Constraints on the Use of Executors in Real-time Systems

Stephen Michell

Brad Moore

Luis Miguel Pinho*

S. Tucker Taft

*CISTER Research Center

CISTER-TR-160404

2016/04/11

Constraints on the Use of Executors in Real-time Systems

Stephen Michell, Brad Moore, Luis Miguel Pinho*, S. Tucker Taft

*CISTER Research Center

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail: lmp@isep.ipp.pt

http://www.cister.isep.ipp.pt

Abstract

Previous work proposed work to include parallelism in Ada to improve the use of multicore processors. This work proposed a model of Tasklets to carry the expressions of parallelism, and introduced the notion of executors to map the execution of Tasklets on hardware. In developing the model, a number of choices arise that effect how well the model will accommodate real-time systems. These choices become constraints which are examined and reasonable choices proposed for adoption in Ada.

Constraints on the Use of Executors in Real-time Systems

Stephen Michell
Maurya Software Inc
stephen.michell@
maurya.on.ca

Brad Moore
General Dynamics
brad.moore@
gdcanada.com

Luis Miguel Pinho
CISTER/ISEP
lmp@isep.ipp.pt

Tucker Taft
AdaCore
taft@adacore.com

Abstract

Previous work proposed the inclusion of parallelism in Ada to improve the use of multicore processors. This work proposed a model of Tasklets to carry the expressions of parallelism, and introduced the notion of executors to map the execution of Tasklets on hardware. In developing the model, a number of choices arise that effect how well the model will accommodate real-time systems. These choices become constraints which are examined and reasonable choices proposed for adoption in Ada.

1. Introduction

As the work on including the ability to use multiple cores to improve the execution time of apparently sequential algorithms has proceeded, there have been attempts to include real-time systems in the parallel computing domain. Mei et al [6] has proposed mechanisms to incorporate Java streams into the Real-Time Java specification, but came to the conclusion that modifications to the Real-Time Java specification were needed to permit such incorporation.

We have proposed ways to include concurrency in Ada's real-time framework [1][2][3]. [1] and [2] have proposed a set of capabilities for Ada to permit the parallelization of sequential code on multicore hardware using tasks, tasklets and executors. In the proposal, tasks carry the high-level scheduled algorithms and contain Parallel OPportunities (POPS) (such as large loops) that are parallelized and the work subdivided and executed in parallel by tasklets. Tasklets are executed by executors which are mapped to execution on the various cores of the hardware. In [2] and [3] we showed how executors are used to carry the execution of the tasklets.

In [3], this proposal was discussed by IRTAW 2015 and a general agreement was reached that the proposals provide a possible path to the provision of capabilities to support parallel programming on multicore computers. A summary of the discussion is contained in [4].

The proposals to IRTAW 2015 [3] identified a set of 13 open issues which needed closure in order to provide a comprehensive proposal to ISO/IEC/JTC 1/SC 22/WG 9 Ada Working Group for possible standardization for Ada. Some of the issues raised did result in firm recommendations, but others did not, largely because of the newness of the issues raised and the challenge in reaching closure on a large proposal.

This paper restates the open issues and makes recommendations for discussion and closure. Readers are requested to reread [2], [3] and [4] for technical details of the basic proposal from which these open issues arise.

2. Open Issues arising from the IRTAW 2015 submission

2.1 Changing task and protected object priorities (and other attributes)

As per the model, when a tasklet executes Set_Priority, it is the base priority of the task that is changed, affecting all tasklets of that task. The changes to the task state is simply the update of state in the task control block, but that change's influence on the executors executing the task are realized as the various tasklets reach a synchronization point.

If such changes happen while there is more than a single executor executing for the task, then there are a wide variety of behaviours which could occur that could lead to starvation of some of executors or the generation of exceptions. We recommend that task attribute changes only occur outside of POPs, and that it is either a bounded error or erroneous execution to set task attributes or to call the package *task attribute.* The issue is the setting of a task attribute in a POP can only be a bounded error or implementation-dependent if the implementation can detect that the operation is being done in parallel.

2.2 Timing Events

A timing event in the Ada Reference Manual [5], section D.15, is handled by a protected object, with a protected procedure called if the event occurs, and a protected procedure or entry used to handle the event. Care is needed to ensure that the presence of multiple tasklets does not result in multiple event creations, nor in multiple tasklets attempting to handle the same event. Are there mechanisms that can be used to prevent erroneous multiple accesses to timing events? Should timing events be restricted to outside of POPs?

2.3 Execution Time Timers

Execution time timers measure the amount of time that a single task (or group of tasks or interrupt routine) uses and notifies a handler if that time is exceeded. Under our proposal, the execution of a tasklet is reflected in the budget of its task. The overhead of managing the parallel update of the budget may make this unfeasible, except if larger quanta are used or budget updates are not immediate (which may lead to accuracy errors). Specific per core quanta may be used to address this issue.

2.4 Parallelizing inside interrupt/timing event handlers

Since interrupts and timing events are expressed as protected operations in Ada, and the proposed model allows tasklets to be spawned inside protected actions, this means that parallelizing the code handling an interrupt is allowed. Although it is not clear why this would be useful (as interrupt handlers are in principle short actions), it is also not clear if it should be forbidden.

One issue which is not yet determined is the relation of interrupt handlers with executors. Ada does not determine which run-time stack is used for the execution of interrupt handlers. It could be interrupt handler executor pool, or it could execute in the stack of the currently executing executor. In the latter case, parallelizing would imply using more executors from the same task.

Should the use of tasklets and tasklets be permitted inside protected operations that can be called from interrupt service routines? Should interrupt handlers have their specific executor pool?

2.5 Relation with Set_CPU /Get_CPU

The Set_CPU call (or the aspect CPU) is used to constrain a task to a single CPU within the execution domain to which the CPU belongs. If the programmer specifies a single CPU for the task it might mean that all executors of the task are pinned to this CPU, thus all tasklets would be executed in the same CPU. This could eventually be considered for the case where there is a single task in the CPU, and parallelization can be used for “software-based hyperthreading”. Or it be that any use of parallel syntax by a task that has the CPU aspect specified as something other than Not_A_Specific_CPU, should result in an implementation generating a compiler warning.

Does there need to be an identifiable call to lock a task to a given CPU but permit its tasklet to execute within the domain that contains the task? Such a call may be Set_Primary_Executor_CPU.

Alternatives would be for Set_CPU to lock only the first tasklet of the task or to add a new version of Set_CPU that assigns a set of CPUs to a task, instead of a single one. This would allow the restriction of the task to execute in a subset of the CPUs in the domain.

Get_CPU can return the CPU where the calling tasklet is being executed, or can be restricted to name the CPU of the parent task.

2.6 Tasklet stealing

As specified in [4] and [3], in the real-time model for parallel Ada, tasklets run-to-completion in the same executor where they started execution. Therefore, tasklets that have already started cannot be stolen, and parent stealing is disallowed (this also means that the main/first tasklet cannot be stolen because it has already been started). So, if No_Executor_Migration is specified, any tasklet after starting in a specific core will not leave that core.

Is this too restrictive, and if so, what is acceptable?

2.7 Distinguishing between number of allowed and active executors

The Ada Europe paper [2] introduced mechanisms to limit the number of executors per task or on a domain. This limit could be either on the number of available executors, or the number of simultaneously allowed active executors (this differentiation makes sense when executors block on tasklet blocking). Supporting both would also be an option.

Is there a preference for what mechanism is used to limit the number of executors?

2.8 Explicitly control of executors

If the programmer is able to explicitly specify the number of executors which are processing a specific POP, it is then possible for the programmer to use some sort of inter-executor synchronization to control the execution of the tasklets (e.g. by doing computation in phases inside a parallel loop). It is not clear if this should be allowed, and, if so, if a model based on language constructs or on a library should be used.

Is there a requirement to specify the number of executors to be used in an individual POP?

2.9 Tasklet minimum execution time

One of the important specifiable parameters affecting the separation into tasklets is the minimum execution time permitted for any given tasklet created by the compiler. This minimum time should be at least as great as the overhead of initiating and waiting for a tasklet (plus increased hardware contention due to parallel execution), to ensure that the critical path execution time for the overall task does not increase as a result of breaking a task into multiple tasklets. This could potentially be a compiler switch or a configuration pragma.

Is this a required capability, i.e. to specify the criteria for determining how many tasklets to create for a single POP, or program wide?

2.10 Relation with simpler runtimes

In principle it would be possible to implement a system with the proposal from [3], also adhering to a set of restrictions used for simpler runtimes such as Ravenscar (or some variant of it). Should we specify a parallel simpler model that can be used with these simpler runtimes? This also has relation with any discussion the workshop may that may have for other profiles than Ravenscar.

2.11 Other preemption models

With the introduction of the lightweight tasklet-based programming model (known as task-based programming model in other programming models), it is important to assess if new preemption models are of interest. In particular, the potential small computation effort of tasklets, as well as the fact that potentially variables exist that do not cross the tasklet boundary, it would be possible to implement a model of (limited) preemption only at tasklet boundaries (when tasklets complete). This could eventually reduce overhead and contention, improving efficiency and analyzability.

2.12 Applicability to high-reliability hard real-time systems

The hard real-time guarantees of applications executing with the proposed model need to be provided by appropriate timing and schedulability analysis approaches. Although extensive works exist in these topics, and the model described in this paper is fit to be used in these works, it is still not possible to know the feasibility of applying these methods for parallel systems. The complexity and combinatorial explosion of interferences between the parallel executions may prove the timing analysis of parallel computations to be unfeasible. Moreover, the analysis requires determinism (and knowledge) of the specific contention mechanisms at the hardware level, something which is more and more difficult to obtain.

Is research into the use of the POP/tasklet/executor model stable enough to be used in high reliability real-time systems?

3. Conclusions

The aggregation of the choices made in this paper and discussed at IRTAW 18 – 2016 should progress the ability to write parallel Ada programs that can control real-time systems.

Bibliography

[1] Luís Miguel Pinho, Brad Moore, Stephen Michell, *"Parallelism in Ada: Status and Prospects*", Proceedings of the Reliable Systems Conference/Ada Europe 2014, Springer LNCS 2014

[2] Luís Miguel Pinho, Brad Moore, Stephen Michell, Tucker Taft "*An execution model for fine grained parallelism in Ada*", Proceedings of the Reliable Systems Conference/Ada Europe 2015, Springer LNCS, 2015

[3] Luís Miguel Pinho, Brad Moore, Stephen Michell, Tucker Taft, "*Real-Time Fine-grained Parallelism in Ada*", International Real Time Ada Workshop 2015, ACM Ada Letters, ACM, New York, NY. 2015

[4] Luís Miguel Pinho, Brad Moore, Stephen Michell "*Session Summary", International Real Time Ada Workshop 2015*", ACM Ada Letters, ACM New York, NY, 2015

[5] ISO IEC 8652:2012 "*Programming Languages and their Environments – Programming Language Ada*", ISO, Geneva, Switzerland 2012.

[6] Mei, HaiTao, Gray, Ian, Wellings, Andy, "*Integrating Java 8 Streams with the Real-Time Specification for Java*", Proceedings of the 13th International Workshop on Java Technologies of Real-time and Embedded Systems, ACM, New York, NY, 2015.