

**SOFTWARE VERIFICATION RESEARCH CENTRE
DEPARTMENT OF COMPUTER SCIENCE
THE UNIVERSITY OF QUEENSLAND**

**Queensland 4072
Australia**

TECHNICAL REPORT

No. 94-11

Formal Specification and Interactive Proof of a Simple Real-Time Scheduler

Colin Fidge, Peter Kearney and Mark Utting

April 1994

Phone: +61 7 365 1003

Fax: +61 7 365 1533

Formal Specification and Interactive Proof of a Simple Real-Time Scheduler

C. Fidge P. Kearney M. Utting
Software Verification Research Centre
Department of Computer Science
The University of Queensland
Queensland 4072, Australia
e-mail: cjf@cs.uq.oz.au

Abstract

A simple round-robin real-time scheduler for the MIPS R3000 RISC processor is modelled. Formal verification of its correctness using an interactive theorem prover is then described.

1 Introduction

Many real-time systems are structured as a set of tasks or processes. Scheduling is required to share computational resources between the tasks in such a way that the real-time requirements of the tasks are met.

This document specifies a simple scheduler and gives MIPS R3000 machine language code for a clock interrupt handler which implements the scheduler. It also gives a strategy for proving correctness of this code, using a (partial) formal specification of the functional and real-time properties of the MIPS RISC R3000 CPU [6, 7] and discusses experience with interactive verification of the scheduler using an automated theorem prover [9]. Verifying code for such an architecture involves a number of unique problems, in particular the need to account for the time taken to fill the instruction pipeline, the possibility of pipeline “stalls” and the seemingly “non-sequential” execution order of some assembler statements.

2 Background

There are two basic types of tasks in real-time systems: periodic and aperiodic [2]. Periodic tasks execute on a regular basis, typically to perform data sampling and control activities. The execution of an aperiodic task is triggered by asynchronous events external to the system. In general, tasks interact to meet system requirements. For example, tasks may read data provided by another task, a task may be initiated by another task, and tasks may compete for the use of non-sharable resources.

Task schedulers of typical commercial real-time kernels are priority based [1, 3, 4]. Application tasks may be assigned priorities dynamically and a running task is pre-empted whenever a higher-priority task becomes executable. In many systems, tasks of equal priority are time sliced. Amongst other services, systems may offer the ability to periodically schedule a task and to delay a task for a specified time.

The scheduler defined here has been chosen to be as simple as possible. It schedules a fixed number of tasks. Tasks are of equal priority. They are scheduled in a round robin fashion with each task being given an equal time slice. It is assumed that each task completes its critical activities, and thus becomes safely interruptible, within its time slice.

3 Model

This section describes some aspects of our model of the environment in which the scheduler operates, and of the tasks to be scheduled.

The time units are natural numbers, representing machine cycles on the target CPU. Time “intervals” are expressed as natural numbers defining for how many such time units some property holds. For instance, if a property has a duration of 3 time units starting from absolute time 10, then this property holds at times 10, 11 and 12, but not at time 13 or later.

3.1 Environment

The following symbolic constants name aspects of the model of the hardware environment in which the scheduler resides.

clocksep: The number of time units between successive clock interrupts.

hold: How many time units each clock interrupt remains active for, *i.e.*, how long the interrupt signal remains high.

It is assumed that no external interrupts occur other than clock interrupts and that no processor resets occur.

3.2 Tasks

The scheduler is required to run periodically a fixed number of tasks, giving all tasks the same period and priority. Tasks are expected to be well-behaved in the sense that they do not generate internal interrupts.

nt: The number of periodic tasks (assumed to be at least one). These are identified by the numbers $0 \dots nt - 1$. (In Section 5 an upper bound for **nt** is derived.)

minslice: The minimum number of uninterrupted time units available in each scheduled time-slice. Each task is obliged to complete its essential activities and become interruptible within this deadline.

tcode(*i*): The starting address of the code for task *i*, following the completion of initialisation activities (Section 3.3).

maxstall: The maximum number of time units any task may stall the processor pipeline, so as to prevent the system from servicing a clock interrupt. (Stalls may be caused by the need to access external memory, or to perform arithmetic operations such as multiplication or division, *etc.*)

The (common) period of all tasks is determined by the number of tasks and the separation between clock interrupts, *i.e.*, the period is $nt * \text{clocksep}$.

Tasks must not hinder the ability to service clock interrupts. Specifically, each task is expected to leave interrupts enabled from **minslice** time units after it begins executing.

For this simple scheduler, tasks are expected to maintain their own register states correctly. Thus, by **minslice** time units from the time a task begins executing, it must have saved all register values needed later, lest they be lost when the interrupt handler is invoked. (An extended version of the scheduler, presented in Appendix A, does not have this requirement.)

When defining **maxstall** we are concerned only with stalls that may affect the ability to service interrupts; the task designer thus need limit only those stalls that may take place **minslice** time units from the time each task begins.

We also require of the scheduled tasks that they do not cause the scheduler code or data to be shifted from instruction and data caches. In practice this requires that tasks do not reside at virtual locations which map to the same cache locations as the scheduler. (Conversely, the tasks will have specific requirements that the scheduler must not violate if the tasks are to be able operate correctly. These are represented by the application-specific invariant **task_inv** in Section 6.1.)

3.3 Initialisation

Our analysis applies to times following an initialisation phase undertaken by a system kernel. During this initialisation phase the scheduler code and task code are loaded into instruction cache and the scheduler data is loaded into data cache (see sections 5 and 3.2). The system then executes a non-stalling busy-wait loop, with interrupts enabled, awaiting the next clock interrupt.

first: Time at which the first clock interrupt occurs following completion of the initialisation phase.

We thus consider systems which are fully initialised at time **first**. The scheduler interrupt handler begins execution for the first time shortly thereafter. (There is a short delay before the interrupt handler code begins to have an effect due to the need to “fill” the instruction pipeline.)

4 Scheduler requirements

The scheduler is required to schedule each task every $\mathbf{nt} * \mathbf{clocksep}$ machine cycles. To help describe the inevitable overheads associated with context switching we define the following constant.

maxdelay: The maximum number of time units allowed to pass from the time a clock interrupt occurs until the next task begins execution. (The maximum delay plus the minimum acceptable time slice for each task are required to equal the clock interrupt separation, *i.e.*, $\mathbf{maxdelay} + \mathbf{minslice} = \mathbf{clocksep}$.)

We can now 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 $(n \bmod \mathbf{nt})$. Furthermore, that task must be allowed to execute without interruption for **minslice** time units from time t .

Figure 1 illustrates the overall requirement for the case $\mathbf{nt} = 3$. The solid line shows the passage of time, punctuated by regular clock interrupts, with a separation of **clocksep** time units. The open circle denotes a non-inclusive endpoint, the filled circle denotes an inclusive endpoint. The first interrupt following the completion of initialisation is denoted **first**. Below the line is shown the task that the scheduler must ensure is executing during each interval.

Figure 2 shows an interval between successive clock interrupts. The marks on the axis represent well-defined (*i.e.*, stable) system states, each separated by one machine cycle. In this illustration **clocksep** is 23, **maxdelay** is 10, **hold** is 6, **maxstall** is 3 and **minslice** is 13.

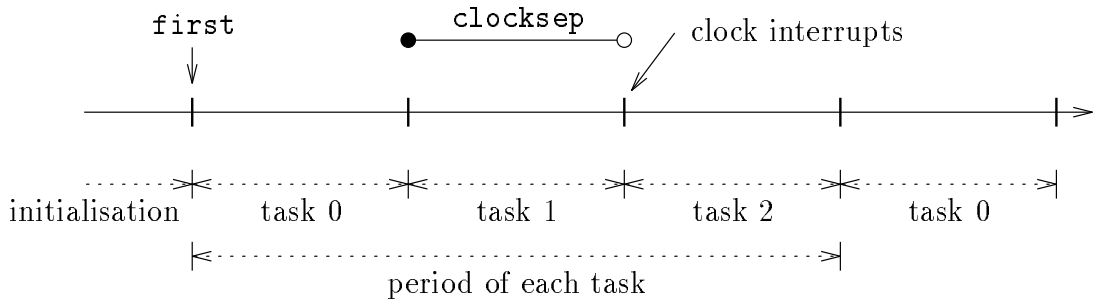


Figure 1: Overall scheduling requirement.

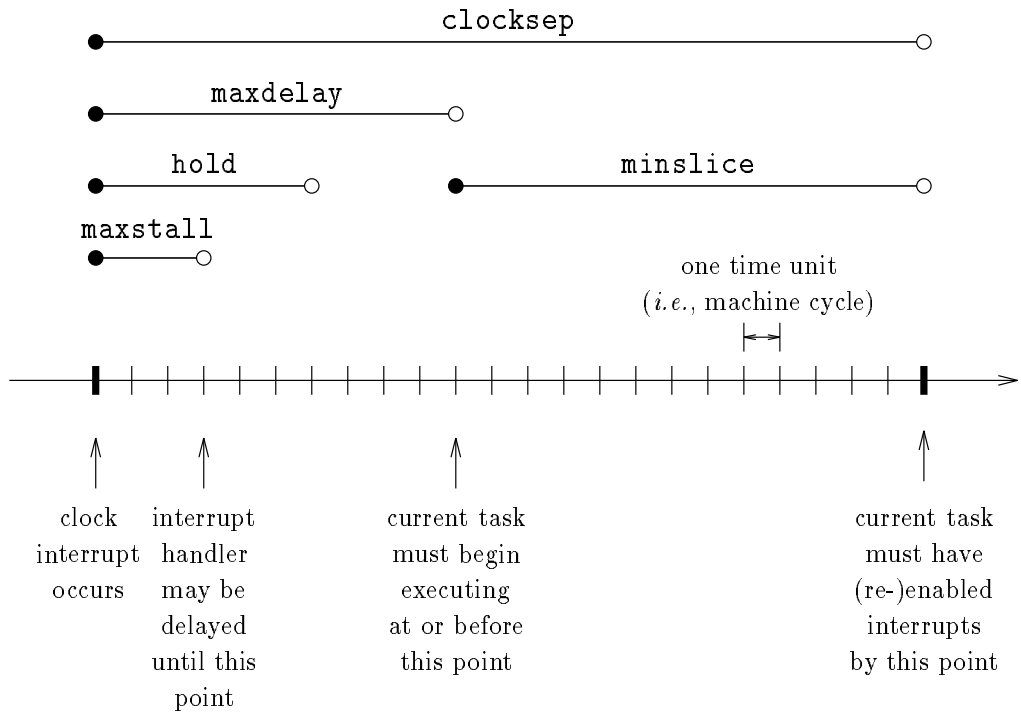


Figure 2: Scheduling requirement for a particular interval.

5 Simple clock interrupt handler

In order to achieve the requirement in Section 4, we consider verification of a very simple scheduler (Figure 3). Nevertheless, verification of this scheduler involves the key problems of timing and interrupt handling. (Appendix A shows an extended version of the scheduler.)

The following symbolic names are used to identify key virtual memory ad-

```

# Get pointers to current task control block and tcb_end.

    lui $k0, ihdata_upper    # Load base pointer to scheduler
                             # data into register $k0.
    lw $k1, currtask($k0)   # Load $k1 with pointer to current
                             # task control block.
    lw $k0, tcb_end($k0)    # Load $k0 with tcb_end pointer.

# Increment the currtask pointer to the next task.

    addiu $k1, $k1, tcb_size # Increment task base address
                             # (in $k1) by adding tcb_size.
    bne $k0, $k1, incr       # Branch around the next change
                             # to $k1 if $k1 is a legal tcb
                             # pointer, i.e., if $k1 does not
                             # equal tcb_end.
    lui $k0, ihdata_upper    # Load base pointer into $k0.
                             # NB. always executed (branch
                             # delay slot).
    addiu $k1, $k0, tcb      # Put address of the first task,
                             # tcb 0, i.e., ihdata + tcb,
                             # into $k1.
incr: sw $k1, currtask($k0)  # Store new value of current task
                             # pointer from $k1 into location
                             # ihdata + currtask.

# Start new task.

    lw $k0, 0($k1)          # Load start pc for current task
                             # from $k1 into $k0.
    nop                    # Do nothing (load delay slot).
    jr $k0                 # Jump to start pc for current
                             # task.
    rfe                    # Restore status reg settings.

```

Figure 3: Scheduler code.

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*. In the extended scheduler in Appendix A the task control blocks also hold register values.

tcb_size: The size of each task control block, in bytes. (When registers are saved, a value of 128 would be typical for `tcb_size`.)

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

Entry to the interrupt handler must be preceded by the initialisation section of kernel code (see Section 3.3). Prior to time `first` the initialisation routine must have set the following interrupt handler data areas.

1. Location `ihdata + tcb_end` must contain the address described above: `ihdata + tcb + nt * tcb_size`.
2. Location `ihdata + currtask` must contain the address of the *last* task control block. When the interrupt handler is called for the first time it will “increment” this value (modulo `nt`) and thus load the *first* task.
3. Each task control block must be suitably initialised. In particular, the task base address for each task *i* must contain the address of the first instruction of that task, *i.e.*, `tcode(i)`.

Figure 4 shows a typical data cache configuration, assuming `nt = 3` and the current task is number 1.

The size of data cache limits the number of tasks which the scheduler can handle. In general `tcb + nt * tcb_size` must be less than the maximum data cache size. With typical values of 128 for `tcb_size` and 64k for the size of data cache this fixes the limit at about 500 tasks!

The MIPS assembly language convention is that registers `$k0` (`$26`) and `$k1` (`$27`) are reserved for the operating system kernel. This convention has been

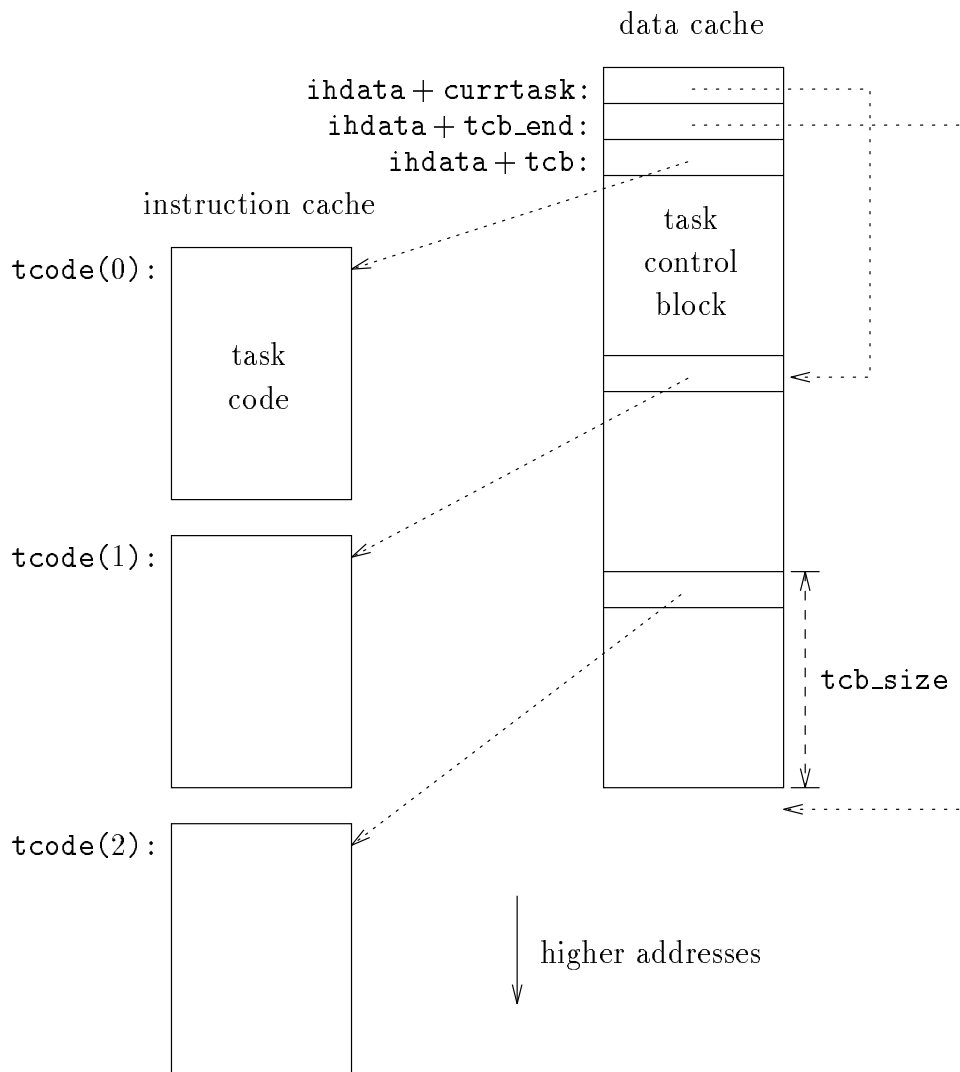


Figure 4: Example scheduler data cache configuration.

followed in the clock interrupt handler: registers `$k0` and `$k1` are used freely in Figure 3.

In calculating the base address of the next task it is necessary to use modulo arithmetic. That is, if the address in the `currtask` location is A , then the next value of `currtask` will be

$$(A - (\text{ihdata} + \text{tcb}) + \text{tcb_size}) \bmod (nt * \text{tcb_size}) + \text{ihdata} + \text{tcb} .$$

This effect is achieved in Figure 3 by a conditional branch that resets the `currtask` pointer back to the first task control block whenever it is incremented past the

last task control block.

Some MIPS assembly language instructions can generate several machine instructions, but no such instructions have been used in Figure 3; this slightly simplifies timing analysis but is not crucial. Due to the instruction pipeline, the instruction *following* a branch or jump is always executed, so the `lui` instruction following the `bne` is always executed, even when the branch is taken. Another noticeable effect of the pipelined architecture is that it is necessary to insert `nop` instructions after any load instruction where we want to use the result of the load in the following instruction [8].

6 Formalisation

This section gives a formalisation of the scheduler requirements in the form of a theorem to be proved in the formal theory of the MIPS processor [7]. The definitions, theorems, *etc.*, in this section are expressed in a machine-readable ASCII form suitable for direct input into the Ergo interactive theorem prover used [9].

The formal theory defined in this section is an extension of the following Ergo theories:

<code>r</code>	— first-order predicate calculus, with weak equality
<code>arith</code>	— integer arithmetic
<code>fun</code>	— functional logic [6, Section 2]
<code>div</code>	— the <code>div</code> and <code>mod</code> operators, plus additional arithmetic properties specific to functional logic.
<code>realtime</code>	— theory of traces and times
<code>twoscomp</code>	— twos complement arithmetic [6, Section 3]
<code>decode</code>	— MIPS machine code instructions [6, Section 6]
<code>cp0</code>	— the MIPS co-processor registers [6, Section 7]
<code>ispec</code>	— the instruction level MIPS specification [7]

6.1 Definitions

This section introduces a number of defined terms to facilitate the formal statement of the theorem.

Firstly we formally define the various names used above. Note that the set A upto B includes A , but excludes B .

```
absolute_constant clocksep: nats.
absolute_constant hold: nats.
absolute_constant nt: nats.
axiom nt_min == nt > 0.
axiom nt_max == tcb + nt*tcb_size : 0 upto dcache_size.
```

```

absolute_constant minslice: nats.
absolute_function tcode(I:0 upto nt): nats.
absolute_constant maxstall: nats.
absolute_constant first: times.
absolute_constant maxdelay: nats.
constant ihcode      === seg(4) + 128.  % ie. 80000080 hex.
constant ihdata      === seg(4) + 2**28. % ie. 90000000 hex.
constant ihdata_upper === ihdata >> 16.
constant currtask    === 0.
constant tcb_end     === 4.
constant tcb         === 8.
constant tcb_size    === 128.
constant k0          === 26.
constant k1          === 27.

```

For definiteness, we define which interrupt is asserted by the clock.

```

constant clk_int === 0.
constant clk_set === int_asserted(clk_int).

```

The predicate `ext_interrupt` is true if any external interrupt `x` is asserted.

```

constant ext_interrupt === (ex x int_asserted(x)).

```

The predicate `sched_code_in_cache` below asserts that each individual instruction of the scheduler code is in instruction cache, starting at the interrupt handler address. Notice that the label “incr” in the assembly code shown in Figure 3 has been translated to an absolute offset in the following machine code representation. Instructions are 4-byte aligned; the offset states that the branch from `ihcode + 16` is to a location $2 * 4$ bytes from the *next* instruction, *i.e.*, to location `ihcode + 28`.

```

constant sched_code_in_cache
===      ival(ihcode)      = lui_instr(k0, ihdata_upper)
      and ival(ihcode + 4) = lw_instr(k1, currtask, k0)
      and ival(ihcode + 8) = lw_instr(k0, tcb_end, k0)
      and ival(ihcode + 12) = addiu_instr(k1, k1, tcb_size)
      and ival(ihcode + 16) = bne_instr(k0, k1, 2)
      and ival(ihcode + 20) = lui_instr(k0, ihdata_upper)
      and ival(ihcode + 24) = addiu_instr(k1, k0, tcb)
      and ival(ihcode + 28) = sw_instr(k1, currtask, k0)
      and ival(ihcode + 32) = lw_instr(k0, 0, k1)
      and ival(ihcode + 36) = nop_instr
      and ival(ihcode + 40) = jr_instr(k0)
      and ival(ihcode + 44) = rfe_instr.

```

The constant `ihdata_size` gives the number of bytes of data used by the scheduler.

```
constant ihdata_size === tcb + nt * tcb_size.
```

Assertion `sched_data_in_cache` then states that the scheduler data is in data cache.

```
constant sched_data_in_cache
=== all x ((x : ihdata upto ihdata + ihdata_size)
=>
in_dcache(x)).
```

It is also convenient to define the following values. The constant `ihcode_size` is the number of bytes occupied by the scheduler code. The other values represent the first locations used by the scheduler in instruction and data caches, and the first addresses *after* these two areas.

```
constant ihcode_size === 12 * 4.
constant sc_low  === ihcode mod icache_size.
constant sd_low  === ihdata mod dcache_size.
constant sc_high === (ihcode + ihcode_size) mod icache_size.
constant sd_high === (ihdata + ihdata_size) mod dcache_size.
```

We define the predicate `sched_inv` which asserts that scheduler code and data are in cache and that `tcb_end` and the task base addresses contain the expected values. The “current” task, as seen by the scheduler, is intended to be task `I`. It is the responsibility of the tasks to maintain this invariant, *i.e.*, not to interfere with the scheduler. The `hw` predicate will be used extensively. It introduces implicit parameters that model all feasible runs of the MIPS R3000 processor [6, p.12].

```
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 predicate `sched_unchanged(t_1, t_2)` asserts that the scheduler code and data are the same at time t_2 as they were at time t_1 .

```
function sched_unchanged(T1,T2)
=== (all x ((is_abs(x)
            and (x : ihdata upto ihdata + ihdata_size
                and at(T1);in_dcache(x)))
            =>
            at(T2);dval(x) = at(T1);dval(x)))
and
(all x ((is_abs(x)
            and (x : ihcode upto ihcode + ihcode_size)
            and at(T1);in_icode(x)
            =>
            at(T2);ival(x) = at(T1);ival(x))).
```

(Predicate `is_abs` is used to state that `x` does not change with time.)

Predicate `task_inv` is not defined here, because its choice depends on the particular set of tasks to be scheduled, but, as seen below, it plays the rôle of an invariant for the set of tasks.

```
constant task_inv.
```

It is not the responsibility of the scheduler to maintain this invariant. Rather the scheduler is shown to not change instruction or data cache locations belonging to the tasks and it is required that this be sufficient to preserve `task_inv` (see axiom `tasks_inv_locality` in Section 6.2.2).

It is convenient to introduce the following predicates which assert that cache locations other than those used by the scheduler have the same values at times $T1$ and $T2$.

```
function d_inv(T1,T2)
=== all x (((x : 0 upto dcache_size)
            and is_abs(x)
            and in_dcache(x)
            and not (x : sd_low upto sd_high))
            =>
            at(T2);dval(x) = at(T1);dval(x)).
```

```
function c_inv(T1,T2)
=== all x (((x : 0 upto icode_size)
            and is_abs(x)
            and in_icode(x)
            and not (x : sc_low upto sc_high))
            =>
            at(T2);ival(x) = at(T1);ival(x)).
```

6.2 Formalising the model

This section re-expresses model from sections 3 and 4 in the formal theory. To make some properties easier to express, we define an *inclusive* form of the `next(E)` operator [6] which leaves the time unchanged if `E` is true at the current time.

```
function next_inc(E) == if(E, id, next(E)).
```

6.2.1 Environmental modelling

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`.

```
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)).
```

(Predicate `during(t_1, t_2, b)` includes time t_1 , but not t_2 . Also note that in the interactive theorem prover free variables in axioms and theorems are implicitly universally quantified, hence there is no need to say “all `N`” above.)

The following assertion states that after initialisation the only external interrupts are clock interrupts. (Pipeline level predicate `int_asserted` refers to *external* interrupts only [6].)

```
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.
```

(The situation where `hold` equals `clocksep`, *i.e.*, the clock interrupts are permanently active, is possible only in the useless situation where `minslice` is 0.) It is implicit in this axiom that `clocksep` is greater than 0.

No processor resets are tolerated at any time after the initialisation phase. Due to the latency inherent in filling the instruction pipeline this means that no resets may be allowed for at least 3 run cycles prior to `first`.

```
axiom no_resets
=== (hw
    and T:times
    and at(first);nth(run^, - 3);time =< T)
=>
    at(T);(not reset^).
```

6.2.2 Task modelling

We frequently refer to the system state in which a task `I` begins or resumes executing. We define this as any time at which the program counter equals the starting address of the code for task `I` and the processor is in a state that allows that code to begin normal execution. The following predicate will be used.

```
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.
```

The following predicate, which asserts that there must be no external interrupts for at least `minslice` time units, is also used in several axioms below in order to guarantee that the task has sufficient time in which to execute.

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

(If a task i is iterative and returns to location `tcode(i)` these predicates may not define a *unique* time in each timeslice, but this is not significant in the following definitions.)

It is convenient to define a function, `next_ih`, which shifts the time to the time at which the next external interrupt is handled.

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

(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.)

Each task must (re-)enable interrupts and be executing in user mode before this time is reached. Furthermore, each task must ensure that it does not cause the next external interrupt to be overridden by a higher priority internal interrupt. (Interrupts due to errors on the bus, *i.e.*, `error_bus_data`, are precluded by axiom `clock_ints_only` above).

```
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(error_interrupt)).
```

A task must not stall the pipeline for more than `maxstall` time units during the period that clock interrupts are expected. In other words, the interrupt handler must begin execution within `maxstall` time units of an interrupt occurring.

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

Each task, if started at a time when the task invariant holds, must ensure that the task invariant is true at the time the next clock interrupt is serviced.

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

The next assumption asserts that the scheduler code and data are unchanged by each task I from the time the task starts executing until the next interrupt is serviced.

```
axiom tasks_avoid_sched
=== task_starts(I)
   and minslice_available
=>
   sched_unchanged(time, next_ih;time).
```

From this assumption we can conclude that if the scheduler invariant was true when a task started then it will still be true when the task finishes (because `sched_inv` is defined using scheduler cache locations only).

The following axiom defines the relationship between `maxdelay`, `minslice` and `clocksep`.

```
axiom tasks_timeslice_defn
=== maxdelay + minslice = clocksep.
```

The task invariant is assumed to remain unchanged if data and instruction cache locations other than those used by the scheduler remain unchanged. (This precludes a task invariant that is dependent on the absolute time.)

```
axiom tasks_inv_locality
=== (hw
   and T1:times
   and T2:times
   and T1 =< T2
   and d_inv(T1, T2)
   and c_inv(T1, T2)
   and at(T1);task_inv)
=> at(T2);task_inv.
```

To ensure that precise bounds can be placed on the execution time of the interrupt handler code, the first two instructions (at least) of each task are kept in the instruction cache. (Otherwise the timing behaviour of the `jr` and `rfe` instructions at the end of the scheduler code may be dependent on how long it takes to load the task instructions from external memory.)

```
axiom tasks_in_cache
=== task_inv
   and (I : 0 upto nt)
=> (is_word_addr(tcode(I))
   and in_icache(tcode(I))
   and in_icache(tcode(I)+4)
   and group(ival(tcode(I))) \= hilo_instr
   and group(ival(tcode(I)+4)) \= hilo_instr
   and tcode(I)+4 < 2**31).
```

6.2.3 Initialisation modelling

By the time the first interrupt is handled the scheduler code and data have been loaded and the “current” task base address is the “last”, *i.e.*, `nt - 1`.

```
axiom init_sched_ready
=== hw
=> at(first);sched_inv(nt - 1).
```

By the time the first interrupt is handled the task invariant has also been established (*e.g.*, task code has been loaded into cache).

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

The interrupt handler code at address `ihcode` is called at the interrupt denoted `first`.

```
axiom init_first_int
=== hw
=> at(first);take_interrupt(error_interrupt).
```

No allowance has been made for an initial stall period. The model assumes that the system is idle when the interrupt designated “first” arrives. However, as indicated by the discussion in Section 7.1.3, it is straightforward to incorporate an allowance for an initial stall period.

The initialisation code must switch off the Boot Exception Vector flag in the status register, thus putting the processor into normal operating mode, rather than bootstrap mode. This is necessary to ensure that interrupts cause the scheduler code (at address `ihcode`) to be called.

```
axiom init_not_bev
=== hw
and T:times
and first =< T
=> at(T);(not cp0_status_bev).
```

6.3 Formalising the user requirements

The user’s requirements for behaviour of the scheduler (Section 4) may be formalised as follows.

```

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
                            and sched_inv(x mod nt)))))).

```

The requirement applies to every timing interval N , where “intervals” are defined by `clocksep` (`letabs` is used to define x as a copy of the current value of the universally quantified variable N , but which does not change with time). It 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.

In order to make this possible in *every* time interval there is also a requirement that `sched_inv` holds at time `tm`. We know from assumption `tasks_avoid_sched` that tasks do not change the scheduler invariant, so adding this scheduler requirement allows us to immediately conclude that the interrupt handler will not be hindered by the tasks on its *next* invocation.

7 Proof strategy

The requirements above were proven using an interactive theorem prover [9]. This section sketches the overall proof and Section 8 discusses some of the lessons learned.

The proof proceeds by induction on N . The base case covers the period between the first clock interrupt and task 0 beginning; the general induction step then covers the period between the start of a task and the start of its successor (Figure 5).

In more detail, the base case involves the period from the time the clock interrupt denoted `first` occurs until the scheduler gives processor control to task 0 (Figure 6). The initial delay for the scheduler code to fill the instruction pipeline is accounted for in the analysis of the scheduler code (see below). Recall that the initialisation code is assumed not to stall the pipeline.

The induction step covers the period between the beginning of task $n \bmod nt$ and the beginning of task $n + 1 \bmod nt$ (Figure 7). This includes two main intervals, where task $n \bmod nt$ and the scheduler control the processor, respectively. The former includes the execution of the task until the next clock interrupt occurs, and the possible “over-run” due to the task having stalled the instruction pipeline just before the interrupt. The latter includes the small initial delay required for

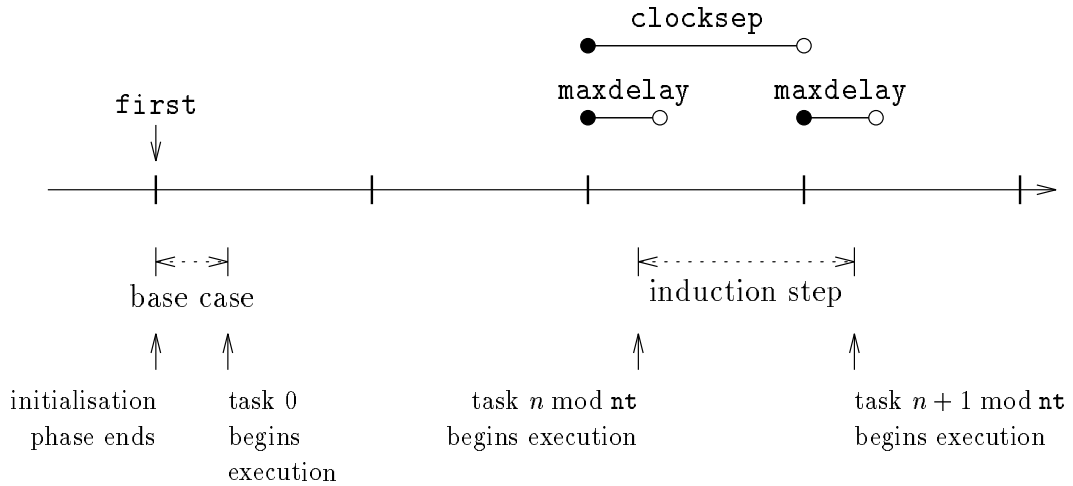


Figure 5: Strategy for inductive proof.

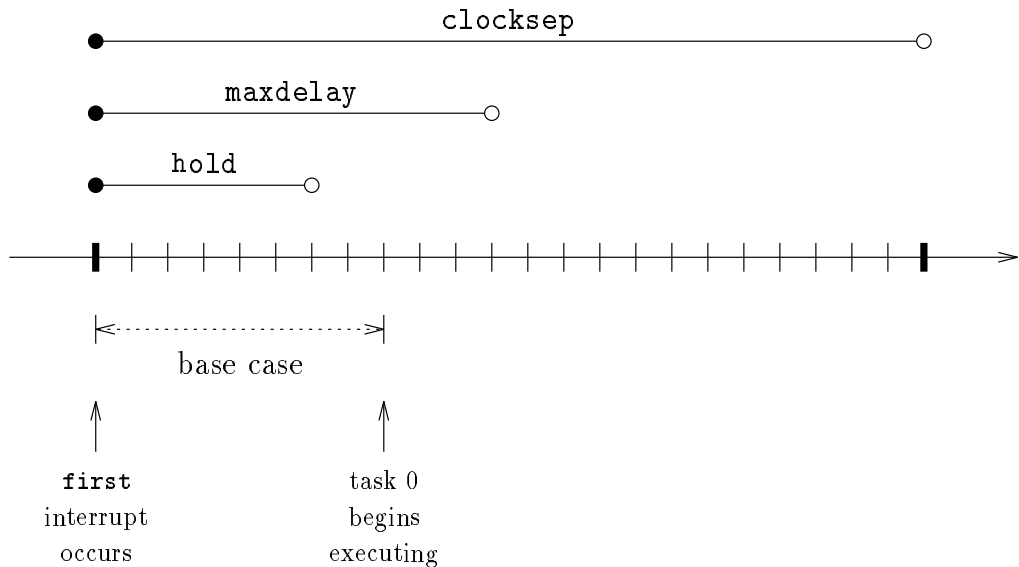


Figure 6: Basis for inductive proof.

the scheduler code to fill the pipeline and the execution of the scheduler code itself. (There is no “fill time” for tasks—as long as their code stays in cache—because they are started explicitly by the final statements of the scheduler code, rather than through an interrupt.) Note that the period of time covered by each induction step is not constant; the time between consecutive task “starts” varies due to the differing length of task stalls (between 0 and `maxstall`) and the dif-

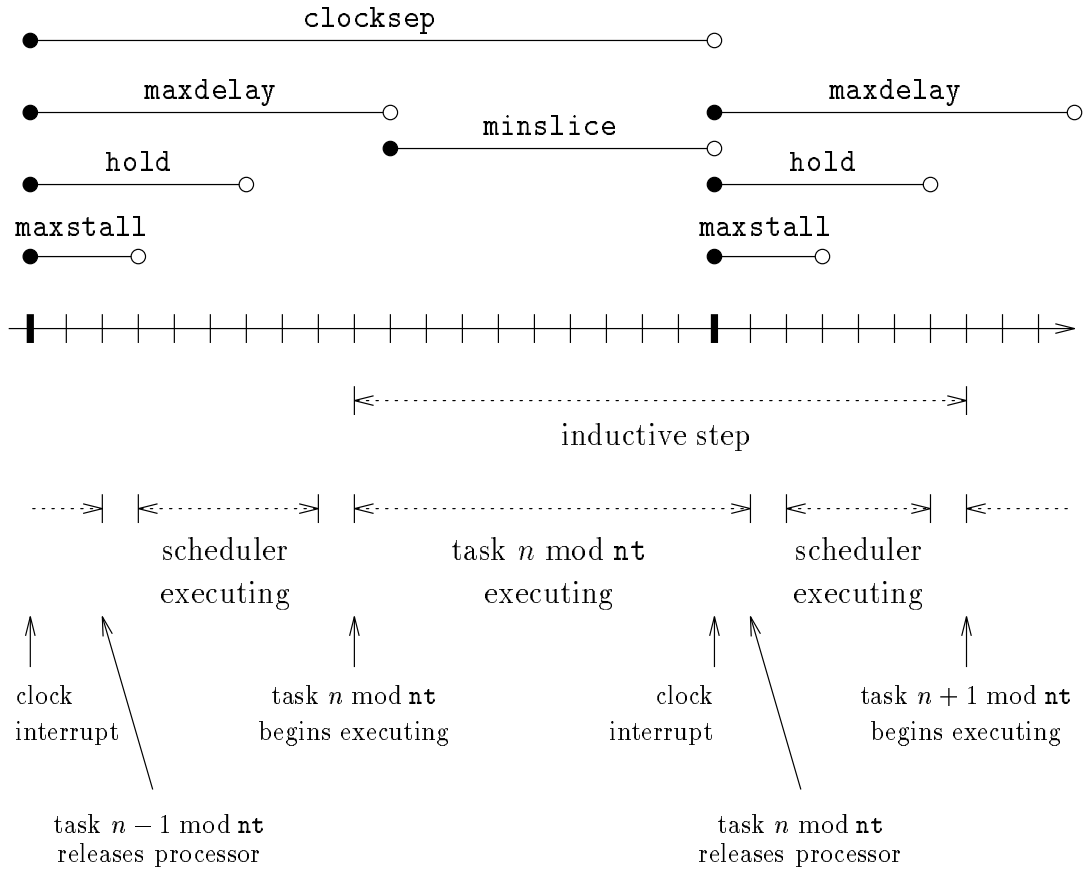


Figure 7: Induction step for proof.

ferent possible execution times for the scheduler. (These are the only sources of timing variation in this system.)

To support the proof there are three major lemmas, whose proofs are discussed in Section 7.2. The first two describe the overall behaviours of the tasks and the scheduler.

Lemma 7.1 *Define the effect of executing a task, from the time it starts until it releases the processor (including a possible final stall).*

```

theorem task_behaviour
=== first =< T
  and at(T);task_starts(I)
  and at(T);minslice_available
  and at(T);sched_inv(I)
  and is_abs(I)
  and hw

```

```

=>
at(T);next_ih;(sched_inv(I)
    and task_inv
    and take_interrupt(error_interrupt)).

```

The next lemma has been proved for specific values of the metavariables *SchedMin*, *SchedMax*, which represent the minimum and maximum permissible execution times for the scheduler, respectively. To emphasise this, these metavariables (whose use and instantiation is supported by the Ergo theorem prover) have been shown here in italics.

Lemma 7.2 *Define the effect of executing the interrupt handler 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(error_interrupt)
    and at(T);task_inv
    and at(T);sched_inv(I)
    and is_abs(I)
    and hw
=>
(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))).

```

In the conclusion of `sched_behaviour` the value of `tm` tells us the acceptable range of times at which the next task can start. We have chosen to derive *SchedMin* and *SchedMax* during the course of the proof (see sections 7.1.1 to 7.1.3), and then to prove that the scheduler code satisfies these bounds (Section 7.2.3).

The third lemma allows us to determine those moments in absolute time where there are at least `minslice` time units before the next clock interrupt.

Lemma 7.3 *Define suitable starting times for tasks.*

```

theorem task_start_times
=== hw
    and N : nats
    and T : (first + N*clocksep + hold)
            upto (first + N*clocksep + maxdelay + 1)
=>
at(T);minslice_available

```

7.1 Top-level proof

This section outlines the overall proof of the user’s scheduling requirements.

7.1.1 A stronger requirement

An attempt to prove the user’s `system_behaviour` requirement reveals that a more efficient proof can be achieved by strengthening the bounds on `tm`.

Consider the base case of `system_behaviour`, when $N = 0$. The conclusion then becomes

```
ex tm (tm : first upto
      (first + maxdelay + 1)
      and at(tm);(task_starts(0)
                  and minslice_available
                  and sched_inv(0))) .
```

However, in order to show that `minslice_available` is true in this first time interval, it is obvious from lemma 7.3 with $N = 0$ that `tm` must be in the range

```
(first + hold) upto (first + maxdelay + 1) .
```

This is not surprising—the scheduler cannot be allowed to start the task while the clock interrupt signal is still high, otherwise the task will be immediately interrupted. Furthermore, it must be assumed that

```
hold ≤ maxdelay ,
```

otherwise no such `tm` exists. This is a natural requirement but was not stated in Section 6.2 because it is neither a property of the tasks nor of the environment; it stems from the interaction between the two.

Having discovered this we determined to prove the original requirement via the following stronger one.

```
theorem system_behaviour_2
=== (hw
     and N:nats)
=>
(letabs x N within
 (ex tm (tm : (first + x*clocksep + hold) upto
             (first + x*clocksep + maxdelay + 1)
             and at(tm);(task_starts(x mod nt)
                         and sched_inv(x mod nt)))))).
```

Notice that predicate `minslice_available` is not explicit here because after `tm` is shown to be in this tighter range, lemma 7.3 allows this predicate to be satisfied immediately.

7.1.2 Proof of basis

In the base case we substitute 0 for N in `system_behaviour_2` to get

```
ex tm (tm : (first + hold) upto
          (first + maxdelay + 1)
          and at(tm);(task_starts(0)
                      and sched_inv(0)))
```

as our goal.

From the initialisation assumptions in Section 6.2.3 we know that

```
at(first);(sched_inv(nt - 1)
           and task_inv
           and take_interrupt(error_interrupt)).
```

This allows us to instantiate lemma 7.2 at time `first` to yield

```
ex tm_2 (tm_2 : first + SchedMin upto first + SchedMax
         and at(tm_2);(task_starts(((nt - 1) + 1) mod nt)
                       and sched_inv(((nt - 1) + 1) mod nt)))
```

which, using the fact that

$$((nt - 1) + 1) \bmod nt = 0 ,$$

gives us

```
ex tm_2 (tm_2 : first + SchedMin upto first + SchedMax
         and at(tm_2);(task_starts(0)
                       and sched_inv(0))).
```

This satisfies the goal above, provided that we adopt the conjectures

$$\text{hold} \leq \textit{SchedMin}$$

and

$$\textit{SchedMax} \leq \text{maxdelay} + 1 .$$

We have thus determined initial constraints on the execution time of the scheduler, if it is to satisfy the user's requirement. (As shown in the next section, the inductive step further tightens the constraints.)

7.1.3 Proof of induction step

For the inductive step assume that there is an N and tm in `system_behaviour_2` such that

```
ex tm (tm : (first + N*clocksep + hold) upto
           (first + N*clocksep + maxdelay + 1)
           and at(tm);(task_starts(N mod nt)
                       and sched_inv(N mod nt)))
```

and attempt to show the same condition for $N + 1$. (For clarity we ignore, in this section, the trivial need to show that N is an absolute constant.)

Lemma 7.3 allows us to conclude `minslice_available` at time tm and we can then substitute tm for T and $N \bmod nt$ for I in the hypothesis of lemma 7.1 to conclude that

```
at(tm);next_ih;(sched_inv(N mod nt)
                 and task_inv
                 and take_interrupt(error_interrupt)).
```

This step represents the period from the time task $N \bmod nt$ started until a clock interrupt occurs and the interrupt handler is invoked.

From here the hypothesis of lemma 7.2 is instantiated, substituting `at(tm);next_ih;time` for tm and $N \bmod nt$ for I , to give the conclusion

```
ex tm_2 (tm_2 : (at(tm);next_ih;time + SchedMin)
               upto (at(tm);next_ih;time + SchedMax)
               and at(tm_2);(task_starts(((N mod nt) + 1) mod nt)
                               and sched_inv(((N mod nt) + 1) mod nt))).
```

Arithmetic tells us that

$$((N \bmod nt) + 1) \bmod nt = (N + 1) \bmod nt$$

thus giving us the desired task identifier.

It then remains to be shown that tm_2 satisfies

```
tm_2: (first + (N+1)*clocksep + hold) upto
      (first + (N+1)*clocksep + maxdelay + 1).
```

Applying lemma 7.4 (Section 7.3) to tm in the inductive assumption determines when the next external interrupt is expected (according to the environmental assumptions in Section 6.2.1). That is,

```
at(tm);next_inc(ext_interrupt);time = first + (N+1)*clocksep.
```

Then, using the knowledge that the interrupt handler will be invoked *after* this time, but within `maxstall` time units (axiom `task_stalls_bounded` in Section 6.2.2), it can be seen that the scheduler code will begin executing in the range

```
at(tm);next_ih;time: (first + (N+1)*clocksep) upto
                    (first + (N+1)*clocksep + maxstall).
```

From this it is clear that the scheduler will be able to satisfy the timing requirement for `tm_2` as long as the following constraints hold. Firstly,

$$\text{hold} \leq \text{SchedMin}$$

so that in the best case, where there was no pipeline stall, the scheduler cannot start the task while the clock interrupt is still active. (This is the same as in the base case.)

Secondly,

$$\text{SchedMax} \leq \text{maxdelay} - \text{maxstall} + 1$$

because 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. The extra “1” is because *SchedMax* is an *exclusive* endpoint in `sched_behaviour`: it represents the worst-case execution time of the scheduler plus one. (This is a stronger constraint on *SchedMax* than was found in the base case because the base case assumed the pipeline would not be stalled initially.)

Thus requirement `system_behaviour_2` can be satisfied as long as the scheduler code adheres to the timing constraints in the following instantiation of lemma 7.2.

```
theorem sched_behaviour
=== first =< T
    and at(T);take_interrupt(error_interrupt)
    and at(T);task_inv
    and at(T);sched_inv(I)
    and is_abs(I)
    and hw
=>
(ex tm_2 (tm_2 : (T + hold) upto
          (T + maxdelay - maxstall + 1)
          and at(tm_2);(task_starts((I+1) mod nt)
          and sched_inv((I+1) mod nt))))).
```

Having proven `system_behaviour_2` the original `system_behaviour` requirement follows easily by using lemma 7.3 to note that `minslices_available` must be true for `tm_2` and observing that the specified range for `tm` in `system_behaviour` is merely a weakening of the proven timing behaviour in `system_behaviour_2`.

7.2 Proofs of major lemmas

This section outlines the proofs of the three major lemmas used in Section 7.1.

7.2.1 Proof of tasks behaviour

Lemma 7.1 is proven readily, using the many assumptions we have made about the nature of tasks and the three principal conjuncts in its hypothesis, *i.e.*,

$$\begin{aligned} & \text{at}(T); \text{task_starts}(I) \\ & \text{and at}(T); \text{minslice_available} \\ & \text{and at}(T); \text{sched_inv}(I) . \end{aligned}$$

There are three conjuncts to be proven in the conclusion. Firstly, to show that

$$\text{at}(T); \text{next_ih}; \text{sched_inv}(I)$$

axiom `tasks_enable_ints` (Section 6.2.2) tells us that

$$\begin{aligned} & \text{at}(T); \text{next_ih}; (\text{ints_enabled} \\ & \quad \text{and not int_masked}(\text{clk_int}) \\ & \quad \text{and user_mode} \\ & \quad \text{and take_interrupt}(\text{error_interrupt})) . \end{aligned}$$

Then axiom `tasks_avoid_sched` is used to yield

$$\text{sched_unchanged}(T, \text{next_ih}; \text{time})$$

which is sufficient to give the desired conclusion using lemma 7.5 (Section 7.3).

The other conjuncts are even simpler. Conclusion

$$\text{at}(T); \text{next_ih}; \text{task_inv}$$

is proven immediately via assumption `tasks_maintain_invar` (Section 6.2.2).

Similarly,

$$\text{at}(T); \text{next_ih}; \text{take_interrupt}(\text{error_interrupt})$$

follows directly from axiom `tasks_enable_ints` (Section 6.2.2).

Having made use of `tasks_enable_ints` it is interesting to note an implicit constraint it imposes on the system variables. In order for this axiom to assert that an interrupt is taken at the next run cycle following the beginning of a clock interrupt it must be the case that

$$\text{maxstall} < \text{hold} ,$$

otherwise a task could stall the pipeline for so long that the clock interrupt is missed. This obvious requirement did not appear explicitly in Section 6.2 because it is not a constraint on any one of the system components but results from the interaction between the tasks and the environment.

7.2.2 Proof of available timeslice

Lemma 7.3 is proven using the assumed properties of the environment. Expanding the definition of `minslice_available`, we must show from some `N` and `T` satisfying

$$\begin{aligned} T : & (\text{first} + N * \text{clocksep} + \text{hold}) \text{ upto} \\ & (\text{first} + N * \text{clocksep} + \text{maxdelay} + 1) \end{aligned}$$

that

$$\text{at}(T); \text{next_inc}(\text{ext_interrupt}); \text{time} \geq T + \text{minslice}.$$

Lemma 7.4 (Section 7.3) allows us to reach this conclusion immediately as long as

$$\text{maxdelay} \geq \text{hold},$$

which is already assumed in Section 7.1.1, and

$$\text{maxdelay} + \text{minslice} \leq \text{clocksep},$$

which is true by axiom `tasks_timeslice_defn` in Section 6.2.2.

7.2.3 Proof of scheduler behaviour

Lemma 7.2 is the most challenging of the three major lemmas in the scheduler proof because it involves reasoning about the scheduler code itself. Ideally it should be possible to step through the code, one instruction at a time; the timing behaviour should then follow trivially from the basic principle of RISC architectures that all instructions take one machine cycle. In practice, however, we must account not only for the control structure of the program itself but also for pipeline stalls, delay slots, and situations involving interrupts and system resets where the overlapping execution of instructions is inevitably visible.

The major difficulty encountered is not the complexity or subtlety of proving code properties, but the sheer number of variables that must be tracked during the proof (CPU registers, CP0 status registers, data and instruction cache contents, *etc.*). Aware of this problem, we developed lemmas for the various assembler instructions used in the proof, to allow each instruction to be handled in a single proof step (Appendix B). These lemmas are more convenient (though less general) than the instruction axioms in the instruction level CPU specification [7], where the functional, timing and annul behaviour of instructions are specified independently.

The lemmas in Appendix B embody the many decisions that must be made in order to know that a particular instruction can be successfully executed in a single cycle. Information typically required includes the value of the program counter, whether the current instruction and its arguments are valid (*e.g.*, is each address argument in the correct range), whether an interrupt is pending, *etc.* Aspects

of the pipelined architecture become apparent as well. In order to check that a system reset has not occurred it is necessary to know not only that a reset does not occur at the current time, but also did not occur during the past three cycles and will not occur in the next cycle, because a reset will affect an instruction at any stage during its journey through the pipeline. Also, in order to know that the pipeline is not stalled it is necessary to know that the instruction two steps in the future from the one currently executing is in cache. This is because this “future” instruction is already being fetched at the same time as the “current” instruction is (nominally) executing and an instruction cache miss would cause the whole pipeline to stall. The lemmas in Appendix B allow much of this repetitive, detailed checking to be hidden.

Since the scheduler code has a very simple control structure we decided to undertake its proof on a case-by-case basis. There are two situations, where the branch is and is not taken, respectively. Since the destination of the branch is known statically it is a simple matter to step through the code for the two possibilities.

The proof itself starts with the hypothesis that for some T and I ,

```

at(T);take_interrupt(error_interrupt)
and at(T);task_inv
and at(T);sched_inv(I).

```

The first step is to model the effect of taking the interrupt. This is handled by the `interrupt_cycles` lemma in Appendix B which models the filling of the pipeline when an interrupt is taken and address `ihcode` is the interrupt vector. It defines the state of the processor three run cycles later and also notes that the caches are invariant, so `task_inv` still holds (see axiom `tasks_inv_locality` in Section 6.2.2). However, `sched_inv(I)` is no longer true because the act of taking the interrupt has disabled interrupts (see the definition of `sched_inv` in Section 6.1).

It is then possible to apply each of the instruction lemmas in turn, for the two possible instruction sequences, thus defining the known characteristics of the processor state at each step. During the course of this proof there are two distinct levels of activity being represented. At the application level we see application-related values and predicates (*e.g.*, `task_inv`, `task_starts(I)`, `tcode((I+1) mod nt)`) appearing in the scheduler code and theorems. At the lower level of the instruction semantics we see those application concepts being implemented and manipulated as values in the processor and CP0 registers. The first seven instructions, for instance, load and manipulate register values but make no changes to cache; effectively they have no impact on application level predicates such as `task_inv`, nor do they change the current task identifier stored at location `ihdata + currtask`.

The key steps in proving the desired functional behaviour of the scheduler are the update made to the current task identifier and the scheduling of the next task. After the pipeline fill delay, the first instruction of the scheduler code (Figure 3) loads `ihdata` into register `$k1`. The second instruction loads the address stored at location `ihdata + currtask` into register `$k1`; this is value “`I`” in the hypothesis of `sched_behaviour`. The third instruction loads the address stored at `ihdata + tcb_end` into `$k0`. The fourth instruction adds constant `tcb_size` to the value in register `$k1`, effectively incrementing the current task identifier. At this stage we have proved:

```

at(T+7);run^
and not at(T+7);user_mode
and not at(T+7);ints_enabled
and not at(T+7);data_bus_error^
and not at(T+7);in_lds^
and not at(T+7);in_bds^
and at(T+7);interruptable
and at(T+7);pc^          = ihcode + 16
and at(T+7);ireg^       = bne_instr(k0,k1,2)
and at(T+7);next_pc^    = ihcode + 20
and at(T+7);next_ireg^  = lui_instr(k0,ihdata_upper)
and at(T+7);reg(k0)^    = ihdata + (tcb + nt*tcb_size)
and at(T+7);reg(k1)^    = ihdata + (tcb + (I+1)*tcb_size)
and cp0_registers_invar(T+3, T+7)
and dcache_invar(T, T+7)
and icache_invar(T, T+7).

```

To allow for the modulo arithmetic, the new task pointer in register `k1` is next compared to the `tcb_end` pointer to see if we have gone “beyond” the set of task control blocks. If not, the code branches to location `incr`. In either case, however, the next instruction, which again loads the base pointer `ihdata_upper` into register `$k0`, is performed. When the branch is not taken, *i.e.*, in the situation where the current task *was* the last, this is followed by an instruction to overwrite the value in register `$k1` with the address of the first task, *i.e.*, `ihdata + tcb`. We prove this section of the code by performing a case analysis, using the condition $I = nt - 1$. When $I = nt - 1$ is true, the branch is not taken, so the `addiu` instruction loads the address of task control block 0 into register `$k1`. When $I = nt - 1$ is false, the branch skips around the `addiu` instruction, leaving the address of task control block $I + 1$ in register `$k1`. After combining both cases,

register $\$k1$ holds the address of task $(I + 1) \bmod nt$ as required and we have proved:

```

ex x   x : 9 upto 11
  and at(T+x);run ^
  and not at(T+x);user_mode
  and not at(T+x);ints_enabled
  and not at(T+x);data_bus_error^
  and not at(T+x);in_lds^
  and not at(T+x);in_bds^
  and at(T+x);interruptable
  and at(T+x);pc^          = ihcode + 28
  and at(T+x);ireg^       = sw_instr(k1,currtask,k0)
  and at(T+x);next_pc^    = ihcode + 32
  and at(T+x);next_ireg^  = lw_instr(k0,0,k1)
  and at(T+x);reg(k0)^    = ihdata
  and at(T+x);reg(k1)^    = ihdata + tcb
                          + ((I+1) mod nt)*tcb_size
  and cp0_registers_invar(T+3, T+x)
  and dcache_invar(T, T+x)
  and icache_invar(T, T+x) .

```

Throughout the above steps the cache locations have been unchanged. It thus remains to store the new `currtask` value and start the appropriate task executing. The eighth instruction stores the new value into cache, and the remaining four instructions start executing code from this location. At the end of the `rfe` instruction we have proved the following theorem.

```

theorem sched_code
===
first =< T
  and at(T);take_interrupt(error_interrupt)
  and at(T);task_inv
  and at(T);sched_inv(I)
  and is_abs(I)
  and hw
=> (ex x_3 x_3 : T + 14 upto T + 16
    and at(x_3);run ^
    and at(x_3);user_mode

```

```

and at(x_3);ints_enabled
and not at(x_3);data_bus_error^
and not at(x_3);in_lds^
and not at(x_3);in_bds^
and at(x_3);interruptable
and at(x_3);pc ^      = tcode((I + 1) mod nt)
and at(x_3);ireg ^    = at(x_3);ival(tcode((I+1) mod nt))
and at(x_3);next_pc ^ = tcode((I + 1) mod nt) + 4
and at(x_3);next_ireg ^ = at(x_3);ival(tcode((I+1) mod nt)+4)
and at(x_3);cp0_status ^ >> 6 = at(T);cp0_status ^ >> 6
and icache_invar(T, x_3)
and dcache_put(T, x_3,
    ihdata + currtask,
    ihdata + (tcb + ((I+1) mod nt) * tcb_size))).

```

From theorem `sched_code` we can establish the two functional requirements in the conclusion of `sched_behaviour`. The first four conjuncts of predicate

$$\text{sched_inv}((I+1) \text{ mod } nt)$$

follow directly from the conclusion of `sched_code` and the remaining conjuncts follow from the invariance of the instruction cache and the fact that all scheduler data cache locations other than `ihdata+currtask` are unchanged. Similarly

$$\text{task_starts}((I+1) \text{ mod } nt)$$

follows from the conclusion of `sched_code` plus the original hypotheses of `sched_behaviour` transformed to reflect the update to the `ihdata+currtask` location.

To complete the proof we must show that the execution times of the scheduler are within the bounds defined in Section 7.1.3 as

$$\text{tm}_2 : (T + \text{hold}) \text{ upto } (T + \text{maxdelay} - \text{maxstall} + 1) .$$

As shown above, our choice of execution environment of the scheduler code allowed us to prove that each instruction will take exactly one machine cycle, once the pipeline has been filled. Actions that could potentially delay the pipeline, such as cache misses, multiplication instructions, or resets have all been precluded and interrupts are disabled while the scheduler code is executing. This makes timing analysis straightforward. The code takes either 11 or 12 machine cycles, if the branch is taken or not taken, respectively. (Although there are two instructions between the `bne` instruction and the `incr` label, the first of these instructions is in the branch delay slot and is thus executed in either case.) Adding the three cycles required to fill the pipeline when the interrupt handler is called tells us

that the scheduler will take either 14 or 15 machine cycles to execute, which is reflected in the `x_3 : T+15 upto T+16` conjunct of the `sched_code` conclusion above.

We can now determine the final constraints imposed on the system variables by this particular scheduler. From the minimum proven execution time for the scheduler, and the constraint on *SchedMin* found in Section 7.1.2, we get

$$\text{hold} \leq 14 .$$

Otherwise the scheduler could start the next task when the interrupt signal was still high, in the case where the scheduler is not stalled and takes the minimum time. This is a constraint on the environment: the scheduler code in Figure 3 cannot be used in an environment whose clock does not adhere to this requirement.

Similarly, from the constraint on *SchedMax* found in Section 7.1.3, and the maximum possible execution time for the scheduler code, we get

$$15 \leq \text{maxdelay} - \text{maxstall} .$$

Without this 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 is a constraint on the user's task requirements: the task designer must select values for `maxdelay` and `maxstall` that obey this constraint if they intend using the scheduler in Figure 3.

7.3 Other lemmas

The following minor lemma was used during the inductive step (Section 7.1.3) and in the proof of lemma 7.3 above.

Lemma 7.4 *Determine when the next external interrupt will occur.*

```

theorem next_ext_interrupt
=== hw
  and N : nats
  and T : (first + N*clocksep + hold) upto
          (first + (N+1)*clocksep + 1)
=>
  at(T);next_inc(ext_interrupt);time = first + (N+1)*clocksep.

```

To prove lemma 7.4 we use axiom `clock_ints_only` (Section 6.2.1) to re-express the desired conclusion as

```

at(T);next_inc(int_asserted(clk_int));time = first + (N+1)*clocksep

```

(because there are no external interrupts other than from the clock). Then, by the definition of `clk_set` (Section 6.1), we get

```
at(T);next_inc(clk_set);time = first + (N+1)*clocksep.
```

The requirement is then a straightforward consequence of axiom `clock_behaviour` in Section 6.2.1. (Notice that the “+ 1” in the hypothesis above means that `clk_set` may be true *at* time T, according to `clock_behaviour`, but this possibility is covered by the use of the inclusive `next_inc` predicate in the conclusion.)

The following verbose, but self-evident, lemma, used in Section 7.2.1, tells us that if the scheduler data and instruction cache locations have the same values at time T2 as they did at time T1 then desirable (time-independent!) properties of the scheduler invariant are maintained.

Lemma 7.5 *Scheduler code and data invariant properties are maintained when scheduler cache locations are unchanged.*

```
theorem sched_maintenance
=== sched_unchanged(T1,T2)
  and (I : 0 upto nt)
  and is_abs(I)
  and at(T1);(sched_code_in_cache
    and sched_data_in_cache
    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))))
=>
at(T2);(sched_code_in_cache
  and sched_data_in_cache
  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))))).
```

8 Interactive verification

The proof outlined above was undertaken using an interactive proof tool, Ergo 4.0 [9]. Earlier versions of Ergo (then called *demo2.x*) had been used for similar real-time proofs [5] of a smaller scale, but this was the first major application of Ergo 4.0. To give some idea of scale, the real-time and MIPS theories used to support the proof contained over 300 axioms, 140 definitions and 170 proven lemmas and theorems. More significantly, to make such proofs practical, it was necessary to write many tactics and heuristics to automate common parts of proofs. Since this was the first large-scale application of Ergo 4.0 it was a learning experience in the use of Ergo as well as in the proof methods for real-time verification. The experience resulted in many changes and extensions to Ergo and to the environment used to develop tactics and heuristics.

The first attempt to prove the induction step (Section 7.1.3), for instance, was hindered by the discovery of a number of bugs in the theorem prover itself. Ultimately this proof required approximately 8 hours of user interaction! However this included time used to correct some of the bugs detected, and to add four new lemmas to the theory base to simplify manipulation of integer and time expressions. This initial proof consumed 13 minutes of CPU time and involved 112 rule applications.

Subsequently, the major lemmas were reorganised to push more of the timing constraints up into the top level proof. Furthermore, we decided to derive the weakest timing constraints for the scheduler code by extracting them from the proof (as described above) rather than postulating them in advance as had been done previously. These changes made the top level proof larger (200 rule applications), but the development of more powerful and efficient tactics meant that it actually took less CPU time (6.5 minutes).

To modularise the proof of the scheduler code (Section 7.2.3), and to reduce proof size and increase execution speed, the proof was split into several lemmas — one for each sequential segment of code.

At the time of writing (April 1994) the `sched_code` theorem (see Section 7.2.3), which expresses the functional and timing behaviour of all twelve instructions of the scheduler code, has been proved using Ergo. The proof that the conclusion of the `sched_code` theorem satisfies the `sched_inv` and `task_starts` invariants has not been checked using Ergo but is expected to be straightforward. To make the code proofs practical, we have developed tactics and heuristics that are able to discharge approximately 90% of the conditions of the instruction lemmas in Appendix B. In many cases, to reason through the next instruction it was sufficient to simply call the `next_instr/2` tactic, which guesses which instruction lemma to apply next (by looking at the current value of `ireg^`), instantiates that lemma appropriately, then discharges all its conditions, which typically takes a

couple of minutes. If the `next_instr/2` tactic fails to discharge a condition, it allows the user to discharge it interactively before continuing with the remaining conditions.

We spent a total of 3 man weeks proving the twelve instructions, but at least two of those weeks were devoted to developing the supporting tactics and heuristics, which will be useful for future code proofs. Our tactics and heuristics are already quite powerful, but we can see many possibilities for improving them further, to make code proofs even more automated. This demonstrates the feasibility of real-time assembler code proofs for small programs (dozens, or perhaps hundreds of instructions).

The proofs that have not been completely checked on the theorem prover at the time of writing are the proofs of lemmas 7.1 and 7.3, the minor lemmas in Section 7.3, and the lemmas in Appendix B, all of which are considered to be straightforward.

9 Implementation constraints

The system specified in Section 6 was parameterised by the number of tasks `nt`, and various timing constants such as `clocksep`, `maxdelay`, `maxstall` and `hold`, *etc.*

Some constraints on these parameters were uncovered during the proof process. For example, we originally did not have any upper bound on the number of tasks that could be scheduled. Rather than predetermining an arbitrary upper bound, we decided to see what (minimal) constraints would be derived during the proof process. During one of the proofs it became necessary to prove that all the task control blocks fitted into the data cache (which has a size of 2^{16} bytes). To satisfy this requirement, we added axiom `nt_max`.

```
axiom nt_max === tcb + nt*tcb_size : 0 upto dcache_size.
```

This constraint effectively limits the number of tasks to be less than 512. We could increase the maximum number of tasks by reducing `tcb_size`, *i.e.*, the amount of information saved for each task (currently equal to 128, but our reduced scheduler only uses the first 4 bytes of each task control block). This illustrates how deriving constraints during the proof process can result in more flexible designs, by imposing only constraints that are necessary for the correctness of the system.

The top level requirement given in Section 6.3 was independent of the actual code chosen to implement the scheduler. However, after implementing a specific scheduling algorithm and deriving tight timing bounds for it, we can derive more specific constraints on the timing parameters of the system. With the 12 instruction scheduler in Figure 3, the following timing constraints were found to be sufficient to ensure that an implementation exists.

The interaction between tasks and the environment showed that

$$\text{maxstall} < \text{hold}$$

must be true so that stalls induced by the tasks do not cause clock interrupts to be missed. Similarly, it was found that

$$\text{hold} \leq \text{maxdelay}$$

again due to the interaction between the environment and the user's task requirements. If this constraint does not hold there is no time at which the scheduler can start the next task without its being immediately interrupted.

Once the absolute execution times for the scheduler code were established it was also possible to further constrain some system variables. Firstly,

$$\text{hold} < 14$$

specifies a maximum limit for the duration of clock interrupts if they are not to interfere with the next task. Secondly,

$$\text{maxstall} + 15 \leq \text{maxdelay}$$

adds the maximum scheduler execution time to the worst case task stall and states that the maximum permissible starting delay must be at least as great as this. These two requirements can be thought of as constraints imposed on the *environment* if we are to use the particular scheduler illustrated in Figure 3.

10 Conclusion

We have formally specified, and given a proof strategy for, a simple real-time scheduler. Furthermore, the proof was undertaken with the aid of an interactive theorem prover, Ergo, which has checked most parts of the proof, including we believe all the less straightforward aspects.

This work is believed to be unique in its use of a pipelined RISC processor as the target architecture. This introduced complexity via

1. the need to account for the time required to fill the pipeline,
2. the possibility of pipeline stalls,
3. the large number of variables that must be kept up to date during proof of the assembler code,
4. the seemingly non-linear execution of some assembler statements due to "delay slots", and

5. other effects of overlapped instruction execution that manifest themselves at the assembler level, especially during interrupt handling.

These challenges were resolved by

1. devising a theorem that defined the effects of an interrupt over the next three run cycles,
2. explicitly modelling the maximum amount of time for which tasks are permitted to stall the pipeline,
3. developing specialised lemmas that enabled the execution of each instruction to be modelled in a single step,
4. making use of features in the instruction level model [7] that allow us to refer to past and future machine cycles, and
5. making use of features of the instruction level model that hide the need to look at instructions other than the one (nominally!) executing at the current moment.

This exercise also gave us considerable experience with the Ergo theorem prover, further populated its theory base and developed skills in the use of the Ergo system for a non-trivial application. It also led to a number of changes and improvements to Ergo itself, notably the introduction of *heuristic environments* [9]. Our experience shows that it is feasible to develop sophisticated tactics and heuristics that automate a large proportion of the tedious condition discharging within proofs about MIPS assembler code.

Acknowledgement

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

References

- [1] Accelerated Technology, ‘Nucleus RTX Technical Notes’, 1992.
- [2] Burns, A., ‘Scheduling Hard Real-time Systems: a Review’, *Software Engineering Journal*, 6(3), May 1991.
- [3] Furht, B., *et al.*, ‘Real-time Unix Systems: Design and Application Guide’, Kluwer Academic Publishers, 1991.

- [4] JMI Software Consultants ‘C Executive Technical Notes’, Release 2.4, April 1, 1992.
- [5] Kearney, P., Staples, J. and Abbas, A., ‘Functional Verification of Hard Real-Time Programs’, in Algorithms, Software, Architecture, Information Processing 92, Volume 1, pp113–119, Elsevier Science Publishers B.V. (North-Holland), 1992.
- [6] Utting, M., and Kearney, P., ‘Pipeline Specification of a MIPS R3000 CPU’, SVRC technical report **92-6**, October 1992.
- [7] Utting, M., and Kearney, P., ‘Instruction Level Specification of a MIPS R3000 CPU’, SVRC technical report **93-25**, February 1994.
- [8] Utting, M., and Kearney, P., ‘Specification Issues for Real-Time Behaviour of RISC Processors’, Proc. Australasian Workshop on Parallel and Real-Time Systems, Melbourne, 7–8 July 1994 (to appear).
- [9] Utting, M., and Whitwell, K., ‘Ergo User Manual’, SVRC technical report **93-19**, February 1994.

A Extended clock interrupt handler

This appendix shows an extended version of the clock interrupt handler from Section 5. This extended handler checks for the cause of interrupts and manages the saving of registers (thus removing two of the restrictions from Section 3.1 and 3.2, respectively). The most obvious additions are the repetitive storage and retrieval of register contents. Application of the same verification techniques to these parts is expected to be straightforward. The difference is one of scale, rather than kind. We estimate that, although this scheduler code is over six times longer than the code in Section 5, because of its repetitive nature this scheduler would require less than double the amount of proof effort that was required for the scheduler in Section 5.

```
# Check cause of interrupt.

mfc0 $k0, $13           # Load $k0 from cause register.
ori $k1, $0, 0x0400    # Load 0x0400 into $k1. (Bit 10,
                        #   i.e., 0x0400 means that
                        #   cp0_cause_code = 0 and that the
                        #   clock is the only interrupt).
andi $k0, $k0, 0xffff  # Isolate lower 16 bits of $k0.
bne $k0, $k1, err      # Error if not the clock interrupt.
```

```

# Get pointers to current task control block and tcb_end.

    lui $k0, ihdata_upper    # Load base pointer to scheduler
                                # data into register $k0.
    lw $k1, currtask($k0)    # Load $k1 with pointer to current
                                # task control block.
    lw $k0, tcb_end($k0)    # Load $k0 with tcb_end pointer.

# Store registers in current task control block.

    sw $1, 4($k1)
    sw $2, 8($k1)
    sw $3, 12($k1)
    sw $4, 16($k1)
    sw $5, 20($k1)
    sw $6, 24($k1)
    sw $7, 28($k1)
    sw $8, 32($k1)
    sw $9, 36($k1)
    sw $10, 40($k1)
    sw $11, 44($k1)
    sw $12, 48($k1)
    sw $13, 52($k1)
    sw $14, 56($k1)
    sw $15, 60($k1)
    sw $16, 64($k1)
    sw $17, 68($k1)
    sw $18, 72($k1)
    sw $19, 76($k1)
    sw $20, 80($k1)
    sw $21, 84($k1)
    sw $22, 88($k1)
    sw $23, 92($k1)
    sw $24, 96($k1)
    sw $25, 100($k1)
    # NB. registers $26 ($k0) and $27 ($k1) are reserved for
    # the operating system kernel and are not saved.
    sw $28, 104($k1)
    sw $29, 108($k1)
    sw $30, 112($k1)
    sw $31, 116($k1)

# Increment the currtask pointer to the next task.

```

```

    addiu $k1, $k1, tcb_size # Increment task base address
                                # (in $k1) by adding tcb_size.
    bne $k0, $k1, incr        # Branch around the next change
                                # to $k1 if $k1 is a legal tcb
                                # pointer, i.e., if $k1 does not
                                # equal tcb_end.
    lui $k0, ihdata_upper    # Load base pointer into $k0.
                                # NB. always executed (branch
                                # delay slot).
    addiu $k1, $k0, tcb      # Put address of the first task,
                                # tcb 0, i.e., ihdata + tcb,
                                # into $k1.
incr: sw $k1, currtask($k0) # Store new value of current task
                                # pointer from $k1 into location
                                # ihdata + currtask.

# Load registers of new task.

    lw $1, 4($k1)
    lw $2, 8($k1)
    lw $3, 12($k1)
    lw $4, 16($k1)
    lw $5, 20($k1)
    lw $6, 24($k1)
    lw $7, 28($k1)
    lw $8, 32($k1)
    lw $9, 36($k1)
    lw $10, 40($k1)
    lw $11, 44($k1)
    lw $12, 48($k1)
    lw $13, 52($k1)
    lw $14, 56($k1)
    lw $15, 60($k1)
    lw $16, 64($k1)
    lw $17, 68($k1)
    lw $18, 72($k1)
    lw $19, 76($k1)
    lw $20, 80($k1)
    lw $21, 84($k1)
    lw $22, 88($k1)
    lw $23, 92($k1)
    lw $24, 96($k1)
    lw $25, 100($k1)

```

```

        lw $28, 104($k1)
        lw $29, 108($k1)
        lw $30, 112($k1)
        lw $31, 116($k1)

# Start new task.

        lw $k0, 0($k1)           # Load start pc for current task
                                   #   from $k1 into $k0.
        nop                       # Do nothing (load delay slot).
        jr $k0                   # Jump to start pc for current
                                   #   task.
        rfe                       # Restore status reg settings.

# Error code, called when an unexpected interrupt arrives.

err: ...

```

B Instruction Lemmas

This appendix shows the lemmas used to ease the proof of the scheduler code (Section 7.2.3). These lemmas are labelled as being *postulates*, which means that their proofs have not yet been checked by the theorem prover. Although not formally proven, these lemmas were all constructed directly from known instruction level properties of the MIPS R3000 architecture, as described in the instruction level model [7], so they are believed to be straightforwardly derivable from that model.

The following lemma, for instance, defines the behaviour of the “load word” instruction.

```

postulate lw_instr_1
=== at(T);ireg^ = lw_instr(Reg,Offset,Base)
    and is_abs(Reg)
    and is_abs(Base)
    and is_abs(Offset)
    and at(T);irun
    and is_word_addr(at(T);pc^)
    and (at(T);user_mode => at(T);pc^ < 2**31)
    and not at(T);data_bus_error^
    and not at(T);int_pending
    and at(T);interruptable
    and (at(T);in_lds^ => Base \= at(T);load_reg^)
    and at(T);in_icache(twos_comp(next_pc^ + 4))

```

```

and group(at(T);ival(twos_comp(next_pc^ + 4))) \= hilo_instr
and hw
=> (letabs x twos_comp(at(T);reg(Base)^ + int_val_n(Offset,16))
    within
    is_word_addr(x)
    and (at(T);user_mode => x < 2**31)
    and at(T);in_dcache(x)
=>      at(T+1);run^
        and at(T+1);pc^          = at(T);next_pc^
        and at(T+1);ireg^        = at(T);next_ireg^
        and at(T+1);next_pc^     = at(T);twos_comp(next_pc^ + 4)
        and at(T+1);next_ireg^
            = at(T);ival(twos_comp(next_pc^ + 4))
        and at(T+1);interruptable
        and at(T);reg_put(Reg, dval(x))
        and not at(T+1);in_bds^
        and at(T+1);in_lds^
        and at(T+1);load_reg^ = Reg
        and not at(T+1);data_bus_error^
        and icache_invar(T,T+1)
        and dcache_invar(T,T+1)
        and cp0_registers_invar(T,T+1)).

```

The hypotheses assert that the current cycle is a run cycle (via `irun` [7]), that the current program counter value is a legal address, that the instruction register does indeed hold a “lw” instruction, and that there are no interrupts pending.

Some influences of the pipelined architecture can also be seen in the last three conjuncts of this hypothesis. Firstly, it is necessary to check that if we are currently in a load delay slot then *this* instruction will not try to use the register still being updated by the *previous* load instruction. Secondly, we must check that the instruction *after the next* instruction is currently in cache. This seemingly premature requirement is essential because while the current instruction is (nominally) executing, *i.e.*, is in the “MEM” stage of the pipeline, the instruction two steps ahead is in the “RD” stage and is thus being fetched from instruction cache. However, if that instruction is not currently in cache then the entire pipeline is stalled until a cache swap is done. Finally, it is also necessary to ensure that the instruction two steps ahead is not a “HI” or “LO” instruction, *i.e.*, one that depends on the result of a multiplication or division, because this can similarly stall the pipeline, awaiting the result.

The conclusion then defines the effect of such an instruction, assuming that the address from which the word is to be loaded is legal. The program counter variables are updated, the fact that we are now in a load delay slot recorded and the particular register of interest is updated.

Most of the instruction lemmas follow this general form. The “store word” lemma below, for instance, is almost identical except that the conclusion records the update to the appropriate cache location.

```

postulate sw_instr_1
=== at(T);ireg^ = sw_instr(Reg,Offset,Base)
   and is_abs(Reg)
   and is_abs(Base)
   and is_abs(Offset)
   and at(T);irun
   and is_word_addr(at(T);pc^ )
   and (at(T);user_mode => at(T);pc^ < 2**31)
   and not at(T);data_bus_error^
   and not at(T);int_pending
   and at(T);interruptable
   and (at(T);in_lds^ => Base \= at(T);load_reg^ )
   and at(T);in_icache(twos_comp(next_pc^ + 4))
   and group(at(T);ival(twos_comp(next_pc^ + 4))) \= hilo_instr
   and hw
=> (letabs x twos_comp(at(T);reg(Base)^ + int_val_n(Offset,16))
    within
    is_word_addr(x)
    and (at(T);user_mode => x < 2**31)
    and at(T);cacheable(x)
=>      at(T+1);run^
        and at(T+1);pc^          = at(T);next_pc^
        and at(T+1);ireg^        = at(T);next_ireg^
        and at(T+1);next_pc^     = at(T);twos_comp(next_pc^ + 4)
        and at(T+1);next_ireg^
          = at(T);ival(twos_comp(next_pc^ + 4))
        and at(T+1);interruptable
        and reg_invar(T,T+1)
        and not at(T+1);in_bds^
        and not at(T+1);in_lds^
        and icache_invar(T,T+1)
        and at(T);dcache_put(x,reg(Reg)^ )
        and not at(T+1);data_bus_error^
        and cp0_registers_invar(T,T+1)).

```

A significant exception are those instructions that define more than one possible behaviour. The “branch not equal” instruction in the scheduler code for instance could be treated in two different ways. A single lemma could be defined for this instruction with a disjunctive conclusion representing the possibilities that the branch is, or is not, taken. However, given that we had already decided to

undertake proof of the scheduler code on a “case-wise” basis it was felt to be more efficient to define two separate lemmas for the two different possibilities. Below is the situation where the branch is not taken. The hypothesis includes a conjunct that asserts that the values in `RegS` and `RegT` are equal and the conclusion therefore merely increments the program counter variables in the usual way. (Again pipeline influences can be seen in the requirements that neither of the two registers being compared are affected by ongoing load operations, *etc.*)

```

postulate bne_instr_1_no
=== at(T);ireg^ = bne_instr(RegS,RegT,Offset)
   and is_abs(RegS)
   and is_abs(RegT)
   and is_abs(Offset)
   and at(T);irun
   and is_word_addr(at(T);pc^ )
   and (at(T);user_mode => at(T);pc^ < 2**31)
   and not at(T);data_bus_error^
   and not at(T);int_pending
   and not at(T);in_bds^
   and (at(T);in_lds^ => RegS \= at(T);load_reg^
        and RegT \= at(T);load_reg^ )
   and at(T);interruptable
   and at(T);reg(RegS)^ = at(T);reg(RegT)^
   and at(T);in_icache(twos_comp(next_pc^ + 4))
   and group(at(T);ival(twos_comp(next_pc^ + 4))) \= hilo_instr
   and hw
=>   at(T+1);run^
     and at(T+1);pc^      = at(T);next_pc^
     and at(T+1);ireg^    = at(T);next_ireg^
     and at(T+1);next_pc^ = at(T);twos_comp(next_pc^ + 4)
     and at(T+1);next_ireg^ = at(T);ival(twos_comp(next_pc^ + 4))
     and at(T+1);interruptable
     and reg_invar(T,T+1)
     and at(T+1);in_bds^
     and not at(T+1);in_lds^
     and not at(T+1);data_bus_error^
     and icache_invar(T,T+1)
     and dcache_invar(T,T+1)
     and cp0_registers_invar(T,T+1).

```

In the case where the branch is taken the hypothesis states that the registers are not equal and the conclusion asserts that `next_pc` is given the value indicated by the offset in the `bne` instruction. Observe that the instruction after the `bne` is still executed, however, because it is in a branch delay slot.

```

postulate bne_instr_1_yes
=== at(T);ireg^ = bne_instr(RegS,RegT,Offset)
    and is_abs(RegS)
    and is_abs(RegT)
    and is_abs(Offset)
    and at(T);irun
    and is_word_addr(at(T);pc^ )
    and (at(T);user_mode => at(T);pc^ < 2**31)
    and not at(T);data_bus_error^
    and not at(T);int_pending
    and not at(T);in_bds^
    and (at(T);in_lds^ => RegS \= at(T);load_reg^
        and RegT \= at(T);load_reg^ )
    and at(T);interruptable
    and at(T);reg(RegS)^ \= at(T);reg(RegT)^
    and hw
=> (letabs x twos_comp(at(T);next_pc^ + int_val_n(Offset,16)*4)
    within
    at(T);in_icache(x)
    and group(at(T);ival(x)) \= hilo_instr
=>      at(T+1);run^
        and at(T+1);pc^      = at(T);next_pc^
        and at(T+1);ireg^    = at(T);next_ireg^
        and at(T+1);next_pc^ = x
        and at(T+1);next_ireg^ = at(T);ival(x)
        and at(T+1);interruptable
        and reg_invar(T,T+1)
        and at(T+1);in_bds^
        and not at(T+1);in_lds^
        and not at(T+1);data_bus_error^
        and icache_invar(T,T+1)
        and dcache_invar(T,T+1)
        and cp0_registers_invar(T,T+1)).

```

The “jump register” instruction always causes a branch, so it is similar to the branch-taken case of the bne instruction above.

```

postulate jr_instr_1
=== at(T);ireg^ = jr_instr(Reg)
    and is_abs(Reg)
    and at(T);irun
    and is_word_addr(at(T);pc^ )
    and (at(T);user_mode => at(T);pc^ < 2**31)
    and not at(T);data_bus_error^

```

```

and not at(T);int_pending
and (at(T);in_lds^ => Reg \= at(T);load_reg^ )
and not at(T);in_bds^
and at(T);interruptable
and at(T);in_icache(reg(Reg)^ )
and group(at(T);ival(reg(Reg)^ )) \= hilo_instr
and hw
=>   at(T+1);run^
      and at(T+1);pc^       = at(T);next_pc^
      and at(T+1);ireg^     = at(T);next_ireg^
      and at(T+1);next_pc^  = at(T);reg(Reg)^
      and at(T+1);next_ireg^ = at(T);ival(reg(Reg)^ )
      and at(T+1);interruptable
      and reg_invar(T,T+1)
      and at(T+1);in_bds^
      and not at(T+1);in_lds^
      and not at(T+1);data_bus_error^
      and icache_invar(T,T+1)
      and dcache_invar(T,T+1)
      and cp0_registers_invar(T,T+1).

```

The `rfe` instruction is significant because it sets the interrupt conditions back to the values they had when the processor was last interrupted (*c.f.*, `interrupt_cycles` below).

```

postulate rfe_instr_1
=== at(T);ireg^ = rfe_instr
      and at(T);irun
      and is_word_addr(at(T);pc^ )
      and (at(T);user_mode => at(T);pc^ < 2**31)
      and not at(T);data_bus_error^
      and not at(T);int_pending
      and (at(T);user_mode => at(T);cp0_usable)
      and at(T);in_bds^
      and at(T);interruptable
      and at(T);in_icache(twos_comp(next_pc^ + 4))
      and group(at(T);ival(twos_comp(next_pc^ + 4))) \= hilo_instr
      and hw
=>   at(T+1);run^
      and at(T+1);pc^       = at(T);next_pc^
      and at(T+1);ireg^     = at(T);next_ireg^
      and at(T+1);next_pc^  = at(T);twos_comp(next_pc^ + 4)
      and at(T+1);next_ireg^ = at(T);ival(twos_comp(next_pc^ + 4))
      and at(T+1);interruptable

```

```

and reg_invar(T,T+1)
and not at(T+1);in_bds^
and not at(T+1);in_lds^
and not at(T+1);data_bus_error^
and icache_invar(T,T+1)
and dcache_invar(T,T+1)
and at(T+1);(cp0_status^ >> 6) = at(T);(cp0_status^ >> 6)
and at(T+1);ints_enabled      = at(T);ints_enabled_prev
and at(T+1);ints_enabled_prev = at(T);ints_enabled_old
and at(T+1);ints_enabled_old  = at(T);ints_enabled_old
and at(T+1);user_mode        = at(T);user_mode_prev
and at(T+1);user_mode_prev   = at(T);user_mode_old
and at(T+1);user_mode_old    = at(T);user_mode_old
and at(T+1);cp0_cause_code   = at(T);cp0_cause_code
and at(T+1);cp0_epc^        = at(T);cp0_epc^ .

```

Finally, for completeness, we give the lemmas for the other two instructions used in the scheduler code; the “add immediate unsigned” and “no operation” instructions.

```

postulate addiu_instr_1
=== at(T);ireg^ = addiu_instr(RegT, RegS, Val)
   and is_abs(RegS)
   and is_abs(RegT)
   and is_abs(Val)
   and at(T);irun
   and is_word_addr(at(T);pc^)
   and (at(T);user_mode => at(T);pc^ < 2**31)
   and not at(T);data_bus_error^
   and not at(T);int_pending
   and (at(T);in_lds^ => RegS \= at(T);load_reg^
        and RegT \= at(T);load_reg^)
   and at(T);interruptable
   and at(T);in_icache(twos_comp(next_pc^ + 4))
   and group(at(T);ival(twos_comp(next_pc^ + 4))) \= hilo_instr
   and hw
=>   at(T+1);run^
     and at(T+1);pc^      = at(T);next_pc^
     and at(T+1);ireg^    = at(T);next_ireg^
     and at(T+1);next_pc^ = at(T);twos_comp(next_pc^ + 4)
     and at(T+1);next_ireg^ = at(T);ival(twos_comp(next_pc^ + 4))
     and at(T+1);interruptable
     and at(T);reg_put(RegT,
                       twos_comp(reg(RegS)^ + int_val_n(Val,16)))

```

```

    and not at(T+1);in_bds^
    and not at(T+1);in_lds^
    and not at(T+1);data_bus_error^
    and icache_invar(T,T+1)
    and dcache_invar(T,T+1)
    and cp0_registers_invar(T,T+1).

postulate nop_instr_1
=== at(T);ireg^ = nop_instr
    and at(T);irun
    and is_word_addr(at(T);pc^ )
    and (at(T);user_mode => at(T);pc^ < 2**31)
    and not at(T);data_bus_error^
    and not at(T);int_pending
    and (at(T);in_lds^ => 0 \= at(T);load_reg^ )
    and at(T);interruptable
    and at(T);in_icache(twos_comp(next_pc^ + 4))
    and group(at(T);ival(twos_comp(next_pc^ + 4))) \= hilo_instr
    and hw
=>    at(T+1);run^
        and at(T+1);pc^      = at(T);next_pc^
        and at(T+1);ireg^    = at(T);next_ireg^
        and at(T+1);next_pc^ = at(T);twos_comp(next_pc^ + 4)
        and at(T+1);next_ireg^ = at(T);ival(twos_comp(next_pc^ + 4))
        and at(T+1);interruptable
        and reg_invar(T,T+1)
        and not at(T+1);in_bds^
        and not at(T+1);in_lds^
        and not at(T+1);data_bus_error^
        and icache_invar(T,T+1)
        and dcache_invar(T,T+1)
        and cp0_registers_invar(T,T+1).

```

In all of the above instructions the timing behaviour is simple; T is incremented by one for each instruction executed. A special case occurs at the very beginning of the scheduler execution, however, because the pipeline must be filled. The following lemma hides this by defining the behaviour of the processor from the time an interrupt is taken, until the first instruction of the interrupt handler code reaches the MEM stage of the pipeline, three cycles later. Like the lemmas above it is closely based on the instruction level model, but, for brevity, we have used the `ihcode` constant from the scheduler model in its formulation. Unlike the above lemmas, therefore, the `interrupt_cycles` lemma is specific to the scheduler model, rather than being provable within the `ispec` (instruction level specification) theory [7].

```

postulate interrupt_cycles
=== hw
  and first =< T
  and at(T);take_interrupt(Cause)
  and at(T);in_icache(ihcode)
  and at(T);in_icache(ihcode+4)
  and group(at(T);ival(ihcode)) \= hilo_instr
  and group(at(T);ival(ihcode+4)) \= hilo_instr
=> at(T+3);irun
  and at(T+3);interruptable
  and at(T+3);pc^ = ihcode
  and at(T+3);ireg^ = at(T);ival(ihcode)
  and at(T+3);next_pc^ = ihcode+4
  and at(T+3);next_ireg^ = at(T);ival(ihcode+4)
  and not at(T+3);in_bds^
  and not at(T+3);in_lds^
  and not at(T+3);ints_enabled
  and not at(T+3);data_bus_error^
  and at(T+3);kernel_mode
  and at(T+3);cp0_status^ >> 6 = at(T);cp0_status^ >> 6
  and at(T+3);ints_enabled_prev = at(T);ints_enabled
  and at(T+3);user_mode_prev = at(T);user_mode
  and at(T+3);cp0_cause_code = at(T);Cause
  and reg_invar(T, T+3)
  and icache_invar(T, T+3)
  and dcache_invar(T, T+3).

```