# Spring Kernel: Added Features (implementation notes)

G. Wallace G. Zlokapa

December 1993
University of Massachusetts at Amherst
Computer Science Department

## System Synchronization

### dispatcher new schedule announcement

status: implemented

> while the dispatcher is waiting for the sst of it's next task to occur, it keeps checking the new schedule flag for an announcement from the scheduler that a new schedule has been made. this way the dispatcher will reevaluate the sst of the next task.

### distributed systems clock synchronization

status: future

> sw clock on scramnet

status: future (long term)

> multiple spring nodes need to have some sort of facility to synchronize their clocks. needed for things like assignment of absolute deadlines, logging event times, etc. (both external and local activities). current need is for the robot demo detection process to be able to timestamp ball detection, and for creation of deadlines based on consuming process info. consuming process needs time to specify its availability interval.

# Error Handling

## RMC error recovery

status: implemented

> bus error handler routine checks Stat2 for RMC error and sets the retry bit in the SSW found on the stack, causing a hardware retry when the RTE from the handler is executed.

## bus error handler feedback

status: implemented

> bus exception handler routine prints out vital statistics when it can't recover (see RMC error recovery). this info includes the stat2 register, the stack pointer, the status register, the program counter, the stack frame type, the vector offset, the software clock.

## generic exception handler

status: future (long term)

> generic exception handler to provide more clues as to what went wrong and better error servicing. hint: one exeception handler that looks at the exception type and vector offset in the exception stack frame, and takes the appropriate recovery action. (this involves updating the exception vector table for all supported exceptions, vector offsets)

## ap processor failure

status: implemented

> scheduler checks for ap processor failure by examining tasks above the cutoff line to make sure they are done if the current time is after the task's sft. in this case we assume the processor has failed (watchdog timer handles overrun errors).

status: future (long term)

> implement recovery procedure along fault tolerant lines

## Task Delay

### task separation interval

status: implemented

> need to be able to specify the minumum and maximum separation time between two precedence related tasks. extend sdl to allow specification of a minimum and a maximum delay. have scheduler use minimum delay to adjust task arrival time (release time) and use maximum delay to adjust task deadline.

### task delay count

status: implemented

> to properly handle delays (in the case where both tasks are rescheduled), each task needs to have a count of the number of predecessors it has that specify a delay. the pap should generate this task delay count.

### delay reschedule problem

status: implemented

> (see task delay count). want to be able to reschedule tasks that have precedence with delays (i.e. the arrival time of the successor has to be reset if both tasks can be moved up). this is done by resetting the working delay count and working arrival times when we calculate the cutoff line. the scheduler maintains these working fields and only uses working arrival time if the working task delay count is equal to the total task delay count. otherwise it has to use the max(old arrival time, working arrival time). old arrival time is saved from previous schedule. (this same scenario should apply to maximum delay and deadlines).

### communication delay

status: future (long term)

the sdl syntax specifies a list of port names as part of the successor list (why this can be a list is unclear at this point). since a comm port has a delay associated with it that is only known at boot time, the pap needs to invoke a system call that returns the delay associated with the named port (use get_sys_info()?). then min_delay = max(min_delay,comm_delay). since all ipc calls use scramnet right now, the comm delay for all ports is a constant value. this means we could have the system service call simply return a constant value (or we can avoid this issue altogether for now if we manually fill in the successor list min_delay).

# Time Sanity

## alarms

status: implemented

consistent alarm mechanism that allows shutdown in a graceful manner. there is one global alarm flag seen by all processors. dispatchers and scheduler check for this alarm condition each time thru their respective processing loop, and cease processing once the alarm is raised. the scheduler also provides feedback to the user based on the alarm information saved when the alarm was raised (e.g. type of alarm, time of alarm, task causing alarm, section of code for time overruns)

status: future (long term)

implement recovery from alarms along fault tolerant lines

## time overrun alarm

status: implemented

use countdown timer to detect time overruns. dispatch loop does stop timer for previous task, check for global timing error (halt in this case), start timer for next task. this handles time overruns that occur in dispatch in, task execution, and dispatch out sections. it also provides external process control for free since external processes are controlled by spring processes that

are handled as normal spring processes by the scheduler. timer isr routine sets both global (for all processors to see) and local alarm (to identify culprit). diagnoses of alarm conditions done by scheduler, this identifies section of code that caused the alarm (dsp in, dsp out, task, timer overhead).

### dispatcher late alarm

status: implemented

dispatcher does a self check to make sure tasks start on time (according to the sst set by the scheduler). if this check fails an alarm is raised and the alarm condition is handled by the scheduler. this alarm catches the situation where the dispatcher takes too long between stop timer and the next start timer, all other dispatch time problems will be caught by the time overrun alarm.

### scheduler late alarm

status: implemented

scheduler does a self check to make sure it finishes by the time represented by the cutoff line. if this check fails, an alarm is raised and the alarm condition is handled by the scheduler.

## Statistics & Measurements

(used to fine tune timing and identify errors)

### dispatcher measurements

status: implemented

the dispatching function is broken into two sections, dispatch in and dispatch out. dispatch in is the the time from start timer until the end of the end of the change context routine. dispatch out is the time from user task exit until stop timer (just before the timer is started for the next task). dispatch in and dispatch out times are measured and the maximums seen are recorded.

### task execution time

status: implemented

> actual execution times are measured by the dispatcher for each instance of a task. the scheduler uses these recorded times to maintain a minimum and maximum for all instances of that task. the scheduler also records the sum of these execution times and the number of instances measured, so an average task execution time can be provided (avg = (sum - min - max) / (num -2)).

### schedule length

status: future (long term)

> have the scheduler gather statistics after each new schedule is announced (only if there are no new events, no external processes have finished, and no tasks have finished). look thru the entire stt and find the min sst and max sft for each instance of a process group. if all tasks are there (number found = total in group) then update group start time. update group finish time anytime at least one task is found. to keep stats for each instance of a process group some sort of table would be needed (the runtime data structures are only templates). if we only maintain a minimum, maximum, and average across instances of task groups we don't need the table (could use process group structure instead).

## Resources

### cheap resource reclaiming

status: implemented

> allow the scheduler to reschedule continuously as long as some task has recently completed (see task done flag). this way we can take advantage of tasks that finish early and release their resources. this shouldn't be penalty because we only reschedule those tasks that are below the cutoff line, the disadvantage of course is the situation when an event arrives while the scheduler is busy rescheduling.

### task done flag

status: implemented

> task done is flag global to the entire spring node. set by the dispatchers, cleared by scheduler. this flag indicates that at least one of the ap's is doing something. the scheduler uses this to determine its course of action, i.e. does it need to reschedule (this takes precedence) or can it gather stats and serve the monitoring interface.

### multiple instance resources

status: future (long term)

> scheduler needs to be able to handle resources (including processors) that have multiple instances in a generic manner (e.g. delivery platform with finite capacity in robotics demo).
>
> resource instance mask that specifies eligible set of resource instances that can be assigned to requesting task. scheduler decides which instance(s) to assign based on availabilty.
>
> number of required resource instances. how many instances this task needs from the specified eligible set.

### request/release resource attributes

status: future

> resource transactions need to include request only (rq), release only (rl), and request/release (rr). this last case is all we have now. rq and rl can span multiple tasks and requires special handling by scheduler.
>
> update eat mechanism should set the eat to infinity for a rq and to sft for rl (to sft for rr, this is the what we do currently). resetting the eat's for rq/rl pairs needs to handle the case where the rq task is gone. do this by updating the eat to infinity iff we find a rl task after the cutoff line and the rq task is before the cutoff line or gone already.

**NB:** for now only implement request/release functionallity required by the RPI - a request task has only a single successor link, and that path leads to the release task. we impose this restriction because there is currently no information that allows request tasks to be matched to the proper release tasks (precedence lists allow multiple paths that may contain more than one request/release pair). it seems that any additional flexiblity would require the programmer to somehow note the matching pairs, so the compiler can pass that information to the pap and scheduler. this limited support also means that unmatched request/release pairs will have catastrophic results! this obviously has to be fixed soon.

# Remote Processes / Scramnet Interrupts

## RPI

status: implemented

a remote process is defined as a computation that is external to the spring programming and runtime environment. need to control execution of remote processes in a predictable manner. use new rq/rl resource attributes and scramnet interrupts to implement this. this mechanism is called the remote process interface (*RPI*).

### remote process specification

status: implemented

remote processes are defined by the SDL node layout for the spring node that will control the booting of all the external nodes (*boot master*). for each RPI, this definition includes node and scramnet acr value pairs that specify the spring node (resource owner) with its scramnet acr mode setting, and the remote node with its mode setting (transmit, receive, transmit and receive). what mode settings make sense for the spring node and what ones make sense for the remote nodes?

the application code will specify calls for execution of external processes. these calls will translate into a request task, followed by a delay (wcet of the external process), followed by a release task (the resource is the csr for the external process). for now we use minimum delay only, if this causes problems we could force the scheduler to strictly honor remote process wcet by specifying maximum delay also.

### remote process boot time

status: implemented

sbug reads the SDL node layout information and sends *RPI* download commands to the specified spring node. only one node should receive this RPI info (even though multiple nodes may exist in the SDL layout), this spring node is called the *boot master*. An error is raised if any RPI definitions specify duplicate RPI names for the same remote node id's. the sp kernel on the boot master node receives the RPI download commands and builds the RPI table in the scramnet RPI region in response. the RPI name and node/acr value pairs are left in the table entry for the other nodes to read.

boot processes on all nodes, including spring nodes (sp kernel), need to initialize scramnet to ensure a proper initial state (e.g. clear all acr's, enable vme interrupts). they also scan the scramnet RPI table looking for entries that match their node id, and initialize the scramnet acr's accordingly.

all remote processes scan the RPI table looking for the entry that matches their process name. they then update the required semantic info in this entry. currently there exists one semantic that requires the address of a flag local to the remote process.

boot synchronization is accomplished via the various RPI status fields (csr, boot, and remote process). the boot master creates and initializes the scramnet RPI region, then sets the RPI csr status to *RPI table ready*. the boot processes on all nodes wait for this RPI csr status, then initialize scramnet and set acr's, and finally update the RPI boot status to *boot process ready*. the

boot master waits for all boot processes to complete, then sets the RPI csr status to *boot process ready*. the remote processes wait for this RPI csr status, then update the semantic info, and finally set the RPI remote process status to *remote process ready*. the boot processes on all nodes wait for all remote processes to be ready, at this point RPI booting is complete. non spring boot processes may simply exit, spring nodes continue with the spring boot sequence of events.

### remote process run time

status: implemented

the scheduler checks the RPI remote status field for a *done* condition, and uses this information to adjust the delay specified between the request and release rpi tasks (this allows the scheduler to reclaim execution time from remote processes).

a spring process that invokes a remote process uses the RPI macro. this macro creates a request-delay-release task precedence. the request task looks up the remote process name in the RPI table, updates the stt address in the rpi table entry, and triggers the interrupt for that entry. the release task checks the remote process status and raises a timing alarm if the remote process hasn't yet completed.

remote processes spin on their own local flag, do their particular computation, and then set the remote process status to *done* upon completion.

scramnet isr's read the address of byte that caused the interrupt (provided with interrupt) to locate the proper RPI table entry. the semantics indicate what action the isr's should take. currently our single semantic case requires that the isr's set the local flag address found in the interrupt entry. this is the address the remote process is spinning on, so it is in effect a *go* signal.

## External world/Spring interface (event mechanism)

### process group parameter passing

status: future

we would like to be able to pass parameters to specific instances of process groups. the event generation process tags each event with a unique event id and generates a process group id (unique within each event) for each process group it combines into a *dynamic process group*. it also stores the parameter lists in a table accessed by event id and process group id.

the scheduler either uses the event id as the instance id, or it updates a table mapping instance id to event id. process group id is saved in each stt entry.

get_sys_info() is enhanced to return event and process group id's. application processes use this call to determine their identity and access their parameter list using the answer as key. this access to the parameter list can be thru shared memory or ipc. an alternate way of implementing this functionallity would require the scheduler to accept a parameter list as part of an event. the scheduler builds the parameter list table keyed by instance id. the dispatcher looks up the argument list and passes the arguments to the task via argc, argv. who cleans up the parameter list table?

### dynamic process group creation

status: future

the event generation process combines known process groups (specified by SDL) into dynamic process groups by creating a precedence graph (successor list). the scheduler parses this successor list and uses it to propagate the precedence information to the stt entries that are created for this event. it seems we may need to keep the begin and end task information that the pap now throws away (remember that a process can have multiple begin/end tasks).

### scheduler request semantics

status: future

in order to provide adequate service to something like an AI planner, the spring scheduler needs to accept different semantics

regarding its action and feedback. provide the following: straight schedule - schedule this event and reject it if no feasible schedule, probe schedule - schedule this event and return feasibility, revert schedule, extra return schedule - schedule this event and reject this event and min value density event if schedule is infeasible, return specific? - remove a specified event from schedule.

removing an event requires that the task group is complete (all tasks lie after the cutoff line). we need to be careful about having the scheduler cycle for a nondeterministic amount of time because the scheduler cutoff line depends on known cost.

# Miscellaneous

## scheduling cost formula

status: future

determine exact formula to be used in determining scheduling cost (using a constant right now)