

A methodology for compilation of high-integrity real-time programs

Karl Lermer Colin Fidge

December 1996

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.

Keywords and phrases: Real-time programming; compilation; timing analysis.

1 Introduction

High-integrity real-time programs must always meet all their ‘hard’ deadlines. Real-time code must exhibit not only correct functional behaviour, but predictable timing behaviour as well. 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 [30, 28], undermining their value for real-time applications. Consequently, safety-critical real-time programs are typically written directly in assembler 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

1. a real-time refinement theory for formally modelling translation of high-level programming language ‘specifications’ to time-verified machine code ‘implementations’, and
2. 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, and illustrate it with a small example. (The data refinement formalism for Topaz is being developed as an extension of the authors' previous work on real-time refinement methods, and will be the subject of a forthcoming report.)

Key features of the methodology are

- a minimal extension to the high-level language for expressing real-time requirements [10],
- separation of timing analysis and functional code development, allowing substantial reuse of existing software,
- the ability to provide precise feedback to high-level language programmers as to which timing requirements are unsatisfiable,
- identification of simple, 'straight-line' timing constraints [10], that potentially allow more accurate timing prediction [21] than previous methods, and
- potential for optimising or re-organising generated object code in order to satisfy the programmer's timing requirements.

2 Enabling technologies

The Topaz methodology is made possible by two recent developments in real-time software engineering.

2.1 Real-time coercions

Hayes and Utting recently proposed a simple form of compiler directive for expressing timing requirements in non-preemptible high-level language code segments, the *real-time coercion* [10]. For some time-valued expression E , this takes the form

before E

and places an *upper* bound of E on the absolute time at which this program statement may be reached. Satisfaction of this requirement must be proven at compile time.

When added to a programming language such as Ada 95 [14], which already implements a **delay until** statement for expressing *lower* absolute timing bounds, the **before** coercion gives us a programming language in which blocks of code can be bracketed by precise timing constraints.

Although numerous experimental programming languages with first-class timing annotations have been proposed previously [32, 6, 17, 18, 31, 7, 5] none has become widely accepted. The simple addition of the **before** directive to an existing language has a greater chance of industrial uptake than an entirely

new language. More importantly, the **before** directive can be generated as a product of stepwise program refinement [10], thus supporting formal real-time program development.

2.2 Accurate RISC code timing prediction

RISC architectures, such as Stanford’s MIPS [29], are proving popular for real-time applications because of their speed [34]. Paradoxically, however, their performance-enhancing features such as instruction pipelines and memory caching inhibit precise timing predictability [30, 28] because they make the time taken to execute each instruction dependent on its context.

Nevertheless, Lim et al [21] recently proposed a highly accurate timing prediction method for machine code programs written for RISC architectures. They showed that good timing predictions can be made for ‘straight-line’ instruction sequences by taking their context into account using *worst-case timing abstraction* matrices for each instruction which indicate those pipeline stages used by that instruction at each cycle. Sequential composition of these matrices allows them to overlap, as long as there is no conflict for access to a pipeline stage. This overlapping yields much less pessimistic timing estimates than previous approaches [27, 3, 23, 16, 24]. In particular, their predictions become better over *longer* instruction sequences, where more contextual information is available. A similar approach is then proposed to model access to cache blocks [21].

Furthermore, tools are becoming available that allow the timing behaviour of RISC code to be predicted. In particular, the SPIM S20 simulator runs assembly language programs for the MIPS R2000 and R3000 architectures [20, 19]. Its latest versions also simulate data and instruction caching overheads including the effects of pipelining [26], so it can be used to accurately predict the timing behaviour of machine code sequences.

3 The Topaz methodology

The Topaz methodology extends traditional high-level language (HLL) program compilation by capitalising on the above developments. A number of distinct instantiations of the methodology are possible, depending on how much effort is expended in changing existing software. In this section we describe the basic strategy and a number of alternative instantiations. Section 4 gives a detailed example illustrating the major steps.

3.1 Basic strategy

Figure 1 gives an overview of the Topaz approach. The *program development*, *compilation* and *assembly* phases are extensions of their traditional (untimed) counterparts. The *timing path analysis* and *timing verification* phases are new.

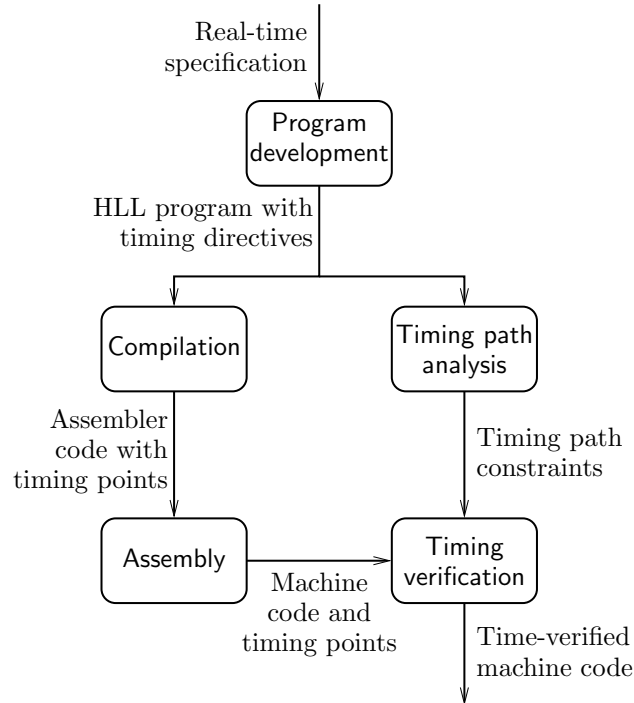


Figure 1: Overview of the Topaz real-time compilation strategy.

Program development The Topaz methodology begins with a high-level language program annotated with real-time compiler directives, such as **delay until** and **before** statements. This program could be written ‘manually’, or formally developed by stepwise refinement from a real-time specification [10].

Timing path analysis The timing path analysis phase simplifies the programmer’s timing directives and checks them for correctness [9]. Complex timing expressions are simplified through analysis of the control paths in the program to produce worst-case execution time (WCET) constraints on straight-line (single-entry/single-exit point [1]) code segments for later verification [10].

Compilation The high-level language program is compiled to functionally-correct assembler code. In addition to the usual compilation process, the compiler must (at least) decorate the generated assembler code with markers identifying those timing points in the program denoted as significant by the programmer’s timing directives.

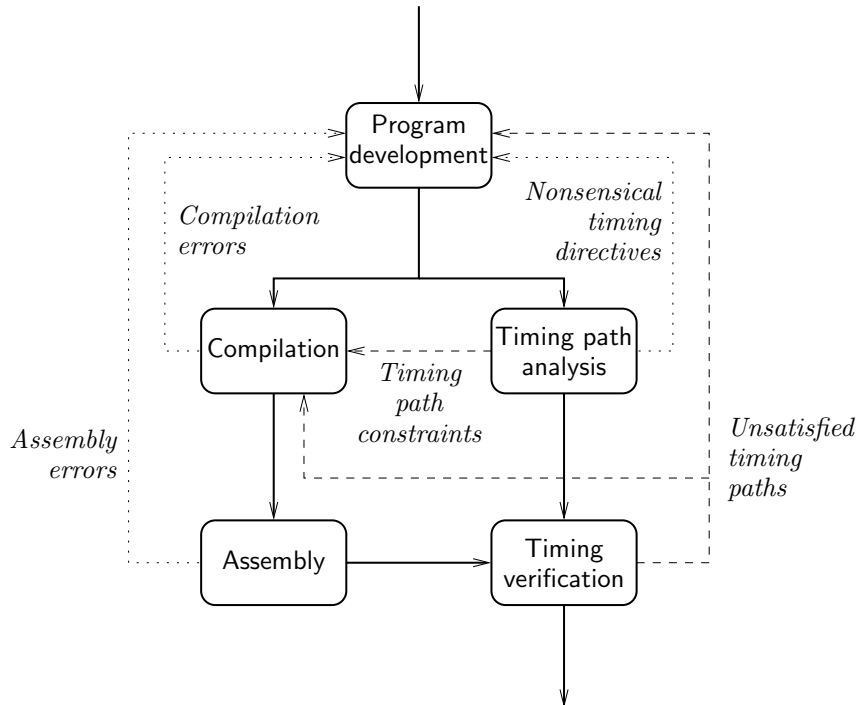


Figure 2: Possible feedback loops in the Topaz strategy.

Assembly The mnemonic assembler code is then assembled to produce executable machine code bytes. In addition to the usual assembly process, the assembler must make a record of the addresses associated with the significant timing points in the program.

Timing verification Timing prediction techniques [21] are used to check correctness of the timing path constraints on the timing points in the final machine code. This is done against a timing model of the particular target architecture.

Programs successfully passing through all these phases will have functional *and real-time* behaviour that respects the programmer's original intention.

3.2 Feedback

In practice, of course, programming errors introduce a need for feedback loops into the methodology, as shown in Figure 2. Exactly which of these feedback

paths is implemented depends on the degree to which existing software, especially the compiler, can be customised for real-time applications. In this section we describe several possible instantiations of the Topaz methodology.

Firstly, some forms of feedback are common to all instantiations of the Topaz methodology.

Static errors Conventional (non-real-time) programming mistakes, such as syntactic and static-semantics errors, are fed back to the programmer as usual, as shown by the two dotted arrows on the left of Figure 2.

The timing path analysis phase also examines the programmer's timing directives for syntactic correctness and reports ill-formed or nonsensical timing requirements back to the programmer as shown by the dotted arrow on the right. This may involve certain static checks, such as identifying timing requirements that are unsatisfiable by *any* implementation, e.g., those that require time to go backwards.

The remaining feedback loops differ depending on the compiler's ability to identify and respond to timing directives.

Off-the-shelf compiler The base case is when as few changes as possible are made to an existing compiler (and assembler). The minimum requirement to support the Topaz methodology is that the compiler is enhanced to mark timing points, corresponding to the HLL timing directives, in the generated code. Other than this the compiler ignores the new timing directives. In this scenario all timing errors must be detected by the timing verification phase and reported back to the programmer, as shown by the dashed arrow on the right of Figure 2. The programmer may then modify and recompile the HLL program, or may try recompiling it with different compiler options set. (In real-time applications it is sometimes better to turn compiler optimisations *off* to achieve greater timing predictability!)

Compiler that optimises timing paths A more powerful approach is to use a compiler that responds to the constraints identified by timing path analysis (horizontal dashed arrow in Figure 2). A naive way of doing this is to merely attempt to shorten or lengthen paths in the generated code depending on the relative 'tightness' of their constraints, e.g., by moving code into, or out of, loop bodies, etc. Furthermore, a compiler with this capability could allow failures detected during the timing verification phase to be returned directly to the compiler, as unsatisfied path constraints, so that an alternative compilation strategy can be tried.

Compiler that performs timing analysis An ideal compiler would be one that itself recognises and analyses the programmer's timing directives, undertakes timing verification of the assembler code as it is generated, and alters its compilation strategy if it predicts that the code will not achieve

```

|| 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:         before start + chsep * n;
C:         delay until start + chsep * n + chdef;
           n := n + 1
       od
     ]|;
D:   before start + chsep * size
||

```

Figure 3: High-level language program annotated with timing directives [10].

the programmer’s timing requirements. In this case the only timing errors likely to remain undetected until the timing verification phase are those introduced by factors not known at compilation time, such as assembler instructions that generate several machine code instructions [15], or additional timing overheads introduced by caching [21]. These remaining errors can be fed back to the compiler or programmer as appropriate.

The first of these scenarios, and perhaps the second, could be supported by modifying an existing compiler. The extensive timing requirements of the third, however, would likely require development of an entirely new compiler.

4 Example

In this section we illustrate key aspects of the Topaz methodology using a small example. It is based on the “transmitter” program developed by Hayes and Utting [10]. The program is intended to transmit a message, consisting of a sequence of characters, via a memory-mapped input/output location. Characters are transmitted periodically and each character must remain stable, i.e., reliably readable, for a fixed minimum time.

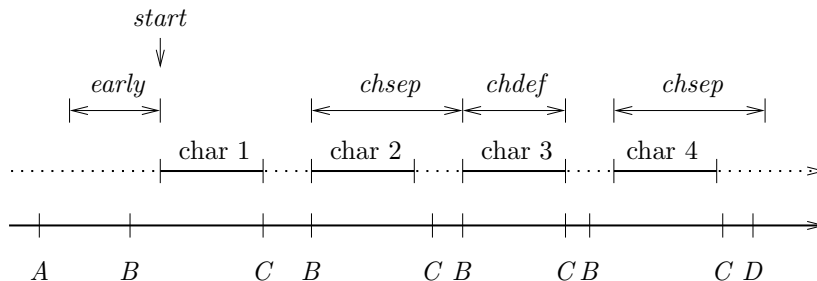


Figure 4: Acceptable timing behaviour of the transmitter program.

4.1 Program development

Figure 3 shows the time-annotated HLL program fragment for our transmitter case study, in guarded command language notation. Hayes and Utting explain in detail how this program can be formally developed using stepwise refinement [10].

Its functional behaviour is straightforward. It assumes the existence of an array *msg*, of length *size*, containing the 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*, using local variable *n* to count iterations.

The required timing behaviour is expressed through the directives labelled *A*, *B*, *C* and *D*. Figure 4 illustrates the desired timing behaviour for a four character message. Solid intervals in the top time line denote periods during which each character *must* be stably defined. Labels *A* to *D* in the second time line show when the program in Figure 3 passes each of its marked timing points.

Statement *A* is an *assumption* [22] that we may expect to be true when this program segment starts executing. Let *start* be the absolute time at which the first character must be transmitted, 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 [22], but we may use them when analysing the program.

Directive *B* is a real-time *coercion* [10], specifying the first significant timing requirement. Let duration *chsep* be the required separation of characters transmitted. The coercion occurs immediately after transmission of the n^{th} character, and tells us that we must reach this point no later than n times *chsep* units from time *start*. This defines the *initial* availability of the n^{th} 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. 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 n^{th} output character, since it prevents the program from performing any further actions, including those 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 $start$.

4.2 Timing path analysis

Hayes and Utting explained how the timing constraints expressed in Figure 3 can be simplified using control path analysis [10]. 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, thus greatly simplifying the later timing verification requirement.

Grundon then explored this concept further, and demonstrated its feasibility via a prototype path identification and simplification tool [9]. The tool recursively follows all control flow paths backwards from significant timing directives to find a preceding timing statement that can be used as the starting point for the path. Since variables appearing in the timing directives may be updated by the statements in the path, the tool retains information about updates made to these variables. The end-to-end timing constraint for each path is then simplified arithmetically.

For the example in Figure 3 Hayes and Utting identify the following worst-case execution time constraints.

$$\begin{aligned} WCET(A \rightsquigarrow D) &= \textit{early} \\ WCET(A \rightsquigarrow B) &= \textit{early} \\ WCET(C \rightsquigarrow B) &= \textit{chsep} - \textit{chdef} \\ WCET(C \rightsquigarrow D) &= \textit{chsep} - \textit{chdef} \end{aligned}$$

Calculation of these values is straightforward for this example, and can be confirmed by inspection, by substituting appropriate values for n and $size$. Path $A \rightsquigarrow D$ 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 $A \rightsquigarrow B$ 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 $C \rightsquigarrow B$ is the time taken to iterate, after the end of the stable period for the n^{th} 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 $C \rightsquigarrow D$ 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 $size - 1$, and again the arithmetic simplification is straightforward.

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 $B \rightsquigarrow C$, because their correctness is guaranteed by correct implementation of the **delay until** statement.

4.3 Compilation

Figure 5 shows a MIPS R3000 [15, 29] assembler code program developed from the HLL program in Figure 3. Here $size$ and msg are symbolic constants representing local addresses for these variables; $test$, $delay$ and end 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 [1], and a memory-mapped output location). Variables $\$t1$ to $\$t6$ are symbolic register names [15, 29]. (This assembler program is deliberately inefficient; we will optimise it in Section 4.6.)

Points in the code corresponding to the labelled timing directives in Figure 3 are prominently marked. A , B and D mark the points where upper timing bounds occurred. No code is generated for these constructs. Timing correctness of the **before** directives must be verified, but it is not helpful to generate instructions in an attempt to reach a point *earlier*! 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 $A \rightsquigarrow B$ and $C^\omega \rightsquigarrow D$, and can be seen to mirror the HLL program closely. Between point A and label $test$, variables $size$ and n are loaded into registers. The branch instruction at $test$ implements the **do** loop guard, exiting the loop if the guard is false. The following instructions implement the first assignment statement in the loop body, storing the value of $msg(n)$ into location out . The instruction immediately following point C^ω implements the second assignment statement, incrementing counter n . An unconditional branch follows this to return to the loop guard.

Due to the way instructions overlap in a RISC pipeline, care must be taken

	A	
	lb \$t1, <i>size</i>	t1 := <i>size</i> (length of <i>msg</i>)
	li \$t2, 0	t2 := 0 (loop counter <i>n</i>)
<i>test</i> :	beq \$t2, \$t1, <i>end</i>	exit if <i>n</i> equals <i>size</i>
	nop	branch delay slot
	lb \$t3, <i>msg</i> + 0(\$t2)	t3 := <i>msg</i> (<i>n</i>)
	nop	load delay slot
	sb \$t3, <i>out</i>	output location := <i>msg</i> (<i>n</i>)
	B	
	C ^α	
	lb \$t4, <i>chsep</i>	t4 := constant <i>chsep</i>
	mult \$t4, \$t2	start calculating <i>n</i> * <i>chsep</i>
	lb \$t5, <i>dt</i>	t5 := first delay time (<i>start</i> + <i>chdef</i>)
	mflo \$t4	t4 := <i>n</i> * <i>chsep</i>
	add \$t5, \$t5, \$t4	t5 := <i>start</i> + <i>chdef</i> + <i>chsep</i> * <i>n</i>
<i>delay</i> :	lb \$t6, <i>clock</i>	t6 := current time
	nop	load delay slot
	sub \$t6, \$t5, \$t6	t6 := delay time – current time
	bgtz \$t6, <i>delay</i>	busy wait if current time < delay time
	nop	branch delay slot
	C ^ω	
	add \$t2, \$t2, 1	increment <i>n</i>
	j <i>test</i>	goto loop test
	nop	jump delay slot
	D	
<i>end</i> :	...	

Figure 5: MIPS R3000 assembler code for the high-level language program in Figure 3.

with instructions that have effects lasting more than one cycle [15]. For instance, the instruction following a branch is *always* executed, whether the branch is taken or not, so it is sometimes necessary to introduce a ‘no-op’ after a branch if no useful instruction can be found to fill this slot. This is done for the jump at the end of Figure 5. Similarly, the result of loads from memory may not be immediately available, so if we want to immediately use such a memory value, a following no-op may be needed to delay the program until the load is completed. In Figure 5 this is done after the instruction that loads the n^{th} *msg* character into register \$t3. (A MIPS assembler can optionally be configured to insert these delays itself [20], but we have chosen to make them explicit here.)

The instructions between points C^α and C^ω implement the **delay until** statement. Initially they calculate the value of expression $start + chsep * n +$

chdef, i.e., the absolute time at which the delay ends. Note that multiplication in the MIPS R3000 architecture uses a separate floating point unit and may take several cycles, therefore possibly delaying the pipeline until the result is available [15, 29]. In Figure 5 this idle time has been partially filled by a ‘load byte’ appearing between the ‘multiply’ and subsequent ‘move from lo register’ instructions. Following this there is a tight loop, labelled *delay*, in which the current time is read and compared with the end delay time. The loop exits when the current time exceeds the delay time.

4.4 Assembly

Figure 6 shows binary machine code generated for our program by the SPIM assembler [20, 19, 26]. The first column contains instruction addresses, and the second the executable 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 5 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’ [15, 29]). Even worse, the fifth assembler instruction from Figure 5, which loads the n^{th} *msg* character into register \$t3, generates *three* machine-level instructions, a ‘load upper immediate’, ‘add unsigned’ and ‘load byte’ (sixth, seventh and eighth instructions in Figure 6)! It is for this reason that we perform timing verification on the final *machine code* program.

To support the timing verification phase some way of associating timing points with particular instructions is still needed. Since we cannot leave ‘comments’ in binary machine code, the assembler must generate a table associating timing point *A* with address 0x00400020, point *B* with address 0x00400048, and so on. (In Figure 6 we have merely indicated the timing points in the mnemonic translation.)

4.5 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 [20, 19, 26] to verify the timing behaviour of the machine code program in Figure 6.

For clarity below, assume there are no caching overheads (although the SPIM tool is capable of simulating such overheads [26]). We can then treat most machine code instructions in Figure 6, with the exception of the ‘multiply’,

		————— <i>A</i> —————
[0x00400020]	0x3c011001	lui \$1, 4097
[0x00400024]	0x80290005	lb \$9, 5(\$1)
[0x00400028]	0x340a0000	ori \$10, \$0, 0
[0x0040002c]	0x11490015	beq \$10, \$9, 84
[0x00400030]	0x00000025	or \$0, \$0, \$0
[0x00400034]	0x3c011001	lui \$1, 4097
[0x00400038]	0x002a0821	addu \$1, \$1, \$10
[0x0040003c]	0x802b0000	lb \$11, 0(\$1)
[0x00400040]	0x00000025	nop
[0x00400044]	0xa38b8000	sb \$11, -32768(\$28)
		————— <i>B</i> —————
		————— <i>C</i> ^α —————
[0x00400048]	0x3c011001	lui \$1, 4097
[0x0040004c]	0x802c0007	lb \$12, 7(\$1)
[0x00400050]	0x018a0018	mult \$12, \$10
[0x00400054]	0x3c011001	lui \$1, 4097
[0x00400058]	0x802d0006	lb \$13, 6(\$1)
[0x0040005c]	0x00006012	mflo \$12
[0x00400060]	0x01ac6820	add \$13, \$13, \$12
[0x00400064]	0x838e8001	lb \$14, -32767(\$28)
[0x00400068]	0x00000025	or \$0, \$0, \$0
[0x0040006c]	0x01ae7022	sub \$14, \$13, \$14
[0x00400070]	0x1dc0fffc	bgtz \$14, -16
[0x00400074]	0x00000025	or \$0, \$0, \$0
		————— <i>C</i> ^ω —————
[0x00400078]	0x214a0001	addi \$10, \$10, 1
[0x0040007c]	0x0810000b	j 0x0040002c
[0x00400080]	0x00000025	or \$0, \$0, \$0
		————— <i>D</i> —————
[0x00400084]	...	

Figure 6: MIPS R3000 machine code for the assembler program in Figure 5.

as taking exactly one cycle. (The goal of RISC design has been to achieve an execution rate of one instruction per machine cycle.) It is then merely necessary to count all instructions a path contains. Let num_i be the number of instructions for each path in Figure 6.

$$\begin{aligned}
 num_i(A \rightsquigarrow D) &= 5 \\
 num_i(A \rightsquigarrow B) &= 10 \\
 num_i(C^\omega \rightsquigarrow B) &= 10
 \end{aligned}$$

$$\text{numi}(C^\omega \rightsquigarrow D) = 5$$

(Keep in mind that we must consider control flow when counting instructions, so, for example, path $C^\omega \rightsquigarrow D$ includes the jump to *test* and back.)

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 4.2). 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 times the machine-specific elapsed time per processor cycle [15]. The ‘lateness’ value is explained below.

$$\begin{aligned} \text{numi}(A \rightsquigarrow D) * t &\leq \text{early} \\ \text{numi}(A \rightsquigarrow B) * t &\leq \text{early} \\ (\text{numi}(C^\omega \rightsquigarrow B) + \text{lateness}(C)) * t &\leq \text{chsep} - \text{chdef} \\ (\text{numi}(C^\omega \rightsquigarrow D) + \text{lateness}(C)) * t &\leq \text{chsep} - \text{chdef} \end{aligned} \quad (1)$$

As mentioned in Section 4.2, we are not required to prove any timing properties of path $B \rightsquigarrow C^\alpha$, which ends with a **delay until** statement. (Provided, of course, that we trust the compiler’s implementation of this HLL construct!) Nevertheless, the possible ‘lateness’ of this path must be considered in analysing the *following* paths $C \rightsquigarrow B$ and $C \rightsquigarrow D$. This is because any practical implementation of the HLL statement

delay until E

cannot guarantee to finish at *exactly* time E , but will typically overrun E by some, hopefully short, time e [14]. This means that we cannot use ‘ E ’ as the starting time of the following path(s), but must consider that the **delay until** may finish as late as ‘ $E + e$ ’.

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. There are two distinct cases, depending on the value of programmer-supplied constant *chdef*.

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.

$$\text{lateness}_{\text{passed}} = 21 - \left\lfloor \frac{\text{chdef}}{t} \right\rfloor$$

It takes 21 cycles to get from C^α to C^ω when the ‘*delay*’ loop in Figure 5 does not iterate. (The multiplication instruction delays the processor for 11 cycles, *excluding* the initial ‘multiply’ and final ‘move from lo’ instructions.) However, we subtract the number of *whole* processor cycles that can be completed before

duration $chdef$ expires, because these do not contribute to lateness. (Remember that $chdef$ is expressed in units of ‘real’ time, rather than processor cycles, so we must divide it by the time-per-cycle t to get the number of instructions it encompasses.)

Secondly, when $chdef$ is large, the **delay until** implementation in Figure 5 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 .

$$lateness_{loops} = 10$$

Thus, in the general case, where the actual value of $chdef$ is not known to us, the most we can say about lateness is as follows.

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

Putting all these calculations together yields the following necessary inequalities.

$$\begin{aligned} 10 * t &\leq early \\ 20 * t &\leq chsep - chdef \\ 31 * t &\leq chsep \end{aligned} \tag{2}$$

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 6 can be considered to conform with the programmer’s specified timing requirements.

4.6 Feedback

Section 3.2 outlined several ways in which timing feedback may affect code generation. Here we illustrate one way in which this might happen. Consider the situation where the programmer specifies the following durations for the constants in Figure 3. For simplicity, let t be 1 so that time units equal processor cycles.

$$\begin{aligned} early &= 40 \\ chsep &= 59 \\ chdef &= 41 \end{aligned}$$

It is now clear that the machine code program in Figure 6 *cannot* satisfy the timing inequalities shown at the end of Section 4.5. It will fail the timing verification phase because inequality (2) does not hold.

$$20 \not\leq 59 - 41$$

In particular, we can determine that it is path $C \rightsquigarrow B$ that contains the timing error! The lateness figure for the **delay until** statement is as follows.

$$\begin{aligned} \textit{lateness}_{\textit{passed}} &= -20 \\ \textit{lateness}_{\textit{loops}} &= 10 \\ \textit{lateness}(C) &= 10 \end{aligned}$$

(The negative value for $\textit{lateness}_{\textit{passed}}$ tells us that this scenario never occurs, because \textit{chdef} is sufficiently large. An idle wait *must* be introduced in this case.) This then tells us that the only WCET constraint *not* satisfied is inequality (1) from Section 4.5, which defines the requirement for path $C \rightsquigarrow B$.

$$10 + 10 \not\leq 59 - 41$$

This information, precisely identifying which part of the HLL program has a timing requirement that cannot be satisfied by the generated machine code, can be fed back to the programmer, or a suitably enhanced compiler, to take corrective action.

The program in Figure 7 offers an alternative compilation of the program in Figure 3 that attempts to solve this problem. The program contains a number of optimisations [15] using the following principles.

Different control flow This compilation strategy places the **do** loop test *after* the loop body. Although this requires an extra jump instruction before the first time the guard is evaluated, it saves the need for a jump instruction back to the top of the loop at *every* iteration.

Loop optimisation Instructions for loading values that remain constant during the lifetime of a loop can be moved outside the loop body. In Figure 7 this is done for compile-time constants \textit{chsep} and \textit{dt} .

Replacing slow operations with faster ones Sometimes instructions that delay the pipeline for several cycles can be replaced. The multiplication instruction in Figure 5 is effectively replaced in Figure 7 by the addition immediately before label C^ω .

Pipeline scheduling Ideally a compiler for a RISC architecture will attempt to fill delay slots with useful, or at least harmless, instructions. In Figure 7 the branch instruction labelled \textit{test} performs a useful addition in its delay slot. (When the loop exits this instruction is performed redundantly, but this is harmless as long as the following code does not make any use of the value in register $\$t5$.)

Passing instructions over time labels In some circumstances the compiler has freedom to move instructions ‘over’ timing markers in order to improve the timing behaviour of certain paths. In particular, instructions other

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

Figure 7: Optimised MIPS R3000 assembler code for the high-level program in Figure 3.

than branches, input/output instructions [8, 5], timer accesses, or any other instruction that has externally observable effects, can usually be safely moved. In Figure 7 the instructions that load constants *dt* and *chsep* are examples of this. Although they ‘belong’ to path $C^\alpha \rightsquigarrow C^\omega$, since they support implementation of the **delay until** statement, they can be harmlessly moved into path $A \rightsquigarrow B$ because their effect is not observable outside this code segment.

Figure 8 shows the assembled machine code for this optimised assembler program. Redoing the timing verification phase on this code then yields the following instruction counts.

$$\begin{aligned} \text{numi}(A \rightsquigarrow D) &= 12 \\ \text{numi}(A \rightsquigarrow B) &= 15 \end{aligned}$$

————— <i>A</i> —————		
[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)
————— <i>B</i> —————		
————— <i>C^α</i> —————		
[0x00400054]	0x838f8001	lb \$15,-32767(\$28)
[0x00400058]	0x00000025	or \$0,\$0,\$0
[0x0040005c]	0x016f7822	sub \$15,\$11,\$15
[0x00400060]	0x1de0ffff	bgtz \$15,-12
[0x00400064]	0x00000025	or \$0,\$0,\$0
[0x00400068]	0x016c5820	add \$11,\$11,\$12
————— <i>C^ω</i> —————		
[0x0040006c]	0x214a0001	addi \$10,\$10,1
[0x00400070]	0x152afff6	bne \$9,\$10,-40
[0x00400074]	0x01486820	add \$13,\$10,\$8
————— <i>D</i> —————		

Figure 8: MIPS R3000 machine code for the assembler program in Figure 7.

$$\begin{aligned} \text{numi}(C^\omega \rightsquigarrow B) &= 6 \\ \text{numi}(C^\omega \rightsquigarrow D) &= 3 \end{aligned}$$

The possible overrun of the **delay until** implementation is, in fact, slightly *increased*, due to the additional **add** instruction following the *delay* loop.

$$\text{lateness}(C) = 11$$

Nevertheless, the overall timing behaviour of path $C \rightsquigarrow B$ is improved due to the reduced number of instructions. Inequality (1) from Section 4.5 is now satisfied.

$$6 + 11 \leq 59 - 41$$

Repeating the calculations for other paths reveals that they too still have correct timing behaviour. All the inequalities in Section 4.5 are now true, and we can finally conclude that the machine code does indeed satisfy the high-level language programmer’s original timing requirements!

5 Related work

A number of recent projects have sought to formalise compilation of real-time programs in order to improve their trustworthiness.

In particular, the European *ProCoS* project aimed to “establish an acceptably high degree of confidence in compilers” by formally specifying and verifying a simple compiler [11, 12, 5]. A companion project, *safemos*, had similar aims but used a different formalism for modelling [2]. Significantly, the complexity of the task led the *safemos* team to conclude that “it seems unlikely that a production compiler will ever be completely verified” [2, p.146]. It is for this reason that the Topaz methodology emphasises reuse of existing, trusted tools.

TCEL is an experimental language for expressing time-constrained programs [13, 7, 8]. It has an associated compilation strategy that allows rearrangement of unobservable code segments in order to achieve timing correctness of observable outputs, a feature entirely compatible with the Topaz approach.

The well-established *MARS* model uses a fully integrated programming environment for construction of real-time systems, including a programming language, development tools, compiler, timing analyser and hardware [24, 25]. The compiler creates a timing graph that acts as a common data structure used to communicate between tools. By relying on timing constraints being reduced to simple straight-line paths, Topaz instead uses the annotated assembler code in this role.

6 Conclusion

The Topaz methodology for development of verified real-time machine code integrates recent advances in real-time programming formalisms into traditional compiler technology. Feasibility of the approach was demonstrated via a detailed example, supported using existing tools [9, 20]. Nevertheless, much work remains in developing formalisms and tools specifically for the Topaz methodology.

Elsewhere we have shown how abstract real-time specifications can be formally refined to multi-tasking programs [4], and then to sequential real-time program code [10, 33]. That work, coupled with the Topaz methodology described herein, will give us a complete development path from abstract real-time specifications, through multi-tasking HLL programs, to time-verified machine code.

Acknowledgements

We are indebted to Ian Hayes, Mark Utting and Steve Grundon for inspiring many of the ideas used in this paper. We also wish to thank Peter Kearney, Ian Hayes, Steve Grundon and Mark Utting for their many valuable comments on drafts of this paper. This work is funded by Australian Research Council grant A49600176, *Verified compilation rules for real-time programs via program refinement*.

References

- [1] S. J. Bharrat and K. Jeffay. Predicting worst case execution times on a pipelined RISC processor. Technical Report 94-072, Department of Computer Science, University of North Carolina, 1994.
- [2] J. Bowen, editor. *Towards Verified Systems*, volume 2 of *Real-Time Safety Critical Systems*. Elsevier, 1994.
- [3] R. Chapman, A. Burns, and A. J. Wellings. Integrated program proof and worst-case timing analysis of SPARK Ada. In *ACM Workshop on language, compiler and tool support for real-time systems*. ACM Press, 1994.
- [4] C. J. Fidge, M. Utting, P. Kearney, and I. J. Hayes. Integrating real-time scheduling theory and program refinement. In M.-C. Gaudel and J. Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *Lecture Notes in Computer Science*, pages 327–346. Springer-Verlag, 1996.
- [5] M. Fränzle and M. Müller-Olm. Towards provably correct code generation for a hard real-time programming language. In P. Fritzson, editor, *Compiler Construction*, volume 786 of *Lecture Notes in Computer Science*, pages 294–308. Springer-Verlag, 1994.
- [6] N. Gehani and K. Ramamritham. Real-time concurrent C: A language for programming dynamic real-time systems. *The Journal of Real-Time Systems*, 3:377–405, 1991.
- [7] R. Gerber and S. Hong. Semantics-based compiler transformations for enhanced schedulability. In *Proc. Real-Time Systems Symposium*, 1993.
- [8] R. Gerber and S. Hong. Compiling real-time programs with timing constraint refinement and structural code motion. *IEEE Transactions on Software Engineering*, 21(5), May 1995.
- [9] S. Grundon. Timing constraint analysis for real-time programming. Honours project report, Department of Computer Science, University of Queensland, November 1996.
- [10] I. J. Hayes and M. Utting. Coercing real-time refinement: A transmitter. In *Proc. Northern Formal Methods Workshop*, Ilkley, U.K., September 1996.
- [11] Jifeng He. *Provably Correct Systems*. McGraw-Hill, 1995.
- [12] Jifeng He and J. Bowen. Specification, verification and prototyping of an optimized compiler. *Formal Aspects of Computing*, 6(6):643–658, 1994.

- [13] S. Hong and R. Gerber. Compiling real-time programs into schedulable code. In *Proc. PLDI*, 1993.
- [14] ISO. *Ada Reference Manual: Language and Standard Libraries*, 6.0 edition, December 1994. International Standard ISO/IEC 8652:1995.
- [15] G. Kane and J. Heinrich. *MIPS RISC Architecture*. Prentice-Hall, 1992.
- [16] K. Kenny and K.-J. Lin. Measuring and analyzing real-time performance. *IEEE Software*, 8(5):41–49, September 1991.
- [17] K. B. Kenny and K.-J. Lin. Building flexible real-time systems using the Flex language. *IEEE Computer*, 24(5):70–78, May 1991.
- [18] E. Kligerman and A. D. Stoyenko. Real-time Euclid: A language for reliable real-time systems. *IEEE Transactions on Software Engineering*, SE-12(9):941–949, September 1986.
- [19] J. R. Larus. *SPIM S20: A MIPS R2000 Simulator*. Computer Sciences Department, University of Wisconsin–Madison, 11 edition, April 1993.
- [20] 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.
- [21] Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, Soo-Mook Moon, and Chong Sang Kim. An accurate worst case timing analysis for RISC processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, July 1995.
- [22] C. Morgan. *Programming from Specifications*. Prentice-Hall, second edition, 1994.
- [23] C. Y. Park and A. C. Shaw. Experiments with a program timing tool based on source-level timing schema. In *Proc. IEEE Real-Time Systems Symposium*, pages 72–81, Florida, December 1990.
- [24] G. Pospischil, P. Puschner, A. Vrchoticky, and R. Zainlinger. Developing real-time tasks with predictable timing. *IEEE Software*, 9(5):35–44, September 1992.
- [25] P. Puschner and Ch. Koza. Calculating the maximum execution time of real-time programs. *Journal of Real-Time Systems*, 1(2):159–176, September 1989.
- [26] A. Rogers and S. Rosenberg. *Cycle Level SPIM*. Department of Computer Science, Princeton University, July 1993.
- [27] A. C. Shaw. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*, 15(7):875–889, July 1989.
- [28] 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.
- [29] P. H. Stakem. *A Practitioner’s Guide to RISC Microprocessor Architecture*. Wiley-Interscience, 1996.
- [30] J. A. Stankovic. Misconceptions about real-time computing. *IEEE Computer*, 21(10):10–19, October 1988.
- [31] A. D. Stoyenko and W. A. Halang. Extending Pearl for industrial real-time applications. *IEEE Software*, 10(4):65–74, July 1993.

- [32] A. D. Stoyenko, T. J. Marlowe, and M. F. Younis. A language for complex real-time systems. *The Computer Journal*, 38(4):319–338, 1995.
- [33] M. Utting and C. J. Fidge. A real-time refinement calculus that changes only time. In Jifeng He, editor, *BCS/FACS Seventh Refinement Workshop*, pages 266–281, Bath, United Kingdom, 1996.
- [34] T. Williams. Performance pushes RISC chips into real-time roles. *Computer Design*, pages 79–86, September 1991.