On the correctness of a branch displacement algorithm*

Jaap Boender[1] and Claudio Sacerdoti Coen[2]

[1] Foundations of Computing Group
Department of Computer Science
School of Science and Technology
Middlesex University, London, UK
`J.Boender@mdx.ac.uk`

[2] Dipartimento di Scienze dell'Informazione,
Università degli Studi di Bologna, Italy
`sacerdot@cs.unibo.it`

Abstract The branch displacement problem is a well-known problem in assembler design. It revolves around the feature, present in several processor families, of having different instructions, of different sizes, for jumps of different displacements. The problem, which is provably NP-hard, is then to select the instructions such that one ends up with the smallest possible program.
During our research with the CerCo project on formally verifying a C compiler, we have implemented and proven correct an algorithm for this problem. In this paper, we discuss the problem, possible solutions, our specific solutions and the proofs.

Keywords: formal verification, interactive theorem proving, assembler, branch displacement optimisation

1 Introduction

The problem of branch displacement optimisation, also known as jump encoding, is a well-known problem in assembler design [3]. Its origin lies in the fact that in many architecture sets, the encoding (and therefore size) of some instructions depends on the distance to their operand (the instruction 'span'). The branch displacement optimisation problem consists of encoding these span-dependent instructions in such a way that the resulting program is as small as possible.

This problem is the subject of the present paper. After introducing the problem in more detail, we will discuss the solutions used by other compilers, present the algorithm we use in the CerCo assembler, and discuss its verification, that is the proofs of termination and correctness using the Matita proof assistant [1].

* Research supported by the CerCo project, within the Future and Emerging Technologies (FET) programme of the Seventh Framework Programme for Research of the European Commission, under FET-Open grant number 243881

Formulating the final statement of correctness and finding the loop invariants have been non-trivial tasks and are, indeed, the main contribution of this paper. It has required considerable care and fine-tuning to formulate not only the minimal statement required for the ulterior proof of correctness of the assembler, but also the minimal set of invariants needed for the proof of correctness of the algorithm.

The research presented in this paper has been executed within the CerCo project which aims at formally verifying a C compiler with cost annotations. The target architecture for this project is the MCS-51, whose instruction set contains span-dependent instructions. Furthermore, its maximum addressable memory size is very small (64 Kb), which makes it important to generate programs that are as small as possible. With this optimisation, however, comes increased complexity and hence increased possibility for error. We must make sure that the branch instructions are encoded correctly, otherwise the assembled program will behave unpredictably.

All Matita files related to this development can be found on the CerCo website, `http://cerco.cs.unibo.it`. The specific part that contains the branch displacement algorithm is in the `ASM` subdirectory, in the files `PolicyFront.ma`, `PolicyStep.ma` and `Policy.ma`.

2 The branch displacement optimisation problem

In most modern instruction sets that have them, the only span-dependent instructions are branch instructions. Taking the ubiquitous x86-64 instruction set as an example, we find that it contains eleven different forms of the unconditional branch instruction, all with different ranges, instruction sizes and semantics (only six are valid in 64-bit mode, for example). Some examples are shown in Figure 1 (see also [4]).

Instruction	Size (bytes)	Displacement range
Short jump	2	-128 to 127 bytes
Relative near jump	5	-2^{32} to $2^{32}-1$ bytes
Absolute near jump	6	one segment (64-bit address)
Far jump	8	entire memory (indirect jump)

Figure 1: List of x86 branch instructions

The chosen target architecture of the CerCo project is the Intel MCS-51, which features three types of branch instructions (or jump instructions; the two terms are used interchangeably), as shown in Figure 2.

Conditional branch instructions are only available in short form, which means that a conditional branch outside the short address range has to be encoded using three branch instructions (for instructions whose logical negation is available, it

Instruction	Size (bytes)	Execution time (cycles)	Displacement range
SJMP (‘short jump’)	2	2	-128 to 127 bytes
AJMP (‘absolute jump’)	2	2	one segment (11-bit address)
LJMP (‘long jump’)	3	3	entire memory

Figure 2: List of MCS-51 branch instructions

can be done with two branch instructions, but for some instructions this is not the case). The call instruction is only available in absolute and long forms.

Note that even though the MCS-51 architecture is much less advanced and much simpler than the x86-64 architecture, the basic types of branch instruction remain the same: a short jump with a limited range, an intra-segment jump and a jump that can reach the entire available memory.

Generally, in code fed to the assembler as input, the only difference between branch instructions is semantics, not span. This means that a distinction is made between an unconditional branch and the several kinds of conditional branch, but not between their short, absolute or long variants.

The algorithm used by the assembler to encode these branch instructions into the different machine instructions is known as the *branch displacement algorithm*. The optimisation problem consists of finding as small an encoding as possible, thus minimising program length and execution time.

Similar problems, e.g. the branch displacement optimisation problem for other architectures, are known to be NP-complete [7,9], which could make finding an optimal solution very time-consuming.

The canonical solution, as shown by Szymanski [9] or more recently by Dickson [2] for the x86 instruction set, is to use a fixed point algorithm that starts with the shortest possible encoding (all branch instruction encoded as short jumps, which is likely not a correct solution) and then iterates over the source to re-encode those branch instructions whose target is outside their range.

Adding absolute jumps

In both papers mentioned above, the encoding of a jump is only dependent on the distance between the jump and its target: below a certain value a short jump can be used; above this value the jump must be encoded as a long jump.

Here, termination of the smallest fixed point algorithm is easy to prove. All branch instructions start out encoded as short jumps, which means that the distance between any branch instruction and its target is as short as possible (all the intervening jumps are short). If, in this situation, there is a branch instruction b whose span is not within the range for a short jump, we can be sure that we can never reach a situation where the span of j is so small that it can be encoded as a short jump. This argument continues to hold throughout the subsequent iterations of the algorithm: short jumps can change into long

```
    jmp X
    ...
L0: ...
% Start of new segment if
% jmp X is encoded as short
    ...
    jmp L0
```

(a) Example of a program where a long jump becomes absolute

```
L0: jmp X
X:  ...
    ...
L1: ...
% Start of new segment if
% jmp X is encoded as short
    ...
    jmp L1
    ...
    jmp L1
    ...
    jmp L1
    ...
```

(b) Example of a program where the fixed-point algorithm is not optimal

jumps, but not *vice versa*, as spans only increase. Hence, the algorithm either terminates early when a fixed point is reached or when all short jumps have been changed into long jumps.

Also, we can be certain that we have reached an optimal solution: a short jump is only changed into a long jump if it is absolutely necessary.

However, neither of these claims (termination nor optimality) hold when we add the absolute jump. With absolute jumps, the encoding of a branch instruction no longer depends only on the distance between the branch instruction and its target. An absolute jump is possible when instruction and target are in the same segment (for the MCS-51, this means that the first 5 bytes of their addresses have to be equal). It is therefore entirely possible for two branch instructions with the same span to be encoded in different ways (absolute if the branch instruction and its target are in the same segment, long if this is not the case).

This invalidates our earlier termination argument: a branch instruction, once encoded as a long jump, can be re-encoded during a later iteration as an absolute jump. Consider the program shown in Figure 3a. At the start of the first iteration, both the branch to X and the branch to L_0 are encoded as small jumps. Let us assume that in this case, the placement of L_0 and the branch to it are such that L_0 is just outside the segment that contains this branch. Let us also assume that the distance between L_0 and the branch to it is too large for the branch instruction to be encoded as a short jump.

All this means that in the second iteration, the branch to L_0 will be encoded as a long jump. If we assume that the branch to X is encoded as a long jump as well, the size of the branch instruction will increase and L_0 will be 'propelled' into the same segment as its branch instruction, because every subsequent instruction will move one byte forward. Hence, in the third iteration, the branch to L_0 can be encoded as an absolute jump. At first glance, there is nothing that prevents us from constructing a configuration where two branch instructions interact in such a way as to iterate indefinitely between long and absolute encodings.

This situation mirrors the explanation by Szymanski [9] of why the branch displacement optimisation problem is NP-complete. In this explanation, a condition for NP-completeness is the fact that programs be allowed to contain *pathological* jumps. These are branch instructions that can normally not be encoded as a short(er) jump, but gain this property when some other branch instructions are encoded as a long(er) jump. This is exactly what happens in Figure 3a. By encoding the first branch instruction as a long jump, another branch instruction switches from long to absolute (which is shorter).

In addition, our previous optimality argument no longer holds. Consider the program shown in Figure 3b. Suppose that the distance between `L`$_0$ and `L`$_1$ is such that if `jmp X` is encoded as a short jump, there is a segment border just after `L`$_1$. Let us also assume that all three branches to `L`$_1$ are in the same segment, but far enough away from `L`$_1$ that they cannot be encoded as short jumps.

Then, if `jmp X` were to be encoded as a short jump, which is clearly possible, all of the branches to `L`$_1$ would have to be encoded as long jumps. However, if `jmp X` were to be encoded as a long jump, and therefore increase in size, `L`$_1$ would be ‘propelled’ across the segment border, so that the three branches to `L`$_1$ could be encoded as absolute jumps. Depending on the relative sizes of long and absolute jumps, this solution might actually be smaller than the one reached by the smallest fixed point algorithm.

3 Our algorithm

3.1 Design decisions

Given the NP-completeness of the problem, finding optimal solutions (using, for example, a constraint solver) can potentially be very costly.

The SDCC compiler [8], which has a backend targeting the MCS-51 instruction set, simply encodes every branch instruction as a long jump without taking the distance into account. While certainly correct (the long jump can reach any destination in memory) and a very fast solution to compute, it results in a less than optimal solution in terms of output size and execution time.

On the other hand, the `gcc` compiler suite, while compiling C on the x86 architecture, uses a greatest fix point algorithm. In other words, it starts with all branch instructions encoded as the largest jumps available, and then tries to reduce the size of branch instructions as much as possible.

Such an algorithm has the advantage that any intermediate result it returns is correct: the solution where every branch instruction is encoded as a large jump is always possible, and the algorithm only reduces those branch instructions whose destination address is in range for a shorter jump. The algorithm can thus be stopped after a determined number of steps without sacrificing correctness.

The result, however, is not necessarily optimal. Even if the algorithm is run until it terminates naturally, the fixed point reached is the *greatest* fixed point, not the least fixed point. Furthermore, `gcc` (at least for the x86 architecture) only uses short and long jumps. This makes the algorithm more efficient, as shown in the previous section, but also results in a less optimal solution.

In the CerCo assembler, we opted at first for a least fixed point algorithm, taking absolute jumps into account.

Here, we ran into a problem with proving termination, as explained in the previous section: if we only take short and long jumps into account, the jump encoding can only switch from short to long, but never in the other direction. When we add absolute jumps, however, it is theoretically possible for a branch instruction to switch from absolute to long and back, as previously explained. Proving termination then becomes difficult, because there is nothing that precludes a branch instruction from oscillating back and forth between absolute and long jumps indefinitely.

To keep the algorithm in the same complexity class and more easily prove termination, we decided to explicitly enforce the 'branch instructions must always grow longer' requirement: if a branch instruction is encoded as a long jump in one iteration, it will also be encoded as a long jump in all the following iterations. Therefore the encoding of any branch instruction can change at most two times: once from short to absolute (or long), and once from absolute to long.

There is one complicating factor. Suppose that a branch instruction is encoded in step n as an absolute jump, but in step $n + 1$ it is determined that (because of changes elsewhere) it can now be encoded as a short jump. Due to the requirement that the branch instructions must always grow longer, the branch encoding will be encoded as an absolute jump in step $n + 1$ as well.

This is not necessarily correct. A branch instruction that can be encoded as a short jump cannot always also be encoded as an absolute jump, as a short jump can bridge segments, whereas an absolute jump cannot. Therefore, in this situation we have decided to encode the branch instruction as a long jump, which is always correct.

The resulting algorithm, therefore, will not return the least fixed point, as it might have too many long jumps. However, it is still better than the algorithms from SDCC and `gcc`, since even in the worst case, it will still return a smaller or equal solution.

Experimenting with our algorithm on the test suite of C programs included with gcc 2.3.3 has shown that on average, about 25 percent of jumps are encoded as short or absolute jumps by the algorithm. As not all instructions are jumps, this does not make for a large reduction in size, but it can make for a reduction in execution time: if jumps are executed multiple times, for example in loops, the fact that short jumps take less cycles to execute than long jumps can have great effect.

As for complexity, there are at most $2n$ iterations, where n is the number of branch instructions. Practical tests within the CerCo project on small to medium pieces of code have shown that in almost all cases, a fixed point is reached in 3 passes. Only in one case did the algorithm need 4. This is not surprising: after all, the difference between short/absolute and long jumps is only one byte (three for conditional jumps). For a change from short/absolute to long to have an effect on other jumps is therefore relatively uncommon, which explains why a fixed point is reached so quickly.

3.2 The algorithm in detail

The branch displacement algorithm forms part of the translation from pseudocode to assembler. More specifically, it is used by the function that translates pseudo-addresses (natural numbers indicating the position of the instruction in the program) to actual addresses in memory. Note that in pseudocode, all instructions are of size 1.

Our original intention was to have two different functions, one function `policy` $: \mathbb{N} \rightarrow \{$`short_jump`, `absolute_jump`, `long_jump`$\}$ to associate jumps to their intended encoding, and a function $\sigma : \mathbb{N} \rightarrow$ `Word` to associate pseudo-addresses to machine addresses. σ would use `policy` to determine the size of jump instructions. This turned out to be suboptimal from the algorithmic point of view and impossible to prove correct.

From the algorithmic point of view, in order to create the `policy` function, we must necessarily have a translation from pseudo-addresses to machine addresses (i.e. a σ function): in order to judge the distance between a jump and its destination, we must know their memory locations. Conversely, in order to create the σ function, we need to have the `policy` function, otherwise we do not know the sizes of the jump instructions in the program.

Much the same problem appears when we try to prove the algorithm correct: the correctness of `policy` depends on the correctness of σ, and the correctness of σ depends on the correctness of `policy`.

We solved this problem by integrating the `policy` and σ algorithms. We now have a function $\sigma : \mathbb{N} \rightarrow$ `Word` $\times$ `bool` which associates a pseudo-address to a machine address. The boolean denotes a forced long jump; as noted in the previous section, if during the fixed point computation an absolute jump changes to be potentially re-encoded as a short jump, the result is actually a long jump. It might therefore be the case that jumps are encoded as long jumps without this actually being necessary, and this information needs to be passed to the code generating function.

The assembler function encodes the jumps by checking the distance between source and destination according to σ, so it could select an absolute jump in a situation where there should be a long jump. The boolean is there to prevent this from happening by indicating the locations where a long jump should be encoded, even if a shorter jump is possible. This has no effect on correctness, since a long jump is applicable in any situation.

The algorithm, shown in Figure 4, works by folding the function F over the entire program, thus gradually constructing *sigma*. This constitutes one step in the fixed point calculation; successive steps repeat the fold until a fixed point is reached. We have abstracted away the case where an instruction is not a jump, since the size of these instructions is constant.

Parameters of the function F are:

```
function F($labels$,$old\_sigma$,$instr$,$ppc$,$acc$)
    $\langle added, pc, sigma \rangle \leftarrow acc$
    if $instr$ is a backward jump to $j$ then
        $length \leftarrow$ jump_size($pc, sigma_1(labels(j))$)                ▷ compute jump distance
    else if $instr$ is a forward jump to $j$ then
        $length \leftarrow$ jump_size($pc, old\_sigma_1(labels(j)) + added$)
    end if
    $old\_length \leftarrow$ old_sigma$_1$($ppc$)
    $new\_length \leftarrow \max(old\_length, length)$                ▷ length must never decrease
    $old\_size \leftarrow$ old_sigma$_2$($ppc$)
    $new\_size \leftarrow$ instruction_size($instr, new\_length$)                ▷ compute size in bytes
    $new\_added \leftarrow added + (new\_size - old\_size)$    ▷ keep track of total added bytes
    $new\_sigma \leftarrow old\_sigma$
    $new\_sigma_1(ppc + 1) \leftarrow pc + new\_size$
    $new\_sigma_2(ppc) \leftarrow new\_length$                ▷ update $\sigma$
return $\langle new\_added, pc + new\_size, new\_sigma \rangle$
end function
```

Figure 4: The heart of the algorithm

- a function *labels* that associates a label to its pseudo-address;
- *old_sigma*, the σ function returned by the previous iteration of the fixed point calculation;
- *instr*, the instruction currently under consideration;
- *ppc*, the pseudo-address of *instr*;
- *acc*, the fold accumulator, which contains *added* (the number of bytes added to the program size with respect to the previous iteration), *pc* (the highest memory address reached so far), and of course *sigma*, the σ function under construction.

The first two are parameters that remain the same through one iteration, the final three are standard parameters for a fold function (including *ppc*, which is simply the number of instructions of the program already processed).

The σ functions used by F are not of the same type as the final σ function: they are of type $\sigma : \mathbb{N} \to \mathbb{N} \times \{\texttt{short\_jump}, \texttt{absolute\_jump}, \texttt{long\_jump}\}$; a function that associates a pseudo-address with a memory address and a jump length. We do this to ease the comparison of jump lengths between iterations. In the algorithm, we use the notation $sigma_1(x)$ to denote the memory address corresponding to x, and $sigma_2(x)$ for the jump length corresponding to x.

Note that the σ function used for label lookup varies depending on whether the label is behind our current position or ahead of it. For backward branches, where the label is behind our current position, we can use *sigma* for lookup, since its memory address is already known. However, for forward branches, the memory address of the address of the label is not yet known, so we must use *old_sigma*.

We cannot use *old_sigma* without change: it might be the case that we have already increased the size of some branch instructions before, making the pro-

```
sigma_policy_specification ≡ $\lambda program.\lambda sigma.$
    $sigma\ 0 = 0 \wedge$
    let $instr\_list \equiv code\ program$ in
      $\forall ppc. ppc < |instr\_list| \rightarrow$
      let $pc \equiv sigma\ ppc$ in
      let $instruction \equiv$ fetch_pseudo_instruction $instr\_list\ ppc$ in
      let $next\_pc \equiv sigma\ (ppc + 1)$ in
        $next\_pc = pc +$ instruction_size $sigma\ ppc\ instruction \wedge$
        $(pc +$ instruction_size $sigma\ ppc\ instruction < 2^{16} \vee$
        $(\forall ppc'. ppc' < |instr\_list| \rightarrow ppc < ppc' \rightarrow$
        let $instruction' \equiv$ fetch_pseudo_instruction $instr\_list\ ppc'$ in
         instruction_size $sigma\ ppc'\ instruction' = 0) \wedge$
        $pc +$ instruction_size $sigma\ ppc\ instruction = 2^{16})$
```

Figure 5: Main correctness statement

gram longer and moving every instruction forward. We must compensate for this by adding the size increase of the program to the label's memory address according to *old_sigma*, so that branch instruction spans do not get compromised.

4 The proof

In this section, we present the correctness proof for the algorithm in more detail. The main correctness statement is shown, slightly simplified, in Figure 5. Informally, this means that when fetching a pseudo-instruction at *ppc*, the translation by σ of $ppc + 1$ is the same as $\sigma(ppc)$ plus the size of the instruction at *ppc*. That is, an instruction is placed consecutively after the previous one, and there are no overlaps. The rest of the statement deals with memory size: either the next instruction fits within memory ($next\_pc < 2^{16}$) or it ends exactly at the limit memory, in which case it must be the last translated instruction in the program (enforced by specfiying that the size of all subsequent instructions is 0: there may be comments or cost annotations that are not translated).

Finally, we enforce that the program starts at address 0, i.e. $\sigma(0) = 0$. It may seem strange that we do not explicitly include a safety property stating that every jump instruction is of the right type with respect to its target (akin to the lemma from Figure 7), but this is not necessary. The distance is recalculated according to the instruction addresses from σ, which implicitly expresses safety.

Since our computation is a least fixed point computation, we must prove termination in order to prove correctness: if the algorithm is halted after a number of steps without reaching a fixed point, the solution is not guaranteed to be correct. More specifically, branch instructions might be encoded which do not coincide with the span between their location and their destination.

Proof of termination rests on the fact that the encoding of branch instructions can only grow larger, which means that we must reach a fixed point after at most $2n$ iterations, with n the number of branch instructions in the program. This worst case is reached if at every iteration, we change the encoding of exactly one branch instruction; since the encoding of any branch instruction can change first from short to absolute, and then to long, there can be at most $2n$ changes.

4.1 Fold invariants

In this section, we present the invariants that hold during the fold of $\texttt{F}$ over the program. These will be used later on to prove the properties of the iteration. During the fixed point computation, the σ function is implemented as a trie for ease of access; computing $\sigma(x)$ is achieved by looking up the value of x in the trie. Actually, during the fold, the value we pass along is a pair $\mathbb{N} \times$ `ppc_pc_map`. The first component is the number of bytes added to the program so far with respect to the previous iteration, and the second component, `ppc_pc_map`, is the actual σ trie (which we'll call *strie* to avoid confusion).

$$\begin{aligned}&\texttt{out\_of\_program\_none} \equiv \lambda prefix.\lambda strie.\\&\quad \forall i.i < 2^{16} \to (i > |prefix| \leftrightarrow \texttt{lookup\_opt}\ i\ (\texttt{snd}\ strie) = \texttt{None})\end{aligned}$$

The first invariant states that any pseudo-address not yet examined is not present in the lookup trie.

$$\begin{aligned}&\texttt{not\_jump\_default} \equiv \lambda prefix.\lambda strie.\forall i.i < |prefix| \to\\&\quad \neg\texttt{is\_jump}\ (\texttt{nth}\ i\ prefix) \to \texttt{lookup}\ i\ (\texttt{snd}\ strie) = \texttt{short\_jump}\end{aligned}$$

This invariant states that when we try to look up the jump length of a pseudo-address where there is no branch instruction, we will get the default value, a short jump.

$$\begin{aligned}&\texttt{jump\_increase} \equiv \lambda pc.\lambda op.\lambda p.\forall i.i < |prefix| \to\\&\quad \textbf{let}\ oj \equiv \texttt{lookup}\ i\ (\texttt{snd}\ op)\ \textbf{in}\\&\quad \textbf{let}\ j \equiv \texttt{lookup}\ i\ (\texttt{snd}\ p)\ \textbf{in}\ \texttt{jmpleq}\ oj\ j\end{aligned}$$

This invariant states that between iterations (with *op* being the previous iteration, and p the current one), jump lengths either remain equal or increase. It is needed for proving termination. We now proceed with the safety lemmas. The lemma in Figure 6 is a temporary formulation of the main property `sigma_policy_specification`. Its main difference from the final version is that it uses `instruction_size_jmplen` to compute the instruction size. This function uses j to compute the span of branch instructions (i.e. it uses the σ under construction), instead of looking at the distance between source and destination. This is because σ is still under construction; we will prove below that after the final iteration, `sigma_compact_unsafe` is equivalent to the main property in Figure 7 which holds at the end of the computation. We compute the distance using

```
sigma_compact_unsafe ≡ $\lambda prefix.\lambda strie.\forall n.n < |prefix| \to$
    match lookup_opt $n$ (snd $strie$) with
        None ⇒ False
        Some $\langle pc, j\rangle$ ⇒
        match lookup_opt $(n+1)$ (snd $strie$) with
            None ⇒ False
            Some $\langle pc_1, j_1\rangle \Rightarrow pc_1 = pc+$
              instruction_size_jmplen $j$ (nth $n$ $prefix$)
```

Figure 6: Temporary safety property

the memory address of the instruction plus its size. This follows the behaviour of the MCS-51 microprocessor, which increases the program counter directly after fetching, and only then executes the branch instruction (by changing the program counter again).

There are also some simple, properties to make sure that our policy remains consistent, and to keep track of whether the fixed point has been reached. We do not include them here in detail. Two of these properties give the values of σ for the start and end of the program; $\sigma(0) = 0$ and $\sigma(n)$, where n is the number of instructions up until now, is equal to the maximum memory address so far. There are also two properties that deal with what happens when the previous iteration does not change with respect to the current one. *added* is a variable that keeps track of the number of bytes we have added to the program size by changing the encoding of branch instructions. If *added* is 0, the program has not changed and vice versa.

We need to use two different formulations, because the fact that *added* is 0 does not guarantee that no branch instructions have changed. For instance, it is possible that we have replaced a short jump with an absolute jump, which does not change the size of the branch instruction. Therefore `policy_pc_equal` states that $old\_sigma_1(x) = sigma_1(x)$, whereas `policy_jump_equal` states that $old\_sigma_2(x) = sigma_2(x)$. This formulation is sufficient to prove termination and compactness.

Proving these invariants is simple, usually by induction on the prefix length.

4.2 Iteration invariants

These are invariants that hold after the completion of an iteration. The main difference between these invariants and the fold invariants is that after the completion of the fold, we check whether the program size does not supersede 64 Kb, the maximum memory size the MCS-51 can address. The type of an iteration therefore becomes an option type: `None` in case the program becomes larger than 64 Kb, or `Some` σ otherwise. We also no longer pass along the number of

```
sigma_safe ≡ $\lambda prefix.\lambda labels.\lambda old\_strie.\lambda strie.\forall i.i < |prefix| \rightarrow$
    $\forall dest\_label.$is_jump_to (nth i prefix) $dest\_label \rightarrow$
    let $paddr \equiv$ lookup $labels$ $dest\_label$ in
    let $\langle j, src, dest\rangle \equiv$ if $paddr \leq i$ then
            let $\langle \_, j\rangle \equiv$ lookup $i$ (snd $strie$) in
            let $\langle pc\_plus\_jl, \_\rangle \equiv$ lookup $(i+1)$ (snd $strie$) in
            let $\langle addr, \_\rangle \equiv$ lookup $paddr$ (snd $strie$) in
            $\langle j, pc\_plus\_jl, addr\rangle$
      else
            let $\langle \_, j\rangle \equiv$ lookup $i$ (snd $strie$) in
            let $\langle pc\_plus\_jl, \_\rangle \equiv$ lookup $(i+1)$ (snd $old\_strie$) in
            let $\langle addr, \_\rangle \equiv$ lookup $paddr$ (snd $old\_strie$) in
            $\langle j, pc\_plus\_jl, addr\rangle$in
      match $j$ with
            short_jump ⇒ short_jump_valid $src$ $dest$
            absolute_jump ⇒ absolute_jump_valid $src$ $dest$
            long_jump ⇒ True
```

Figure 7: Safety property

bytes added to the program size, but a boolean that indicates whether we have changed something during the iteration or not.

If the iteration returns `None`, which means that it has become too large for memory, there is an invariant that states that the previous iteration cannot have every branch instruction encoded as a long jump. This is needed later in the proof of termination. If the iteration returns `Some` σ, the fold invariants are retained without change.

Instead of using `sigma_compact_unsafe`, we can now use the proper invariant:

```
sigma_compact ≡ $\lambda program.\lambda sigma.$
    $\forall n.n < |program| \rightarrow$
    match lookup_opt $n$ (snd $sigma$) with
        None ⇒ False
        Some $\langle pc, j\rangle$ ⇒
        match lookup_opt $(n+1)$ (snd $sigma$) with
            None ⇒ False
            Some$\langle pc1, j1\rangle$ ⇒
              $pc1 = pc +$ instruction_size $n$ (nth $n$ $program$)
```

This is almost the same invariant as `sigma_compact_unsafe`, but differs in that it computes the sizes of branch instructions by looking at the distance between position and destination using σ. In actual use, the invariant is qualified: σ is compact if there have been no changes (i.e. the boolean passed along is `true`). This is to reflect the fact that we are doing a least fixed point computation: the result is only correct when we have reached the fixed point.

There is another, trivial, invariant in case the iteration returns `Some` σ: it must hold that `fst` *sigma* $< 2^{16}$. We need this invariant to make sure that addresses do not overflow.

The proof of `nec_plus_ultra` goes as follows: if we return `None`, then the program size must be greater than 64 Kb. However, since the previous iteration did not return `None` (because otherwise we would terminate immediately), the program size in the previous iteration must have been smaller than 64 Kb.

Suppose that all the branch instructions in the previous iteration are encoded as long jumps. This means that all branch instructions in this iteration are long jumps as well, and therefore that both iterations are equal in the encoding of their branch instructions. Per the invariant, this means that *added* $= 0$, and therefore that all addresses in both iterations are equal. But if all addresses are equal, the program sizes must be equal too, which means that the program size in the current iteration must be smaller than 64 Kb. This contradicts the earlier hypothesis, hence not all branch instructions in the previous iteration are encoded as long jumps.

The proof of `sigma_compact` follows from `sigma_compact_unsafe` and the fact that we have reached a fixed point, i.e. the previous iteration and the current iteration are the same. This means that the results of `instruction_size_jmplen` and `instruction_size` are the same.

4.3 Final properties

These are the invariants that hold after $2n$ iterations, where n is the program size (we use the program size for convenience; we could also use the number of branch instructions, but this is more complex). Here, we only need `out_of_program_none`, `sigma_compact` and the fact that $\sigma(0) = 0$.

Termination can now be proved using the fact that there is a $k \leq 2n$, with n the length of the program, such that iteration k is equal to iteration $k+1$. There are two possibilities: either there is a $k < 2n$ such that this property holds, or every iteration up to $2n$ is different. In the latter case, since the only changes between the iterations can be from shorter jumps to longer jumps, in iteration $2n$ every branch instruction must be encoded as a long jump. In this case, iteration $2n$ is equal to iteration $2n + 1$ and the fixed point is reached.

5 Conclusion

In the previous sections we have discussed the branch displacement optimisation problem, presented an optimised solution, and discussed the proof of termination and correctness for this algorithm, as formalised in Matita.

The algorithm we have presented is fast and correct, but not optimal; a true optimal solution would need techniques like constraint solvers. While outside the scope of the present research, it would be interesting to see if enough heuristics could be found to make such a solution practical for implementing in an existing compiler; this would be especially useful for embedded systems, where it is important to have as small a solution as possible.

In itself the algorithm is already useful, as it results in a smaller solution than the simple 'every branch instruction is long' used up until now—and with only 64 Kb of memory, every byte counts. It also results in a smaller solution than the greatest fixed point algorithm that `gcc` uses. It does this without sacrificing speed or correctness.

The certification of an assembler that relies on the branch displacement algorithm described in this paper was presented in [6]. The assembler computes the σ map as described in this paper and then works in two passes. In the first pass it builds a map from instruction labels to addresses in the assembly code. In the second pass it iterates over the code, translating every pseudo jump at address *src* to a label l associated to the assembly instruction at address *dst* to a jump of the size dictated by $(\sigma\ src)$ to $(\sigma\ dst)$. In case of conditional jumps, the translated jump may be implemented with a series of instructions.

The proof of correctness abstracts over the algorithm used and only relies on `sigma_policy_specification` (page 5). It is a variation of a standard 1-to-many forward simulation proof [5]. The relation R between states just maps every code address *ppc* stored in registers or memory to $(\sigma\ ppc)$. To identify the code addresses, an additional data structure is always kept together with the source state and is updated by the semantics. The semantics is preserved only for those programs whose source code operations $(f\ ppc_1\ \ldots\ ppc_n)$ applied to code addresses $ppc_1 \ldots ppc_n$ are such that $(f\ (\sigma\ ppc_1)\ ...\ (\sigma\ ppc_n) = f\ ppc_1\ ppc_n))$. For example, an injective σ preserves a binary equality test f for code addresses, but not pointer subtraction.

The main lemma (fetching simulation), which relies on `sigma_policy_specification` and is established by structural induction over the source code, says that fetching an assembly instruction at position ppc is equal to fetching the translation of the instruction at position $(\sigma\ ppc)$, and that the new incremented program counter is at the beginning of the next instruction (compactness). The only exception is when the instruction fetched is placed at the end of code memory and is followed only by dead code. Execution simulation is trivial because of the restriction over well behaved programs w.r.t. sigma. The condition $\sigma\ 0 = 0$ is necessary because the hardware model prescribes that the first instruction to be executed will be at address 0. For the details see [6].

Instead of verifying the algorithm directly, another solution to the problem would be to run an optimisation algorithm, and then verify the safety of the result using a verified validator. Such a validator would be easier to verify than the algorithm itself and it would also be efficient, requiring only a linear pass over the source code to test the specification. However, it is surely also interesting to formally prove that the assembler never rejects programs that should be accepted, i.e. that the algorithm itself is correct. This is the topic of the current paper.

5.1 Related work

As far as we are aware, this is the first formal discussion of the branch displacement optimisation algorithm.

The CompCert project is another verified compiler project. Their backend [5] generates assembly code for (amongst others) subsets of the PowerPC and x86 (32-bit) architectures. At the assembly code stage, there is no distinction between the span-dependent jump instructions, so a branch displacement optimisation algorithm is not needed.

References

1. Asperti, A., Sacerdoti Coen, C., Tassi, E., Zacchiroli, S.: User interaction with the Matita proof assistant. Automated Reasoning 39, 109–139 (2007)
2. Dickson, N.G.: A simple, linear-time algorithm for x86 jump encoding. CoRR abs/0812.4973 (2008)
3. Hyde, R.: Branch displacement optimisation. http://groups.google.com/group/alt.lang.asm/msg/d31192d442accad3 (2006)
4. Intel: Intel 64 and IA-32 Architectures Developer's Manual. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html
5. Leroy, X.: A formally verified compiler back-end. Journal of Automated Reasoning 43, 363–446 (2009), http://dx.doi.org/10.1007/s10817-009-9155-4, 10.1007/s10817-009-9155-4
6. Mulligan, D.P., Sacerdoti Coen, C.: On the correctness of an optimising assembler for the intel mcs-51 microprocessor. In: Hawblitzel, C., Miller, D. (eds.) Certified Programs and Proofs, Lecture Notes in Computer Science, vol. 7679, pp. 43–59. Springer Berlin Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-35308-6_7
7. Robertson, E.L.: Code generation and storage allocation for machines with span-dependent instructions. ACM Trans. Program. Lang. Syst. 1(1), 71–83 (Jan 1979), http://doi.acm.org/10.1145/357062.357067
8. Small device C compiler 3.1.0. http://sdcc.sourceforge.net/ (2011)
9. Szymanski, T.G.: Assembling code for machines with span-dependent instructions. Commun. ACM 21(4), 300–308 (Apr 1978), http://doi.acm.org/10.1145/359460.359474