Interactively Verifying a Simple Real-time Scheduler

Colin Fidge, Peter Kearney, Mark Utting

Software Verification Research Centre,
Department of Computer Science,
The University of Queensland, Brisbane, Australia.
Email: {cjf,pk,marku}@cs.uq.oz.au

Abstract. This paper describes the interactive verification of a simple interrupt-driven real-time scheduler written in the machine code language of the MIPS R3000 RISC processor. The formal verification was carried out using the interactive theorem prover Ergo.

1 Introduction

Many real-time systems are structured as sets of tasks or processes. Scheduling is required to share resources between the tasks so that their real-time requirements are met. Real-time schedulers are often a core part of safety critical real-time applications.

This paper describes the formal verification of a simple scheduler which schedules a fixed number of periodic tasks with each task being given an equal time slice. The scheduler (or cyclic executive) is implemented by a clock-driven interrupt routine, where the period of the clock is equal to the time-slice.

The scheduler is written in the machine code language of the MIPS[1] R3000 RISC processor. RISC processors are increasingly used in real-time applications [11], but they introduce certain complexities into the analysis of real-time behaviour, through the use of memory caches and instruction pipelining. To provide a basis for verifying real-time R3000 machine code, a formal specification of the real-time behaviour of the processor was written [5, 8, 9]. The instruction level of that specification provides the basis for the verification discussed here.

2 Formal Proof and Constraint Extraction

The verification was carried out using the Ergo interactive theorem prover [10]. Ergo was used to

- formalise *functional logic* [3, 4, 7] – the modal logic that was used for reasoning about real-time systems;
- ensure that the proof was carried out correctly;
- provide automated assistance in finding proofs and managing proof detail;

[1] MIPS is a registered trademark of MIPS Computer Systems.

- support code verification through the use of application-specific tactics and heuristics for the R3000 machine language;
- help derive required timing constraints as part of the proof process.

The scheduler system is parameterised by seven timing constants and over a dozen constraints on these parameters are necessary to make the specified system feasible. Some of these constraints come from the environment, while others come from our chosen implementation, even the scheduler itself. The interaction between these constraints is subtle and we found we could not easily identify a minimal set of constraints that would ensure feasibility. There is a fine balance between having sufficient constraints to make the system feasible and over-constraining the system unnecessarily.

The theorem prover was used not only to ensure that sufficient constraints were present, but also to assist in deriving the weakest necessary constraints. Sections 6.2 through 6.6 discuss this constraint derivation.

3 Assumptions and Requirements

The verification is carried out by proving a theorem which formalises the scheduling requirement. However, the requirement can only be met under certain assumptions about the environment: for example, concerning the kind and frequency of interrupts, and about task behaviour. The verification theorem is proven in the theory which describes the behaviour of the MIPS R3000, augmented with these assumptions.

This section introduces the scheduling requirement and environmental assumptions under which the requirement is to be met. Formalisation of these is discussed in Section 4. As indicated in the last section, further constraints on certain timing parameters were derived during the proof and are given in Sections 6.2, 6.3 and 6.6.

3.1 Initialisation

Our analysis applies only to times following an initialisation phase undertaken by a system kernel. During initialisation the scheduler code and task code are loaded into instruction cache and the scheduler data is loaded into data cache. The system then executes a non-stalling busy-wait loop, with interrupts enabled, awaiting a clock interrupt. The time at which the first clock interrupt occurs following completion of the initialisation phase is denoted `first`.

3.2 Interrupt Behaviour

The clock interrupt signal is assumed to go high every `clocksep` time units and stay high for exactly `hold` time units. These values are determined by the interrupt hardware used, and are configurable. The use of self-clearing interrupts simplifies the verification slightly, since the scheduler does not need to clear the interrupts. It is also assumed that no external interrupts occur other than clock interrupts.

3.3 The Scheduling Requirement: Overview

The number of tasks to be scheduled is denoted **nt**. Tasks are identified by the numbers $0 \ldots \mathbf{nt} - 1$, and the starting address of the code for task i, following the completion of initialisation activities, is denoted **tcode**(i). The timeslice allocated to each process is **clocksep**. Figure 1 illustrates the scheduling requirement for the case $\mathbf{nt} = 3$.

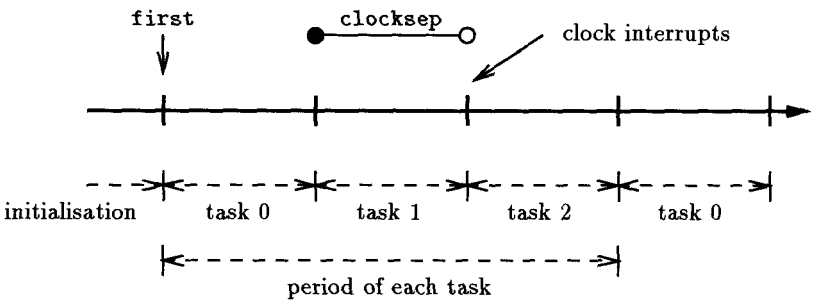

Fig. 1. Overview of the scheduling requirement for $\mathbf{nt} = 3$.

To account for the time consumed by the scheduler itself, we introduce the constant **maxdelay**, which equals the maximum number of time units allowed to pass from the time the clock interrupt occurs until the next task begins execution. Defining the minimum acceptable time-slice for each task to be $\mathbf{minslice} = \mathbf{clocksep} - \mathbf{maxdelay}$, we can state the scheduler requirement more precisely as follows. For each integer $n \geq 0$ there must be a time t, where $(\mathbf{first} + n * \mathbf{clocksep}) \leq t \leq (\mathbf{first} + n * \mathbf{clocksep} + \mathbf{maxdelay})$, such that at time t control resides at the starting address of task (n mod **nt**). Furthermore, that task must be allowed to execute without interruption for **minslice** time units from time t. See Fig. 2.

The scheduling requirement is parameterised by **clocksep** and **maxdelay**. To satisfy the requirement, **clocksep** may be chosen by suitable configuration of the clock interrupt hardware. The requirement can be met by a scheduler only if **maxdelay** satisfies a further constraint, which takes account of the maximum time to execute the scheduler. The particular constraint required for the scheduler we verify is given in Section 6.6.

3.4 Task Behaviour

The scheduling requirement can be met only under certain assumptions about the tasks, which are introduced below.

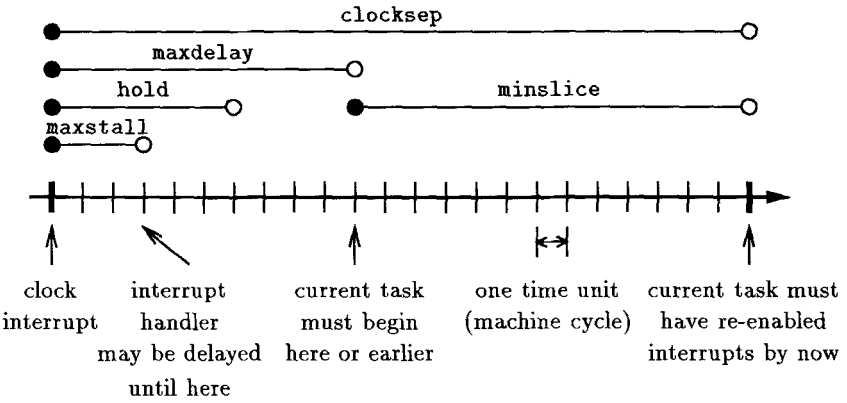

Fig. 2. Scheduling requirement for a particular interval.

Stalls-Bounded: In order to analyse the time taken by the scheduling activity, it is important to know the time taken to service the clock interrupt and transfer control to the scheduler. However, interrupt processing is delayed by pipeline stalls, which may happen when a task accesses external memory or performs multiplication or division operations. To permit real-time analysis of scheduling, tasks are required to limit the duration of stalls during the critical period when they may be interrupted (that is, after `minslice` time units) to a maximum of `maxstall` time units.

Interrupts-Enabled: A task must ensure that the clock interrupt can be taken. This requires ensuring that interrupts are enabled, that the clock interrupt is not masked, and that no internal interrupts (exceptions) are being generated which would take priority over the clock interrupt. See Fig. 2.

Scheduler-Memory-Unchanged: The tasks must not corrupt the scheduler code or data. Further, we require that tasks do not cause scheduler code or data to be moved out of the instruction and data caches. This is achieved by requiring that specified instruction and data cache locations associated with the scheduler are unchanged.

Initial-Task-Instructions: It is an intrinsic property of pipelined architectures that the timing of the 'current instruction' may be influenced by other instructions [5]. In our case study, the timing of the last few scheduler instructions may be dependent on properties of the first few task instructions, since they will be executing at the same time in the pipeline.

If the first two instructions of a task are not in instruction cache, the last two scheduler instructions will be delayed. Also, if one of the first two of a task's instructions accesses the `HI/LO` registers (associated with multiplication and division instructions), a delay of the order of hundreds of cycles is possible. In order to eliminate this dependence, we require that the first two instructions of each task are in cache, and that they do not access the `HI/LO` reg-

isters. These requirements can be discharged during the verification of each task. Without these requirements, we would obtain a weaker bound on scheduler timing, and a longer **maxdelay**.

3.5 A Task Invariant

We have required a number of conditions to be fulfilled by the tasks. However, we have not so far introduced any requirement that the scheduler not corrupt or interfere with the code or data of the tasks! In order to introduce that requirement in an application independent way, we introduce a *task invariant*, denoted **task_inv**. We make only two assumptions about the task invariant:

1. if all locations other than those associated with scheduler code and data are unchanged, then the task invariant is maintained;
2. the tasks themselves maintain the task invariant.

We introduce the additional requirement on the scheduler that it guarantees the task invariant at the time each task is re-started. Given the assumptions just introduced, this requires showing that the scheduler does not change any locations other than its own. In addition, the initialisation phase must establish the task invariant (at time **first**).

We also make the assumptions about task behaviour given in Section 3.4 conditional on the task invariant holding at the time a task starts. Thus the tasks guarantee certain properties, assuming that the task invariant is true when they start, and in turn the scheduler guarantees that the task invariant is true when tasks start (assuming that previous task behaviour has satisfied its guarantees). The usefulness of this type of mutual assumption-guarantee structure in the verification of concurrent systems has been recognised for some time [6].

4 Formalisation

This section discusses aspects of formalising the assumptions and requirements introduced in the previous section. The full formalisation is available in a separate report [1].

4.1 Model and Notation

We do not attempt to introduce formally the notation of the specification here, instead giving intuitive descriptions of the meaning of the notation as it is used. Readers interested in the formal basis of the notation may consult [3, 4, 7].

Our underlying model of a real-time system is a set of traces, where each trace represents a possible evolution of the system. A trace is indexed by time, where time is represented by the natural numbers, and the unit of time is the clock cycle of the processor. Specification assertions are relative to a trace and a time. Operators are used to make assertions at a particular time, or during a sequence of times, and to shift the time to the next occurrence of some event in

the trace. The constant **time** denotes the current time and the operator **at(T)** shifts the current time to the time **T**.

Our specification is given in a theory which formalises a model of the processor. Amongst other state components in this model are: two program counters (**pc** and **next_pc**), two instruction registers (**ireg**, **next_ireg**) and a location **run** which indicates whether the current cycle is a run cycle, that is, not a stall cycle. The postfix operator ^ is used to extract the contents of a location. All notation in typewriter font can be input directly to the Ergo interactive theorem prover.

4.2 Interrupt Behaviour

The following assertion formalises the required clock behaviour, stating that interrupts occur with separation **clocksep** (from **first** onwards) and remain set for **hold** time units in each timeslice interval **N**. The predicate **during**(t_1, t_2, b) includes time t_1, but not t_2.

```
axiom clock_behaviour
=== hw and N:nats
    => during(first+N*clocksep, first+N*clocksep+hold,
              clk_set) and
       during(first+N*clocksep+hold, first+(N+1)*clocksep,
              not clk_set).
```

We also require that after initialisation, clock interrupts are the only external interrupts.

```
axiom clock_ints_only
=== hw and first =< T and at(T);int_asserted(X)
    => X = clk_int.
```

To be detectable, a clock interrupt must "hold" for at least one machine cycle. Furthermore, it would be nonsensical for the hold period to exceed the clock interrupt separation.

```
axiom clock_ints_duration
=== 0 < hold and hold =< clocksep.
```

4.3 Task Behaviour

In order to state assumptions about task behaviour, we frequently refer to the system state in which a task **I** begins or resumes executing. The following predicate is used to define this as a state in which the program counter equals the starting address of the code for task **I** and the processor is in a state that allows code to begin normal execution (this includes requiring the task invariant) .

```
function task_starts(I)
=== hw                              and (I : 0 upto nt)
    and task_inv                    and irun
    and pc^ = tcode(I)              and next_pc^   = tcode(I) + 4
    and ireg^ = ival(pc^)           and next_ireg^ = ival(next_pc^)
    and user_mode                   and ints_enabled
    and not int_masked(clk_int)     and not in_lds^
    and not in_bds^                 and not data_bus_error^
    and interruptable.
```

```
# NB. The $k0 and $k1 registers are reserved for scheduler use.
# Get pointers to current task control block (TCB) and tcb_end.
lui $k0, ihdata_upper      # $k0 := address of scheduler data.
lw $k1, currtask($k0)      # $k1 := pointer to current TCB.
lw $k0, tcb_end($k0)       # $k0 := address after last TCB.

# Increment the currtask pointer to the next task.
addiu $k1, $k1, tcb_size   # $k1 := $k1 + tcb_size.
bne $k0, $k1, 2            # Jump over the addiu instruction if
                           #   $k1 is a legal tcb pointer.
lui $k0, ihdata_upper      # NB. always executed (branch delay slot).
                           # $k0 := address of scheduler data.
addiu $k1, $k0, tcb        # $k1 := address of TCB 0. (=ihdata+tcb)
sw $k1, currtask($k0)      # Store new current task pointer ($k1)
                           #   into location ihdata + currtask.
#Start new task.
lw $k0, 0($k1)             # $k0 := start pc for current task.
nop                        # Do nothing (load delay slot).
jr $k0                     # Jump to start pc of current task.
rfe                        # Restore status reg settings.
```

Fig. 3. An assembler version of the scheduler code.

Tasks can assume, when they resume executing, that there are at least `minslice` time units available before the next external interrupt. The following predicate is used to assert this in the preconditions of the task assumptions. The expression `next_inc(ext_interrupt)` shifts the current time to the time when an external interrupt next occurs (possibly the current time). Thus the expression `next_inc(ext_interrupt);time` denotes the time of the next external interrupt.

```
function minslice_available
=== next_inc(ext_interrupt);time >= time+minslice.
```

It is convenient to define a function, **next_ih**, which shifts the current time to that at which the next external interrupt is handled. There may be a delay between the time the interrupt occurs and the next available run cycle, due to pipeline stalls, hence the use of **next_inc** to skip to the next **run** cycle.

```
function next_ih
=== next_inc(ext_interrupt);next_inc(run^).
```

The following axioms formalise some of the most important task assumptions discussed in Sections 3.4 and 3.5. The predicate **take_interrupt(0)** means that an external interrupt is taken, and includes in its definition [1] requirements sufficient to ensure that an external interrupt is not preempted by an internal interrupt (exception).

```
axiom tasks_stalls_bounded
=== task_starts(I) and minslice_available
    => (next_ih;time - next_inc(ext_interrupt);time) =< maxstall.

axiom tasks_enable_ints
=== task_starts(I) and minslice_available
    => next_ih;(ints_enabled  and not int_masked(clk_int)
                and user_mode and take_interrupt(0)).

axiom tasks_maintain_invar
=== task_starts(I) and minslice_available => next_ih;task_inv.

axiom init_tasks_ready
=== hw => at(first);task_inv.
```

4.4 Formalising the Requirement

The scheduling requirement introduced in Section 3.3 is formalised as:

```
theorem system_behaviour
=== hw and N:nats
   => (letabs x N within
      (ex tm (tm:(first+x*clocksep) upto (first+x*clocksep+maxdelay+1)
              and at(tm);(task_starts(x mod nt)
                          and minslice_available))))
```

The requirement applies to every timing interval **N**, where 'intervals' are defined by **clocksep**. The quantifier **letabs** is used to define **x** as an 'absolute' value equal to **N**; absolute values do not change with time. The requirement states that, within **maxdelay** time units of the interval beginning, the appropriate task must begin executing, with at least **minslice** time units in which to do so.

5 The Scheduler Code

We verify the interrupt routine shown in Fig. 3. The following symbolic names are used to identify key virtual memory addresses and constants used by the scheduler.

ihcode: Address at which the interrupt handler code resides following the initialisation phase. (On the MIPS R3000 this is address `0x80000080`, the location to which external interrupts are vectored.)

ihdata: The base address of a contiguous area of memory used to store data for the interrupt handler (address `0x90000000` on the MIPS R3000). For convenience, we also define `ihdata_upper` to equal `ihdata >> 16` (*i.e.*, `0x9000`).

tcb: The offset from `ihdata` to an array of *task control blocks*. These task control blocks contain state information for each task to be scheduled. The first word in each task control block holds the starting address of the task code. This is called the *task base address*.

tcb_size: The size of each task control block, in bytes.

currtask: The offset from `ihdata` to a location which contains a pointer to the task control block of the current task. This offset effectively records the current task.

tcb_end: The offset from `ihdata` to a location which contains the value `ihdata+tcb+nt*tcb_size`. This constant is the address of the first location *after* the array of task control blocks. It is stored to avoid the overhead of recalculating it each time the interrupt handler runs.

6 The Verification

6.1 A Scheduler Invariant

In order to verify that this scheduler implements the scheduling requirement, we introduce a *scheduler invariant*: `sched_inv(I)` asserts that scheduler code and data are in cache, `tcb_end` and the task base addresses contain the expected values and that the 'current' task recorded by the scheduler is task `I`.

```
function sched_inv(I)
=== hw                          and user_mode
    and ints_enabled            and not int_masked(clk_int)
    and sched_code_in_cache     and sched_data_in_cache
    and (I : 0 upto nt)
    and dval(ihdata+currtask) = ihdata+tcb+I*tcb_size
    and dval(ihdata+tcb_end)  = ihdata+tcb+nt*tcb_size
    and (all y ((y : 0 upto nt)
                => dval(ihdata+tcb+y*tcb_size) = tcode(y))).
```

The notation `dval(X)` extracts the contents of data cache location `X`.

Now we prove the scheduling requirement strengthened by the assertion of the scheduler invariant, as follows:

```
theorem system_behaviour2
=== hw and N:nats
    => (letabs x N within
       (ex tm (tm: (first+x*clocksep) upto
                   (first+x*clocksep+maxdelay+1)
                and at(tm);(task_starts(x mod nt)
                           and minslice_available
                           and sched_inv(x mod nt)))))
```

6.2 Proof Strategy

The overall proof strategy is to prove the **system_behaviour2** theorem using induction on **N**. This leads to a base case and an inductive step. Figure 4 illustrates the inductive step of the proof. The inductive assumption is that there exists a time **tm** in the required range at which task **n** *mod* **nt** starts executing in the expected state, with **minslice** time units available, and the scheduler invariant holding. We must show that these conditions obtain again at an appropriate later time for task (**n** + **1**) *mod* **nt**. This involves reasoning through the execution of task **n** *mod* **nt**, and then through the occurrence of the clock interrupt and execution of the scheduler code.

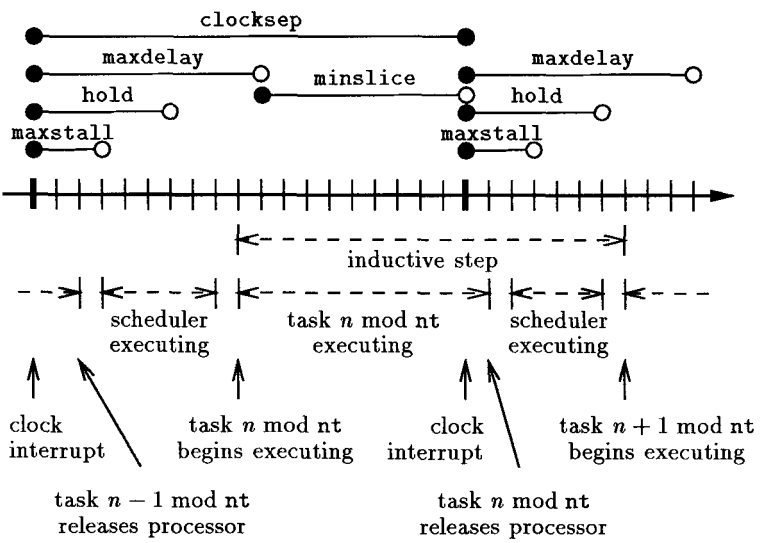

Fig. 4. Induction step for proof.

We conducted a top-down proof, in which the overall induction was proved with the aid of postulated lemmas about task behaviour and scheduler behaviour. These postulated lemmas were then proven in the next phase.

The **sched_behaviour** lemma defines the required behaviour of the scheduler code, from the time it starts, until the appropriate task is scheduled (including the initial pipeline fill time).

```
theorem sched_behaviour
=== first =< T          and  at(T);take_interrupt(0)
    and is_abs(I)       and at(T);task_inv
    and hw              and at(T);sched_inv(I)
    => (ex tm_2 (tm_2 : T+SchedMin upto T+SchedMax
                and at(tm_2);(task_starts((I+1) mod nt)
                              and sched_inv((I+1) mod nt)))).
```

The required minimum and maximum execution times of the scheduler were initially left as uninstantiated meta-variables. Constraints on those variables were postulated as required during the top-level proof, then a specific instantiation of them was chosen so as to satisfy these constraints. That instantiated version of the **sched_behaviour** lemma was then proven. (*SchedMin* and *SchedMax* here stand for those ultimate instantiations).

6.3 Constraints Extracted Interactively in Top-Level Proof

It transpired that the proof could be carried out only if **hold** $\leq$ **maxdelay**. This is a natural requirement, since the task cannot be started when the interrupt is still high, otherwise it will be interrupted straight away.

Further, to carry through the proof, the following constraints on *SchedMin* and *SchedMax* were required:

```
hold =< SchedMin  and  SchedMax =< maxdelay-maxstall+1 .
```

The first of these derives from the fact that the scheduler cannot start the task while the clock interrupt is still active. The second follows from the fact that the task must be started before **maxdelay** time units have expired, but in the worst case the previous task may have stalled the pipeline by up to **maxstall** time units.

A further constraint, **maxstall** < **hold**, was extracted during the proof of the task behaviour lemma [1] (not given here). Without this constraint, a task could stall the pipeline for so long that the clock interrupt is missed.

6.4 Constraint Extraction Technique

Interactively, additional constraints were derived when a relationship between the relevant parameters was required in order to progress. For example, as part of the base case of the induction the following predicate had to be proved (**time0** stands for the time at which task 0 starts in the base case):

```
time0 < ((first + 0*clocksep) + maxdelay) + 1
```

To do that, its negation was added to the current hypotheses, and automatically simplified. The simplified negation appears as hypothesis 5 in the following list.

```
3: SchedMin + first =< time0
4: 1 + time0 =< SchedMax + first
5: 1 + first + maxdelay =< time0
```

Deriving a contradiction from the augmented hypothesis list became the current goal. A linear arithmetic tactic [2] was then used to eliminate variables, in search of a contradiction. In this case, there was no contradiction, so we experimented with eliminating various variables. Eliminating `elim(first)` resulted in the derived hypothesis `2 + maxdelay =< SchedMax`, which corresponds to a sensible constraint on the scheduler implementation, so the negation of this was adopted as the weakest conjecture that permitted the deduction of false:

```
conjecture(SchedMax =< maxdelay + 1)
```

Then the command `elim(maxdelay)` resulted in the contradiction `1 =< 0`. Note that, in general, the choice of which constraint to add cannot be automated, since it is an application-specific design choice.

6.5 Proof of Scheduler Behaviour

Lemma `sched_behaviour` is instantiated so as to satisfy the constraints identified in Section 6.3: *SchedMin* is instantiated to `hold` and *SchedMax* is instantiated to `maxdelay - maxstall + 1`.

The instantiated lemma is now proven by stepping through the code, carrying the machine state forward through each instruction. Since the scheduler code has a very simple control structure we undertook its proof on a case-by-case basis. There are two situations, where the branch is and is not taken, respectively. The proof was split into several lemmas – one for each sequential segment of code.

The large number of variables that must be tracked (CPU registers, CP0 status registers, data and instruction cache contents, *etc.*) necessitated the development of lemmas for each of the instructions used by the scheduler. Each lemma has about a dozen antecedents that check various pipeline conditions and a consequent of similar complexity that defines the (timed) transition to the next machine state. Tactics and heuristics were developed which allowed approximately 90% of the antecedent conditions to be discharged automatically.

In many cases, to reason through the next instruction it was sufficient to make a single tactic call, which guessed which instruction lemma to apply next (by looking at the current value of `ireg^`), instantiated that lemma appropriately, and then discharged all its conditions (this typically took a couple of minutes). If the tactic failed to discharge a condition, it allowed the user to discharge it interactively before continuing with the remaining conditions.

A total of three person weeks was spent reasoning through the twelve machine code instructions, but at least two of those weeks were devoted to developing the supporting application-specific tactics and heuristics, which can be re-used in other code proofs.

6.6 Scheduler Specific Constraints

It turns out that under the imposed conditions, the scheduler code takes 11 or 12 machine cycles to execute. Adding the three cycles required to fill the pipeline when the interrupt handler is called, we deduce that the scheduler will take 14 or 15 machine cycles to execute altogether. The instantiated scheduler lemma cannot be proven without imposing further constraints on system parameters:

```
hold =< 14  and  15 =< maxdelay-maxstall .
```

Without the first of these, the scheduler could start the next task when the interrupt signal was still high. This is a constraint on the environment: the scheduler code in Fig. 3 cannot be used in an environment whose clock does not adhere to this requirement.

Without the second constraint, it would be impossible for the scheduler to start the next task within the user's specified deadline in the case where the previous task has stalled the pipeline for the maximum amount of time and the scheduler takes the maximum time to execute. This imposes a constraint on the user's requirement: the task designer must select values for `maxdelay` and `maxstall` that obey this constraint if they intend using the scheduler in Fig. 3.

7 Conclusions

We have given an overview of the verification of an interrupt-driven real-time scheduler for a pipelined processor. While the scheduler is very simple, the verification demonstrates a number of points which we believe are generally applicable to the formal analysis of real-time systems:

1. Careful analysis of assumptions about the execution environment is required. In the present case, a number of non-obvious assumptions were required about task behaviour, for example, about bounded stall time (Section 3.4), and properties of the first few task instructions (Section 3.4).
2. Timing parameters derived from the requirement and from the environment need to satisfy a number of constraints in order to make the specified system feasible in a given environment. There can be a large number of these constraints with subtle interactions. We found it difficult to (and we did not) identify all of these constraints at specification time. A number were derived in the proof process.
3. Formal, machine assisted proof is highly desirable to ensure that sufficient constraints have been identified to ensure the system is implementable. Further, as we have demonstrated, machine assisted proof can be used to *derive* the required (weakest) constraints. Some form of interaction is essential for this, since the choice of constraints is a design decision.
4. In general, all required constraints can be identified only when development is complete. For example, the scheduler can only satisfy the requirement if the requirement obeys a constraint deriving from the timing properties of the scheduler. This is an instance of a general phenomenon: some timing properties are satisfiable only using particular machines, compilers, etc.

5. Verification of real-time code requires tracking a large number of state variables – more than would be required in a purely functional verification. We have demonstrated the feasibility of automating much of this management, through the use of appropriate lemmas and tactics. Further, we found that machine assistance was necessary: our attempts at paper proofs were overwhelmed by detail.

References

1. Fidge, C., Kearney, P., and Utting, M., 'Formal Specification and Interactive Proof of a Simple Real-time Scheduler', Technical Report **94-11**, Software Verification Research Centre, Department of Computer Science. April 1994.
2. Griffiths, A., and Utting, M., 'The Automatic Proof of Theorems Involving Linear Inequalities', Technical Report **94-29**, Software Verification Research Centre, Department of Computer Science.
3. Kearney, P., Staples, J., Abbas, A., 'Functional Verification of Hard Real-Time Programs', in *Algorithms, Software, Architecture*, ed. L. van Lueewen, Information Processing 92, Volume I, North-Holland 1992, 113-119.
4. Kearney, P., Staples, J., Abbas, A. and Liu, C., 'Functional Verification of Real-Time Procedural Code: a Simplified RS232 Software Repeater Problem', Technical Report **91-2**, Software Verification Research Centre, Department of Computer Science. Revised, May 1992.
5. Kearney, P., and Utting., M., 'A Layered Real-time Specification of a RISC Processor', in: Langmaack, H., de Roever, W.-P. and Vytopil J., eds., *Formal Techniques in Real-time and Fault-Tolerant Systems*, Lecture Notes in Computer Science **863**, September 1994, pp. 455-475.
6. de Roever, W.P., 'The Quest for Compositionality', in:E.J. Neuhold and G. Chroust (eds.) *Formal Models in Programming*, North Holland, 1985.
7. Staples J., Robinson, P., Hazel, D., 'A functional logic for higher level reasoning about computation', *Formal Aspects of Computing*, Vol. 6, pages 1-38, 1994.
8. Utting, M., 'Instruction-level Specification of a MIPS R3000 CPU', Software Verification Research Centre Technical Report 93-26, February 1994, revised April 1994, 27 pages.
9. Utting, M., Kearney, P., 'Pipeline Specification of a MIPS R3000 CPU', Software Verification Research Centre Technical Report 92-6, October 1992, revised April 1994, 57 pages.
10. Utting, M. and Whitwell, K., 'Ergo 4.0 Users Manual', Technical Report **93-19**, Software Verification Research Centre, Department of Computer Science, University of Queensland, 1994.
11. Williams, Performance pushes RISC chips into real-time roles, *Computer Design*, September 1991, 79-86.

8 Acknowledgements

This work was supported by the Information Technology Division of the Australian Defence Science and Technology Organisation.