A Methodology for Compilation
of High-Integrity Real-Time Programs

Karl Lermer   Colin Fidge

Software Verification Research Centre
School of Information Technology
The University of Queensland

Abstract. A practical methodology for compilation of trustworthy real-time programs is introduced. It combines new program development and timing analysis techniques with traditional compilation and assembly technologies.

1   Introduction

Programming real-time systems in a high-level language is difficult because it is the machine code generated by the compiler and assembler, not the high-level source program, that ultimately determines timing correctness. Contemporary compilers make no attempt to generate code with *predictable* timing characteristics [8], undermining their value for real-time applications. Consequently, safety-critical real-time programs are often written in assembly language, forsaking the well-established productivity benefits of high-level language programming.

The *Topaz* project is currently applying formal methods to compilation of trustworthy real-time programs. Topaz comprises

- a real-time refinement theory for formally translating high-level programming language 'specifications' to time-verified machine code 'implementations', and
- a practical methodology for instantiating this theory in existing, or planned, programming environments.

In this article we introduce the second of these two aspects, the Topaz methodology, via a small example.

2   The Topaz methodology

As shown in Figure 1, the Topaz methodology extends traditional high-level language (HLL) program compilation. The *program development*, *compilation* and *assembly* phases are extensions of their traditional (untimed) counterparts. The *timing path analysis* and *timing verification* phases are new. Below we illustrate key aspects of the methodology using a small "transmitter" example [1].
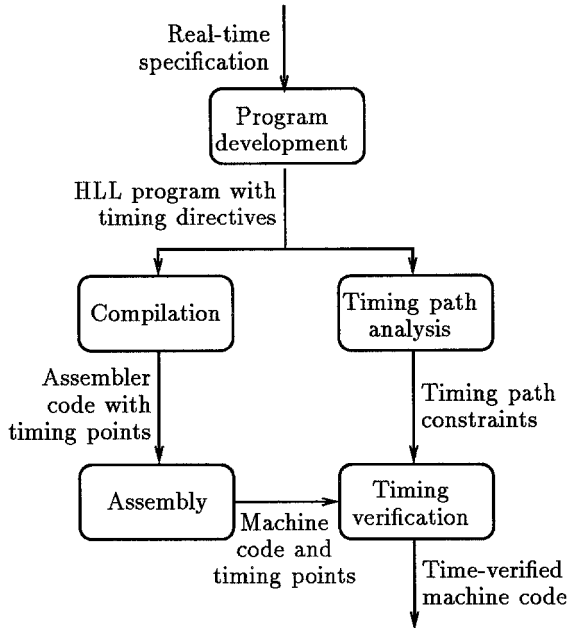
Fig. 1. Overview of the Topaz real-time compilation strategy.

2.1 Program development

Figure 2 shows the time-annotated HLL program fragment for our case study, in guarded command language notation. Hayes and Utting explain in detail how this program can be formally developed using stepwise refinement [1].

Its functional behaviour is straightforward. It assumes the existence of an array *msg*, of length *size*, containing a message to be transmitted. Special variable *out* is a memory-mapped output location. The program fragment of interest, between labels A and D, simply writes each character in *msg* to *out*.

The required timing behaviour is expressed through the directives labelled A, B, C and D. Statement A is an *assumption* [6] that we may expect to be true when this program segment starts executing. Let *start* be the absolute time by which the first character must be available, and *early* be the minimum duration before *start* at which the program fragment begins. Statement A tells us that initially the current time, denoted by imaginary specification variable **now**, is expected to be at least *early* time units before *start*. No executable code is generated for assumptions [6], but we may use them when analysing the program.

Directive B is a so-called real-time *coercion* [1]. Let duration *chsep* be the required separation of characters transmitted. The **deadline** directive occurs immediately after transmission of the nth character, and tells us that we must reach this point no later than n times *chsep* units from time *start*. This de-

```
|[ var size : nat;
      msg : array (0 .. (size − 1)) of char;
      out : char •
          ⋮
A:      {now + early ⩽ start};
      |[ var n : nat •
          n := 0;
          do n ≠ size →
              out := msg(n);
B:            deadline start + chsep ∗ n;
C:            delay until start + chsep ∗ n + chdef;
              n := n + 1
          od
      ]| ;
D:      deadline start + chsep ∗ size
   ]|
```

Fig. 2. High-level language program annotated with timing directives [1].

fines the *initial* availability of the nth output character, since it constrains the preceding assignment statement to finish *before* this time.

Statement C is a conventional **delay until** statement of the form found in a programming language like Ada 95 [2]. Let *chdef* be the minimum duration for which each output character must be stably defined. The delay statement tells us that we may pass this point no earlier than n times *chsep* plus *chdef* units from time *start*. This defines the *final* availability of the nth output character, since it prevents the program from performing any actions that may change *out* until *after* this time.

Finally, statement D is another coercion, this time placing an overall timing constraint on the whole code segment to finish within a time proportional to the number of characters in *msg* from *start*.

2.2 Timing path analysis

Hayes and Utting explain how the timing constraints expressed in Figure 2 can be simplified using control flow analysis [1]. All control paths between significant pairs of timing directives can be statically identified, and an overall worst-case execution time constraint for each such path defined. This reduces the timing requirements to simple execution time bounds on 'straight-line' code segments.

For the example in Figure 2, Hayes and Utting identify the following worst-case execution time constraints.

$$WCET(A \rightsquigarrow D) = early \qquad\qquad WCET(A \rightsquigarrow B) = early$$
$$WCET(C \rightsquigarrow B) = chsep - chdef \qquad WCET(C \rightsquigarrow D) = chsep - chdef$$

Path $A \leadsto D$ is that followed when the loop is not entered at all, i.e., when *size* is 0. In this case the expression at point D simplifies to '*start*', so the time available to get from A to D may be as low as '*early*'.

Path $A \leadsto B$ is the time taken to transmit the first character, including the loop initialisation activities. In this case n is 0 at point B, so again as few as *early* time units may be available to reach this point from A.

Path $C \leadsto B$ is the time taken to iterate, after the end of the stable period for the nth character at time $start + chsep * n + chdef$, up to the beginning of the stable period for the $(n + 1)^{\text{th}}$ character , and includes the time required to evaluate the (true) loop guard. In this case n is incremented in going from C to B, so '$n + 1$' must be substituted for 'n' in the expression at point B. Simplification then yields the overall constraint shown above.

Path $C \leadsto D$ is the time required to exit the program after the last character has been transmitted at time $start + chsep * (size - 1) + chdef$, and includes the overhead of evaluating the (false) guard for the last time. In this case it is known at point C that n equals $size - 1$.

The final machine code program must be proven to satisfy these worst-case path constraints. However, it is not necessary to check the timing behaviour of 'best-case' paths, such as $B \leadsto C$, because their correctness is guaranteed by correct implementation of the **delay until** statement.

2.3 Compilation

Figure 3 shows a MIPS R3000 [3] assembler code program developed from the HLL program in Figure 2. Here *size* and *msg* are symbolic constants representing local addresses for these variables; *body*, *delay* and *test* are symbolic instruction addresses; *chsep* and *dt* are compile-time constants, where *dt* equals *start* + *chdef*; and *clock* and *out* are hardware-dependent global addresses used to read the current absolute time and write to the output location, respectively (we assume an on-board synchronous clock that can be read with a normal load instruction, and a memory-mapped output location). Variables $t0 to $t7 are symbolic register names [3].

Keep in mind that, due to the way instructions overlap in a RISC pipeline, the instruction following a branch is *always* executed, whether the branch is taken or not [3]. Also the result of a load from memory that takes more than one cycle is not immediately available to the following instruction [3].

Timing points corresponding to the labelled directives in Figure 2 are prominently marked. A, B and D mark the points where upper timing bounds occurred. No code is generated for these constructs. C^α and C^ω mark the beginning and end of the **delay until** statement, respectively. The compiler generates several instructions to implement the lower timing bound required by HLL statement C. Since this statement is no longer atomic we separately mark both its beginning and end. The functional code appears in paths $A \leadsto B$ and $C^\omega \leadsto D$.

The **delay until** implementation in path $C^\alpha \leadsto C^\omega$ is a busy wait on label *delay*. At each iteration it reads the clock value, and compares it with the time at which the delay expires.

```
                    —— A ——
              la $t0, msg        t0 := msg base address
              lb $t1, size       t1 := size (length of msg)
              li $t2, 0          t2 := 0 (loop counter n)
              lb $t3, dt         t3 := first delay time (start + chdef)
              lb $t4, chsep      t4 := constant chsep
              j test             goto loop test
              nop                jump delay slot
      body :  lb $t6, ($t5)      t6 := msg(n)
              nop                load delay slot
              sb $t6, out        output location := msg(n)
                    —— B ——
                    —— Cᵅ ——
      delay : lb $t7, clock      t7 := current time
              nop                load delay slot
              sub $t7, $t3, $t7  t7 := delay time − current time
              bgtz $t7, delay    busy wait if current time < delay time
              nop                branch delay slot
              add $t3, $t3, $t4  next delay time := delay time + chsep
                    —— Cᵚ ——
              add $t2, $t2, 1    increment n
      test :  bne $t1, $t2, body goto loop body if n ≠ size
              add $t5, $t2, $t0  address of msg(n) := msg base + n
                    —— D ——
```

Fig. 3. MIPS R3000 assembler code for the high-level program in Figure 2.

2.4 Assembly

Figure 4 shows binary machine code generated for our program by the SPIM assembler [4]. The first column contains instruction addresses, and the second the instructions themselves. A mnemonic version is shown on the right.

Symbolic constants, addresses and register names have been substituted with particular values. More significantly, however, some assembler instructions have generated several machine code instructions. For instance, the first assembler 'load byte' from Figure 3 puts the value of local variable *size* in register '$t1'. The equivalent machine code consists of two instructions: firstly, a 'load upper immediate' stores data area base pointer 4097 in machine register number 1; secondly, a machine-level load byte instruction stores the contents of memory location 4097 plus offset 5 in register number 9 (the register number corresponding to name '$t1' [3]). It is for this reason that we can undertake accurate timing verification only on the final *machine code* program.

To support the timing verification phase some way of associating timing points with particular instructions is needed. The assembler must generate a table associating timing point A with address 0x00400020, point B with address 0x00400054, and so on.

```
                                          —— A ——
[0x00400020] 0x3c081001   lui $8, 4097
[0x00400024] 0x3c011001   lui $1, 4097
[0x00400028] 0x80290005   lb $9, 5($1)
[0x0040002c] 0x340a0000   ori $10, $0, 0
[0x00400030] 0x3c011001   lui $1, 4097
[0x00400034] 0x802b0006   lb $11, 6($1)
[0x00400038] 0x3c011001   lui $1, 4097
[0x0040003c] 0x802c0007   lb $12, 7($1)
[0x00400040] 0x0810001c   j 0x00400070
[0x00400044] 0x00000025   or $0, $0, $0
[0x00400048] 0x81ae0000   lb $14, 0($13)
[0x0040004c] 0x00000025   or $0, $0, $0
[0x00400050] 0xa38e8000   sb $14, −32768($28)
                                          —— B ——
                                          —— Cᵅ ——
[0x00400054] 0x838f8001   lb $15, −32767($28)
[0x00400058] 0x00000025   or $0, $0, $0
[0x0040005c] 0x016f7822   sub $15, $11, $15
[0x00400060] 0x1de0fffd   bgtz $15, −12
[0x00400064] 0x00000025   or $0, $0, $0
[0x00400068] 0x016c5820   add $11, $11, $12
                                          —— Cʷ ——
[0x0040006c] 0x214a0001   addi $10, $10, 1
[0x00400070] 0x152afff6   bne $9, $10, −40
[0x00400074] 0x01486820   add $13, $10, $8
                                          —— D ——
```

Fig. 4. MIPS R3000 machine code for the assembler program in Figure 3.

2.5 Timing verification

The timing verification phase checks the timing behaviour of machine code paths against the timing constraints derived from the HLL timing directives. For the purposes of this paper we used the SPIM S20 simulator [4] to verify the timing behaviour of the machine code program in Figure 4.

For clarity below, assume there are no caching overheads (although the SPIM tool is capable of simulating such overheads [7]). We can then treat all machine code instructions in Figure 4 as taking exactly one cycle. It is then merely necessary to count all instructions a path contains. Let $numi$ be the number of instructions for each path in Figure 4:

$$numi(A \rightsquigarrow D) = 12 \qquad\qquad numi(A \rightsquigarrow B) = 15$$
$$numi(C^\omega \rightsquigarrow B) = 6 \qquad\qquad numi(C^\omega \rightsquigarrow D) = 3$$

We must then compare the number of instructions in each machine-level path with the WCET timing constraints calculated during the HLL timing analysis

phase (Section 2.2). This tells us that we must satisfy the following four equations. Let constant t be the elapsed 'real' time per instruction, calculated as the number of cycles per instruction multiplied by the machine-specific elapsed time per processor cycle [3]. The *'lateness'* value is explained below.

$$numi(A \leadsto D) * t \leqslant early$$

$$numi(A \leadsto B) * t \leqslant early$$

$$(numi(C^\omega \leadsto B) + lateness(C)) * t \leqslant chsep - chdef$$

$$(numi(C^\omega \leadsto D) + lateness(C)) * t \leqslant chsep - chdef$$

We are not required to prove any timing properties of path $B \leadsto C^\alpha$, which ends with a **delay until** statement. Nevertheless, the possible 'lateness' of this path must be considered in analysing the *following* paths $C \leadsto B$ and $C \leadsto D$. This is because any practical implementation of the HLL statement '**delay until** E' cannot guarantee to finish at *exactly* time E. Thus *lateness* above represents the number of instructions by which a path ending in a **delay until** statement may exceed its specified finishing time. For the implementation of statement C above we can calculate this as follows.

Firstly, if *chdef* is very small, then the delay time may pass while we are still executing the first few instructions following point C^α. In the worst case *every* instruction executed between C^α and C^ω contributes to an overrun.

$$lateness_{passed} = 6 - \left\lfloor \frac{chdef}{t} \right\rfloor$$

It takes 6 cycles to get from C^α to C^ω when the *'delay'* loop in Figure 3 does not iterate. However, we subtract the number of *whole* processor cycles that can be completed before duration *chdef* expires, because these do not contribute to lateness.

Secondly, when *chdef* is large, the **delay until** implementation in Figure 3 may overrun by going around the *'delay'* loop too often. The worst case is where the delay period expires fractionally *after* reading the time from the hardware clock. In this situation it may take up to two entire iterations beyond the desired finishing time to reach point C^ω.

$$lateness_{loops} = 11$$

Thus, since the actual value of *chdef* is unknown, the most we can say about lateness is as follows.

$$lateness(C) = max\{lateness_{passed}, lateness_{loops}\} = 11$$

Substituting these calculated execution times into the WCET constraints, and simplifying, yields the following inequalities.

$$15 * t \leqslant early \qquad\qquad 17 * t \leqslant chsep - chdef$$

If the programmer-supplied values for durations *early*, *chsep* and *chdef*, together with the machine-specific constant t, satisfy these equations then the program in Figure 4 can be considered to conform with the programmer's timing requirements.

3 Conclusion

The Topaz methodology for development of verified real-time machine code integrates recent advances in real-time program specification and timing analysis into traditional compiler technology. At the time of writing we are furthering the approach through the development of a formal model for representing program compilation [5].

References

1. I. J. Hayes and M. Utting. Coercing real-time refinement: A transmitter. In D. J. Duke and A. S. Evans, editors, *BCS-FACS Northern Formal Methods Workshop*, Electronic Workshops in Computing. Springer-Verlag, 1997. http://www.springer.co.uk/ewic/workshops/NFM96/.
2. ISO. *Ada Reference Manual: Language and Standard Libraries*, 6.0 edition, December 1994. International Standard ISO/IEC 8652:1995.
3. G. Kane and J. Heinrich. *MIPS RISC Architecture*. Prentice-Hall, 1992.
4. J. R. Larus. Assemblers, linkers and the SPIM simulator. In J. L. Hennessy and D. A. Patterson, editors, *Computer Organization and Design—The Hardware / Software Interface*. Morgan Kaufmann, 1994.
5. K. Lermer and C. J. Fidge. Compilation as refinement. In *Proc. Formal Methods Pacific '97*, Wellington, New Zealand, July 1997. To appear.
6. C. Morgan. *Programming from Specifications*. Prentice-Hall, second edition, 1994.
7. A. Rogers and S. Rosenberg. *Cycle Level SPIM*. Department of Computer Science, Princeton University, July 1993.
8. K. G. Shin and P. Ramanathan. Real-time computing: A new discipline of computer science and engineering. *Proceedings of the IEEE*, 82(1):6–24, January 1994.