Andrea Corallo
akrl@sdf.org

Luca Nassi
luknax@sdf.org

Nicola Manca
nicola.manca@spin.cnr.it
CNR-SPIN
Genoa, Italy

ABSTRACT

Emacs Lisp (Elisp) is the Lisp dialect used by the Emacs text editor family. GNU Emacs can currently execute Elisp code either interpreted or byte-interpreted after it has been compiled to byte-code. In this work we discuss the implementation of an optimizing compiler approach for Elisp targeting native code. The native compiler employs the byte-compiler's internal representation as input and exploits *libgccjit* to achieve code generation using the GNU Compiler Collection (GCC) infrastructure. Generated executables are stored as binary files and can be loaded and unloaded dynamically. Most of the functionality of the compiler is written in Elisp itself, including several optimization passes, paired with a C back-end to interface with the GNU Emacs core and *libgccjit*. Though still a work in progress, our implementation is able to bootstrap a functional Emacs and compile all lexically scoped Elisp files, including the whole GNU Emacs Lisp Package Archive (ELPA) [6]. Native-compiled Elisp shows an increase of performance ranging from 2.3x up to 42x with respect to the equivalent byte-code, measured over a set of small benchmarks.

CCS CONCEPTS

• **Software and its engineering** → **Compilers**; *Dynamic compilers*; **Software performance**; *Development frameworks and environments*.

KEYWORDS

GNU Emacs, Elisp, GCC, libgccjit

ACM Reference Format:
Andrea Corallo, Luca Nassi, and Nicola Manca. 2020. Bringing GNU Emacs to Native Code. In *Proceedings of the 13th European Lisp Symposium (ELS'20).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.5281/zenodo.3736363

1 INTRODUCTION

GNU Emacs is known as the extensible, customizable, free/libre text editor [19]. This is not only one of the most iconic text editors, GNU Emacs (from now on just "Emacs" for simplicity) represents metaphorically the hearth of the GNU operating system. Emacs can be described as a Lisp implementation (Emacs Lisp) and a very broad set of Lisp programs written on top that, capable of a surprising variety of tasks. Emacs' design makes it one of the most popular Lisp implementations to date. Despite being widely employed, Emacs has maintained a remarkably naïve design for such a long-standing project. Although this makes it didactic, some limitations prevent the current implementation of Emacs Lisp to be appealing for broader use. In this context, performance issues represent the main bottleneck, which can be broken down in three main sub-problems:

- lack of true multi-threading support,
- garbage collection speed,
- code execution speed.

From now on we will focus on the last of these issues, which constitutes the topic of this work.

The current implementation traditionally approaches the problem of code execution speed in two ways:

- Implementing a large number of performance-sensitive primitive functions (also known as *subr*) in C.
- Compiling Lisp programs into a specific assembly representation suitable for targeting the Emacs VM called Lisp Assembly Program (LAP) and assembling it into byte-code. This can be eventually executed by the byte-interpreter [3, Sec.1.2], [15, Sec. 5.1].

As a result, Emacs developers had to implement a progressively increasing amount of functions as C code primarily for performance reasons. As of Emacs 25, 22% of the codebase was written in C [3, Sec. 1.1], with consequences on maintainability and extensibility [24]. The last significant performance increase dates back to around 1990, when an optimizing byte-compiler including both source level and byte-code optimizations was merged from Lucid Emacs [15, Sec. 7.1]. However, despite progressive improvements, the main design of the byte-code machine stands unmodified since then. More recently, the problem of reaching better performance has been approached using Just-In-Time (JIT) compilation techniques, where three such implementations have been attempted or proposed so far [15, Sec. 5.11], [23]. Possibly due to their simplistic approaches none of them proved to introduce sufficient speed-up, in particular if compared to the maintenance and dependency effort to be included in the codebase. In contrast, state-of-the-art high-performance Lisp implementations rely on optimizing compilers targeting native code to achieve higher performance [12]. In this context, C-derived toolchains are already employed by a certain number of Common Lisp implementations derived from KCL [2, 9, 17, 18, 31], where all these, except CLASP, target C code generation.

In this work we present a different approach to tackle this problem, based on the use of a novel intermediate representation (IR) to bridge Elisp code with the GNU Compiler Collection [20]. This intermediate representation allows to effectively implement a number of optimization passes and for Elisp byte-code to be translated

ELS'20, April 27–28 2020, Zürich, Switzerland

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.5281/zenodo.3736363

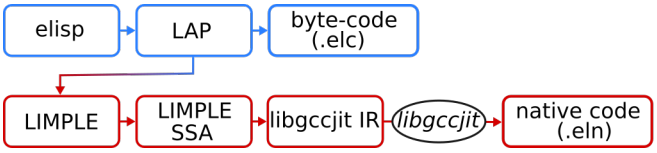

Figure 1: Program representation formats used by byte-compiler (blue) native compiler (red) pipelines.

to a C-like semantic, compatible with the full pipeline of GCC optimization passes. This process relies on *libgccjit* to plug into the GCC infrastructure and achieve code generation without having to target any intermediate programming language [14]. The result of the compilation process for a compilation unit (CU) is a file with `.eln` extension (Emacs Lisp Native). This is a new file extension we have defined to hold the generated code and all the necessary data to have it re-loadable over different Emacs runs. This last characteristic, in contrast with typical JIT-based approaches, saves from having to recompile the same code at each run and allows for more time expensive optimization passes. Also, the classical Lisp image dump feature is supported.

From a more general point of view, here we demonstrate how a Lisp implementation can be hosted on top of *libgccjit*. Although different libraries for code generation, such as *libjit* [13] or LLVM [11], have been successfully employed by various Lisp implementations so far [10, 18], we are not aware of any leveraging *libgccjit*. Moreover, the proposed infrastructure introduces better support for functional programming style in Emacs Lisp with a pass performing tail recursion elimination [30] and the capability to be further extended in order to perform full tail call optimization.

2 IMPLEMENTATION

The proposed compiler pipeline can be divided in three main stages:

- Front-end: Lisp programs are compiled into LAP by the current byte-compiler.
- Middle-end: LAP is converted into "LIMPLE", a new intermediate representation named after GCC GIMPLE [8] which is the very core of the proposed compiler infrastructure. LIMPLE is a sexp-based IR used as static single assignment (SSA) representation [7, 26]. Middle-end passes manipulate LIMPLE by performing a series of transformations on it.
- Back-end: LIMPLE is converted into the *libgccjit* IR to trigger the final compilation through the conventional GCC pipeline.

The sequence of program representation formats is presented in Figure 1. The compiler takes care of type and value propagation through the program control flow graph. We point out that, since Emacs Lisp received in 2012 lexical scope support, two different sub-languages are currently coexisting [15, Sec. 8.1]. The proposed compiler focuses on generating code for the new lexically scoped dialect only, since the dynamic one is considered obsolete and close to deprecation.

2.1 LAP to *libgccjit* IR

Here, we briefly discuss the two endpoints of our compilation pipeline: the Lisp Assembly Program and the *libgccjit* IR, by showing different representations of an illustrative and simple code.

LAP representation is a list of instructions and labels, expressed in terms of S-expressions. Instructions are assembled into byte-code, where each one is associated with an operation and a manipulation of the execution stack, both happening at runtime and defined by the opcode corresponding to the instruction. When present, control flow instructions can cause a change in the execution flow by executing jumps to labels. As an example, the Lisp expression `(if *bar* (+ *bar* 2) 'foo)` is compiled into the following LAP representation:

```
(byte-varref *bar*)
(byte-goto-if-nil TAG 8)
(byte-varref *bar*)
(byte-constant 2)
(byte-plus)
(byte-return)
(TAG 8)
(byte-constant foo)
(byte-return)
```

where `byte-varref` pushes the value of a symbol into the stack, `byte-goto-if-nil` pops the top of the stack and jumps to the given label if it is `nil`, `byte-constant` pushes the value from an immediate into the stack, `byte-plus` pops two elements from the stack, adds them up and pushes the result back into the stack and `byte-return` exits the function using the top of the stack as return value. An extensive description of these instructions is available in Ref. [3, Sec. 1.3].

libgccjit allows for describing code programmatically in terms of `gcc_jit_objects` created through C or C++ API [14, Sec. *Objects*]. The semantic it can express can be described as a subset of the one of the C programming language. This includes `l` and `r` values, arithmetic operators, assignment operators and function calls. The most notable difference with respect to C is that conditional statements such as `if` and `else` are not supported and the code has to be described in terms of basic blocks. Inside GCC, *libgccjit* IR is mapped into GIMPLE when the actual compilation is requested. One key property of Emacs Lisp LAP is that it guarantees that, for any given program counter, the stack depth is fixed and known at compile time. The previous LAP code can be transformed in the following pseudo-code, suitable to be described in the *libgccjit* IR:

```
Lisp_Object local[2];

bb_0:
  local[0] = varref (*bar*);
  if (local[0] == NIL) goto bb_2;
  else goto bb_1;

bb_1:
  local[0] = varref (*bar*);
  local[1] = two;
  local[0] = plus (local[0], local[1]);
  return local[(int)0];

bb_2:
  local[0] = foo;
  return local[0];
```

This transformation accomplishes the following:

- performs opcode decoding during the transformation so that it is not needed anymore at runtime.
- decodes and compiles all the operations within the original stack into assignments.
- splits the initial list of LAP instructions into basic blocks.

These tasks are performed by means of an intermediate translation into LIMPLE, which enables standard code optimization routines present in GCC as well as dedicated optimization passes.

2.2 LIMPLE IR

As previously introduced, in order to implement a series of optimization passes, we defined an intermediate representation, that we called "LIMPLE", whose main requirement is to be SSA. The description of every variable inside the compiler is accomplished with instances of a structure we called `m-var` and reported in Appendix A. This represents the Lisp objects that will be manipulated by the function being compiled. A function in LIMPLE is a collection of basic blocks connected by edges to compose a control flow graph where every basic block is a list of `insn` (instructions). The format of every LIMPLE `insn` is a list `(operator operands)` whose valid operands depend on the operator itself, such as:

- `(set dst src)` Copy the content of the slot represented by the `m-var` `src` into the slot represented by `m-var` `dst`.
- `(setimm dst imm)` Similar to the previous one but `imm` is a Lisp object known at compile time.
- `(jump bb)` Unconditional jump to basic block whose name is represented by the symbol `bb`.
- `(cond-jump a b bb_1 bb_2)` Conditional jump to `bb_1` if `a` and `b` are `eq` or to `bb_2` otherwise.
- `(call f a b . . . )` Call a primitive function `f` where `a, b, . . .` are `m-vars` used as parameters.
- `(comment str)` Include annotation `str` as comment inside the `.eln` debug symbols (see Sec. 5.3).
- `(return a)` Perform a function return having as return value the `m-var a`.
- `(phi dst src1 . . . srcn)` Conventional Φ node used by SSA representation. When all `m-vars` `src1 . . . srcn` have the same immediate value this is assigned to `dst`. Similarly it happens for the type (no handling for type hierarchy is implemented up to date).

3 COMPILATION PASSES

The compilation pipeline is composed by a sequence of passes that, starting from the input Elisp source code, apply a succession of transformations to finally produce the executable code in the form of a `.eln` file. The following sections describe each of the compilation passes, all of them are implemented in Lisp with the exception of `final`.

However, before getting into the details of each pass, it is useful to discuss the reason why the data-flow analysis and optimization algorithms already present in the GCC infrastructure are not enough for the Elisp semantic, and dedicated ones had to be developed in Lisp.

Type propagation. Emacs Lisp is a strong dynamically-typed programming language. Typing objects is done through tagging pointers [27]. While GCC has passes to propagate both constants and ranges, it has no visibility of the Lisp type returned by Lisp primitive functions and, as a consequence, on the tag bits set.

Pure functions. Similarly, GCC does not know which Lisp functions can be optimized at compile time having visibility only on the local compilation unit. Optimizable functions are typically pure functions or functions that are pure for a specific set of parameters.

Reference propagation. Another useful property to be propagated is if a certain object will or will not be referenced [29] during function calls. This information is required to generate a more efficient code, as discussed in Sec. 3.8.

Unboxing. GCC does not offer infrastructure for unboxing values. Although not yet implemented, the proposed infrastructure is designed to host further improvements, such as unboxing, requiring data-flow analysis [28].

Compiler hints. The data-flow analysis can be fed with compiler hints about the type of certain expressions, included as high-level annotations in the source code by the programmer.

Warning and errors. A data-flow analysis engine as the one proposed in this work could be used in the future to provide more accurate warnings and errors during the compilation phase.

GCC optimization constraints. GCC optimization passes often adopt conservative strategies not to break the semantic of all the supported programming languages. As an example, the GCC tail call optimization pass does not perform transformations whenever an instruction referencing memory is present in the compiled function. Given the specific semantic of the code generated by the proposed work, conditions as the one mentioned may be too restrictive resulting in missed optimizations.

3.1 *spill-lap*

As already discussed, the main input for the compilation process is the Lisp Assembly Program Intermediate Representation (LAP IR). *spill-lap* runs the byte-compiler infrastructure with the Elisp source as input collecting all top-level forms and spilling the LAP before it is assembled into final byte-code.

3.2 *limplify*

This pass is responsible for translating LAP IR into LIMPLE IR. In general, LAP is a sequence of instructions, labels and jumps-to-label. Since the Emacs byte-interpreter is a stack-based machine, every LAP instruction manipulates the stack [3], [15, Sec. 5.1]. It is important to highlight that at this stage all the stack manipulations performed by LAP instructions are compiled into a series of `m-var` assignments. Spurious moves will eventually be optimized out by GCC. This pass is also responsible for decomposing the function into lists of LIMPLE `insns`, or basic blocks. The code necessary for the translation of most LAP instructions is automatically generated using the original instruction definition specified in the byte-compiler.

3.3 *static single assignment (SSA)*

This pass is responsible for bringing LIMPLE into minimal SSA form, discussed in [7, Sec. 2.2], as follows:

a) Edges connecting the various basic blocks are created.
b) The dominator tree is computed for each basic block [4].
c) Dominator frontiers are computed for each basic block.
d) Φ functions are placed as described in [7, Sec. 3.1].
e) `m-vars` goes through classic SSA renaming.

Once LIMPLE is in SSA form every `m-var` object appears as destination of an instruction only in one place within the SSA lattice. The same object can be referenced multiple times as source though, but each `m-var` can be identified by its unique `id` slot.

3.4 *forward data-flow analysis*

For each `m-var`, this pass propagates the following properties within the control flow graph: value, type and where the `m-var` will be allocated (see Sec. 3.8). Initially, all immediate values set at compile time by `setimm` are propagated to each destination `m-var`. Afterwards, for each `insn` in each basic block the following operations are iteratively performed:

a) If the `insn` is a Φ, the properties of `m-vars` present as source operands are propagated to the destination operand when in agreement.
b) If a function call has a known return type, this is propagated to the result.
c) If a function call to a pure function is performed with all arguments having a known value, the call is optimized out and the resulting value is substituted.
d) Assignments by `set` operators are used to propagate all `m-vars`.

This sequence is repeated until no more changes are performed in the control flow graph.

3.5 *call-optim*

This pass is responsible for identifying all function calls to primitives going through the `funcall` trampoline and substitute them with direct calls. The primitive functions most commonly used in the original LAP definition are assigned dedicated opcodes, as described in [3, page 172]. When a call to one of these functions is performed, the byte-interpreter can thus perform a direct call to the primitive function. All the remaining functions are instead called through the `funcall` trampoline, which carries a considerable overhead. This mechanism is due to the intrinsic limit of the opcode encoding space. On the other hand, native-compiled code has the possibility to call all Emacs primitives without any encoding space limitation. After this pass has run, primitive functions have all equal dignity, being all called directly irrespective of the fact that they were originally assigned a dedicated byte-opcode or not. The same transformation is performed for function calls within the compilation unit when the compiler optimization level is set to its maximum value (see Sec. 5.1). This will improve the effectiveness of inlining and other inter-procedural optimizations in GCC. Finally, recursive functions are also optimized to prevent `funcall` usage.

3.6 *dead-code*

This pass cleans up unnecessary assignments within the function. The algorithm checks for all `m-vars` that are assigned but not used elsewhere, removing the corresponding assignments. This pass is also responsible for removing function calls generated by compiler type hints (see Sec. 5.2) if necessary.

3.7 *tail recursion elimination (TRE)*

This peephole pass [16, Chap. 18] performs a special case of tail call optimization called tail recursion elimination. The pass scans all LIMPLE `insns` in the function searching for a recursive call in tail position. If this is encountered it is replaced with the proper code to restart executing the current function using the new arguments without activating a new function frame into the execution stack. This transformation is described in [16, Chap. 15.1].

3.8 *final (code layout)*

This pass is responsible for converting LIMPLE into *libgccjit* IR and invoking the compilation through GCC. We point out that the code we generate for native-compiled Lisp functions follows the same ABI of Elisp primitive C functions. Also, a minimal example of pseudo C code for a native-compiled Elisp function is listed in Appendix B.

When optimizations are not engaged, `m-vars` associated to each function are arranged as a single array of Lisp objects. This array has the length of the original maximum byte-code stack depth. Depending on the number of their arguments, Elisp primitive functions present one of the following two C signatures [22]:

a) `Lisp_Obj fun (Lisp_Obj arg01, ..., Lisp_Obj argn)`, for regular functions with a number of arguments known and smaller or equal to 8.
b) `Lisp_Obj fun (ptrdiff_t n, Lisp_Obj *args)`, otherwise.

where `ptrdiff_t` is an integral type, `n` is the number of arguments and `args` is a one-dimensional array containing their values. When a call of the second kind is performed, GCC clobbers all the `args` array content, regardless the number of arguments `n` involved in the call. This means that the whole array content after the call is considered potentially modified. For this reason the compiler cannot trust values already loaded in registers and has to emit new load instructions for them. To prevent this, when the optimization we have called "advanced frame layout" is triggered, each `m-var` involved in a call of the second kind is rendered in a stack-allocated array dedicated to that specific call. All other `m-vars` are rendered as simple automatic variables. The advanced frame layout is enabled for every compilation done with a non zero `comp-speed`, as discussed in Sec. 5.1.

This pass is also responsible for substituting the calls to selected primitive functions with an equivalent implementation described in *libgccjit* IR. This happens for small and frequently used functions such as: `car`, `cdr`, `setcar`, `setcdr`, `1+`, `1-`, or `-` (negation). As an example, the signature for function `car` implemented in *libgccjit* IR will be:

```
1 static Lisp_Object CAR (Lisp_Object c,
2                         bool cert_cons)
```

If compared to the original `car` function a further parameter has been added, `cert_cons` which stands for "certainly a `cons`". *final* will emit a call to `CAR` setting `cert_cons` to `true` if the data-flow analysis was able to prove `c` to be a `cons` or setting it to `false` otherwise. This mechanism is used in a similar fashion with most inlinable functions injected by this pass in order to provide the information obtained by the data-flow analysis to the GCC one. Since the GCC implementation has the full definition of these functions, they can be optimized effectively.

4 SYSTEM INTEGRATION

4.1 Compilation unit and file format

The source for a compilation unit can be a Lisp source file or a single function and, as already mentioned, the result of the compilation process for a compilation unit is a file with `.eln` extension. Technically speaking, this is a shared library where Emacs expects to find certain symbols to be used during load. The conventional load machinery is modified such that it can load `.eln` files in addition to conventional `.elc` and `.el` files.

In order to be integrated in the existing infrastructure we define the `Lisp_Native_Comp_Unit` Lisp object. This holds references to all Lisp objects in use by the compilation unit plus a reference to the original `.eln`. Every `.eln` file is expected to contain a number of symbols including:

- `freloc_link_table`: static pointer to a structure of function pointers used to call Emacs primitives from native-compiled code.
- `text_data_reloc`: function returning a string representing all immediate constants in use by the code of the compilation unit. The string is formed using `prin1` so that it is suitable to be read by Lisp reader.
- `d_reloc`: static array containing the Lisp objects used by the compiled functions.
- `top_level_run`: function responsible of performing all the modifications to the environment expected by the load of the compilation unit.

4.2 Load mechanism

Load can be performed conventionally as: `(load "test.eln")`. Loading a new compilation unit translates into the following steps:

a) Load the shared library into the Emacs process address space.
b) Given that `.eln` plugs directly into Emacs primitives, forward and backward version compatibility cannot be ensured. Because of that each `.eln` is signed during compilation with an hash and this is checked during load. In case the hash mismatches the load process is discarded.
c) Lookup the following symbols in the shared library and set their values: `current_thread_reloc`, `freloc_link_table`, `pure_reloc`.
d) Lookup `text_data_reloc` and call it to obtain the serialized string representation of all Lisp objects used by native-compiled functions.
e) Call the reader to deserialize the objects from this string and set the resulting objects in the `d_reloc` array.
f) Lookup and call `top_level_run` to have the environment modifications performed.

We show in Appendix B an example of pseudo C code for a native-compiled function illustrating the use of `freloc_link_table` and `d_reloc` symbols.

When loaded, the native-compiled functions are registered as *subr*s as they share calling convention with primitive C functions. Both native-compiled and primitive functions satisfies `subrp` and are distinguishable using the predicate `subr-native-elisp-p`.

4.3 Unload

The unload of a compilation unit is done automatically when none of the Lisp objects defined in it is referenced anymore. This is achieved by having the `Lisp_Native_Comp_Unit` object been integrated with the garbage collector infrastructure.

4.4 Image dump

Emacs supports dumping the Lisp image during its bootstrap. This technique is used in order to reduce the startup time. Essentially a number of `.elc` files are loaded before dumping the Emacs image that will be invoked during normal use. As of Emacs 27 this is done by default by relying on the portable dumper, which is in charge of serializing all allocated objects into a file, together with the information needed to revive them. The final Emacs image is composed by an executable plus the matching dump file. Image dump capability has been extended to support native-compiled code, the portable dumper has been modified to be able to dump and reload `Lisp_Native_Comp_Unit` Lisp objects.

4.5 Bootstrap

Since the Elisp byte-compiler is itself written in Elisp, a bootstrap phase is performed during the build of the standard Emacs distribution. Conventionally this relies on the Elisp interpreter [15, Sec. 5.2.1]. We modified the Emacs build system to allow for a full bootstrap based on the native compiler. The adopted strategy for this is to follow the conventional steps to produce `.eln` files instead of `.elc` when possible (lexically scoped code) and fall-back to `.elc` otherwise. More than 700 Elisp files are native-compiled in this process.

4.6 Documentation and source integration

The function documentation string, `describe-function`, and "goto definition" mechanism support have been implemented and integrated such that native-compiled code behaves as conventional byte-compiled code.

4.7 Verification

A number of tests have been defined to check and verify the compiler. These include some micro test cases taken from Tom Tromey's JIT [15, Sec. 5.11], [23]. A classical bootstrap compiler test has also been defined, where the interpreted compiler is used to native-compile itself, and then the resulting compiler is used to compile itself. Finally, the two produced binaries are compared. The test is successful if the two objects are bytewise identical.

5 ELISP INTERFACE

5.1 Code optimization levels

Some special variables are introduced to control the compilation process, most notably `comp-speed`, which controls the optimization level and safety of code generation as follows:

(0) No optimization is performed.
(1) No Lisp-specific optimization is performed.
(2) All optimizations that do not modify the original Emacs Lisp semantic and safeness are performed. Type check elision is allowed where safe.
(3) Code is compiled triggering all optimizations. Intra compilation unit inlining and type check elision are allowed. User compiler hints are assumed to be correct and exploited by the compiler.

`comp-speed` also controls the optimization level performed by the GCC infrastructure as indicated by the table below.

`comp-speed`	0	1	2	3
propagate	n	n	y	y
call-optim	n	n	y	y
call-optim (intra CU)	n	n	n	y
dead-code	n	n	y	y
TRE	n	n	n	y
advanced frame layout	n	y	y	y
GCC -Ox	0	1	2	3

5.2 Language extensions

In order to allow the user to feed the data-flow analysis with type suggestions, two entry points have been implemented:

- `comp-hint-fixnum`
- `comp-hint-cons`

These can be used to specify that a certain expression evaluates to the specified type. For example, `(comp-hint-cons x)` ensures that the result of the evaluation of the form itself is a `cons`. Currently, when `comp-speed` is less or equal to 2, type hints are compiled into assertions, while they are trusted for type propagation when using `comp-speed` 3. These low level primitives are meant to be used to implement operators similar to Common Lisp `the` and `declare` [1].

5.3 Debugging facility

libgccjit allows for emitting debug symbols in the generated code and dumping a pseudo C code representation of the *libgccjit* IR. This is triggered for compilations performed with `comp-debug` set to a value greater than zero. Debugging the generated code is achieved using a conventional native debugger such as `gdb` [21]. In this condition, the `final` pass emits additional code annotations, which are visible as comments in the dumped pseudo C code to ease the debugging (see Appendix B).

6 PERFORMANCE IMPROVEMENT

In order to evaluate the performance improvement of the native code, a collection of Elisp benchmarks has been assembled and made available as `elisp-benchmarks` in the Emacs Lisp Package Archive (ELPA) [5]. It includes the following set of programs:

- List processing: traverse a list incrementing all elements or computing the total length.
- Fibonacci number generator: iterative, recursive and tail-recursive implementations.
- Bubble sort: both destructive in-place and non destructive.
- Dhrystone: the famous synthetic benchmark ported from C to Elisp [25].
- N-body simulation: a model of the solar gravitation system, intensive in floating-point arithmetic.

The benchmarking infrastructure executes all programs in sequence, each for a number of iterations selected to have it last around 20 seconds when byte-interpreted. The sequence is then repeated five times and the execution times are averaged per each benchmark. The results reported in Table 1 are obtained from an Intel i5–4200M machine. They compare the execution time of the benchmarks when byte-compiled and run under the vanilla Emacs 28 from master branch against their native-compiled versions at `comp_speed` 3. The native-compiled benchmarks are run under Emacs compiled and bootstrapped at `comp_speed` 2 from the same revision of the codebase.

The optimized native-code allows all the benchmarks to run at least two times faster, with most of them reaching much higher performance boosts. Despite the analysis being still preliminary, the reason behind these improvements can be explained with several considerations. First of all the removal of the byte-interpreter loop which, implementing a stack-machine, fetches opcodes from memory, decodes and executes the corresponding operations and finally pushes the results back to the memory. Instead the native compiler walks the stack at compile time generating a sequence of machine-level instructions (the native code) that works directly with the program data at execution time (see Sec. 3.2). The result of this process is that executing a native-compiled program takes a fraction of machine instructions with respect to byte-interpreting it. Analyzing the instructions mix also reveals a smaller percentage of machine instructions spent doing memory accesses, in favor of data processing ones. This is the fundamental upgrade of native compilation against interpretation and is probably the major source of improvement for benchmarks with smaller speed-ups, where no other optimizations apply.

On the other hand, benchmarks with larger improvements also take advantage of Lisp specific compiler optimizations, in particular from call optimizations (see Sec. 3.5). Function calls avoid the trampoline when targeting subroutines defined in the same compilation unit or when calling pure functions from the C codebase. Moreover, the data-flow analysis step allows to exploit the properties of the structures manipulated by the compiler in order to produce code with less overheads. Function calls can also be completely removed, along with corresponding returns, and replaced by simple jumps for optimized tail recursive functions, avoiding at the same time new allocations on the execution stack.

Finally, the data-flow analysis can be made even more effective when paired with compiler hints. Without these, the only types known at compile time are the ones belonging to constants or values returned by some primitive functions. Type hints greatly increase the chances for the native compiler to optimize out expensive type

benchmark	byte-comp runtime (s)	native-comp runtime (s)	speed-up
inclist	19.54	2.12	9.2*x*
inclist-type-hints	19.71	1.43	13.8*x*
listlen-tc	18.51	0.44	42.1*x*
bubble	21.58	4.03	5.4*x*
bubble-no-cons	20.01	5.02	4.0*x*
fibn	20.04	8.79	2.3*x*
fibn-rec	20.34	7.13	2.9*x*
fibn-tc	21.22	5.67	3.7*x*
dhrystone	18.45	7.22	2.6*x*
nbody	19.79	3.31	6.0*x*

Table 1: Performance comparison of byte-compiled and native-compiled Elisp benchmarks

checks. In our measurements, the same benchmark (*inclist*) annotated with type hints earns a further improvement of 50% in terms of execution speed, while compiled under the same conditions.

7 CONCLUSIONS

In this work we discussed a possible approach to improve execution speed of generic Elisp code, starting from LAP representation and generating native code taking advantage of the optimization infrastructure of the GNU Compiler Collection. Despite its early development stage, the compiler successfully bootstraps a usable Emacs and is able to compile all lexically scoped Elisp present in the Emacs distribution and in ELPA. The promising results concerning stability and compatibility already led this work to be accepted as *feature branch* in the official GNU Emacs repository. Moreover, a set of benchmarks was developed to evaluate the performance gain and preliminary results indicate an improvement of execution speed between 2.3x and 42x, measured over several runs. At last, we point out that most of the optimization possibilities allowed by this infrastructure are still unexplored. Already planned improvements include: supporting `fixnum` unboxing and full tail call optimization, exposing more primitives to the GCC infrastructure by describing them in the *libgccjit* IR during the `final` pass, and allowing to signal warnings and error messages at compile time based on values and types inferred by the data-flow analysis.

ACKNOWLEDGMENTS

Stefan Monnier and Kyrylo Tkachov for reviewing this manuscript. Luca Benso for inspiration, useful discussions and artichokes, and Daniela Ferraro for cooking them.

REFERENCES

[1] American National Standards Institute and Computer and Business Equipment Manufacturers Association. *Draft proposed American National Standard programming language Common LISP: X3.226-199x: Draft 12.24, X3J13/92-102*. pub-CBEMA, pub-CBEMA:adr, July 1992. July 1, 1992.

[2] Giuseppe Attardi. The embeddable common lisp. *Pap. 4th Int. Conf. LISP Users Vendors, LUV 1994*, 8(1):30–41, 1995. doi: 10.1145/224139.1379849.

[3] Rocky Bernstein. *GNU Emacs Lisp Bytecode Reference Manual*, 2018. URL https://rocky.github.io/elisp-bytecode.pdf.

[4] Keith Cooper, Timothy Harvey, and Ken Kennedy. A simple, fast dominance algorithm. *Rice University, CS Technical Report 06-33870*, 01 2006.

[5] Andrea Corallo and Luca Nassi. Elisp benchmarks collection. URL https://elpa.gnu.org/packages/elisp-benchmarks.html.

[6] Emacs Developer Community. Gnu emacs lisp package archive. URL https://elpa.gnu.org/.

[7] National Institute for Research in Digital Science and Technology, editors. *Static Single Assignment Book*. 2018. URL http://ssabook.gforge.inria.fr/latest/book.pdf.

[8] GCC Developer Community. GIMPLE – GNU GCC Internals. URL https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html.

[9] GNU Common LISP Developer Community. GNU Common LISP. URL https://www.gnu.org/software/gcl.

[10] Jaffer, Aubrey and Lord, Tom and Bader, Miles and GNU Project. Gnu ubiquitous intelligent language for extensions (gnu guile). URL https://www.gnu.org/software/guile/.

[11] Chris Lattner and LLVM Foundation. LLVM. URL https://llvm.org.

[12] Robert A. MacLachlan. The Python compiler for CMU Common Lisp. In *Proc. 1992 ACM Conf. LISP Funct. Program. - LFP '92*, pages 235–246, New York, New York, USA, 1992. ACM Press. ISBN 0897914813. doi: 10.1145/141471.141558. URL http://portal.acm.org/citation.cfm?doid=141471.141558.

[13] David Malcolm and GCC Developer Community. LibJIT, . URL https://www.gnu.org/software/libjit/.

[14] David Malcolm and GCC Developer Community. LibgccJIT, . URL https://gcc.gnu.org/onlinedocs/jit.

[15] Stefan Monnier and Michael Sperber. Evolution of emacs lisp. 2018. URL https://www.iro.umontreal.ca/~monnier/hopl-4-emacs-lisp.pdf.

[16] Steven Stanley Muchnick. *Advanced Compiler Design and Implementation*. 1997. ISBN 1558603204.

[17] Christian E. Schafmeister. Clasp — A common Lisp that interoperates with C++ and uses the LLVM Backend. In *Proceedings of the 8th European Lisp Symposium, ELS2015*, pages 90–91, 2015. URL https://www.european-lisp-symposium.org/static/proceedings/2015.pdf.

[18] Christian E. Schafmeister and Alex Wood. CLASP Common Lisp Implementation and Optimization. In *Proceedings of the 11th European Lisp Symposium, ELS2018*, pages 59–64, 2018. doi: 10.5555/3323215.3323223. URL https://dl.acm.org/doi/10.5555/3323215.3323223.

[19] Richard M. Stallman and Emacs Developer Community. GNU EMACS text editor. URL https://www.gnu.org/software/emacs/.

[20] Richard M. Stallman and GCC Developer Community. GNU C Compiler. URL https://gcc.gnu.org.

[21] Richard M. Stallman and GDB Developer Community. GNU Debugger. URL https://www.gnu.org/software/gdb/.

[22] Stallman, Richard M. and Elisp Developer Community. GNU Emacs Lisp Reference Manual. URL https://www.gnu.org/software/emacs/manual/html_node/elisp/Writing-Emacs-Primitives.htm.

[23] Tom Tromey. Jit compilation for emacs, 2018. URL https://tromey.com/blog/?p=982.

[24] Tom Tromey. el-compilador, 2018. URL https://github.com/tromey/el-compilador.

[25] Reinhold P. Weicker. Dhrystone: a synthetic systems programming benchmark. *Commun. ACM*, 27(10):1013–1030, oct 1984. ISSN 00010782. doi: 10.1145/358274.358283. URL http://portal.acm.org/citation.cfm?doid=358274.358283.

[26] Wikipedia contributors. Static single assignment form — Wikipedia, the free encyclopedia, 2020. URL https://en.wikipedia.org/wiki/Static_single_assignment_form.

[27] Wikipedia contributors. Tagged pointer — Wikipedia, the free encyclopedia, 2020. URL https://en.wikipedia.org/wiki/Tagged_pointer.

[28] Wikipedia contributors. Object type (object-oriented programming)#boxing — Wikipedia, the free encyclopedia, 2020. URL https://en.wikipedia.org/wiki/Object_type_(object-oriented_programming)#Boxing.

[29] Wikipedia contributors. Reference (computer science) — Wikipedia, the free encyclopedia, 2020. URL https://en.wikipedia.org/wiki/Reference_(computer_science).

[30] Wikipedia contributors. Tail call — Wikipedia, the free encyclopedia, 2020. URL https://en.wikipedia.org/wiki/Tail_call.

[31] Taiichi Yuasa. Design and implementation of Kyoto Common Lisp. *j-J-INF-PROCESS*, 13(3):284–295, 1990. ISSN 0387-6101.

A DEFINITION OF M-VAR

```
(cl-defstruct (comp-mvar (:constructor make--comp-mvar))
    "A meta-variable being a slot in the virtual-stack."
    (id nil :type (or null number)
        :documentation "SSA unique id number when in SSA form.")
    (const-vld nil :type boolean
        :documentation "Validity signal for the following slot.")
    (constant nil
        :documentation "When const-vld is non-nil this is used for holding a known value.")
    (type nil
        :documentation "When non-nil indicates the type known at compile time."))
```

B EXAMPLE OF A COMPILATION UNIT

Below we show an example of an elementary compilation unit followed by the pseudo C code generated dumping the *libgccjit* IR. The compilation process is performed using `comp-speed = 3` and `comp-debug = 1`.

```
;;;  -*- lexical-binding: t -*-
(defun foo ()
  (if *bar*
      (+ *bar* 2)
      'foo))
```

```
extern union comp_Lisp_Object
F666f6f_foo ()
{
  union comp_Lisp_Object[2] arr_1;
  union comp_Lisp_Object local0;
  union cast_union union_cast_28;
entry:
  /* Lisp function: foo */
  goto bb_0;
bb_0:
  /* const lisp obj: *bar* */
  /* calling subr: symbol-value */
  local0 = freloc_link_table->R73796d626f6c2d76616c7565_symbol_value (d_reloc[0]);
  /* const lisp obj: nil */
  union_cast_28.v_p = (void *)NULL;
  /* EQ */
  if (local0.num == union_cast_28.lisp_obj.num) goto bb_2; else goto bb_1;
bb_2:
  /* foo */
  local0 = d_reloc[2];
  /* const lisp obj: foo */
  return d_reloc[2];
bb_1:
  /* const lisp obj: *bar* */
  /* calling subr: symbol-value */
  arr_1[0] = freloc_link_table->R73796d626f6c2d76616c7565_symbol_value (d_reloc[0]);
  /* const lisp obj: 2 */
  arr_1[1] = d_reloc[3];
  /* calling subr: + */
  local0 = freloc_link_table->R2b_ (2, (&arr_1[0]));
  return local0;
}
```