CISTER
Research Center in
Real-Time & Embedded
Computing Systems

Technical Report

Real-Time Programming on Accelerator Many-Core Processors

Stephen Michell

Brad Moore

Luis Miguel Pinho*

*CISTER Research Center

CISTER-TR-131112

2013/11/10

Real-Time Programming on Accelerator Many-Core Processors

Stephen Michell, Brad Moore, Luis Miguel Pinho*

*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

Multi-core platforms are challenging the way software isdeveloped, in all application domains. For the particular case ofreal-time systems, models for the development of parallelsoftware must be able to be shown correct in both functional andnon-functional properties at design-time. In particular, issues suchas concurrency, timing behaviour and interaction with theenvironment need to be addressed with the same caution as for thefunctional requirements.

This paper proposes an execution model for the parallelization ofreal-time software, based upon a fine-grained parallelism supportbeing proposed to Ada, a programming language particularlysuited to the development of critical, concurrent software. Wealso show the correctness of the proposed model in terms ofsatisfying constraints related to execution order and unboundedpriority inversions.

Real-Time Programming on Accelerator Many-Core Processors

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

Brad Moore
General Dynamics
Canada
brad.moore@gdcanada.com

Luís Miguel Pinho
CISTER/INESC-TEC, ISEP
Portugal
lmp@isep.ipp.pt

ABSTRACT

Multi-core platforms are challenging the way software is developed, in all application domains. For the particular case of real-time systems, models for the development of parallel software must be able to be shown correct in both functional and non-functional properties at design-time. In particular, issues such as concurrency, timing behaviour and interaction with the environment need to be addressed with the same caution as for the functional requirements.

This paper proposes an execution model for the parallelization of real-time software, based upon a fine-grained parallelism support being proposed to Ada, a programming language particularly suited to the development of critical, concurrent software. We also show the correctness of the proposed model in terms of satisfying constraints related to execution order and unbounded priority inversions.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features – *Concurrent programming structures.*

Keywords

Multi-core; real-time; programming language; Ada; dispatching domains

1. INTRODUCTION

The importance of parallel computations has grown significantly as the trend to use multi-core and many-core platforms spreads to new application domains, and parallelization is the only means to continue to be able to support increasingly complex software in hardware architectures which no longer evolve to faster speeds. We are thus witnessing an immense growth in parallel programming methodologies and models, put forward to address the inherent complexity of developing reliable software on these platforms.

HILT'13, November 10–14, 2013, Pittsburgh, PA, USA.

ACM 978-1-4503-2467-0/13/11…$15.00.
http://dx.doi.org/10.1145/2527269.2527270.

This is the case even for domains which are traditionally more conservative in evolving to new hardware or software models, such as real-time applications. In this domain, systems are built in such a way as to guarantee at design time both functional behaviour, and timing behaviour in addition to other constraints. These systems present significant challenges to the development of applications, as they require the guarantee of predictable timing behaviour as they interact with, and react to, the external environment.

To meet these challenges, models and technologies incorporate intrinsically the notion of time, priorities and concurrency. Programming models therefore need to be based in languages which integrate these notions, and any solution to the development of parallel software must adhere to the same requirements. Any approach which considers parallelization must be rigorous and amenable to verification.

Within this context, it is necessary to address the integration of fine-grained parallelism in the Ada programming language [9]. Ada's sound specification of concurrency is based on the direct support for tasks, supporting coarse-grain multi-core programming.

A fine-grain parallel model for Ada has recently been proposed [13], based on the notion of tasklets, which are non schedulable computation units (similar to Cilk [8] or OpenMP [12] "tasks"). Tasklets may be executed by a pool of worker tasks.

This paper starts from this existent work [13] to propose a model of execution for the parallelization of real-time software based upon a separation of domains for the execution of the application tasks and the execution of their parallel components. The tasks of the application are executed in a single core, while the remaining cores are used as accelerators, to execute parallel code blocks on behalf of the application tasks. We also show analytically that the model can preserve important properties of such systems, such as avoidance of unbounded priority inversions, deadlocks[4] and race conditions. Further work is needed to include analysis of timing properties, including issues such as contention for common busses and shared global state.

This work contrasts with other work on scheduling real-time tasks in multi-core systems (in [7] the reader will find a survey of the major directions being followed and approaches being proposed). In contrast to other proposals for parallel real-time tasks [6][11][17][2]) this approach tends to be much simpler and maintains the structure, methodologies, code, and verification techniques currently being used for real-time systems while providing extra processing power when needed in a less intrusive way.

Presently Ada does not have the necessary syntax and libraries to support the proposals given here. Paraffin [14][15] implements a set of generics which can already be used to achieve the results of [13], but require more explicit rearrangement of loops and function calls than can be done with dedicated syntax.

The paper is structured as follows. The next section provides the required background, which is then further detailed in Section 3 for the case of real-time programming models. Section 4 then provides some definitions, while Section 5 describes the ongoing work to address parallelism within Ada. Section 6 describes the model of computation and shows its correctness. Section 7 is conclusions and Future work. An annex is also included to show a complete example.

2. BACKGROUND

2.1 Brief Summary of Real-Time Systems

Real-time systems are systems in which some or all events in the system must result in the correct response within a bounded fixed time interval [4]. Real-time systems are usually divided into 2 domains. Soft real-time systems are ones where some calculations can exceed expected time (i.e. the system can accommodate some slippage). Hard real-time systems are characterized by the property that all of the bounds on calculations are absolute – i.e., a late answer is as bad as no answer or a wrong answer, and an early answer can also be problematic. Some examples of real-time systems are industrial processing systems, transportation systems such as airplane control, train braking systems and communication, such as network controllers for wifi and ethernet controllers.

Real-time programming presents significant challenges even for single-CPU programming. It is event driven, feedback-oriented, time critical in that:

- important calculations have “best before” (for soft real-time), “must before” and “not before” (for hard real-time) times; and
- they are highly concurrent with a minimum of interrupt routines, event routines, and likely tasks (threads) to handle natural concurrency that occurs.

In order to work with such systems, a notion of priority (imperativeness) is an important concept to ensure that the most imperative calculations are done in preference to other calculations, and to maintain schedulability (i.e. to meet the hard real-time deadlines for the critical tasks). Priority is so important that all CPU's have a notion of priority embedded in the hardware to control which classes of hardware get preferential service when conflict exists, and all operating systems and run times have a notion of software priority that extends the hardware priority right down to the “idle task”.

A large real-time program may have:

- Interrupt handlers delivering external I/O to the system;
- Clock handlers managing time and making decisions about time;
- Event handlers working with “softer” events with high software priority but low hardware priority;
- Worker tasks interacting with sets of interrupt and event handlers;
- Calculation engines maintaining real-world properties, such as velocity, position, acceleration; and
- Managers managing system state, task states and modes.

Such a program uses time, priority and programmed state to schedule and manage the interaction of the multitude of system components to keep the system functioning safely, securely and correctly. The precise interaction of many components is vital to that correct functioning [10].

2.2 Introduction of Parallel Computation

Like all other domains, the real-time computation domain has been demanding more and more CPU power to solve its problems, as new applications such as vision systems are introduced, very large data sets are processed, and more fidelity is demanded of calculations. To date, almost all real-time systems have resided on single CPUs [10], or if using multiple CPUs, have highly restricted the ways that they can interact so that the constraints of the real-time domain can be satisfied. The stagnation in the growth of CPU speed means that real-time systems must seriously consider how they can make use of multi-core and many-core systems.

Even real-time systems sometimes need a bigger calculation window than can be delivered by a single CPU in the time constraints available, such as processing vision system frames to detect and identify obstacles. Many calculations are approximate, and increasing the processor power increases the fidelity of the calculation (and permits some complex algorithms that may not be permissible on a single CPU). Dividing the calculation across multiple cores permits increased calculation efficiency, provided that it can integrate into real-time systems constraints.

To satisfy the constraints, one must verify before the system is fielded that it will operate correctly always, that all eventualities have been considered and that all corner cases have been considered. Typical approaches are static verification-based analysis at design time that may include formal verification, Worst Case Execution Time (WCET) analysis, Worst Case Response Time (WCRT), measurement, modelling, and extensive human review.

Multi-core systems dramatically change the way that components interact with each other and with the external world in that:

- Tasks executing on separate cores often have very different access to memory, network, busses and registers, interfering with each other via those covert resources;
- Some hardware or interrupts may not be available on all CPU's;
- Memory access may be orders of magnitude slower, depending upon locality of reference and cache issues; and
- Objects such as spin locks may give (unintended) preferential treatment to some cores or tasks. So, although multi-core systems are assumed to be homogenous, bottlenecks such as memory bus speeds and number of cores trying to access common memory are leading to non-homogeneous systems, such as CC-NUMA.

The items discussed above simply add to the complication that multiprocessing systems already present to any previously sequential algorithm.

3. DEFINITIONS

Task – in Ada is both a unit of design mapping of concurrency and unit of concurrent execution. In terms of execution model, tasks are similar to threads in POSIX. In fact they are typically mapped to OS threads unless accompanied by a specialized runtime.

Application Task (*AT*) – A task that is declared by the programmer. Application tasks are declared to be in a single domain that, in this example, execute on a single processor. All real-time *AT*s have a unique priority that is higher than any non-real-time application task. An application task with priority *i* is labelled AT_i.

Parallelism OPportunity (POP) – The place in an application where sequential code can be executed by parallel units (but is not mandated to be executed in parallel).

Dispatching Policy – The policy which governs the allocation and scheduling behaviour of tasks on specific processors.

Dispatching Domain – a named set of CPUs within which tasks assigned to that domain are scheduled according to the dispatching policy for that domain.

Worker Task – A task that belongs to a task pool and executes parallel code on behalf of an *AT* at a POP. All *q* worker tasks $WT_i(1..q)$ of a dedicated task pool service a single application task AT_i. In this model, all WT_i execute on a specific dispatching domain.

Spawn – To create an object representing a parallel code unit and submitting it to be processed by a worker task.

4. PARALLEL ADA MODEL PROPOSAL

Having identified the lack of direct support for use of fine-grained parallelism in Ada, we recently proposed a mechanism that the programmer can use and precisely control fine-grain parallelism in loops and subroutine calls [13]. The basic mechanism leverages from the new Ada (2012) aspects syntax to permit an aspect "with parallel" to suggest to the compiler that work be parallelized across processors, together with a set of library package interfaces to support user-defined or user-augmented fine-grained parallelism.

In order to effectively describe the new concurrent behaviour, this work introduced a unit of parallelism called a "tasklet" (similar to the Cilk concept of task). Unlike Ada tasks, tasklets are not nameable or directly visible in a program. A tasklet carries the execution of a subprogram or of a code fragment (such as part of a "for" loop) in parallel with other tasklets executing the same code fragment (with different state) and possibly in parallel with other tasklets executing code fragments from other Ada tasks.

This proposal incorporates logical units of parallelism in the semantic model of the language, allowing potential parallelism to be expressed both for task/control parallelism, where the control structures of the code (e.g. loops and subprograms) which are amenable to parallelization are identified, and for data parallelism where data structures (arrays or records) are potentially processed in parallel, based on the notion of a logical unit of parallelism.

The programmer identifies these potential parallel opportunities in the code, guiding the compiler in generating code that creates the logical tasklets. During execution, the runtime executes the tasklets in parallel, if the load of the system allows it. These tasklets may actually not exist as runtime identifiable objects (it depends on actual compiler and libraries implementation) but exist as logical entities of the program. Note that this model also allows integrating vectorization, as logically the compiler can decompose parallel processing in several tasklets which are directly executed in hardware.

There are two types of tasklets. The first is created by the compiler when it can determine that an operation can be parallelized and submitted to multiple processors, and hence is not visible to the programmer. Usages of this could include default initialization, assignment of values to arrays of records, copying large structures using the Ada assignment operator, or compiler identifiable parallelizable loops, as shown in Figure 1.

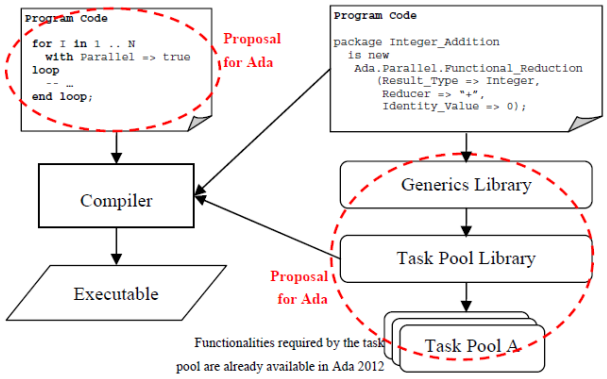

Figure 1 - Proposal for Tasklets in Ada 2012 [13]

The second tasklet type is created by the compiler upon instruction from the programmer, who uses explicit syntax to guide the compiler and runtime in deciding how much parallelism should be provided (e.g. by "chunking"), and whether the tasklets should process work bundles using a work-sharing, work-seeking[1] or work-stealing model. The syntax includes the use of aspects on subprograms and loops.

Tasklets are meant to augment, not replace tasking as the unit of concurrency. Programmers will declare an intent that code fragments be executable in parallel, but do not necessarily concern themselves with the details of the parallelism itself, or how it interoperates with other tasks. They can, however, as we will show, extend the syntax and add runtime mechanisms to achieve specific concurrency behaviours.

Each tasklet behaves as if it were executed by a single Ada task that was explicitly created for the execution of the tasklets and terminated immediately after execution of the code fragment. In order to make tasklets integrate smoothly with the tasking mechanism, priority, and real-time bounds, tasklets can be executed by worker tasks. The Ada tasking model is then used to express the concurrency since tasks in Ada already have a computationally sound model that addresses the issues (i.e. proven support for real-time systems) raised here. To not base this

[1] Work seeking is similar to work stealing, but the worker with extra work participates directly in process by frequently checking to see if idle workers are available and offering work directly to the idle worker. We believe (but have not confirmed) that work seeking is safe from priority inversions. See [14][15].

concurrency on tasks puts at risk the priority model of Ada for any real-time programs.

In a generic system the compiler is free to create as many tasks as it needs to execute tasklet code, and any such tasks that execute tasklets are not visible to application code. This can be augmented with user-defined pools of tasks to execute tasklet code by matching the interface that the compiler exposes; a set of packages and generics to let the pool provide the service.

4.1 Syntax

The most obvious opportunities for parallelism are the subprogram call and the loop. For a subprogram call one can declare to the compiler the desire to execute the subprogram in parallel with its caller by writing

```
A_Value :=  Some_Function(Value1) with Parallel
          + Some_Function(Value2);
```

Here the subprogram `Some_Function(Value2)` will be executed in parallel with `Some_Function(Value1)` and the caller waits at a point before its return value is consumed by the "+" operation [2].

For loops the basic syntax is

```
for I in Integer 1..N with Parallel loop
   -- some calculation on I
end loop;
```

or if we wanted to control chunking of the algorithm (say to split among C cores using work-sharing)

```
for I in integer 1..N
      with Parallel, Chunk_Size => N/C
loop
   -- some calculation on I
end loop;
```

More details can be found on [13] on the syntax and how issues such as managing complex calculations that need reduction, identity values, and other tuning parameters, are addressed.

Using the same model, data-level parallelism can be supported by allowing the notion of potentially parallel data types, where operations can be parallelized (operations in data types are actually subprograms). These data types can have the operations overridden by specifying "with Parallel => true" and what would be the parallelizable units.

In this model, the compiler is free to optimize and use SIMD hardware when available (as it already can), but may also generate logical tasklets, within the same generic model as above, and share the same task pools.

For instance, the following example describes a simple parallel array, which the compiler can vectorize in some architectures:

```
-- this can be vectorized
type Par_Arr is array 1 .. 100 of Integer
   with Parallel => true;
function "+"(Left, Right: Par_Arr) return Par_Arr
   with Parallel_By_Element =>
              function "+" (Left, Right: Integer)
                           return Integer;
```

For more complex data types, the model would be the same:

```
type My_Type is record
   -- whatever
end record;

function "+"(Left, Right: My_Type) return My_Type;
-- implements addition of two My_Type objects

type My_Type_Arr is array 1 .. 100 of My_Type
   with Parallel => true;

function "+"(Left, Right: My_Type_Arr)
   return My_Type_Arr
   with Parallel_By_Element =>
              function "+" (Left, Right: My_Type)
                           return My_Type,
         Chunk_Size => 50;

  -- any "+" operation on My_Type_Arr can be
  -- parallelized by compiler
  -- even automatically vectorized when possible

function "*"(Left, Right: My_Type_Arr)
   return My_Type is  -- this cannot be
                      by_element
begin
   -- implement dot product with parallel loop
end "*";
```

Aspects could be allowed on the statement of execution to control the level of chunking to perform, either in the specification of the type, or in the actual code performing the operation:

```
My_Arr_1, My_Arr_2: My_Type_Arr;
-- ...
... My_Arr_1 + My_Arr_2 with  Parallel,
                              Chunk_Size => 10;
```

Assignment into a parallel data type could be automatically parallelized by the compiler using the same context as the parallel operation being performed (or freely if no other operation was being performed). Note that for expressions, a "with Parallel" gives instructions to the compiler to parallelize as much as possible. If the programmer wishes finer control of the parallelization of the operations and subprograms she may need to rewrite the expression. Further research is needed on the best suitable approach for this finer control.

4.2 Facilities for Programmer-defined Task Pools

The examples given above are enough to have the compiler generate a set of parallel dispatches to tasks or processors to execute their work component and return partial results for final reduction. Many situations exist, however, in which more control is needed, such as when the priority of the application task requires a set of worker tasks with the same priority. Therefore, there are times when the application needs to define its own task pool, and to have the compiler invoke these explicit worker tasks. In order to integrate the worker tasks, there needs to be an interface between the worker tasks and the application program.

This interface is provided by the addition of package `Ada.Parallel` together with a set of child packages to the Ada runtime library to support user-defined or user-augmented fine-grained parallelism. This library contains interfaces to mechanisms (among other things) to support the creation of task pools to permit the dispatching of fine-grained parallel work to

[2] We also permit the "with Parallel" aspect to be placed on the subprogram specification, letting all calls to execute in parallel with the caller. In this case one would need "with Parallel => False" to prohibit it from happening.

user-written pools of worker tasks, and parallel manager objects to control exactly how the parallel work is to be dispatched and controlled. The details of this work can be found in [13] and [16].

The `Ada.Parallel` interfaces also include generic packages to implement function reducers and loop iterators that are shown in [13] as well as work plans to permit work to be processed by a much smaller number of processors (and tasks) than there are work packages to be done. For example, load balancing may improve performance in some situations but not in others. Thus, user-defined task pools can be created to satisfy specialized dispatch conditions, such as a bounded set of worker tasks, or set priority for all worker tasks, or even a set of Ravenscar [3] compliant tasks for very specialized runtimes.

Let us return to the basic syntax to invoke a tasklet, `with parallel`. In order to access the user-defined task pools, we need more machinery. Here detailed aspects can be used for that purpose.

```
for I in 1 .. 1000
      with Parallel          => True,
           Worker_Count      => 10,
           Parallel_Manager  =>
                     WSL.Work_Sharing_Manager,
           Task_Pool         => My_Worker_Pool,
           Chunk_Size        => 100,
           Priority          =>
                     System.Priority'Last,
           Load_Sensitive    => True
loop
   --...
end loop;
```

In the example above, `WSL.Work_Sharing_Manager` [4] is a user-defined package that is an instantiation of a generic child package of `Ada.Parallel` [16].

An important point to note is that the communication between the application task that contains the POP and the worker tasks that execute the tasklets is always via an Ada protected object(s). Such protected objects obey the ceiling priority protocol [4], which means that priority-based scheduling is supported by sound scheduling theory. It is also important to note that worker task pools can be assigned to domains that match characteristics of the hardware [3], whether it be a few multi-cores in a homogenous environment or a many-core system without shared memory.

5. THE ACCELERATOR MODEL – A POTENTIAL FOR REAL-TIME MULTI-CORE SYSTEMS

Building on this existent work, this section presents a model for real-time programming for multi-core and many-core processors, using available cores as accelerators of the real-time application tasks. We also present a couple of examples of alternative constructions of a real-time program that follows our model and shows how a real-time analysis of such a program could be undertaken.

[3] The Ravenscar Tasking Profile is a highly restrictive subset of Ada tasking with fixed priority tasks that can only be statically declared and that communicate by protected objects that can only have a single entry with a single queue element.

[4] WSL refers to Work_Sharing_Loops.

For our real-time system model, we propose a system where all application tasks execute on a single core using the priority mechanism and communicate with each other and with interrupts and events via protected objects that obey the ceiling priority protocol (e.g. with FIFO spinning [5] or other applicable protocol). We assume that every application task has a unique priority to express its degree of urgency, and that the priority of all real-time tasks is higher than the priority of non-real-time tasks. For normal inter-task interactions, each protected object shared by two or more tasks has a priority equal to the highest priority task that can call a protected subprogram or entry of the object [5].

For our system, we assume that there are P tasks with unique priorities $1..P$ (as in Ada higher numbers indicate higher priority), called application task $1..P$ and denoted $AT_1 .. AT_P$. Furthermore, for the examples below, we assume that task AT_1 and AT_2 need more computational power than is available from the first core, but we have M (in the examples below M=7) additional cores available. We now show two different configurations for distributing the work and show how the real-time properties of the program are preserved for each configuration.

5.1 Mapping 1 – Independent Worker Dispatching Domains

The first mapping (Figure 2) is used when each application task AT_i ($\forall i \in 1..P$) is assigned a non-overlapping subset of the M accelerator cores, within a AT_i-specific worker dispatching domain. In the example, we create three dispatching domains, AD, WD_1 and WD_2, where AD contains core 1 upon which all AT tasks execute. Dispatching domain WD_2 contains 3 cores and has a task pool containing 3 (or more) [6] worker tasks $WT_2(1..3)$, each at priority 2, the same as AT_2. The communication between AT_2 and $WT_2(1..3)$ occurs via work manager protected object 2 ($WMPO_2$) with a ceiling priority of 2. Similarly, AT_1 is supported by dispatching domain WD_1 consisting of 4 cores and a task pool containing 4 worker tasks $WT_1(1..4)$, and communication between AT_1 and $WT_1(1..4)$ occurs via $WMPO_1$.

When AT_i (i=1 or 2 in the example) dispatches work to its worker tasks, e.g. using

```
for j in 1..N with Parallel => XXX loop,
```

AT_i calls a protected procedure of $WMPO_i$ to schedule up to N worker tasks and then spin-waits for a final result. Worker tasks $WT_i(1..q)$ iteratively collect a work packet, calculate a result, return the result, until all work packets have completed and a final answer can be returned. At this point, AT_i is unblocked and

[5] The model and arguments provided herein rely upon the fact that all application tasks execute on a single CPU and rely upon the ceiling priority protocol to let correctness calculations be performed. If work is distributed to worker tasks in other cpu's, and these worker tasks cannot communicate or share variables with worker tasks from other applications, then they have no dependency with each other outside of the application tasks.

[6] It may permissible to create more tasks in a single pool than there are available CPUs. All can be dispatched by a "with Parallel" call and will sort themselves out to do the calculation. Some mechanisms, such as work sharing, may have better efficiency when there are more tasks than CPUs.

returns with its result. In our model, parallelizable code cannot share resources with other application tasks. This means that worker tasks will not share resources between domains.

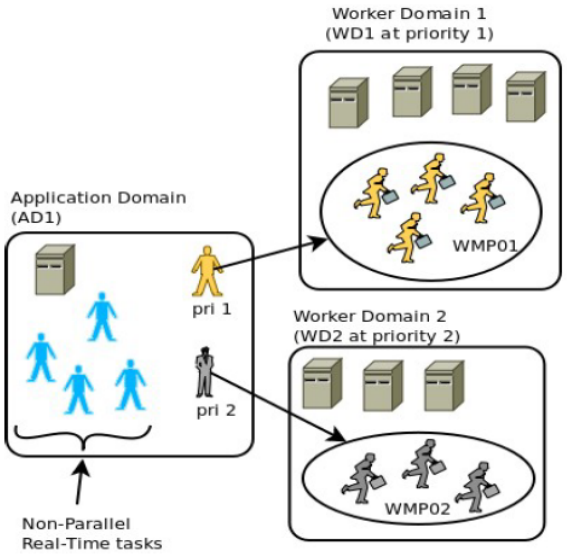

Figure 2 - Real-time Tasks Parallelizing in Dedicated Worker Domains

The challenge is to show that real-time schedulability and analyzability of the application are preserved. For example, we need to show an absence of priority inversion where for priority(*i*)>priority(*j*), an AT_i is ready to execute but cannot because an AT_j is executing (excluding cases where AT_j is calling a protected operation with priority >priority(*i*)).

By priority rules, AT_j can only execute when all higher priority tasks are waiting on a suspend, block or delay operation. It may be interrupted while it is doing its work, but cannot interrupt any $AT_i(3..P)$ except when calling a protected object to communicate with some AT_i. Without loss of generality, in the example, AT_2 can be interrupted when scheduling work for $WT_2(1..3)$ because $WMPO_2$ has priority 2, and can be interrupted by all higher priority tasks.

When AT_2 has dispensed all of its work, and $WT_2(1..3)$ interacts with $WMPO_2$, all interactions occur within the processor of each WT_2 task and therefore do not impact AT_i, i>2. Even when the work has completed and AT_i must do some computation in $WMPO_2$, the priority is such that higher priority tasks always get service.

The argument made above for AT_2 applies also to AT_1, except that AT_2 is now added to the set of application tasks that cannot be blocked by AT_1 or its $WT_1(1..4)$. Now however, we must also consider interactions between $WT_2(1..3)$ and $WT_1(1..4)$. There is none, because each belongs on independent dispatching domains, there is no sharing of data between parallel opportunities in the AT_i and AT_j, and even the execution of *WMPO* code called by worker tasks is independent because of the independent domains.

5.2 Mapping 2 – Shared Worker Domain

The second mapping (Figure 3) considers the case where we only create a single worker domain of *M* cores and map all *WT* task pools to this *WD* domain. We further permit any *AT* requiring tasklets to be somewhere in the range of tasks, not necessarily only the lowest.

In order to accomplish their work, application tasks make liberal use of tasklets implemented by pools of worker tasks executing in the single dispatching domain (*WD*) that contains all remaining processors. For this example we change the previous scenario by placing worker tasks into a single worker task domain, but here we permit the notion that application tasks at any priority can use tasklets. We continue to show for the example only AT_1 and AT_2 using worker tasks but the analysis and verification is generalized for all combinations.

In this case, when 2 or more application tasks compete for resources and they are at different priorities then all worker tasks for each task will compete on the multi-core domain with the same priority rules, meaning that all tasks for the highest priority work will receive computing resources, and lower priority worker tasks will only proceed when there are more available cores in *WD* than there are higher priority tasks left to execute.

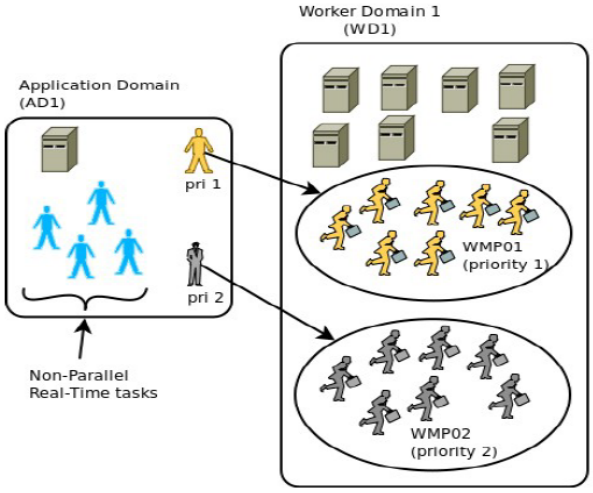

Figure 3 - Accelerator Example - Single Worker Domain

For this model, we assume that there are *N* cores and *M*=*N*-1 cores in *WD*. Without loss of generality we consider 2 arbitrary application tasks, AT_i and AT_j. We further assume that $priority(AT_i) > priority(AT_j)$. Each AT_i (and AT_j) has a set of dedicated worker tasks $WT_i(1..Q<N-1)$ and $WT_j(1..R<N-1)$ and each communicates through its dedicated work manager protected object *WMPOi* and *WMPOj*. We further assume that

$priority(WMPO_i) = priority(AT_i) = priority(WT_i)$,

and similarly for AT_j, WT_j and $WMPO_j$. We assume that AT_i and AT_j spin-wait on completion of worker tasks. We also discuss implications when $AT_{i,j}$ are free to block.

We claim that this scenario does not introduce priority inversions (i.e. it never happens that AT_i or any worker task $WT_i(q)$ is ready to execute but cannot because either AT_j or $WT_j(r)$ occupies a processor) [7].

[7] The notion of priority inversion can only be applied on a single dispatching domain. If a task T_i, i>j is ready to run in domain D_1 and T_j is executing domain D_2, this is a design decision, not a priority inversion.

5.3 Verification

Assumptions

- All tasks, by the nature of Ada protected objects when supported by the Ada real-time annex, and configured to use the ceiling priority protocol, follow the ceiling priority rules. This means that all communication between application tasks happens through protected operations at a priority higher than the highest priority task that uses them. This also means that all nested interactions occur at increasing levels of priority. Priority rules guarantee deadlock freedom and absence of unbounded priority inversion, in a single core [4].
- All real-time application tasks (*AT*) have a unique CPU priority and we label each AT_i by its priority *i* and the lowest priority real-time task has priority greater than the highest priority non-real-time task. This is to guarantee that they complete their task in bounded time, which accounts for the total time taken by higher priority tasks.
- All real-time AT_i execute in a dispatching domain that consists of a single core, relying upon priority to give the most urgent task the processor, and relying upon tasklets to perform CPU-intensive calculations while application tasks spin-wait for results. We nevertheless present a case (8a) where this is relaxed and application tasks may suspend.
- Each AT_i that requires additional computational power has a bounded dedicated pool of worker tasks *WT* to implement the tasklets. It also has a dedicated protected object (or set of protected objects in the case of a Ravenscar implementation of this model). We label *WT* for AT_i as $WT_i(1..q)$ and they have the same priority in *WD* as AT_i has in *AD*.
- AT_i and AT_j do not share memory resources except via protected objects. Also $WT_i(1..q)$ and $WT_j(1..r)$ do not share memory resources.
- Spawning tasklets to worker tasks is not allowed in protected objects and no potentially blocking operations are called from any WT_i. Note that this limits considerably the model and in particular prevents the use of blocking mechanisms when implementing nested or divide-and conquer parallelism. It provides a safer model for analysis at the expense of restricting parallelism. In domains where it is acceptable, these restrictions may be relaxed to allow for instance workers to spin-wait for the results of another worker, which would allow more efficient and expressive parallelism.
- All $WT_i(1..q)$ execute in a single dispatching domain that includes all of the remaining cores after the allocation of the application task core. The model is independent of WT_i being allowed or not to migrate within the cores of the domain. However, the analysis focuses on the case where WT_i tasks are statically assigned to cores.
- Non-real-time priority tasks may share the application task domain, if already foreseen in the application, but will not share the worker task dispatching domains. These tasks will execute with priorities lower than those of the real-time application tasks hence will not interfere with the progress of work for these tasks. Any communication between real-time and non-real-time tasks needs to be performed in a controlled and correct way, as it would already need to be (independently of the model here proposed), and cannot occur while real-time tasks are in a parallelized region.
- The worker task domain manager (*WTDM*) for each application task contains the protected object(s) $WMPO_i$ used for communication between AT_i and $WT_i(1..q)$. This protected object has the same priority as does the AT_i and WT_i that communicate through it.
- The blocking model for application tasks waiting for replies from worker tasks is spin waiting at the priority of AT_i.
- All WCET calculations for each application task and worker tasks are determined as usual and include preemptions, migration, communications time, cache misses, bus contention.

Claim

No priority inversion is introduced by the model – i.e. there will be no task AT_i ready to run with work with no available core while AT_j, $i>j$ and *priority(i)>priority(j),* is executing. Similarly there will be no task WT_i ready to run with no available core while task WT_j is executing.

Proof: Break into cases.

Case 1 – application task AT_i is executing and has not initiated any work for $WT_i(1..q)$. By priority rules on the single core, task AT_j cannot be executing.

Case 2 – application task AT_i is calling the work management PO ($WMPO_i$), setting up the Q<N-1 work items for workers. Task AT_j is on the same core as AT_i and cannot be executing by priority rules.

Case 3 – task AT_i is spin-waiting on the return of results. Task AT_j cannot commence execution.

Case 4 – Task AT_j is executing but has not reached a Parallel Opportunity and AT_i is resumed. AT_j is preempted and AT_i executes, spawns its worker tasks, collects results, and finishes. AT_j then completes execution, scheduling its worker tasks, collecting results and completing.

Case 5 – AT_i does not schedule to execute in this scenario and AT_j wakes up, calls $WMPO_j$ and schedules $WT_j(1..r)$ on *WD*. The WT_j complete, return values and AT_j proceeds back to a suspend state.

Case 6 – AT_j executes, initiates worker tasks $WT_j(1..r)$, then spin-waits for its workers to complete. While AT_j is spinning, AT_i commences execution, preempts AT_j, initiates $WT_i(1..q)$ and spins waiting on results. In a shared domain, $WT_i(1..q)$ will preempt some or all $WT_j(1..r)$ and all proceed to completion, releasing AT_i to finish its calculations. While AT_i is finishing its calculations, $WT_j(1..r)$ workers have resumed. Once AT_i completes, then AT_j resumes spinning for its results, which may or may not already be there.

Case 7 – AT_i and AT_j have initiated worker tasks and AT_i is spin waiting for a result with AT_j preempted still in the protected object $WMPO_j$. $WT_i(1..q)$ complete, with the last one releasing AT_i. AT_j continues to be preempted on $WMPO_j$ waiting for the completion of $WT_j(1..r)$. $WT_j(1..r)$ execute while AT_i executes on the application domain core. If some or all $WT_j(1..r)$ complete before AT_i finishes, they try to access a protected procedure of $WMP0_j$ to deposit their results, but cannot acquire the protected object, since AT_j has not released it. After AT_i finishes execution, then AT_j exits $WMP0_j$ and spin-waits on the results from $WT_j(1..r)$. $WT_j(1..r)$ acquire $WMP0_j$

to deposit their results. Once all WT_j *(1..r)* complete, then AT_j is released.

Case 8 – same as case 6, except that $WT_i(1..q)$ do not use all cores, or as $WT_i(1..q)$ complete, cores are released and all $WT_j(1..r)$ complete before all $WT_i(1..q)$ complete. At this point AT_j is freed the next time that it checks for completed work, but the spinning of AT_i does not let this happen until all $WT_i(1..q)$ complete and AT_i completes and blocks for the next release. AT_j then continues.

If we permit blocking by AT_i (i.e. self-suspend waiting for $WT_i(1..q)$ to complete instead of spin-waiting), then the following additional cases exist (Note that the system will not be ICPP compliant but some real-time systems analysis permits blocking by tasks in more than one place):

Case 8a – same as Case 8 except that AT_j is free to execute upon release as AT_i is blocked waiting on the WT_i to complete. AT_j may or may not finish its iteration before AT_i is released. Spawning tasklets inside a protected object is not allowed, thus we are guaranteed that AT_i is not using any resources when suspending.

The discussion and analysis above only shows that there is no structural contention that could cause deadlock or introduce priority inversions. It does not address platform-specific issues such as bus contention, DMA contention, cache-flush/cache-miss issues, or local memory/global memory access times: all of which are critical issues for real-time systems. Such analysis is the subject of several current (and future) research works.

The discussion also does not discuss the role of high priority real-time tasks that share the application domain but do not use tasklets. These tasks are the highest priority tasks that interact with the external environment but consume few processor cycles (e.g. Interrupt handlers). If such a task AT_i preempts an executing application task that has not yet called its $WMPO_i$, processing happens normally. If it is preempted while a call to $WMPO_i$ is in progress, the priority rules mean that the call waits until the preempting operation completes, as would happen in a single CPU system. If the higher priority task executes in the application domain while WT_i are executing, AT_i is preempted so no interruption occurs, and the processing of any returned values waits until the higher priority task completes.

The model above was chosen specifically to closely match the existing knowledge base and verification approaches for real-time programs based on single CPUs and priority to control scheduling. The extension of the single core to a worker domain that matches the main single core extends the priority model to the worker tasks. The choice of a dedicated worker task protected object for each application task, and setting its priority to be identical to that of its application task guarantee that higher priority tasks will always get the computing resource upon demand, even at the expense of blocking possible execution cycles of worker tasks on different domains. We note that the goal in real-time systems is not to use the algorithm that extracts the most available work from the cores, but to use algorithms that can be verified to satisfy the time bounds as well as generate correct calculations. Furthermore, this approach allows not breaking the Ravenscar model in the application tasks single core, whilst allowing accelerating computation in worker cores.

Other systems exist that can take advantage of the simple model presented here (e.g. for runtime simplicity) but that may not have the same strict requirements on static analysis. For those systems we may want to remove some of the assumptions presented in section 5.3, which may lead to a more efficient and balanced utilization of the system resources. This is outside of the scope of this paper, and subject of future research.

Other mappings are clearly possible and supported by the mechanisms that we propose. We have already shown a mapping that dedicates a worker domain for each worker task needing such a resource and one sharing a domain. In particular, where it is known that processor layouts give preference to certain couplings of cores, then these cores can be combined into dispatching domains with work allocation managers defined that optimize such couplings in the configuration portion of the program, and invoked using the straightforward, analyzable methods shown here.

The question naturally arises as to the applicability to other languages. There are real-time operating systems and kernels, as there are other languages and add-ons that permit some level of fine-grained parallelism. The challenge is in pulling them together so that the integration of the combination satisfies the tough requirements of real-time and of converting/dispatching work into multi-core domains. Certainly it is possible, but clearly Ada has a level of integration in the way that it combines real-time tasking with all of the paradigms of a modern programming language necessary to do this today.

6. CONCLUSIONS AND FUTURE WORK

This paper builds on recent work to introduce a model for parallel real-time programming in Ada. We develop and analyze a model where all cores but one are used to provide extra computational power to the application tasks executing in a single core. Because it is rooted in the real-time methodologies prevalent today it leverages those models and techniques to extend the traditional real-time approaches on single core systems to a variety of multi-core possibilities.

In the domain of non-uniform multi-core applications, further exploration of the effects that localized protected object calls have viz-a-viz spin-locks, fair-locks and message-exchanging systems would be useful. On systems where CPU architecture is heterogeneous, the interaction of Ada partitions, shared passive partitions, protected objects and tasklets may permit real-time behaviour across such systems, but further exploration is required.

7. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable comments. This work was partially supported by Portuguese Funds through FCT (Portuguese Foundation for Science and Technology) and by ERDF (European Regional Development Fund) through COMPETE (Operational Programme 'Thematic Factors of Competitiveness'), within VipCore (ref. FCOMP-01-0124-FEDER-015006) project and FCT and the EU ARTEMIS JU funding, within project ref. ARTEMIS/0003/2012, JU grant nr. 333053 (CONCERTO).

8. REFERENCES

[1] H. Ali and L. M. Pinho. A parallel programming model for Ada. In *Proceedings of the 2011 ACM SIGAda International Conference.* ACM, November 2011.

[2] S. K. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie and A. Wiese. A Generalized Parallel Task Model for Recurrent Real-time Processes. In *Proceedings of the 33rd IEEE Real-Time Systems Symposium*, pp. 63-72, 2012.

[3] G. Bosch. Synchronization cannot be implemented as a library. In *Proceedings of the High Integrity Language Technology Conference 2012*, ACM, 2012.

[4] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. 4th Edition, Pearson Education Ltd, Edinburg, UK, 2009.

[5] A. Burns and A. Wellings. Locking Policies for Multiprocessor Ada. In *Proceedings 16th International Real-Time Ada Workshop IRTAW 2013*, York, UK, ACM Ada Letters (to be published).

[6] S. Collette, L. Cucu and J. Goossens. Integrating job parallelism in real-time scheduling theory. *Information Processing Letters*, vol. 106, pp. 180–187, May 2008.

[7] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Survey*, 43(4):35:1–35:44, October 2011.

[8] M. Frigo, C. E. Leiserson and K. H. Randall. The implementation of the Cilk-5 multithreaded language. *SIGPLAN Not.*, 33:212-223, May 1998.

[9] ISO IEC 8652:2012. *Programming Languages and their Environments – Programming Language Ada*. International Standards Organization, Geneva, Switzerland, 2012.

[10] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer, 2011.

[11] K. Lakshmanan, S. Kato and R. Rajkumar. Scheduling parallel realtime tasks on multi-core processors. In *Proceedings of the 31st IEEE Real-Time Systems Symposium*, pp. 259 –268, December 2010.

[12] A. Marowka. Parallel computing on any desktop. *Communications of the ACM*. 50:74-78, ACM, September 2007.

[13] S. Michell, B. Moore and L. M. Pinho. Tasklettes – a Fine Grained Parallelism for Ada on Multicores. In *International Conference on Reliable Software Technologies - Ada-Europe 2013*, LNCS 7896, Springer, 2013.

[14] B. Moore. Parallelism generics for Ada 2005 and beyond. In *Proceedings of the ACM SIGAda Annual International Conference.* ACM, 2010.

[15] B. Moore. A comparison of work-sharing, work-seeking, and work-stealing parallelism strategies using Paraffin with Ada 2005. *Ada User Journal,* Volume 32 Number 1, published by Ada Europe, March 2011.

[16] B. Moore, S. Michell and L. M. Pinho. Parallelism in Ada: General Model and Ravenscar. In *Proceedings 16th International Real-Time Ada Workshop IRTAW 2013*, York, UK, ACM Ada Letters (to be published).

[17] A. Saifullah, K. Agrawal, C. Lu and C. Gill. Multi-core real-time scheduling for generalized parallel task models. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium*, Vienna, Austria, December 2011.

APPENDIX – EXAMPLE

This appendix provides an example with the application structured in a one-core system domain and two worker domains. We assume 5 tasks ($P = 5$) all in the application domain. Tasks 3 to 5 read sensors, then update actuators. The devices that manipulate the sensor data and process actuator outputs are assumed to be external to this application. Tasks 1 and 2 perform computations which are amenable to parallelization. These tasks parallelize to worker domains D_1 and D_2 respectively.

A generic interface for a Parallel Manager is provided in the backend. A Parallel Manager implementation may utilize a task pool, and interfaces to sharable task pools are also provided in the backend. The restrictions associated with the Ravenscar Profile, however necessitate a different task pool interface than for the general case. Since task pools only interface with Parallel Managers and not with user code, the backend design allows flexibility in supporting multiple task pool interfaces. This interface contains a set of routines that a pool of Ada tasks call to obtain work to be done, and to return the results of work. A task pool interface which follows the code restrictions of Ravenscar (but not fully the Ravenscar model) is shown here, since it is more relevant to real-time.

```
pragma Profile (Ravenscar);

package Ada.Parallel.Ravenscar_Task_Pools is

   --  A Pool_Index uniquely identifies a worker
   --  within the Task Pool
   type Pool_Worker_Count is new
      Worker_Count_Type;

   subtype Pool_Index is Pool_Worker_Count;

   --  A Plan_Index uniquely identifies a worker
   --  within the work plan and Parallel Manager
   type Plan_Worker_Count is new
      Worker_Count_Type;

   subtype Plan_Index is Plan_Worker_Count;

   --  A Work Plan gives the task pool client (the
   --  Parallel Manager) the full control on
   --  how the worker manages and approaches its
   --  work. The task pool only provides
   --  the workers, the work plan defines the work
   --  to be done.

   type Work_Plan is limited interface;

   procedure Engage (Plan   : in out Work_Plan;
                     Worker : Pool_Index;
                     Item   : Plan_Index)
      is abstract;
   --  When a worker starts executing, it engages
   --  the work plan. The parallelism manager
   --  client decides how to execute the work
   --  (tasklets). Engage is called once per
   --  tasklet and executes the plan.
   --  Upon returning, the Worker is once again
   --  idle and returns to the task pool

   procedure Starting (Plan      : in out
                                   Work_Plan;
                       Requester : Plan_Index;
                       Item      : out Plan_Index)
       is null;
   --  Routine that gets called before a work plan
   --  is engaged, to allow the plan to initialize
   --  any internal state. This routine is meant
   --  to be called from within a protected object
   --  associated with the pool, and therefore
   --  must not be potentially blocking

   procedure Completing (Plan : in out Work_Plan;
                         Item : Plan_Index)
       is null;
   --  Routine that gets called immediately after
   --  the work plan executed the tasklet,
   --  to allow the plan to update any internal
   --  state. This routine is intended to be
   --  called from within a protected object
   --  associated with the pool, and therefore
   --  must not be potentially blocking.

   type Task_Pool_Interface is limited interface;

   procedure Reserve (Pool : in out
                             Task_Pool_Interface;
                      Worker_Count :
                             Positive_Worker_Count)
       is abstract
   with Pre'Class =>
       Pool.Available_Workers >= Worker_Count;
   -- Allows a POP to request and reserve a number
   -- of workers from the pool.

   procedure Release (Pool : in out
                             Task_Pool_Interface;
                      Worker_Count :
                             Positive_Worker_Count)
       is abstract
   with Pre'Class =>
       Pool.Total_Workers - Pool.Available_Workers
       >= Worker_Count;
   --  Allows a POP to release the workers it had
   --  reserved back to the pool

   function Available_Workers (Pool :
                               Task_Pool_Interface)
       return Worker_Count_Type is abstract;
   --  Returns the number of workers that may be
   --  reserved in the pool.

   function Idle_Workers (Pool :
                          Task_Pool_Interface)
       return Worker_Count_Type is abstract;
   --  Returns the number of workers that are idle
   --  in the pool

   function Total_Workers (Pool :
                           Task_Pool_Interface)
       return Positive_Worker_Count is abstract;
   --  Returns the total number of workers in the
   --  pool

   procedure Offer_Work (Pool : in out
                            Task_Pool_Interface;
                         Plan : aliased in out
                            Work_Plan'Class;
                         Item : Plan_Index)
     is abstract
   with Pre'Class => Pool.Available_Workers > 0;
   --  Allows a Parallel Manager to request a
   --  worker from the task pool. The Work plan is
   --  offered to the task pool, which is then
   --  engaged by an available worker. Note: This
   --  routine is intended to be invoked by the
   --  parallelism manager, and not exposed to the
   --  user client code.
```

```
   procedure Offer_Work_To_Group (Pool :
                        in out Task_Pool_Interface;
                                  Plan :
                   aliased in out Work_Plan'Class;
                                  Worker_Count :
                              Positive_Worker_Count)
       is abstract
   with Pre'Class =>
       Pool.Available_Workers >= Worker_Count;
   --  Allows a Parallel Manager to request a
   --  group of multiple workers from the task
   --  pool. The Work plan is then engaged by each
   --  worker up to the requested Worker_Count.
   --  Note: This routine is intended to be called
   --  by the parallelism manager, and not exposed
   --  to the user client code.

   function Priority (Pool : Task_Pool_Interface)
     return System.Priority is abstract;
   --  Get the priority of the task pool

   procedure Next_Worker_Id (Pool :
                        in out Task_Pool_Interface;
                               Plan :
                   aliased in out Work_Plan'Class;
                               Requester :
                                       Plan_Index;
                               Item :
                                    out Plan_Index)
       is null;
   --  Returns the next Plan_Index which will be
   --  uniquely associated with a worker and
   --  corresponding tasklet while it executes the
   --  work plan.

   procedure Finished_Work (Pool   :
                      in out Task_Pool_Interface;
                             Worker : Pool_Index;
                             Plan   :
                aliased in out Work_Plan'Class;
                             Item   : Plan_Index)
       is null;
   --  Allows a Parallel Manager to indicate to
   --  the task pool that a tasklet has completed
   --  execution. This provides the protected
   -- subprogram context for calling the work plan
   --  Completing primitive to allow the Parallel
   --  Manager to perform any final processing
   --  with synchronization and protection from
   --  other workers.

end Ada.Parallel.Ravenscar_Task_Pools;
```

The specification for a possible implementation of this interface follows.

```
pragma Profile (Ravenscar);

with System.Storage_Elements;
with System.Multiprocessors; use System;
with Ada.Parallel.Ravenscar_Task_Pools; use
Ada.Parallel.Ravenscar_Task_Pools;

generic

   Storage_Size :
       System.Storage_Elements.Storage_Count :=
                     Default_Worker_Storage_Size;
   Worker_Priority :
       System.Priority  :=
                     System.Default_Priority;
   Number_Of_Workers :
       Pool_Worker_Count := 100;
```

```
package Ravenscar_Pool_Implementation is

   type Worker (Core : Multiprocessors.CPU_Range)
       is limited private;

   type Worker_Array is array (1 ..
       Number_Of_Workers) of access Worker;
   --  The Ada tasks in the task pool

   type Task_Pool (Workers : access Worker_Array)
       is limited new Task_Pool_Interface
       with private;
   --  task pool object type that has a pool of
   --  real Ada tasks to process
   --  tasklets that are submitted to the pool for
   --  processing.

private
   ...  Implementation Defined
end Ravenscar_Pools_Implementation;
```

The Application specification identifies and configures the application tasks and associated task pools.

```
private with Ravenscar_Pools_Implementation;

package The_Ravens_Car_Application is

   --  We assume here that Sensors and Actuators
   --  are maintained by external devices. Sensors
   --  can be read by the Ada application, and
   --  actuators can be set by the Ada
   --  application.

   type External_Device is new Float with Atomic;

   subtype Sensor_Type   is External_Device;
   subtype Actuator_Type is External_Device;

   type External_Data_Buffer
       is array (Integer range <>) of Integer
       with Atomic_Components;

   Camera_Data :
       External_Data_Buffer  (1 .. 1_000_000) :=
                                    (others => 0);
   --  Video capture

   Brakes : Actuator_Type;
   --  Controls the brakes of the vehicle

   Audio_Data :
        External_Data_Buffer (1 .. 2**20) :=
                                     (others => 0);
   Voice_Command : Actuator_Type;
   --  Indicates current voice command to process

   Desired_Temperature : Sensor_Type;
   --  Monitors Desired temperature
   Thermostat : Actuator_Type;
   --  Controls the thermostat

   Desired_Direction : Sensor_Type;
   -- Monitors GPS direction
   Steering_Wheel : Actuator_Type;
   -- Controls direction of vehicle

   Desired_Velocity : Sensor_Type;
   --  Monitors the desired velocity

   Speed : Actuator_Type;
   --  Controls the speed
```

```
private
   --  Tasks T1 and T2 get the Camera and Audio
   --  data and calculate the actuator
   --  output. We assume that this can be
   --  parallelized.  Create the application tasks

   task AT1 with Priority => 1, CPU => 1;
   --  Controls vehicles brakes via camera input

   task AT2 with Priority => 2, CPU => 1;
   --  Interprets voice data commands

   task AT3 with Priority => 3, CPU => 1;
   --  Controls vehicle direction via GPS input

   task AT4 with Priority => 4, CPU => 1;
   --  Controls vehicle speed

   task AT5 with Priority => 5, CPU => 1;
   --  Controls air temperature

   --  Create the task pools
   package D1_Task_Pool is new
       Ravenscar_Pools_Implementation
             (Storage_Size      =>
          Parallel.Default_Worker_Storage_Size,
              Worker_Priority   => 1,
              Number_Of_Workers => 4);

   Worker1 :
       aliased D1_Task_Pool.Worker (Core => 2);
   Worker2 :
       aliased D1_Task_Pool.Worker (Core => 3);
   Worker3 :
       aliased D1_Task_Pool.Worker (Core => 4);
   Worker4 :
       aliased D1_Task_Pool.Worker (Core => 5);

   D1_Workers :
       aliased D1_Task_Pool.Worker_Array :=
       (1 => Worker1'Access,
        2 => Worker2'Access,
        3 => Worker3'Access,
        4 => Worker4'Access);

   TP1 : aliased D1_Task_Pool.Task_Pool (
              Workers => D1_Workers'Access);
   --  Task Pool for AT1

   package D2_Task_Pool is new
       Ravenscar_Pools_Implementation
             (Storage_Size      =>
         Parallel.Default_Worker_Storage_Size,
              Worker_Priority   => 2,
              Number_Of_Workers => 2);

   Worker5 :
       aliased D2_Task_Pool.Worker (Core => 6);
   Worker6 :
       aliased D2_Task_Pool.Worker (Core => 7);

   D2_Workers :
       aliased D2_Task_Pool.Worker_Array :=
       (1 => Worker5'Access,
        2 => Worker6'Access);

   TP2 : aliased D2_Task_Pool.Task_Pool (
              Workers => D2_Workers'Access);
   --  Task pool for AT2

end The_Ravens_Car_Application;

--  Change Cores of Worker 5-6 to overlap cores of
--  Worker 1-4  to change to Mapping 2
--  Note that to switch to MAPPING 2, nothing
--  needs to be done in the tasks AT(1 or 2) or WT
```

The actual code of the applications can be based on existent sequential code, with only adding parallelization information in the loops in tasks T_1 and T_2. For completeness we show the code.

```
with Ada.Real_Time; use Ada;
with Parallel.Functional_Reducing_Loops.
                              Ravenscar_Work_Seeking;
with Parallel.Functional_Reducing_Recursion_
                              Ravenscar_Work_Sharing;
with Parallel.One_Shot_Wait_Free_
                              Synchronous_Barriers;
use Parallel.One_Shot_Wait_Free_
                              Synchronous_Barriers;

package body The_Ravens_Car_Application is

   Start_Time :
       constant Real_Time.Time := Real_Time.Clock;
   use type Real_Time.Time;

   task body AT1
      --  Suppose AT1 controls the brakes of the
      --  vehicle by monitoring camera views of
      --  the road
   is
      package Integer_Loops is
        new Parallel.Functional_Reducing_Loops
           (Result_Type => Integer,
            Reducer => Integer'Max,
            Identity_Value => Integer'First,
            Iteration_Index_Type => Integer);

      package Max_Loop is new
           Integer_Loops.Ravenscar_Work_Seeking;

      Max_Value : Integer := Integer'First;

      Next_Execution :
           Real_Time.Time := Start_Time;
      Period         :
           constant Real_Time.Time_Span :=
                       Real_Time.Milliseconds (1);

   begin  -- AT1 body
      loop
         delay until Next_Execution;

         for I in Camera_Data'Range
             with Parallel,
                  Task_Pool => TP1,
                  Accumulator => Max_Value,
                  Parallel_Manager =>
                     Max_Loop.Work_Seeking_Manager
         loop
            Max := Integer'Max (
                           Max, Camera_Data (I));
         end loop;

         --  Assume that the brake value is the
         --  maximum value found in the camera
         --  data. (Not at all realistic, a
         --  realistic computation would be too
         --  complex to show here)
         Brakes := Actuator_Type (Max_Value);
         Next_Execution := Next_Execution +Period;
      end loop;
   end AT1;

-------------------------------------------------
   task body AT2
   is
      --  Suppose AT2 processes voice command
      --  audio data, and acts on the interpreted
      --  commands
```

```
   --  The processing shown here is not
   --  realistic, but nevertheless shows
   --  parallel processing in two phases.

   --  The simplistic algorithm shown here
   --  calculates the average of the  array
   --  segment, then adds 1 to all values
   --  above the average, and then substracts
   --  1 from all values below the average.

   --  eg: An array of 8 elements with values
   --  from 1 .. 8
   --  there are only two worker tasks in the
   --  task pool, therefore we indicate a
   --  chunk_Size of 4 and the compiler  will
   --  divide the work in two

   --  data: 1,  2,  3,  4,  5,  6,  7,  8
   --
   --  pass 1 (two workers in parallel):
               -------------  -------------
   --   Sum:       10,             26
   --   Reduced Sum:       36

   --  Average calculated by AT2: 4.5

   --  pass 2 (two workers in parallel):
               -------------  -------------
   --  data: 0, 1,  2,  3,  6,  7,  8,  9

   package Integer_Loops is
     new Parallel.Functional_Reducing_Loops
        (Result_Type => Integer,
         Reducer => "+",
         Identity_Value => 0,
         Iteration_Index_Type => Integer);

   package Avg_Loop is new
        Integer_Loops.Ravenscar_Work_Sharing;

   Sum_Value : Integer := 0;
   Avg_Value : Float;


   Next_Execution  :
        Real_Time.Time := Start_Time;
   Period          :
        constant Real_Time.Time_Span :=
                    Real_Time.Milliseconds (5);


begin --  AT2 body

   loop
      delay until Next_Execution;

      for I in 1 .. 8
         with  Parallel,
               Task_Pool => TP2,
               Accumulator => Sum_Value,
               Parallel_Manager =>
                 Avg_Loop.Work_Sharing_Manager,
               Chunk_Size => 4
      loop
         Sum_Value := Audio_Data(I) +Sum_Value;
      end loop;
      --  Parallel first phase calculating
      --  aggregated sum

      Avg_Value := Float(Sum_Value) / 8.0;
      --  Sequential phase calculating the
      --  average

      for I in 1 .. 8
         with  Parallel,
               Task_Pool => TP2,
               Chunk_Size => 4
      loop
          if (Float(Audio_Data(I)) >
                                Avg_Value) then
            Audio_Data(I) := Audio_Data(I) + 1;
          elsif (Float(Audio_Data(I)) <
                                Avg_Value) then
            Audio_Data(I) := Audio_Data(I) - 1;
          end if;
      end loop;
      --  Parallel second phase updating values

      Next_Execution := Next_Execution +Period;
   end loop;
end AT2;

-------------------------------------------------
   task body AT3 is
      --  Suppose AT3 maintains the direction of
      --  the vehicle according to
      --  current route instructions from a
      --  GPS device
      Next_Execution :
          Real_Time.Time := Start_Time;
      Period :
          constant Real_Time.Time_Span :=
                     Real_Time.Milliseconds (10);
   begin
      loop
         delay until Next_Execution;
         Steering_Wheel := Desired_Direction;
         Next_Execution := Next_Execution +Period;
      end loop;
   end AT3;

-------------------------------------------------
   task body AT4 is
      --  Suppose AT4 maintains the
      --  cruise control speed
      Next_Execution :
          Real_Time.Time := Start_Time;
      Period :
          constant Real_Time.Time_Span :=
                    Real_Time.Milliseconds (100);
   begin
      loop
         delay until Next_Execution;
         Speed := Desired_Velocity;
         Next_Execution := Next_Execution +Period;
      end loop;
   end AT4;

-------------------------------------------------
   task body AT5 is
      --  Suppose AT5 maintains the cabin
      --  temperature by controlling heating/AC
      Next_Execution :
          Real_Time.Time := Start_Time;
      Period :
          constant Real_Time.Time_Span :=
                          Real_Time.Minutes (1);
   begin
      loop
         delay until Next_Execution;
         Thermostat := Desired_Temperature;
         Next_Execution := Next_Execution +Period;
      end loop;
   end AT5;

end The_Ravens_Car_Application;
```