Evaluation of Delay Queues for a Ravenscar Hardware Kernel

Gustaf Naeser
Dept. of Computer Science and Engineering
Mälardalen University
Sweden
gustaf.naeser@mdh.se

Johan Furunäs
Dept. of Computer Science and Engineering
Mälardalen University
Sweden
johan.furunas@mdh.se

Abstract

In this paper we present and evaluate four delay queues designed for application tailored Ravenscar hardware real-time kernels. The properties of the different queues and optimisations of them are discussed and both formal models and actual hardware implementation of the queues are presented. A transformation from timed automata to VHDL is described during the translation of the timed automata of the formal model into the corresponding VHDL state machines. Our study of the queues shows that even though parallelism costs much in terms of chip area, there are system configurations where it is the most space conservative. We also show that the queues meet the timing requirements of Ravenscar and that they can be fitted onto an FPGA.

1 Introduction

Establishing the temporal behaviour of a real-time system remains a complicated challenge since it depends not only on the application but also on the performance of real-time kernel (RTK). The RTK performance depends on the scheduling the kernel describes, the algorithms used, and the hardware used to implement the RTK. Knowledge of the kernel's performance is required for more accurate calculations of the full system's (application's and RTK's) temporal behaviour. A tasking profile defines the behaviour the system, and hence the kernel, should exhibit.

The Ravenscar profile [3] defines a deterministic profile for Ada tasking to be used in high-integrity real-time systems [4]. The profile restricts the use of programming constructs that complicates the temporal behaviour of a system. Still, the need for analysis of the kernel times and identification of them in a Ravenscar kernel remains, as presented in [17, 18]. Analysis of a RTK implemented in software involves calculating the timing not only of the kernel but also of the hardware the kernel is running on. We have chosen to study an RTK implemented in hardware partly to avoid the double analysis required by a software RTK and partly since we want to create a platform where we easier can move parts of the system between hardware and software. An important property of the system parts placed in hardware will be their area requirements as small area usage allows more software or processing components to be placed in hardware. Placing kernel functionality in hardware can improve the execution speed of the kernel [9]. Some problems that exist in software kernels can be avoided in hardware kernels. For example, as clock functionality can be integrated in the kernel rather than used as an external component, the clock inaccuracy problem can be eliminated. The times used for kernel operation in a hardware kernel can by analysis of the synthesised hardware be decided on kernel clock cycle level, i.e. with extremely high precision. Execution time analysis often focus only on the task level sequential code where most time is spent. Kernel times, accounting for a fraction of the time spent in a system, is often neglected and this analysis is enough for most real-time applications. However, timing analysis without consideration of the kernel is not sufficient for safety critical systems where a higher level of detail is required to secure correct operation.

Formal verification is used to ensure that the behaviour of a model of a system conforms to its specification. Using formal methods for verifying parts of kernel functionality during its design can help eliminating design flaws and give better knowledge of system operation [2, 5, 16]. A formal model of a Ravenscar kernel (Lundqvist and Asplund's Model of Ravenscar, LAMR) was been presented in [7] and an implementation of it was presented in [15]. The kernel model we use in this paper is a refined version of LAMR with a clearer component design and extended functionality, e.g., support for multiple processors.

The Open Ravenscar Run Time Kernel(ORK) [13, 14] also implements the Ravenscar profile. Dynamic validation by software faults injection of ORK is described in [8] where verification of an implemented kernel is attempted. This approach does not suit the kernel described in this paper since the kernel we describe is specialised for the final system's actual characteristics. For example, the delay queue can be specialised for the actual task setup when the number of delaying tasks is known or easily can be decided using code inspection. This kind of optimisations will help not only to reduce the size both of the final hardware implementation but also the size of the states saved during verification and allow for larger systems to be verified.

The design of the queues was first made in UPPAAL [6], in timed automata, where queue operation could be verified with other kernel components and a models of high-integrity applications. The queues, and the rest of the kernel, are part of a project aiming for analysis of the temporal properties of safety critical systems that are implemented in both hardware and software. Though there are some tools for hardware analysation [11, 12], the ability to analyse the temporal behaviour of the full system, where not the whole system resides in hardware, made us design the queues in the UPPAAL tool. The transformation of the timed automata to VHDL and implementation metrics of the FPGA implementations are discussed later in the paper.

2 A Ravenscar Delay Queue

The real-time kernel in a real-time system manages the system resources, like processor allocation and the access to shared objects. The different tasks of the kernel can be implemented in separate components, like the ready queue, the delay queue, the protected objects handler and the interrupt handler. This separation makes it easier to modify the design and implementation of each individual part of the kernel to meet a system's specific demands and requirements. The desired properties of the components are the same as those for a software kernel's components, i.e. high speed and small size. The timing properties of the different kernel components is also important since they will impact on the level of possible parallelism, a slower component can become a bottleneck if interacting components operate faster.

The interface of the delay queue is shown in Table 1. The basic operation of the delay queue is as follows:

1. When a task is delayed a preliminary quick check to decide if the task will be suspended is done. If the delay-time is right now or in the past, the task should not be suspended and this is signalled with an unblock. On receiving an unblock the ready-queue will move the task to the last position among tasks that have the same priority.

2. If the delay-time is in the future the task should be suspended and a suspend is signalled. Suspension makes the ready-queue remove the task from the running tasks and preempt it from the processor where it is running.

3. When the release time of a task is reached, the ready-queue is signalled to make that task runnable again.

Table 1. Delay queue interface description. The input interface contains the signals that the delay queue will react to and the output interface contains those signals used to update task states in the ready-queue.

(a) Input signals consumed by the delay queue.

delay(T_{id}, $time$)	Delay task T with identity id, T_{id}, until $time$ is reached.
tick	Signaled when the system clock increase.

(b) Output signals produced by the delay queue.

suspend(T_{id})	Remove T_{id} from ready-queue.
unblock(T_{id})	Put T_{id} last within its priority.
runnable(T_{id})	Signal that T_{id} ready to run.

The resources the delay queue uses to store information about the delayed tasks and the way in which it monitors the releases are varied in the different queues models describe below. The variation explores different runtime characteristics of the described interface.

3 Models

The UPPAAL tool contains an editor, simulation tool and verification tool for timed automata. The timed automata consists of labelled transition systems with time and can contain clocks, boolean variables, integer variables and synchronous channels. Shared variables and synchronous channels can be used to communicate data and control between automata. The value of variables are initiated to nothing, zero, false, etc. Each automata consists of an initial location, indicated by an inner circle, a fixed number of locations and transitions between the locations. In the explanation of the queues below the notation $n_1 \longrightarrow n_2$ represents a transition from location n_1 to location n_2. Transitions contain guards, synchronisation and assignments. An automata can take a transition from a state if the guard on the transition is satisfied, any synchronisation on the transition is possible. When a transition is taken the assignment part of the transition is executed. During a synchronous step, where two automata communicate over a channel, the assignments of the sending task are made before those of the receiving task. A transition can at most synchronise one channel. An exclamation mark after the channel name is used to indicate that the channel is used for sending and a question mark is used to indicate reception. A late addition to UPPAAL was the introduction of broadcast channels where one sender synchronise with multiple receivers. Locations can be marked as committed or urgent, Ⓒ respective Ⓤ, to force specific temporal behaviour. Committed locations are used to create atomic chains of transitions. and an automata in a committed location must leave the location before any other transition (that is not committed) may be taken in the system. Committed locations can be used to synchronise over multiple channels in a chain of transitions. Urgent location indicate that outgoing transitions from the location have precedence over time transitions. Time transitions can be taken whenever there are no automata in committed or urgent locations that can take transitions. (Timed transitions are not used in the models presented here.) Failure is reported during verification or simulation if an automaton cannot leave a committed location.

The verification part of UPPAAL is used to explore user defined properties in all possible executions of a given system. If a property cannot be verified (proven correct) a counter proof can be explored in the simulator part of the tool. Keeping the size of the kernel components down allows larger systems to be verified. There are parameters that can be changed to optimise the implementation size of the delay queue, *a*) the size of the stored delay times, *b*) how the delays are stored, and *c*) the amount of parallelism used. The behaviour of the queue, i.e. if and when the queue will cause unwanted stalling of the RTK depends on if work is done when delaying or when releasing, and again the amount of parallelism used. Some of the parameters depend on each other and some combinations make no sense, e.g. sorted arrays with the work at release time. The models we present below explore different combinations of the parameters.

The delay times, can be stored and used as *absolute times* or as *delta times*. An absolute time, T, is the actual release time of the task and, as discussed in [19], requires a minimum of 41 bits to represent the 50 years at 1 ms resolution required by the Ada Reference Manual [1]. However, the number of bits required can be reduced in a system of periodic tasks where the cycle times of all delaying tasks are known. A delta time,Δ_T, represents the number of ticks remaining until the release of a task and can be used with countdown timers to delay tasks. A safe estimation of the number of bits needed for the delta times is the number of bits needed to represent the cycle time of the task with the longest period.

The array (or queue) where information about the delayed tasks is stored can be managed in two ways, either as a sorted queue ordered by the closeness of the tasks' release or as an array indexed by the task identities. The two forms of storage prompts the work of the queue to either be more when delaying, the queue, or when releasing, the indexed array. A delay queue using a sorted list will have to re-sort the queue of delayed tasks when a task is delayed whereas an indexed queue will have to find the next task to release whenever a task is released. The sorted queue can at the time a task is delayed be made to respect the order in which the tasks are released and implement, e.g., FIFO or a priority release policy. An indexed array cannot keep this kind of information and will hence release the tasks in some kind of identifier indexed order. However, a priority based release can be achieved by ordering the task identities in priority order. Note that the first position of the arrays is not needed since the task id zero is reserved for null processes (which will not delay).

The amount of parallelism that can be imposed on the kernel will reduce the time kernel components can be blocked by each other but introduce the possibility of communication delays. However, parallelism must be used carefully since it increases the amount of chip area the hardware implementation will use.

To ensure the correct operation of the different delay queues, their behaviour were formally verified. The verification used models of the other kernel components and sample application systems. After having passed the formal verification the designs were transformed into VHDL and finally synthesised for a FPGA. The initial model, $\mathcal{Q}_1$ below, was designed for verification with no thought of hardware synthesis. It was used for experimental verification of application properties, it was suspected to use too much hardware resource to be useful in real systems. The properties governing the queues resource usage were distilled and explored in reasonable combinations as shown in this paper.

3.1 Delay queue $\mathcal{Q}_1$

The first delay queue design, shown in Fig. 1, uses an indexed array of absolute release times. The computation needed to delay a task is minimal, $\mathrm{n}_0 \longrightarrow \mathrm{n}_1 \longrightarrow \mathrm{n}_0$, the queue writes the release time in the position corresponding to the task in the array DQd. The queue records the index of the task with the closest release time, in the variable next. If there are several releases at the same time the one with the lowest identity is stored. When the release time of next is reached, $\mathrm{n}_0 \longrightarrow \mathrm{n}_2$, the task scheduled to be released then is made ready to run and the array is searched for the next task to release. In n_3 the first delayed task is found and set to be the next task, and then further searching is continued in n_4. If several tasks are scheduled to be released at the same clock tick the queue will release all of them, $n_4 \longrightarrow n_2 \longrightarrow n_3$. Tasks released at the same tick are released in index order to enforce a deterministic behaviour of the releases. As will be discussed in Section 6, this forced order makes it possible to easily achieve better performance. Ticks from the clock will initiate no action if no task is scheduled to be released, $n_0 \longrightarrow n_0$.

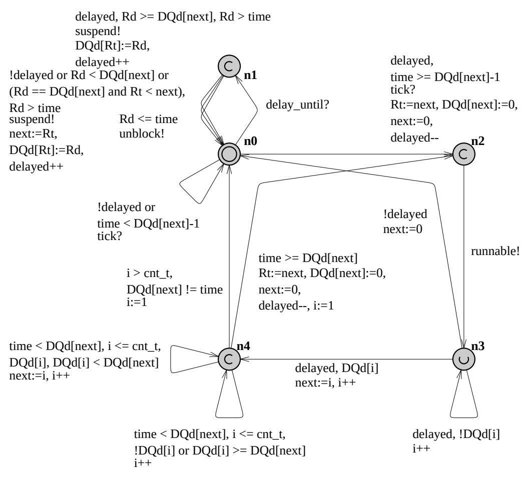


Figure 1. UPPAAL **model of $\mathcal{Q}_1$.**

The worst release case for a single task occurs in the case where all task are scheduled to be released at the same time and the task with the highest index would be released after the release of all other tasks. This property is the same for all queues.

The expected area requirements for the queue is proportional to the size of the number of delaying tasks and some additional variables used to remember the number of currently delayed tasks and next.

3.2 Delay queue $\mathcal{Q}_2$

The second delay queue, shown in Fig. 2, manages delayed tasks by using a time counter for every delayed task. Each position in the array DQd holds a counter for the associated task. The counter for a task is set to the delta time, Δ_T, when the task delays, $n_0 \longrightarrow n_1 \longrightarrow n_0$, and all stored delta times are decremented by one at every system clock tick, in the transitions leading to and from location n_2. Task associated with a Δ_T decremented to zero during the tick handling are released using transition $n_2 \longrightarrow n_3$. When all positions in the array have been processed the automata returns to its initial location and waits for the next tick.

The expected area requirements is lower than that of $\mathcal{Q}_1$ since delta times are used rather than absolute times. No attempts to improve the queues performance by tracking the lowest and highest indexes of the delayed tasks has been attempted since the mechanisms for tracking them are likely to be a waste of valuable chip area.

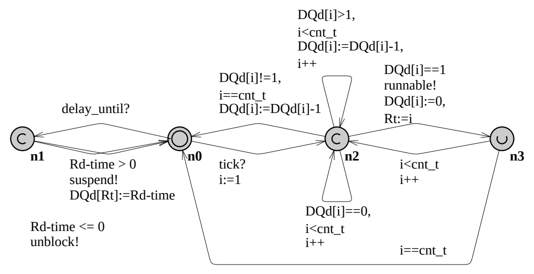


Figure 2. UPPAAL **model of $\mathcal{Q}_2$.**

3.3 Delay queue $\mathcal{Q}_3$

The third queue, see Fig. 3, is a parallelised version of $\mathcal{Q}_2$ where each delaying task gets a countdown timer of it own. Parallelism is used to minimise the time used by the kernel to delay and release the tasks. The delay queue functionality is achieved by the work of all the counters running in parallel. The location to the left of the initial location n_0 is used if a task tries to delay to a time not in the future, and the locations to the right of n_0 are used if the task is delayed. When a task is delayed, the Δ_T for the delay is calculated and stored in a time counter, activating the automaton. The individual values of active counters are decremented on every clock tick, $n_3 \longrightarrow n_3$. When the release time for a task is reached, the counter releases the task and returns to the initial location.

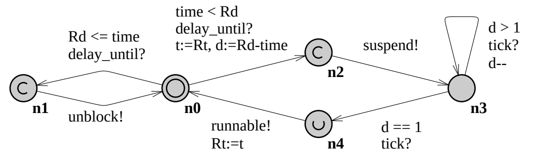


Figure 3. UPPAAL **model of $\mathcal{Q}_3$.**

Location n_4 is urgent (and not committed) since there can be several delay queues contending to release their delayed tasks at the same time. In the model presented here there is no way to enforce a specific release order of tasks released at the same tick. The UPPAAL tool will explore all possible executions of the releases. If the tasks are of the same priority this will introduce a temporal behaviour which can be calculated, though undesired. Ways to handle the releases are discussed in the final discussions of this paper.

The area used by each individual queue is minimal but area will be needed to accommodate one automata for each task that delays. As will be shown in the section on implementation Section 4, the area cost for the parallelism is very high.

3.4 Delay queue $\mathcal{Q}_4$

The fourth delay queue, see Fig. 4, uses two memory arrays. The DQt array contains the task queue and the DQd array is used to store delta times. The delta times represent the time to release tasks once a task is the next task to be released. Every time a task is delayed the queue takes $n_0 \longrightarrow n_1$. If the task is the only task that is delayed the transition back to the initial state is taken. If there are delayed tasks the new task should be queued in DQt according to its delta time. DQt is a circular queue whose head is pointed at by next and the position where the delayed task should be inserted is found using $n_2 \longrightarrow n_2$. When the insertion position has been found all tasks to the right of it are shifted right, $n_3 \longrightarrow n_3$. When the delta time of a next is reached, $n_0 \longrightarrow n_4$, all tasks delayed to the same time will be released using $n_4 \longrightarrow n_4$.

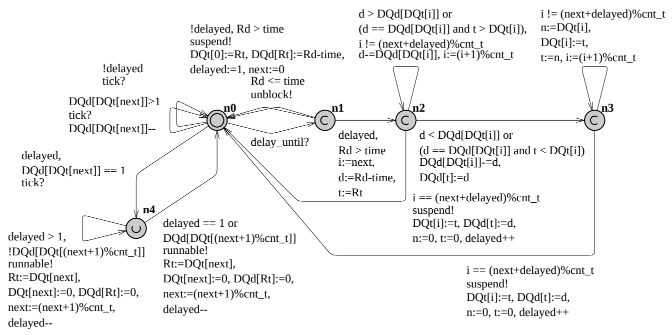


Figure 4. UPPAAL **model of $\mathcal{Q}_4$.**

A possible area optimisation for this queue is if the greatest number of tasks that can be delayed at a single time is known. If the maximum number of simultaneously delayed tasks is known the DQt array can be reduced to at most accommodate this number of tasks. A possible speed optimisation is to check if the array should grow at the beginning rather than at the end. If there are n tasks delayed, shifting the task identities to the left is preferable if the task to be inserted is among the $n/2$ tasks with the closest release times.

4 Implementation

The UPPAAL models of the four delay queues were manually translated to VHDL state machines with extra glue logic, e.g. for communication, and then synthesised to the target device. The Xilinx ISE Foundation 6.2.03i tool [20] was used for synthesis and the target device was a Virtex2Pro `2vp7ff672-7` FPGA [21]. The FPGA has an on-chip PowerPC [10] processor on which the actual tasks are run.

The basic translation of the UPPAAL automata to VHDL finite state machines (FSM) is straight forward but constructs like UPPAAL's channels, urgent locations and committed locations are not present in VHDL and these constructs have to be handled with care. A transition from an urgent location should be taken before the next system clock tick and this can be accomplished if the implementation makes sure that the state machines finish urgent parts within a system clock tick. The way we have ensured this is by having the clock speed of the kernel run so fast that all work in a finite state machine (FSM) can be completed within time. As described in Section 3, committed locations are used for atomic transactions, e.g. in the UPPAAL model of $\mathcal{Q}_1$, Fig. 1, the transitions $n_0 \longrightarrow n_2 \longrightarrow n_3$ form an atomic chain where the automata first receives over a channel and then sends over a channel. This behaviour can be optimised in the implementation by using separate Rt for input respectively output, cf. Rt and rdy_Rt in Table 2. We have chosen to translate the communication and synchronisation channels of UPPAAL into a call-and-acknowledge protocol, shown in Fig. 5. With our translation the transition $n_0 \longrightarrow n_1$ corresponds to a delay call and $n_1 \longrightarrow n_2$ corresponds to an acknowledge, i.e., the call has been handled. In this case the call is handled in n_1; additional locations needed before the acknowledgement should replace n_1. Transition $n_2 \longrightarrow n_0$ is used to complete the communication/synchronisation sequence. An alternative translation is the more complex channel implementation described in [15].

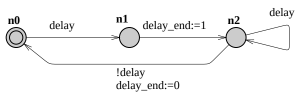


Figure 5. State graph for communication.

Delay queue designs $\mathcal{Q}_1$, $\mathcal{Q}_2$ and $\mathcal{Q}_4$ use arrays to store information like task identities and delay values. For the target technology we use, arrays can be implemented with registers and/or with block ram memory. A register implementation is larger since it requires a register to be coded in the FPGA while a memory implementation can use memory blocks the FPGA already have. The performance penalty for using a block ram, where an access takes one clock cycle, instead of registers, where an access takes zero clock cycles, in Virtex2Pro is not significant in our case. An example of how we handle block ram accesses is showed in Fig. 6. Transition $n_0 \longrightarrow n_1$ is the DQt[next] access and transition $n_1 \longrightarrow n_2$ is the DQd[DQt[next]] access. The last transition $n_2 \longrightarrow n_3$ is the DQt[next]:=0 access. In other words, we try to set the address in advance to not lose an extra clock cycle and when that is not possible we insert an extra state.

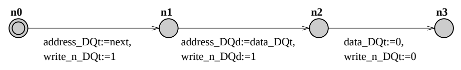


Figure 6. State graph for assigning the value 0 **to** DQd[DQt[next]],DQt[next]**.**

A reset state, shown in Fig. 7, looping through arrays and initialising variables has been introduced to initialise memory arrays. This state implements UPPAAL's initialisation of its variables but is not shown in the state graphs of the different queue implementations, e.g., Fig. 8.

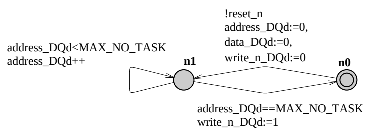


Figure 7. State graph for reseting the DQd **array.**

The hardware signals/busses described in Table 2 are a translation of the component design interface shown in Table 1. The reset and clock signals that resets respectively clocks the FSMs are not included since they generally exists in any FSM implementation and do not contribute to the understanding.

4.1 Delay queue $\mathcal{Q}_1$

The implementation of $\mathcal{Q}_1$ consists of the FSM shown in Fig. 8. The state machine basically has the same states as the UPPAAL model, cf. Section 3.1, and block memory is used to implement the DQd array. The transition $\mathrm{n}_0 \longrightarrow \mathrm{n}_1$ is replaced with $\mathrm{in}_0 \longrightarrow \mathrm{in}_1 \longrightarrow \mathrm{in}_5$ to implement the call and acknowledge protocol. The extended parallelism of the implementation allows Rt_rdy to be set to runnable when the Rt value is received, allows $\mathrm{n}_4 \longrightarrow \mathrm{n}_2$ to be translated into $\mathrm{in}_4 \longrightarrow \mathrm{in}_3$. Address_DQd replaces the temporary i variable used for looping in the UPPAAL model. Since address_DQd is cnt_t bit wide, transition $\mathrm{n}_4 \longrightarrow \mathrm{n}_0$ has been changed to handle the fact that address_DQd can't be greater than cnt_t.

4.2 Delay queue $\mathcal{Q}_2$

The implementation of $\mathcal{Q}_2$ is similar to that of $\mathcal{Q}_1$. The state machine has basically the same states that the $\mathcal{Q}_2$ model has, cf. Section 3.2, and like the FSM for $\mathcal{Q}_1$ it uses block memory to implement the DQd array. The implementation also uses the same variable and location eliminations as described for $\mathcal{Q}_1$. The cycle need to access the ram block storing DQd is implemented by introducing an extra location in the FSM.

Table 2. Hardware signals/busses.

(a) Signals

tick	Clock signal generated at system level frequency that decides the delay accuracy available to the application.
delay	Signal triggering the state machines to insert a task (Rt) with delay time (Rd).
delay_end	Signal to synchronise with a bus-interface that a delay call has finished.
runnable/ suspend/ unblock	Signals used to request the ready queue (scheduler) to make a task runnable, to inform that the task was suspended, or to signal the the task was unblocked, i.e., that the task should be placed last within its priority in the ready queue.
rdy_end	Signal from the delay queue to inform that a runnable/ suspend/ unblock request has finished.

(b) Busses

time	The current system time. The system time starts from zero and is increased at every clock_tick.
Rt	Task identity of the task that perform a delay call.
rdy_Rt	Task identity of the task that will be runnable/suspend/ unblock.
Rd	Absolute delay-time the task should be delayed until. Queues that use delta time will subtract the system time from this time to get Δ_T.

4.3 Delay queue $\mathcal{Q}_3$

The countdown timers of the third queue design is implemented using the finite state machines shown in Fig. 9. One of the state machines, FSM_n with locations in_0–in_3, is an interface for distribution of delayed tasks to their dedicated state machine. The dedicated state machines are described by machine FSM_m with location im_0. FSM_n calculates Δ_T and performs the Δ_T checks to decide if a delaying task should unblock or suspend. FSM_n also informs the ready queue when a task's delay time has expired and the task becomes runnable. The FSM_p, i.e. ip_0, serialise the run signals from the FSM_ms for FSM_n. Each FSM_m implements a register counter, which is decremented at each clock

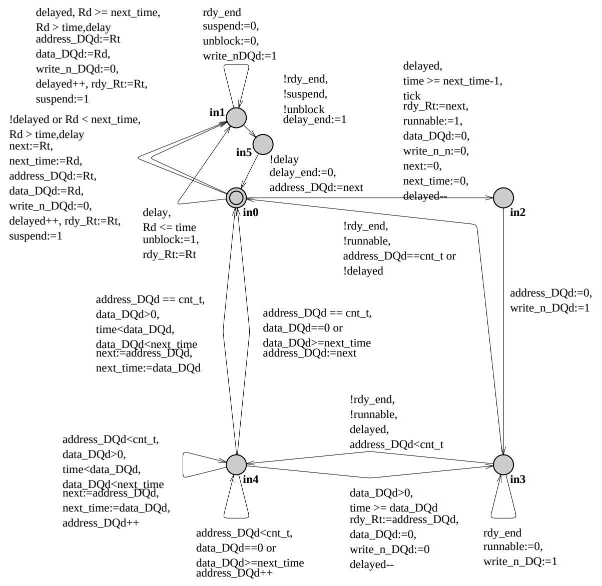
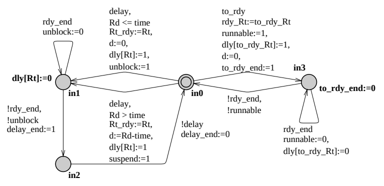
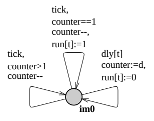


Figure 8. State diagram for delay queue $\mathcal{Q}_1$.

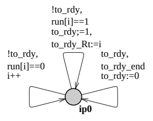
tick until it become zero. The machine signals that the task should be released when the counter value to be decreased is one. The model of the counters, cf. Section 3.3, basically corresponds to FSM_n, FSM_m and FSM_p. The reason for



(a) FSM_n



(b) FSM_m



(c) FSM_p

Figure 9. State diagrams for delay queue $\mathcal{Q}_3$.

partitioning the model is to save hardware area and to enable a priority ordered task activation which is further discussed in Section 6. If priority ordered activation is used it can be managed by FSM_p. The area required by the design is reduced by using one interface state machine FSM_n, instead of multiple similar interfaces for each FSM_m.

The choice of using a register rather than the smaller memory blocks is that the access speed mentioned above, one clock cycle, applies to the access of a RAM block which no other FSM uses. Using a block to hold information about a single task is resource waste and having several queues use and access the same block will be another implementation of $\mathcal{Q}_2$.

4.4 Delay queue $\mathcal{Q}_4$

The FSM created while implementing $\mathcal{Q}_4$ closely resembles the automaton in Fig. 4. Both the time-counter and task-queue array, DQd and DQt, are implemented in block memory.

The same variable and location eliminations described for $\mathcal{Q}_1$ and $\mathcal{Q}_2$ are used and extra locations are added to handle the extra clock cycles needed when accessing the DQd and DQt memories.

5 Results

To investigate the properties of the implementations we synthesised systems with different numbers of delaying tasks and timer widths. The bit times are selected to represent systems with high rate cyclic executives (16 bit time), medium rate (32 bit time), and then the 50 years required by the Ada standard (41 bit time).

5.1 Area

The results we present in this section are based on synthesis with the clock timing constraint set to 10 ns, i.e., creating a kernel running with a kernel clock frequency of 100 MHz, and without any area constraints. Besides that, the synthesis tool default settings has been used. The gate count is roughly equivalent to the chip area used by the implementations. The gate counts used by the synthesised systems are presented in Fig. 10.

It is not unexpected that it is resource effective to use memory rather than registers when the number of tasks increases. The small gate count growth between 4–16 tasks configurations for designs $\mathcal{Q}_1$, $\mathcal{Q}_2$ and $\mathcal{Q}_4$ is due to that the synthesis tool use 16x1 memory primitives for the arrays. The size growth of the queues is, as expected, close to linear. Queues $\mathcal{Q}_1$, $\mathcal{Q}_2$ and $\mathcal{Q}_4$ use RAM blocks to store data. The number of RAM blocks used to implement the single array is the same for $\mathcal{Q}_1$ and $\mathcal{Q}_2$ while $\mathcal{Q}_4$ use a little bit more to implement its two arrays. Since $\mathcal{Q}_3$ does not use RAM blocks to code its registers, the cost for the stored variables is taken by the gate count.

The $\mathcal{Q}_3$ design, which uses registers instead of memory for the time counters does not show this behaviour. The area usage of the different queues is such that $\mathcal{Q}_2$ is smallest, followed by $\mathcal{Q}_1$ and then by $\mathcal{Q}_4$. However, $\mathcal{Q}_3$ uses the least area for systems with 4 delaying tasks but using registers is not resource effective for large task sets and $\mathcal{Q}_3$ quickly outgrows the other implementations.

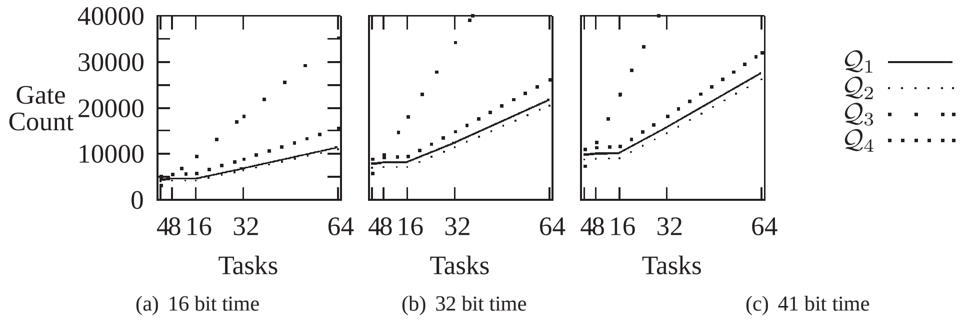


Figure 10. Gate usage of the implementations. The x-axis shows the number of tasks the queues handle and the y-axis shows the gate count.

The memory utilisation of the designs is very small compared to that available on the target system. For example, the 4200 gates used by a 16 task 16 bit time $\mathcal{Q}_2$ implementation is not much compared to the 811008 gates that the target Virtex2Pro device supports. A 4 tasks 16 bit time synthesised system for any of the queues uses 1%–3% of the FPGA's resources in slices, 4 input LUTs and slice flip flops. Queues $\mathcal{Q}_1$, $\mathcal{Q}_2$ and $\mathcal{Q}_4$ uses 6%–10% of the slices and LUTs and 1%–2% of the slice flip flops when synthesised for a 64 tasks system with 41 bit time but the register queue, $\mathcal{Q}_3$, uses about 80% of the available slices and LUTs and close to 30% of the slice flip flops for this configuration. Fitting the queues, besides the register queue, on the target FPGA can easily be accomplished with the better part of the resources left for the rest of the kernel and other system components.

5.2 Speed

The execution properties of the delay queue implementations depend on the behaviour of the rest of the kernel. The communication times between kernel components will influence the execution time of the delay queue. Other kernel components can force the delay queue to wait, e.g. when a batch of tasks is released the ready queue will accept them in serial at the rate it can process them. The application will also impact on the execution time of the delay queue since it will instantiate the queue with, e.g., the number of tasks that delay.

Let $\mathcal{S}_{T_{Dly}}$ be the set of tasks that delay in an application and let $|\mathcal{S}_{T_{Dly}}|$ be the cardinality of that set. Let $\mathcal{C}_{run}$ be the number of clock cycles the ready queue uses to make a task runnable and let $\mathcal{C}_{sus}$ be the number of clock cycles it uses to suspend a task.

The worst case execution for $\mathcal{Q}_1$ and $\mathcal{Q}_4$ to delay a task occurs when the delay request arrives during the release of a batch of tasks and the time is described in Equation 1.

$$((|\mathcal{S}_{T_{Dly}}| - 1) * (3 + \mathcal{C}_{run}) + 1) + (4 + \mathcal{C}_{sus}) \qquad (1)$$

The first part of the expression, $((|\mathcal{S}_{T_{Dly}}| - 1) * (3 + \mathcal{C}_{run}) + 1)$, describes the number of kernel clock cycles used to time out all tasks in the delay queue, i.e. the task with the lowest priority will have to wait for all other tasks to be handled by the ready queue, and the second part, $(4 + \mathcal{C}_{sus})$, describes the number of cycles used to manage the insertion of the call into the queue.

The worst delay for $\mathcal{Q}_2$, shown in Equation 2, is similar to the one of $\mathcal{Q}_1$ and $\mathcal{Q}_4$.

$$((|\mathcal{S}_{T_{Dly}}| - 1) * (2 + \mathcal{C}_{run}) + 1) + (4 + \mathcal{C}_{sus}) \qquad (2)$$

The worst case for $\mathcal{Q}_3$ is different since delays are made to private time counters and in parallel. Equation 3, takes the shared interface machine into consideration.

$$(3 + \mathcal{C}_{run}) + (4 + \mathcal{C}_{sus}) \qquad (3)$$

Note that $\mathcal{Q}_3$ prioritises delay calls before each clock tick and that is possible because it uses one FSM for each task's delay counter. It is not possible to prioritise delay calls with the other delay queues since this would risk a system clock tick to be missed.

The frequency the kernel clock, KerClk, needs to ensure that the queue's work, together with any time added by interaction with other kernel components, can be completed within a system clock tick. If this can not be guaranteed the kernel risks missing system ticks. Besides this, the kernel clock frequency must support the Ravenscar profile delay accuracy of 1 ms of the system ticks. Table 3 shows the maximum clock frequency the queues can be synthesised for a 16 task configuration. To check that the queues satisfy the 1 ms requirement we made a coarse overestimation of the worst number of kernel cycles used to delay a task in a kernel with 16 tasks would have the kernel working, and found this to be 350 kernel cycles. All these cycles must be completed within a system clock tick for the operation of the kernel to be guaranteed correct. In Table 3 we see that slowest queue, $\mathcal{Q}_1$, can be synthesised to a maximum of 132 MHz. This speed would allow the system to be synthesised with a system clock frequency of 0,38 MHz which clearly supports the 1 KHz that Ravenscar requires.

Table 3. Maximal clock frequency (in MHz) that the different queues can be synthesised for, when 16 tasks are supported.

Time Width	$\mathcal{Q}_1$	$\mathcal{Q}_2$	$\mathcal{Q}_3$	$\mathcal{Q}_4$
16	145	173	283	155
32	142	156	242	144
41	132	150	225	138

The calculation presented here makes no optimisations of the system clock tick management. An optimisation, which can allow the kernel to run at a slower speed, is the use of a buffer for the system clock ticks. A management with a buffer that can store ticks would allow the delay queue to spread its worst case work over the number of ticks the buffer can hold. This is based on the simple reasoning that a worst case cannot be followed by another equally bad case since the first case will lead the system to a system state where the equally bad state is impossible. For example, if the worst case is that all tasks are delayed and released at the same time they will not be delayed during the next tick making it impossible to repeating the release. A system with a buffer could make it easier to synthesise the system and produce an efficient hardware kernel.

6 Discussion

The delay queue (and the whole kernel) is designed to be synthesised for a specific target application system. This specialisation enables some interesting resources optimisations. In Section 3 an optimisation of the number of bits used to represent delta times was presented. The optimisation used knowledge about the cycle time of cyclic executive tasks. The length of the memory array needed to remember release times can be optimised if tasks that delay are given task identities in sequence.

The normal procedure when releasing tasks is to lock (stop) the dispatching before releasing tasks, e.g. as done in [13]. The need for the locking is only necessary in a system where a task of lower priority, T_l, can be released from the delay queue ahead of a task of higher priority, T_h, when a batch of tasks are released at the same time. If the dispatcher is not locked a situation can occur where T_l is loaded on a processor only to be preempted when T_h is released. The situation is avoided if tasks are released in priority order, with the release of the highest prioritised task first. It is safe to dispatch and start loading T_h since no task in the same release batch can force T_h to be preempted. A FIFO order within each priority makes the release behaviour even more deterministic if several tasks can have the same priority. In a system where all tasks have their unique priorities the index order can be used as priority order. Queues $\mathcal{Q}_1$ and $\mathcal{Q}_2$ enforce priority ordered release if the task indexes are ordered in priority order. To achieve FIFO release these queues would have to be extended with memories to carry the priorities of the tasks and the arrival order of the tasks. The dispatch order of $\mathcal{Q}_3$ can be defined if the communication between the counters and the ready queue is defined to follow a specific protocol, i.e. one which prioritises the signals from the counters according to the priorities of the tasks. FIFO order is however outside the immediate reach of $\mathcal{Q}_3$ since it would place to many requirements on communication or synchronisation to be usable. The $\mathcal{Q}_4$ queue releases according to index order and FIFO order. Like the first two queue, $\mathcal{Q}_4$ will have to be extended with more memory to implement FIFO if several tasks can have the same priority.

The Ravenscar profile only allows absolute delays where the release time is given explicitly, i.e. there are no relative delays where a task delays for a given time. The main reason for not supporting relative delays is that it makes system analysis easier with only a delay-until mechanism. Also, the kind of systems the profile focuses on, cyclic executives, use delay until. The queues we present can easily be extended to handle relative delays by adding an extra interface function that doesn't calculate the delta-times. The formal automata should then be extended in the same way.

The delay queues presented are not limited to use in a hardware RTK but can be used as standalone components to help a processor manage delayed tasks. A memory mapped bus interface to the delay queue allows them to exist in a hardware/software co-design. The interface should contain the system time and implement the clock-tick generation. The task status, runnable/suspend/unblock, should be included in a readable register and it should also be wired to an interrupt pin at the processor. Additionally, a readable task identity register should be included.

One weakness with the current hardware implementations is that they have been manually translated using the formal models as blueprint. The implemented queues have not been verified to ensure that they describe exactly the same behaviour as those described in the formal models. We are currently looking at how verification of the implementations can be made using available hardware tools and using the formal models for input.

7 Conclusions

In this paper we present formal models and hardware implementations of four delay queues suited for multiple processor systems. The queues express different properties regarding hardware requirements, possible parallelism, and execution times. Different task release policies and how they can be supported by the queues is discussed.

The translation from the original designs, made in timed automata to VHDL, is described and metrics of the hardware implementations are presented.

Surprisingly, the queue using most parallelism, $\mathcal{Q}_3$, shows not only the best response time properties but also the least chip area usage for systems where four or fewer tasks use the delay queue. In systems with more than five delaying tasks that queue quickly outgrows the other queues in area. Otherwise $\mathcal{Q}_2$ uses the least amount of chip area. All queues can meet the Ravenscar timing demand of a granularity of 1 ms.

Though not attempted in this paper, an interesting study would be that of a framework where the properties verified in the initial design, made in a high level verification tool, could be transformed into properties of the hardware tool used for synthesis to hardware. Enabling verification of the high level properties could be a step in validating software to hardware translation.

References

[1] "The Consolidated Ada Reference Manual", Springer–Verlag, LNCS 2219, 2001.

[2] S. Bradley, W. Henderson, D. Kendall, and A. Robson, "A Formally Based Real-Time Kernel", *Fundamental Approaches to Software Engineering* LNCS 1382, Springer–Verlag, 1998.

[3] A. Burns, B. Dobbing, and G. Romanski, "The Ravenscar Tasking Profile for High Integrity Real-Time Programs", *Reliable Software Technologies — Ada-Europe 1998*, LNCS 1411, Springer–Verlag, 1998.

[4] A. Burns, B. Dobbing, and T. Vardanega, "Guide for the use of the Ada Ravenscar Profile in hight integrity systems", *University of York Technical Report YCS-2003-348*, 2003.

[5] W. Hussak, "Temporal analysis of a microkernel", *Software Engineering Journal*, Software Engineering Journal, vol. 10, issue 1, IEEE, 1995.

[6] K. Larsen, P. Pettersson, and W. Yi, "Uppaal in a Nutshell", Int. Journal on Software Tools for Technology Transfer, Springer–Verlag, 1997.

[7] K. Lundqvist, and L. Asplund, "A Ravenscar-Compliant run-time kernel for safety critical systems", Real-Time Systems, 24(1), 2003.

[8] R. Maria, et.al., "Verifying, Validating and Monitoring the Open Ravenscar Real Time Kernel", *12th International Ada Real-Time Workshop*, Ada Letters, vol. XXIII, no. 4, 2003.

[9] A. Morton, and W. Loucks, "A Hardware/Software Kernel for System on Chip Designs", *ACM Symposium on Applied Computing*, 2004.

[10] IBM Microelectronics and Motorola Inc., "The PowerPC Microprocessor Family: The Programming Environments", IBM Microelectronics Document MPRPPCFPE-01, Motorola Document MPCFPE/AD (9/94).

[11] M. Laramie, "Automating Analog Verification in a Mixed-Mode Simulation", *SNUG Boston 2004*, 2004.

[12] SafeLogic (property based verification), Combining dynamic and static property checking, `http://www.safelogic.se/news/downloads/Dynamic_and_static.pdf`

[13] J. de la Puente, et.al., "Open Ravenscar Real-Time Kernel – Operations Manual", 2001.

[14] J. de la Puente, et.al., "The design and implementation of the open Ravenscar kernel", *10th international workshop on Real-time Ada workshop*, Ada Letters, vol. XXI, no. 1, pp. 85–90, 2001

[15] A. Silbovitz, K. Lundqvist, "A hardware implementation of a Ravenscar-compliant run-time kernel", *Digital Avionics Systems Conference*, IEEE, 2003.

[16] P. Tullman, et.al, "Formal Methods: A Practical Tool for OS Implementors", *The Sixth Workshop on Hot Topics in Operating Systems*, IEEE, 1997.

[17] T. Vardanega, J. Zamorano, and J-A. de la Puente, "On the Dynamic Semantics and the Timing Behaviour of Ravenscar Kernels", Real-Time Systems, vol. 29, 2005.

[18] J. Zamorano, and J. de la Puente, "Precise Response Time Analysis for Ravenscar Kernels, Ada Letters, vol. XXII, no. 4,, 2002.

[19] J. Zamorano, J. Ruiz, and J-A. de la Puente, "Implementing Ada.Real_Time.Clock and Absolute Delays in Real-Time Kernels", *6th Ade-Europe International Conference Leuven on Reliable Software Technologies*, 2001.

[20] Xilinx Inc., "Xilinx ISE 6 Software Manuals and Help", 2004.

[21] Xilinx Inc., "Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet", 2004.