Table of Contents

Newsletter Information .................................................. 1
From the Editor’s Desk ................................................... 3
Editorial Policy .................................................................. 4
Key Contacts ...................................................................... 6

IRTAW-16 Workshop Papers .............................................. 9

Parallel Ada - A Requirement for Ada 2020 - A. Burns .......... 9
Parallelism in Ada: General Model and Ravenscar - Brad Moore, Stephen Michell and Luis Miguel Pinho .... 14
On real-time partitioned multicore systems - Juan Zamorano and Juan A. de la Puente ..... 33

Ada and Many-core Platforms - Luis Miguel Pinho, Stephen Michell and Brad Moore .... 40
Incorporating the Deadline Floor Protocol in Ada - Mario Aldea, Alan Burns, Marina Gutiérrez and Michael González Harbour ................................................................. 49
Locking Policies for Multiprocessor Ada - A. Burns and A.J. Wellings .................................................... 59
Lock-Free Protected Types for Real-Time Ada - Geert Bosch ................................................................. 66
Execution time timers for interrupt handling - Kristoffer Nyborg Gregersen and Amand Skavhaug .......... 87
Deferred and Atomic Setting of Scheduling Attributes for Ada - Sergio S´aez, Jorge Real and Alfons Crespo .... 97
An extended Ravenscar proﬁle for execution time control - Kristoffer Nyborg Gregersen .............................................. 109

Session Summary: Parallel and Multicore Systems .......... 115
Chair: Luis Miguel Pinho and Rapporteurs: Stephen Michell and Brad Moore ....................

Session Summary: Locking Protocols .......................... 123
Chair: Alan Burns and Rapporteur: Andy Wellings ..........................

Session Summary – Improvements to Ada .................... 126
Chair: Tullio Vardanega and Rapporteur: Rod White

Session Summary: Open Issues ................................. 131
Chair: Jorge Real and Rapporteur: Juan Antonio de la Puente

Trudy Levine’s Column .................................................. 133
Reusable Software Components - Trudy Levine ..........................

SIGAda 2013 Conference ............................................. 141
Ada Europe Conference 2014 ......................................... 145
The ACM Digital Library

www.acm.org/dl

The Ultimate Online INFORMATION TECHNOLOGY Resource!

Powerful and vast in scope, the ACM Digital Library is the ultimate online resource offering unlimited access and value!

The ACM Digital Library interface includes:

- **The ACM Digital Library** offers over 45 publications including all ACM journals, magazines, and conference proceedings, plus vast archives, representing over two million pages of text. The DL includes full-text articles from all ACM publications dating back to the 1950s, as well as third-party content with selected archives. www.acm.org/dl

- **The Guide to Computing Literature** offers an enormous bank of more than one million bibliographic citations extending far beyond ACM’s proprietary literature, covering all types of works in computing such as journals, proceedings, books, technical reports, and theses! www.acm.org/guide

- **The Online Computing Reviews Service** includes reviews by computing experts, providing timely commentary and critiques of the most essential books and articles.

Available only to ACM Members.

Join ACM online at www.acm.org/joinacm

To subscribe to the ACM Digital Library, contact ACM Member Services:

Phone: 1.800.342.6626 (U.S. and Canada) +1.212.626.0500 (Global)

Fax: +1.212.944.1318

Hours: 8:30 a.m.-4:30 p.m., Eastern Time

Email: acmhelp@acm.org

Mail: ACM Member Services

General Post Office

PO Box 30777

New York, NY 10087-0777 USA

ACM Professional Members can add the ACM Digital Library for only $99 (USD). Student Portal Package membership includes the Digital Library, Institutional, Corporate, and Consortia. Packages are also available.

ACM Special Interest Group on Ada Programming Language (SIGAda) provides a forum on all aspects of the Ada language and technologies, including usage, education, standardization, design methods, and compiler implementation. Among the topics that SIGAda addresses are software engineering practice, real-time applications, high-integrity & safety-critical systems, object-oriented technology, software education, and large-scale system development. SIGAda explores these issues through an annual international conference, special-purpose Working Groups, active local chapters, and its Ada Letters publication.

The Association for Computing Machinery (ACM) is an educational and scientific computing society which works to advance computing as a science and a profession. Benefits include subscriptions to Communications of the ACM, MemberNet, TechNews and CareerNews, full and unlimited access to online courses and books, discounts on conferences and the option to subscribe to the ACM Digital Library.

- SIGAda (ACM Member) ........................................... $ 10
- SIGAda (ACM Student Member & Non-ACM Student Member) ........................................... $ 10
- SIGAda (Non-ACM Member) ........................................... $ 25
- ACM Professional Membership (S99) & SIGAda (S25) ........................................... $124
- ACM Professional Membership (S99) & SIGAda (S25) & ACM Digital Library (S99) ........................................... $223
- ACM Student Membership (S19) & SIGAda (S10) ........................................... $ 29
- Ada Letters only ........................................... $ 53
- Expedited Air for Communications of the ACM (outside N. America) ........................................... $ 58
Table of Contents

**Newsletter Information** 1
From the Editor’s Desk 3
Editorial Policy 4
Key Contacts 6

**IRTAW-16 Workshop Papers**
Parallel Ada - A Requirement for Ada 2020 - A. Burns 9
Parallelism in Ada: General Model and Ravenscar - Brad Moore, Stephen Michell and Luis Miguel Pinho 14
On real-time partitioned multicore systems - Juan Zamorano and Juan A. de la Puente 33
Ada and Many-core Platforms - Luis Miguel Pinho, Stephen Michell and Brad Moore 40
Incorporating the Deadline Floor Protocol in Ada –
    Mario Aldea, Alan Burns, Marina Gutiérrez and Michael González Harbour 49
Locking Policies for Multiprocessor Ada - A. Burns and A.J. Wellings 59
Lock-Free Protected Types for Real-Time Ada - Geert Bosch 66
Programming Simple Reactive Systems in Ada: Premature Program Termination -
    A.J. Wellings, A. Burns, A.L.C. Cavalcanti and N.K. Singh 75
Execution time timers for interrupt handling - Kristoffer Nyborg Gregertsen and Amund Skavhaug 87
Deferred and Atomic Setting of Scheduling Attributes for Ada - Sergio S’aez, Jorge Real and Alfons Crespo 97
An extended Ravenscar profile for execution time control - Kristoffer Nyborg Gregertsen 109

**Session Summary: Parallel and Multicore Systems** 115
Chair: Luis Miguel Pinho and Rapporteurs: Stephen Michell and Brad Moore

**Session Summary: Locking Protocols** 123
Chair: Alan Burns and Rapporteur: Andy Wellings

**Session Summary – Improvements to Ada** 126
Chair: Tullio Vardanega and Rapporteur: Rod White

**Session Summary: Open Issues** 131
Chair: Jorge Real and Rapporteur: Juan Antonio de la Puente

**Trudy Levine’s Column:** 133
Reusable Software Components – Trudy Levine

SIGAda 2013 Conference 141
Ada Europe Conference 2014 145

A Publication of SIGAda, the ACM Special Interest Group on Ada
ACM SIGAda Executive Committee

CHAIR
David Cook, Stephen F. Austin State University, Dept. of Computer Science, P.O. Box 13063, SFA Station, Nacogdoches, TX 75962, USA, Phone: +1 (936) 468-2508, CookDA@sfasu.edu

VICE-CHAIR
Tucker Taft, AdaCore, 24 Muzzey St., 3rd Floor, Lexington, MA 02421, USA
Phone: +1 (646) 375-0730, Taft@adacore.com

SECRETARY/TREASURER
Clyde Roby, Institute for Defense Analyses, 4850 Mark Center Drive, Alexandria, VA 22311 USA
Phone: +1 (703) 845-6666, Roby@ida.org

INTERNATIONAL REPRESENTATIVE
Dirk Craeynest, c/o K.U.Leuven, Dept. of Computer Science, Celestijnenlaan 200-A, B-3001 Leuven (Heverlee) Belgium, Dirk.Craeynest@cs.kuleuven.be

PAST CHAIR
Ricky E. Sward, The MITRE Corporation, 1155 Academy Park Loop Colorado Springs, CO 80910 USA
Phone: +1 (719) 572-8263, RSward@Mitre.org

EDITOR, ACM ADA LETTERS
Alok Srivastava, TASC Inc., 475 School Street, SW, Washington, DC 20024
Phone: +1 (202) 314-1419, Alok.Srivastava@TASC.Com

ACM PROGRAM COORDINATOR SUPPORTING SIGAda
Irene Frawley, 2 Penn Plaza, Suite 701, New York, NY 10121-0701
Phone: +1 (212) 626-0605, Frawley@ACM.Org

For advertising information contact:
Advertising Department
2 Penn Plaza, Suite 701, New York, NY 10121-0701
Phone: (212) 869-7440; Fax (212) 869-0481

Is your organization recognized as an Ada supporter? Become a SIGAda INSTITUTIONAL SPONSOR! Benefits include having your organization’s name and address listed in every issue of Ada Letters, two subscriptions to Ada Letters and member conference rates for all of your employees attending SIGAda events. To sign up, contact Rachael Barish, ACM Headquarters, 2 Penn Plaza, Suite 701, New York, NY 10121-0701, and email: MEETING@ACM.ORG, Phone: 212-626-0603.

Interested in reaching the Ada market? Please contact Jennifer Booher at Worlddata (561) 393-8200 Ext. 131, email: platimer@worlddata.com. Please make sure to ask for more information on ACM membership mailing lists and labels.

Ada Letters (ISSN 1094-3641) is published three times a year by the Association for Computing Machinery, 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA. The basic annual subscription price is $20.00 for ACM members.

POSTMASTER: Send change of address to Ada Letters:
ACM, 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA

Notice to Past Authors of ACM-Published Articles
ACM intends to create a complete electronic archive of all articles and/or other material previously published by ACM. If you have written a work that has been previously published by ACM in any journal or conference proceedings prior to 1978, or any SIG Newsletter at any time, and you do NOT want this work to appear in the ACM Digital Library, please inform permissions@acm.org, stating the title of the work, the author(s), and where and when published.
Welcome to this issue of ACM Ada Letters. In this issue you will find very interesting papers presented at the 16th International Real-Time Ada Workshop (IRTAW 2013) held in cooperation with Ada Europe at University of York, Department of Computer Science from 17-19 April 2013 in Kings Manor, York, England. Since the late Eighties the International Real-Time Ada Workshop series has provided a forum for identifying issues with real-time system support in Ada and for exploring possible approaches and solutions, and has attracted participation from key members of the research, user, and implementer communities worldwide. Recent IRTAW meetings have significantly contributed to the Ada 2005 standard and to the proposals for Ada 2012, especially with respect to the tasking features, the real-time and high-integrity systems annexes, and the standardization of the Ravenscar profile. The summary and outcome of various focused sessions during the IRTAW-15 are presented in this issue.

In this issue you will find details on High-Integrity Language Technology SIGAda 2013 conference to be held from November 10-14, 2013 in Pittsburgh, Pennsylvania (USA) and another major Ada event, the 19th International Conference on Reliable Software Technologies – Ada-Europe 2014 which will take place from June 23rd to 27th in Paris, France.

Our regular contributor Trudy Levine has provided updates to the Ada Reusable Software Components.

A new team of elected SIGAda officers has started working from July 1st 2013. Dr. David Cook, a former US Air Force Officer and a well known academician, is the new SIGAda Chair. Under his leadership, SIGAda will focus more on Ada education, awareness of Ada and in promoting its advantages in safety-conscious software development. Other SIGAda elected officers are Tucker Taft as Vice-Chair, Clyde Roby as Secretary/Treasurer and Dirk Craeynest as International Representative. Ada community is offering their best wishes to carry the responsibilities.

Ada Letters is a great place to submit articles of your experiences with the language revision, tips on usage of the new language features, as well as to describe success stories using Ada. We’ll look forward to your submission. You can submit either a MS Word or Adobe PDF file (with 1” margins and no page numbers) to our technical editor:

Pat Rogers, Ph.D.
AdaCore, 207 Charleston, Friendswood, TX 77546 (USA)
+1 281 648 3165, rogers@adacore.com

We look forward to hearing from you!

Alok Srivastava, Ph.D.
Technical Fellow, TASC Inc.
475 School St, SW; Washington, DC 20024 (USA)
+1 202 314 1419 Alok.Srivastava@TASC.Com
Editorial Policy (from Alok Srivastava, Managing Editor)

As the editor of Ada Letters, I’d like to thank you for your continued support of ACM SIGAda, and encourage you to submit articles for publication. In addition, if there is some way we can make Ada Letters more useful to you, please let me know. Note that Ada Letters is now on the web! See http://www.acm.org/sigada/ada_letters/index.html. The two newest issues are available only to SIGAda members. Older issues beginning March 2000 are available to all.

Now that Ada is standing on its own merits without the support of the DoD, lots of people and organizations have stepped up to provide new tools, mechanisms for compiler validation/assessment, and standards (especially ASIS). The Ada 2005 language version is fulfilling the market demand of robust safety and security elements and thereby generating a new enthusiasm into the software development. Ada Letters is a venue for you to share your successes and ideas with others in the Ada community. Be sure to take advantage of it so that we can all benefit from each other’s learning and experience.

As some of the other ACM Special Interest Group periodicals have moved, Ada Letters also transitioned from quarterly to a tri-annual publication. With exception of special issues, Ada Letters now is going to be published three times a year, with the exception of special issues. The revised schedules and submission deadlines are as follows:

<table>
<thead>
<tr>
<th>Deadline</th>
<th>Issue</th>
<th>Deadline</th>
<th>Issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>October 1st, 2013</td>
<td>December, 2013</td>
<td>February 1st, 2014</td>
<td>April, 2014</td>
</tr>
</tbody>
</table>

Please send your article to Dr. Pat Rogers at rogers@adacore.com

Guidelines for Authors

Letters, announcements and book reviews should be sent directly to the Managing Editor and will normally appear in the next corresponding issue.

Proposed articles are to be submitted to the Technical Editor. Any article will be considered for publication, provided that topic is of interest to the SIGAda membership. Previously published articles are welcome, provided the previous publisher or copyright holder grants permission. In particular, keeping with the theme of recent SIGAda conferences, we are interested in submissions that demonstrate that “Ada Works.” For example, a description of how Ada helped you with a particular project or a description of how to solve a task in Ada are suitable.

Although Ada Letters is not a refereed publication, acceptance is subject to the review and discretion of the Technical Editor. In order to appear in a particular issue, articles must be submitted far enough in advance of the deadline to allow for review/edit cycles. Backlogs may result in an article's being delayed for two or more issues. Contact the Managing Editor for information on the current publishing queue.

Articles should be submitted electronically in one of the following formats: MS Word (preferred) Postscript, or Adobe Acrobat. All submissions must be formatted for US Letter paper (8.5” x 11”) with one inch margins on each side (for a total print area of 6.5” x 9”) with no page
numbers, headers or footers. Full justification of text is preferred, with proportional font (preferably Times New Roman, or equivalent) of no less than 10 points. Code insertions should be presented in a non-proportional font such as Courier.

The title should be centered, followed by author information (also centered). The author's name, organization name and address, telephone number, and e-mail address should be given. For previously published articles, please give an introductory statement (in a distinctive font) or a footnote on the first page identifying the previous publication. ACM is improving member services by creating an electronic library of all of its publications. Read the following for how this affects your submissions.

**Notice to Contributing Authors to SIG Newsletters:**
By submitting your article for distribution in this Special Interest Group publication, you hereby grant to ACM the following non-exclusive, perpetual, worldwide rights:

- to publish in print on condition of acceptance by the editor
- to digitize and post your article in the electronic version of this publication
- to include the article in the ACM Digital Library
- to allow users to copy and distribute the article for noncommercial, educational or research purposes

However, as a contributing author, you retain copyright to your article and ACM will make every effort to refer requests for commercial use directly to you.

**Notice to Past Authors of ACM-Published Articles**
ACM intends to create a complete electronic archive of all articles and/or other material previously published by ACM. If you have a work that has been previously published by ACM in any journal or conference proceedings prior to 1978, or any SIG Newsletter at any time, and you do NOT want this work to appear in the ACM Digital Library, please inform permissions@acm.org, stating the title of the work, the author(s), and where and when published.

**Back Issues**
Back issues of Ada Letters can be ordered at the price of $6.00 per issue for ACM or SIGAda members; and $9.00 per issue for non-ACM members. Information on availability, contact the ACM Order Department at 1-800-342-6626 or 410-528-4261. Checks and credit cards only are accepted and payment must be enclosed with the order. Specify volume and issue number as well as date of publication. Orders must be sent to:

ACM Order Department, P.O. Box 12114, Church Street Station, New York, NY 10257 or via FAX: 301-528-8550.
KEY CONTACTS

Technical Editor
Send your book reviews, letters, and articles to:

Pat Rogers
AdaCore
207 Charleston
Friendswood, TX 77546
+1-281-648 3165
Email: rogers@adacore.com

Managing Editor
Send announcements and short notices to:

Alok Srivastava
TASC Inc.
475 School Street, SW
Washington DC 20024
+1-202-314-1419
Email: Alok.Srivastava@auatac.com

Advertising
Send advertisements to:

William Kooney
Advertising/Sales Account Executive
2 Penn Plaza, Suite 701
New York, NY 10121-0701
Phone: +1-212-869-7440
Fax: +1-212-869-0481

Local SIGAda Matters
Send Local SIGAda related matters to:

Vice Chair
Tucker Taft
AdaCore
24 Muzzey St., 3rd Floor
Lexington, MA 02421
Phone: +1.646.375.0730
Fax: +1.646.329.7458
Email: taft@adacore.com

Ada CASE and Design Language Developers
Matrix
Send ADL and CASE product Info to:

Judy Kerner
The Aerospace Corporation
Mail Stop M8/117
P.O. Box 92957
Los Angeles, CA 9009
+1-310-336-3131
Email: kerner@aero.org

Ada Around the World
Send Foreign Ada organization info to:

Dirk Craeynest
c/o K.U.Leuven, Dept. of Computer Science,
Celestijnenlaan 200-A, B-3001 Leuven (Heverlee)
Belgium
Email: Dirk.Craeynest@cs.kuleuven.be

Reusable Software Components
Send info on reusable software to:

Trudy Levine
Computer Science Department
Fairleigh Dickinson University
Teaneck, NJ 07666
+1-201-692-2000
Email: levine@fdu.edu
SIGAda Working Group (WG) Chairs
See http://www.acm.org/sigada/ for most up-to-date information

Ada Application Programming Interfaces WG
Geoff Smith
Lightfleet Corporation
4800 NW Camas Meadows Drive
Camas, WA 98607
Phone: +1-503-816-1983
Fax: +1-360-816-5750
Email: gsmith@lightfleet.com

Ada Semantic Interface Specification WG
http://www.acm.org/sigada/wg/asiswg/asiswg.html
Bill Thomas
The MITRE Corp
7515 Colshire Drive
McLean, VA  22102-7508
Phone: +1-703-983-6159
Fax: +1-703-983-1339
Email: BThomas@MITRE.Org

Education WG
http://www.sigada.org/wg/eduwg/eduwg.html
Mike Feldman
420 N.W. 11th Ave., #915
Portland, OR 97209-2970
Email: MFeldman@seas.gwu.edu

Standards WG
Robert Dewar
73 5th Ave.
New York, NY 10003
Phone: +1-212-741-0957
Fax: +1-232-242-3722
Email: dewar@cs.nyu.edu
**Ada Around the World**
*(National Ada Organizations)*

From: [http://www.ada-europe.org/members.html](http://www.ada-europe.org/members.html)

---

**Ada-Europe**
Tullio Vardanega
University of Padua
Department of Pure and Applied Mathematics
Via Trieste 63
I-35121, Padova, Italy
Phone: +39-049-827-1359
Fax: +39-049-827-1444
E-mail: tullio.vardanega@math.unipd.it
http://www.ada-europe.org/

**Ada-Belgium**
Dirk Craeynest
C/o K.U.Leuven, Dept. of Computer Science, Celestijnenlaan 200-A, B-3001
Leuven (Heverlee), Belgium
Phone: +32-2-725 40 25
Fax: +32-2-725 40 12
E-mail: Dirk.Craeynest@cs.kuleuven.be

**Ada in Denmark**
Jørgen Bundgaard
E-mail: Info at Ada-DK.org
http://www.Ada-DK.org/

**Ada-Deutschland**
Peter Dencker, Steinackerstr. 25
D-76275 Ettlingen-Spessartt, Germany
E-mail: dencker@parasoft.de
http://www.ada-deutschland.de/

**Ada-France**
Association Ada-France
c/o Jérôme Hugues
Département Informatique et Réseau École Nationale Supérieure des Télécommunications, 46, rue Barrault
75634 Paris Cedex 135, France
E-mail: bureau@ada-france.org
http://www.ada-france.org/

**Ada Spain**
J. Javier Gutiérrez
P.O. Box 50.403
E-28080 Madrid, Spain
Phone: +34-942-201394
Fax: +34-942-201402
E-mail: gutierjj@unican.es
http://www.adaspain.org/

**Ada in Sweden**
Rei Stråhle
Box Saab Systems
S:t Olofsgatan 9A
SE-753 21 Uppsala, Sweden
Phone: +46-73-437-7124
Fax: +46-85-808-7260
E-mail: Rei.Strahle@saabgroup.com
http://www.ada-i-sverige.se/

**Ada in Switzerland**
Ahlan Marriott
White Elephant GmbH
Postfach 327
CH-8450 Andelfingen, Switzerland
Phone: +41 52 624 2939
Fax: +41 52 624 2334
E-mail: ada@white-elephant.ch
http://www.ada-switzerland.org/

**Italy**
Contact: tullio.vardanega@math.unipd.it

**Ada-Europe Secretariat**
e-mail: secretariat@ada-europe.org
Parallel Ada - A Requirement for Ada 2020

A. Burns
Department of Computer Science,
University of York, UK

Abstract

Much of the focus with multi-core hardware has been on the mapping of tasks to cores. It is important that languages can support both static and dynamic forms of this mapping. However, as the number of cores increase and platforms become more heterogeneous it becomes necessary to identify and support parallel execution within tasks. Various forms of ‘parallel’ statement have been discussed in the literature. Here we argue for the need for simple changes to the language that can go a long way towards exploiting fine grain parallelism.

1 Introduction

During a Panel on language support for multi-core platforms at the Ada Europe conference in 2011 an informal classification [3] was made between platforms with

- one core, or
- a few cores, or
- many cores.

The point was made that most languages were defined for single CPUs. A ‘few cores’ was defined to be a number less than the natural number of tasks found in embedded software. For such platforms the ‘task’ is the most straightforward means of exploiting the platform’s parallelism. Tasks are allocated (either statically or dynamically) to cores, and at run-time there is usually more executable tasks than cores and hence an adequate exploitation of the hardware is being made.

With ‘many cores’ however there are more cores than tasks (by definition) and hence the parallelism can only be exploited if parallelism within tasks is identified and utilised. Unfortunately, the considerable effort that has been exerted in trying to automatically extract parallelism from sequential code has, in large part, failed. It therefore seems that programmers must directly address the issue, and programming languages must support adequately fine grain parallelism. This paper considers the form of support this might take for Ada as this languages moves towards its 2020 definition.

2 Fine Grain Concurrency

Few programming languages support fine grain concurrency. Most, if they support concurrency at all, do so by a task, thread or process notion. A program consists of a relatively small number of relatively large/long sequential tasks. There are however some examples of sub-task concurrency. For example, OCCAM [2, 6, 10] defined parallel composition to be equivalent (syntactically) to sequential composition:
Only when there is an explicit need to constrain the code is the `SEQ` statement used. Of course `PAR` does not mean parallel execution, it just indicates that parallel execution is allowed. A better label would have been `CON` for concurrency.

Whereas extracting parallel execution from sequential code is problematic, extracting parallel execution from fine grain concurrency is relatively straightforward. Both the compiler and the run-time support software have a role. The compiler can convert too fine grain code to sensibly sized parallel ‘chunks’. And the run-time can dynamically decide on how to partition code. With the above example, the assignments to A and B really should be sequential and the compiler can accomplish this. But in

```ada
case for PAR
  F1
  F2
end case
```

where F1 and F2 were arbitrary function calls then the compiler would retain in the executable code the potential for parallel execution and the run-time would indeed execute the functions in parallel if there were cores available. Of course if F1 and F2 turn out to be executed in sequence then there should be no (or very little) extra overhead for the programmer using `PAR` rather than `SEQ`. For `PAR` and `SEQ` to be equivalent then F1 and F2 cannot share any program entities.

For programming languages with pure functions (i.e. no side effects) then the parallel execution of F1 and F2 is easily guaranteed to be correct. It is perhaps an issue for Ada 2020 whether pure functions should be supported more explicitly.

### 3 Requirements

Exploiting highly parallel hardware will require run-time entities that are simpler (more light-weight) than processes or even threads. A number of terms are used in the OS literature for such entities, including, micro-threads and pico-threads. Typical constraints/requirements for these entities are:

- They do not block.
- They can potentially start execution as soon as they are created.
- They have no internal state (they are more event like).
- They terminate (and cease to exist) as soon as they complete the execution of their code.
- They are not named and are not therefore directly addressed in the program.

Clearly Ada tasks are much more heavy-weight. Support for parallelism must come at the statement level (from within tasks). As any particular task will only execute in a single (Ada) partition, the following is focussed on partitions with many-cores.

For statement-level parallelism there are really only two main requirements, and these were well supported in OCCAM. It must be possible to state that a block of code statements can be executed in parallel (`parbegin/parend` perhaps). Also it must be possible to say that each iteration of a loop can execute in parallel (perhaps indicated by `parloop/end parloop`).

```ada
case for PAR
  A := 1
  B := 2
end case
```

```ada
case for SEQ
  A := 1
  B := A + 2
end case
```
Both of these constructs have the form: create parallelism, execute immediately, wait for completion of all parallel components, continue. So really a simple fork and join construct but with a higher branching factor.

It might be possible to get new key words into Ada 2020, or if not then perhaps a syntactical form using the new aspect feature could be utilised. In the code examples used below the simple addition of ‘#’ to either loop or begin is used to identify potential parallelism. This use of a simple single character is used to show that little does actually have to change to existing code.

There have of course been a number of papers that have advocated the introduction of some sort of ‘parallel’ key words into Ada [1, 4, 5, 7–9]. The motivation for this paper is to bring the issue forward for discussion at IRTAW16, and to illustrate that in fact the requirements on the language are not too excessive.

The above has focused on control parallelism, there is also the requirement to support data parallelism. Concurrent parts of a program might be assessing different parts of a large data structure (for example an array), but if the underlying hardware serialises access to the memory via either the bus or the memory itself, then much of the potential advantage available from the parallel processing hardware will be lost due to non-parallel memory. This issue must be addressed by any proposed change to the language.

4 Examples

The above additions to Ada are quite simple. Are they sufficient to write programs that can exploit highly parallel hardware? The answer to this is probably yes, but for Ada the question should be: are they sufficient to write correct programs that can easily exploit highly parallel hardware?

As mentioned above pure functions would help structure parallel code. Otherwise the programmer must ensure that parallel entities cannot interfere with each other (e.g. use shared variable). As our micro-threads cannot block then they cannot make use of protected objects (POs). POs are for tasks (which would still exist), for parallel constructs it is necessary to use different programming patterns. Here we give one such example which might help to define what Ada 2020 needs to support (or need not support).

Lets start with code that is clearly easy to parallelise – a loop that adds one to each element of an array.

So rather than

```ada
for I in 1..100_000 loop -- i.e. a large N
    Store(I) := Store(I) + 1;
end loop;
```

the programmer could put

```ada
for I in 1..100_000 loop#
    Store(I) := Store(I) + 1;
end loop#;
```

A compiler might be able to put this into sequential chunks of say 100 iterations; if not the programmer may need to:

```ada
for I in 0..999 loop#
    for J in I*100+1 .. (I+1)*100 loop
        Store(J) := Store(J) + 1;
    end loop;
end loop#;
```

Ideally the code would not need to be explicit about the size of chunk, rather this information could be given directly to the compiler.

A more general problem is to sum the elements in this array:

```ada
Sum := 0;
for I in 1..100_000 loop
    Sum := Sum + Store(I);
end loop;
```
Clearly it would not be sensible to simple replace the `loop` by a `loop#. But again the programmer could make the level of parallelism explicit by summing up parts of the array in parallel to obtain subtotals that are then added together:

```ada
Sum := 0;
for I in 0..999 loop#
    Summ(I) := 0;
    for J in I*100+1..(I+1)*100 loop
        Summ(I) := Summ(I) + Store(J);
    end loop;
end loop;
for I in 0..999 loop
    Sum := Sum + Summ(I);
end loop;
```

An alternative means of achieving this parallelism is to use recursion rather than an explicit loop. Now let \texttt{Summer} be a function, with two parameters:

```ada
function Summer(A, Z : integer) return integer is
    temp, temp2 : integer := 0;
    begin
    if Z-A <= 10 then -- unit of sequential execution
        for I in A..Z loop
            temp := temp + Store(I);
        end loop;
        return temp;
    else
        begin
            temp := Summer(A,A+(Z-A)/2);
            temp2 := Summer(A+(Z-A)/2+1,Z);
        end;
        return temp + temp2;
    end if;
end Summer;
```

Here the programmer has decided that the minimum chunk of code is 10 summations. At run-time a dynamic decision can be made as to how much parallel execution is exploited. At each `begin#`, if there is an available core, then the code will genuinely run in parallel. For example, if there are 4 cores then the first call of \texttt{Summer}(1,100_000) will result in parallel calls to \texttt{Summer}(1,50_000) and \texttt{Summer}(50_001,100_000). The first of these may result in parallel calls to \texttt{Summer}(1,25_000) and \texttt{Summer}(25_001,50_000); and the second may result in the parallel execution of \texttt{Summer}(50_001,75_000) and \texttt{Summer}(75_001,100_000). The subsequent recursive calls will result in sequential execution as there are no further available cores to exploit.

5 Conclusions

The need to write programs that can easily and efficiently exploit highly parallel hardware is an increasingly significant one. For Ada this means that the programmer should be able to designate code that can be executed in parallel. And this code will need to be at a granularity well below that of the task. Sufficient implementation freedom must be available to allow the compiler and the run-time support system to jointly target the capabilities of the hardware - whether this is a multicore SMP or heterogeneous hardware directly fabricated to meet the specific needs of the program.

Although an important challenge, significant support for parallelism can be obtained, in a language like Ada, by small changes to the language. In particular, including the ability to designate blocks of code to be executed in parallel and loops to have all iterations executed in parallel is a straightforward but nevertheless important facility.
Functional programming languages have been shown to more easily exploit parallel hardware because they contain pure functions (functions without side effects). Ada 2020 should therefore look at ways of designate such functions.

References

Parallelism in Ada: General Model and Ravenscar

Brad Moore¹  Stephen Michell² Luis Miguel Pinho³

¹ General Dynamics, Canada, brad.moore@gdcanada.com
² Maurya Software Inc, Canada, stephen.michell@maurya.on.ca
³ CISTER/INESC-TEC, ISEP Polytechnic Institute of Porto, Portugal, lmp@isep.ipp.pt

Abstract

Parallel programming is expected to become more the norm as multi-core and many-core processors gain more widespread use. Ada has always had excellent concurrency support, but could be improved in the area of parallel programming. Specifically, divide and conquer parallelism via parallel loops and subprograms are difficult to write without some sort of library support. In this paper we describe a proposal that combines the use of task pools and parallelism managers to provide parallelism capabilities to real-time Ada applications, including Ravenscar applications. This work complements the syntax enhancements that we previously proposed, so that together they facilitate the writing of parallel applications.

1 Introduction

The trend towards multi-core and many-core will require new approaches to make effective use of the available processing capabilities. Parallelism is one of the areas in computing that should see significant impact as a result of the new hardware. Past experience has shown that parallel programming is inherently difficult to do correctly without the benefit of a solid framework of abstractions to decouple the sequential algorithms from their parallel counterparts.

Part of such a framework should explore ways to simplify the expression of parallelism in an algorithm using syntactic improvements to the language. In the case of C and C++, approaches such as Cilk [1] and Intel's TBB [2] extensions to the languages provide ways to express parallel algorithms. In the case of Ada, there have been proposals for introducing parallelism such as parallel loops in the past [3]. More recently, other proposals [4][5] have covered a broader set of possibilities including parallel subprograms, but to date have not addressed any real-time issues associated with multi-core.

The real-time computing world similarly is considering the possibilities of using parallelism, as the demands for increasing processing power approach the limits that can be handled by a single core.

Ada[6] is a language that was designed from the start to support parallelism and concurrency and in many senses was ahead of its time. While Ada¹ provides a solid and safe framework for concurrency with tasks and protected objects, there is room for improvements in the area of parallelism support for multi-core and manycore.

This paper extends our design proposals for a parallel framework in Ada [7]². While the earlier work describes a more general approach, this paper is tailored specifically for the real-time community.

In particular, this paper discusses extensions to that previous proposal, to provide a framework for executing parallel loops and subprograms that satisfy the restrictions of the Ravenscar Profile.

An open source reference implementation of these interfaces (minus the language syntax) can be found in Paraffin[8], a suite of generics adapted to the Ada 2012 standard, that also includes reusable utilities such as fast fourier transforms, quicksort, and red-black tree containers that utilize the parallelism generics.

The paper is structured as follows. Section 2 presents some examples that illustrate new syntax we are proposing for the language, that is described in more detail in [7]. Then Section 3 presents the high level design model for the parallelism framework. Section 4 describes the benefits of introducing the notion of task pools to the framework. Section 5 presents a design proposal for implementing task pools for the non-Ravenscar case, and then goes on to present a design proposal for parallelism managers that interact with this task pool. Section 6 similarly presents a design proposal for implementing Ravenscar task pools, which then goes on to present a design proposal for Ravenscar parallelism managers that utilize that

---

¹ The Ada 2012 version is necessary as we use aspects to implement our proposal.
² The referenced paper is provided as an attachment for the IRTAW to read.
2 Parallel Framework: Syntactic Enhancements

The desire is for the programmer to easily express that code should be executed in parallel, if possible.

Below is a very introductory and simple example to aid in understanding the concept. The reader can find in [7] a more detailed description of the syntax extensions that have been proposed.

```ada
Sum := 0;
for I in 1 .. 1_000_000_000
  with Parallel,_
    Accumulator => Sum
loop
  Sum := Sum + I;
end loop;
```

In [7], we proposed a new aspect, Parallel, that can be combined with other related proposed aspects to indicate to the compiler that a loop should be executed in parallel.

Similarly, the parallel aspect could be applied to a subprogram call to indicate that the call should be executed in parallel. Our proposal in [7] provides several levels of control. At the highest level, illustrated in the loop example shown above, the underlying supporting software objects are hidden from view in the application, other than as indicated by the parallel aspect. When finer grain control is needed, it may be necessary to provide more guidance to the compiler to correctly transform the problem or to use user-controlled parallelism strategies.

```ada
with Work_Seeking_Integer.Addition_Loops;
with Parallel_Configuration;
...;
declarer
  Sum : Integer := 0;
begin
  for I in 1 .. N with Parallel,
    Parallel_Manager => Work_Seeking_Integer.Addition_Loops.Work_Seeking_Manager,
    Task_Pool => Parallel_Configuration.Worker_Pool,
    Chunk_Size => 100,
    Accumulator => Sum
loop
  Sum := Sum + I;
end loop;
end;
```

This example illustrates the use of a specific parallelism strategy, and a programmer specified task pool object.

The interfaces described in this paper illustrate how the lower level generic code could be structured and implemented.

Our proposal in [7] discussed a general solution that could be applied to the full set of Ada. However, the hard real-time community has generally shied away from the use of multiple cores, due to the undeterministic behaviour such usage would entail. For example, spin-locks typically used to synchronize tasks across multiple cores can result in unbounded times to acquire a lock, due to the inherent unfairness of the mechanism [9]. Ada implementers have found ways to safely work around these limitations through techniques such as fair locks [10]. This raises the question whether a parallel framework such as the one we proposed could be extended to satisfy hard real-time timing constraints. In particular, the goal is to see if the restrictions of the Ravenscar [11] profile can be satisfied with our approach from the language perspective, after hard real-time timing constraints have been considered.

3 Framework summary
The underlying framework behind the syntactic improvements consists of a set of generics that implement an abstraction that tackles a parallelism opportunity (Figure 1). The notion of a parallelism opportunity comes up frequently in any discussion of parallelism, so we introduce the acronym POP as an abbreviation for Parallelism OPportunity. The generics consist of two software abstractions that work together to execute an algorithm in parallel.

The first abstraction is the task pool, which provides a set of general-purpose worker tasks that can be assigned to a POP, and the second is a parallelism manager that coordinates the workers and delivers any overall results to the application task (Figure 2).

We give the name “tasklette” to the code fragments assigned to the workers from the task pool. Tasklettes do not have any syntactic construct but merely serve to describe the abstraction of a lightweight thread of execution. Task pool objects can be declared at library level and interact only with the parallelism manager. The parallelism manager is an object of a generic instantiation that is declared at the site of the POP, with the users’ code.

4 Need for Task Pools

A potential reusable component of a parallelism framework is the notion of a task pool. A task pool is a pool of general purpose worker tasks that can be made available to tackle parallelism opportunities by providing workers to a parallelism manager that divides the work between the workers which in turn are executed on different cores.

A common pattern for exploiting a POP involves a single task dividing the work to be executed into smaller chunks, and giving the smaller chunks to a set of worker tasks, which then execute in parallel on the available cores until the work is complete, after which the single task proceeds sequentially once again.

In the general case, tasks can be dynamically created and destroyed for each POP, or can be put in a general-use task pool with more added as needed. For real-time programs, however, the non-determinism of task creation and destruction means that a static task pool is essential.

In Ada, the Ravenscar profile [11] was devised specifically to meet the demands of the real-time community. The restrictions of the Ravenscar profile do however offer certain challenges if one wants to find ways to take advantage of parallelism opportunities when multiple cores are available for use (as presented in Section 6). One of these challenges is that all tasks need to be statically declared at library level. If worker tasks are to be employed to provide algorithm acceleration in a general purpose reusable way, these tasks need to be declared statically as well. The task pool abstraction
is ideal for this situation, as it allows the worker tasks to be managed, and the allocation of these processing resources are controlled from a central task pool manager. Task pool objects can easily be declared at library level with the declarations for the workers.

The Ravenscar Tasking Profile implementation of our parallelism mechanism differs in one key aspect from the general purpose model.

Task pools are supported by parallelism manager objects (one per POP – either specified by the programmer or generated by the implementation), which usually are in a nested scope. With Ravenscar, such objects cannot be protected objects, because the Ravenscar profile disallows nested protected objects. The parallelism managers typically need to synchronize at certain points during the processing, such as when the results of workers need to be combined (reduced) into a single result. A task pool is a logical place to provide the needed synchronization primitives, since the task pool can be declared at library level.

In the general model, protected objects are created at the site of the POP, to manage that specific parallel execution. In the Ravenscar model, protected objects are in the task pool manager (declared at the library level), managing all parallel opportunities which use the same task pool. In the Ravenscar model, worker tasks are requested from the pool, and when finishing the work parcel, need to return to the pool object to block for the next work parcel. In the non-Ravenscar, worker tasks are requested from the pool and only return when all work parcels are finished, as they can use the protected object associated with the parallelism manager declared at the site of the POP to block and wait.

As one might expect, the non-Ravenscar model results in a simpler interface with the disadvantage that it cannot support the restrictions of Ravenscar.

5 Non-Ravenscar Task Pools

The following presents the proposed interface for the non-Ravenscar task pool model. This package also provides a facility for specifying a default task pool.

```ada
package Ada.Parallel.Task_Pools is

   -- A Work Plan gives the task pool client (the parallelism manager) the full
   -- control on how the work division is managed. The task pool only provides
   -- the worker, the work plan defines the work to be done.
   --
   type Work_Plan is limited interface;

   procedure Engage (Plan : Work_Plan) is abstract;
   -- When a worker starts executing, it engages the work plan. The parallelism
   -- manager defines the work. The Engage is called once and executes the plan.
   -- Upon returning, the Worker is once again idle and returns to the task pool.

   type Task_Pool_Interface is limited interface;

   procedure Offer_Work (Pool : in out Task_Pool_Interface;
                         Item   : aliased in out Work_Plan'Class;
                         Worker_Count : Positive_World_Count) is abstract;
   -- Allows a work plan to request workers from the task pool. The Work plan is
   -- offered to the task pool, which 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 application client code

   function Default_Task_Pool return not null access Task_Pool_Interface'Class;

   procedure Set_Default_Task_Pool
                        (Pool : aliased in out Task_Pool_Interface'Class);

end Ada.Parallel.Task_Pools;
```

The prescribed use for this pool is that the Parallelism Manager derives its own work plan object from the Work_Plan interface provided by this package. Such a work plan can contain any needed state information associated with the POP, but the main purpose of this object is to provide the Engage callback that the task pool calls when releasing a task from the pool.

When the parallelism manager wishes to utilize the pool, it calls Offer_Work to specify the number of workers needed to perform the work, as well as the work plan object that provides the Engage callback. The parallelism manager performs all
needed processing within the Engage callback, and then returns from the callback. This causes the enclosed worker task to
return idle to the task pool. This approach permits strategies such as work-seeking and work-stealing where workers request
more work from the parallelism manager when they complete their work assignments. This all happens within the
parallelism manager code, and does not involve any interactions with the task pool manager.

5.1 Derivations of the Task Pool Interface

The task pool interface allows for various schemes and implementations. For example, one could derive a bounded task
pool that manages a static, fixed set of workers, while another derivation could implement an unbounded task pool that
dynamically creates more worker tasks if needed, and possibly terminating workers which are idle during extended periods
of time. Furthermore, discriminants on the derived objects may be used to provide extra controls and extra primitive
subprograms may be added if necessary, such as priority control or storage size.

The task pool interface allows for flexibility so that programmers can either rely on a default task pool, or can specify
separate task pool objects for different parts of the application. The idea is that there is a different parallelism manager
object declared at the site of each POP, and each parallelism manager could reference different task pools if necessary.

5.2 A Non-Ravenscar Bounded Task Pool Example

The following illustrates a potential bounded task pool design. Note that, when a bounded task pool object is declared, the
number of workers in the pool is specified as one of the discriminants in the declaration. This allows different bounded task
pools to have different worker counts. Similarly, the storage size of the worker tasks can be specified, as well as the ceiling
priority of the task pool's protected object that manages the pool. While the task pool object does not have default
discriminants, the Create calls do have parameters that default to reasonable values to provide ease of use for cases where
programmers are not concerned with those details. The second Create call allows all tasks in the task pool to be assigned to
a specific dispatching domain. Otherwise, the dispatching domain of a task pool defaults to the dispatching domain of the
task that declared the task pool.

with System.Storage_Elements;
with System.Multiprocessors.Dispatching_Domains;

package Ada.Parallel.Task_Pools.Bounded is

type Task_Pool (Number_Of_Workers : Positive_Worker_Count;
   Storage_Size   : System.Storage_Elements.Storage_Count;
   Ceiling_Priority : System.Priority)
   is limited new Task_Pool_Interface with private;
   -- task pool object type that has a pool of real Ada tasks to process virtual
   -- thread fibers that are submitted to the pool for processing.

function Create (Number_Of_Workers : Positive_Worker_Count;
   Storage_Size   : System.Storage_Elements.Storage_Count
                     := Default_Worker_Storage_Size;
   return Task_Pool;
   -- This call provides a means to create a task pool without having to specify
   -- all the discriminants. Alternatively, a Task_Pool object can just be
   -- declared without calling Create.

function Create (Number_Of_Workers : Positive_Worker_Count;
   Storage_Size   : System.Storage_Elements.Storage_Count
                     := Default_Worker_Storage_Size;
   Domain          : in out Dispatching_Domains.Dispatching_Domain)
   return Task_Pool;
   -- This Create call is identical to the first except that it also assigns all
   -- the tasks in the task pool to the specified dispatching domain. This call can
   -- only be made during elaboration before the main program has started, and thus
   -- can only assign workers from the system dispatching domain to a specific domain.

private

use Ada;

task type Worker
   (Pool       : access Task_Pool'Class := null;
Id : Worker_Id := Worker_Id'Last;
Storage_Size : System.Storage_Elements.Storage_Count := Default_Worker_Storage_Size
pragma Storage_Size (Storage_Size);

entry Work_Offered (Item : aliased in out Work_Plan'Class;
Priority : System.Priority);
end Worker;

type Worker_Array is array (Worker_Id range <>) of Worker;
-- The Ada tasks in the task pool

type Task_Manager;

function Create_Worker
(Pool : access Task_Pool'Class;
Storage_Size : System.Storage_Elements.Storage_Count) return Worker;
-- Creates a task in the task pool

type Idle_List is array (Worker_Id range <>) of Worker_Id;

function Create_Idle_List
(Number_Of_Workers : Positive_Worker_Count) return Idle_List;

protected type Task_Manager
(Pool : access Task_Pool'Class;
Number_Of_Workers : Positive_Worker_Count;
Priority : System.Priority;
Storage_Size : System.Storage_Elements.Storage_Count) with Priority => Priority is

entry Request_Workers
(Worker_Count : Positive_Worker_Count;
First_Worker : out Worker_Id);

entry Offer_Work
(Item : aliased in out Work_Plan'Class;
Priority : System.Priority);

procedure Finished_Offer;

entry Completed_Work (Worker : Worker_Id);

procedure Assign
(Domain : in out Dispatching_Domains.Dispatching_Domain);

private

entry Allocate_Workers
(Worker_Count : Positive_Worker_Count;
First_Worker : out Worker_Id);

Outstanding_Workers : Worker_Count_Type := 0;
Idle_Workers : Idle_List (1 .. Number_Of_Workers)
:= Create_Idle_List (Number_Of_Workers);
Workers : Worker_Array (1 .. Number_Of_Workers)
:= (others => Create_Worker (Pool => Pool,
Storage_Size => Storage_Size));

Busy : Boolean := False;
Requested_Workers : Positive_Worker_Count;

end Task_Manager;

type Task_Pool
(Number_Of_Workers : Positive_Worker_Count;
Storage_Size : System.Storage_Elements.Storage_Count;
Ceiling_Priority : System.Priority) is limited new Task_Pool_Interface with
record

-- Temporary used for initializing worker ids
Next_Id : Worker_Id := 1;

Manager : Task_Manager (Task_Pool'Access,
Number_Of_Workers,
Ceiling_Priority,
overriding
procedure Offer_Work
(Pool          : in out Task_Pool;
Item          : aliased in out Work_Plan'Class;
Worker_Count  : Positive_Worker_Count)
-- Allows a work plan to request workers from the task pool. The Work
-- plan is offered to the task pool, which is then engaged by each
-- worker up to the requested Worker_Count
pragma Inline {Offer_Work};
package body Ada.Parallel.Task_Pools.Bounded is
protected body Task_Manager is
entry Allocate_Workers
(Worker_Count : Positive_Worker_Count;
 First_Worker : out Worker_Id)
when Outstanding_Workers + Requested_Workers <= Number_Of_Workers is
begin
  -- Get the index of the first worker of a block of workers in the Idle worker list
  First_Worker :=
      Number_Of_Workers - (Outstanding_Workers + Worker_Count);
  Busy := True;
end Allocate_Workers;
procedure Assign
(Domain : in out Dispatching_Domains.Dispatching_Domain) is
begin
  for I in Workers'Range loop
    Dispatching_Domains.Assign_Task (Domain => Domain,
      CPU    => Multiprocessors.Not_A_Specific_CPU,
      T      => Workers (I)'Identity);
  end loop;
end Assign;
entry Completed_Work (Worker : Worker_Id) when not Busy is
begin
  Outstanding_Workers := Outstanding_Workers - 1;
  Idle_Workers (Number_Of_Workers - Outstanding_Workers) := Worker;
end Completed_Work;
procedure Finished_Offer is
begin
  Busy := False;
end Finished_Offer;
entry Offer_Work (Item      : aliased in out Work_Plan'Class;
  Priority : System.Priority) when True is
begin
  Worker_Index := Positive_Worker_Count;
  Worker_Index := Idle_Workers (Number_Of_Workers - Outstanding_Workers);
  Outstanding_Workers := Outstanding_Workers + 1;
  requeue Workers (Worker_Index).Work_Offered;
end Offer_Work;
entry Request_Workers
(Worker_Count : Positive_Worker_Count;
 First_Worker : out Worker_Id) when True is
begin
  Requested_Workers := Worker_Count;
  requeue Allocate_Workers;
end Request_Workers;
task body Worker is
  Plan : access Work_Plan'Class;
begin -- Worker

Work_Loop : loop

select
  accept Work_Offered (Item : aliased in out Work_Plan'Class;
                       Priority : System.Priority)
  do
    Dynamic_Priorities.Set_Priority (Priority);
    Plan := Item'Unchecked_Access;
  end Work_Offered;

or
  terminate;
end select;

Plan.Engage;
Pool.Manager.Completed_Work (Id);

exception
  when others =>
    -- Should probably at least log something here, but we don't want exceptions to
    -- take tasks out of the task pool. The handling and reporting of exceptions
    -- is handled in the parallelism manager
    null;
end loop Work_Loop;
end Worker;

function Create
  (Number_Of_Workers : Positive_Worker_Count;
   Storage_Size      : System.Storage_Elements.Storage_Count := Default_Worker_Storage_Size;
return Task_Pool is
begin
  return Pool : Task_Pool (Number_Of_Workers,
                           Storage_Size,
                           Ceiling_Priority)
do null;
  end return;
end Create;

function Create
  (Number_Of_Workers : Positive_Worker_Count;
   Storage_Size      : System.Storage_Elements.Storage_Count := Default_Worker_Storage_Size;
   Domain             : in out Dispatching_Domains.Dispatching_Domain)
return Task_Pool is
begin
  return Pool : Task_Pool (Number_Of_Workers,
                           Storage_Size,
                           Ceiling_Priority)
do
    -- Assign all the workers in the pool to the specific domain
    Pool.Manager.Assign (Domain);
  end return;
end Create;

function Create_Idle_List
  (Number_Of_Workers : Positive_Worker_Count) return Idle_List is
begin
  return Result : Idle_List (1 .. Number_Of_Workers) do
    for I in Result'Range loop
      Result (I) := I;
    end loop;
  end return;
end Create_Idle_List;

function Create_Worker
  (Pool      : access Task_Pool'Class;
   Storage_Size : System.Storage_Elements.Storage_Count) return Worker
is
  Id : constant Worker_Id := Pool.Next_Id;
begin
  Pool.Next_Id := Pool.Next_Id + 1;
  return New_Worker : Worker (Pool,
    Id,
    Storage_Size)
  do
    null;
  end return;
end Create_Worker;

procedure Offer_Work
  (Pool         : in out Task_Pool;
   Item         : aliased in out Work_Plan'Class;
   Worker_Count : Positive_Worker_Count)
is
begin
  Pool.Manager.Request_Workers (Worker_Count, First_Worker);
  for I in 1 .. Worker_Count loop
    Pool.Manager.Offer_Work (Item, Current_Priority);
  end loop;
  Pool.Manager.Finished_Offer;
end Offer_Work;
end Parallel.Task_Pools.Bounded;

5.3 Non-Ravenscar Parallelism Managers

As outlined in [7], the instantiation of parallelism managers involves low level parent generic instantiations that specify the reduction operations and loop iterator types, as well as child generic instantiation that creates the parallelism manager object type. To illustrate this for a parallel loop involving integer reduction, the following generic is the low level generic specifying the properties of the reduction. This includes the reduction type, the reducing function, and the identity value.

generic
  type Result_Type is private;
  -- Final Result type
  with function Reducer (Left, Right : Result_Type) return Result_Type;
  -- Reducing operation used to compute final result. The operation
  -- needs to take two values and reduce into a single value (the Left)
  -- parameter.
  Identity_Value : Result_Type;
  -- A special value that when applied as the right operand of the
  -- Reducing function, does not change the value of the reducing result.

package Ada.Parallel.Functional_Reduction is
  use Ada.Parallel.Functional_Reduction;
end Ada.Parallel.Functional_Reduction;

This generic does not require a body and can be instantiated at library level. For instance:

with Ada.Parallel.Functional_Reduction;
package Integer_Addition is new Ada.Parallel.Functional_Reduction
  (Result_Type => Integer,
   Reducer    => "+",
   Identity_Value => 0);

This instantiation could be shared between parallel loops and parallel subprograms.

The second level instantiation specific to parallel loops specifies the iterator type. It also specifies the interface that a parallelism manager library writer would need to implement.

generic
  type Iteration_Index_Type is <>;
  -- Loop Index type
package Ada.Parallel.Functional_Reduction.Loops is

  type Parallelism_Manager is limited interface;

  procedure Execute_Parallel_Loop
  (Manager      : Parallelism_Manager;
   From         : Iteration_Index_Type := Iteration_Index_Type'First;
   To           : Iteration_Index_Type := Iteration_Index_Type'Last;
   Worker_Count : Worker_Count_Type := Default_Worker_Count;
   Process      : not null access procedure (Start, Finish ; Iteration_Index_Type;
                                                  Item         : in out Result_Type);
   Result       : aliased in out Result_Type) is abstract;

end Ada.Parallel.Functional_Reduction.Loops;

This generic also does not require a body and can be instantiated at library level. For instance:

  with Integer_Addition;
  with Ada.Parallel.Functional_Reduction.Loops;

package Integer_Addition_Loops is new Integer_Addition.Loops (Iteration_Index_Type => Positive);

The final instantiation creates the parallelism manager, which implements some strategy such as work sharing, work-seeking, or work-stealing. The following shows an implementation that could be used for this instantiation.

  with Ada.Parallel.Task_Pools;
  with System.Multiprocessors.Dispatching_Domains; use System.Multiprocessors;

generic
package Ada.Parallel.Functional_Reduction.Recursion.Work_Seeking is

  type Recursion_Dispatcher is limited interface;

  function Recurse (Dispatcher : Recursion_Dispatcher;
                     Item       : Work_Type) return Result_Type
      is abstract;

  type Recursion_Dispatcher_Access is access all Recursion_Dispatcher'Class;

  type Work_Seeking_Manager is access all Recursion_Dispatcher_Access;

  type Work_Seeking_Manager_Access is access all Recursion_Dispatcher_Access;

  type Work_Seeking_Manager with record
    Dispatcher : aliased Recursion_Dispatcher_Access := null;
  end record;

  not overriding function Create
  (Others_Seeking_Work : aliased in out Work_Seeking_State;
   Other_Workers       : access Work_Seeking_State := null;
   Affinity            : access Dispatching_Domains.CPU_Set := null;
   Allow_Migration     : Boolean := True)
      return Work_Seeking_Manager;

private

  overriding function Execute_Parallel_Subprogram
  (Manager      : in out Work_Seeking_Manager;
   Item         : Work_Type;
   Worker_Count : Worker_Count_Type := Default_Worker_Count;
   Process      : not null access function (Item : Work_Type) return Result_Type)
      return Result_Type
  with Pre'Class => Manager.Dispatcher = null and then
  Worker_Count <= Manager.Workers.Available_Workers,
  Post'Class => Manager.Dispatcher = null;
package body Ada.Parallel.Functional_Reduction.Recursion.Work_Seeking is

function Create (Others_Seeking_Work : aliased in out Work_Seeking_State;
Workers             : not null access Task_Pools.Task_Pool_Interface;'Class
Affinity            : access Dispatching_Domains.CPU_Set := null;
Allow_Migration     : Boolean := True)
return Work_Seeking_Manager is
begin
return Manager : Work_Seeking_Manager
(Workers, 
Others_Seeking_Work'Access,
Ceiling_Priority, 
Affinity, 
Allow_Migration)
do
null;
end return;
end Create;

function Execute_Parallel_Subprogram
(Manager      : in out Work_Seeking_Manager;
Item         : Work_Type;
Worker_Count : Worker_Count_Type := Default_Worker_Count;
Process      : not null access function (Item : Work_Type) return Result_Type)
return Result_Type is
type Work_Seeking_Plan is new Parallel.Task_Pools.Work_Plan with null record;
overriding
procedure Engage (Plan : Work_Seeking_Plan);
protected type Internal_Recursion_Dispatcher is new Recursion_Dispatcher with
pragma Priority (Manager.Ceiling_Priority);
... procedure Check_Completion;
procedure Save_Exception (E : Exceptions.Exception_Occurrence);
private
Exception_Raised    : Boolean := False;
Saved_Exception     : Ada.Exceptions.Exception_Occurrence;
...
end Internal_Recursion_Dispatcher;
overriding function Recurse
(Dispatcher : Internal_Recursion_Dispatcher;
Item : Work_Type) return Result_Type;
protected body Internal_Recursion_Dispatcher is
procedure Check_Completion is
begin
if Exception_Raised then
Ada.Exceptions.Reraise_Occurrence (X => Saved_Exception);
end if;
end Check_Completion;
procedure Save_Exception (E : Exceptions.Exception_Occurrence) is
begin
Exception_Raised := True;
Ada.Exceptions.Save_Occurrence (Target => Saved_Exception, Source => E);
end Save_Exception;
...
end Internal_Recursion_Dispatcher;

-- Allow client to call recursion routine
Internal_Dispatcher : aliased Internal_Recursion_Dispatcher;

procedure Engage
(Plan : Work_Seeking_Plan)
is
    Value : Work_Type;
    Result : Result_Type := Identity_Value;
begin -- Engage
...
    Internal_Dispatcher.Request_Work (Item => Value);
...
    Result := Reducer (Left => Result,
        Right => Process (Value));
exception
    when E : others =>
        Work_Sharing_Manager.Save_Exception (E);
end Engage;

function Recurse
(Dispatcher : Internal_Recursion_Dispatcher;
    Item : Work_Type) return Result_Type
is
    Work_Accepted : Boolean := False;
begin
...
    Internal_Dispatcher.Offer_Work
    (Item, 
    Work_Accepted);
    if Work_Accepted then
        return Identity_Value;
    end if;
    return Process (Item);
end Recurse;

Result : Result_Type;
Plan   : aliased Work_Seeking_Plan;
begin -- Execute_Parallel_Subprogram
    -- Allow client to call recursion routine
    Manager.Dispatcher := Internal_Dispatcher'Unchecked_Access;
    -- Get the workers needed to do the work from the task pool
    Manager.Workers.Offer_Work (Item => Plan,
        Worker_Count => Worker_Count);
...
    Internal_Dispatcher.Wait_For_Completion (Result);
    Manager.Dispatcher := null;
    Internal_Dispatcher.Check_Completion;
    return Result;
end Execute_Parallel_Subprogram;

6 Ravenscar Task Pools
As mentioned previously, the Ravenscar restrictions impact the task pool design enough that a separate task pool interface was needed. In particular, the following restrictions impacted this design.

No Dynamic Priorities
No Local Protected Objects
No Requeue Statements
No Select Statements
No Task Allocators
No Task Hierarchy
No Task Termination
Simple Barriers
Max Entry Queue Length=>1
Together, these restrictions require that:

1. Worker tasks are declared statically at library level.
2. Parallelism managers cannot hold on to workers and reassign work when work is complete. In other words, work-seeking and work-stealing managers need to return workers to the task pool in these cases, since the manager is typically declared on the stack, and cannot internally declare any protected objects.
3. Each worker will need to wait when idle on its own protected-object or suspension object. There cannot be a single protected object that workers queue on when idle.
4. The Protected object for the task pool cannot directly release workers when work is offered to the pool. The Offer_Work call cannot be an entry, since multiple tasks may be offering work to the pool. Therefore, the Offer_Work call needs to determine which workers should be released, and interact with the protected objects associated with those worker tasks to perform the release.
5. Ravenscar Task Pools are bounded task pools, since tasks cannot be created dynamically.
6. Parallelism managers that return intermediate workers to the pool before the total work is complete need to have a means to associate worker state managed by the parallelism manager with workers that are getting reassigned to the parallelism manager. Similarly, the task pool needs to maintain its own index associated with a worker. To provide these associations, the concept of a Pool_Index, and a Plan_Index is needed. A Pool_Index is the identifier for the task that the task pool uses for its own purposes, while the Plan_Index is an identifier that the parallelism manager uses to associate a worker with the internal worker state of the parallelism manager.
7. The assignment of Plan_Ids within the parallelism manager has to involve interactions with the task pool, because the parallelism manager cannot declare its own protected objects. Instead, the protected object of the pool provides the protected operations that guarantee that the next Plan_Id can safely be determined by incrementing a local state variable.
8. Since dynamic task priorities are disallowed, there will need to be a separate task pool object declared for each priority level. Application tasks that use the task pool will need to ensure that the task pools used for a particular application task have the same priority level.

```ada
pragma Profile (Ravenscar);
package Ada.Parallel.Ravenscar_Task_Pools is

  type Pool_Worker_Count is new Worker_Count_Type;
    -- A Pool_Index identifies a worker within the Task Pool
  subtype Pool_Index is Pool_Worker_Count;
    -- A Plan_Index identifies a worker within the work plan
  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 parallelism 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 : Work_Plan;
    Worker : Pool_Index;
    Item : Plan_Index) is abstract;
    -- When a worker starts executing, it engages the work plan. The
    -- parallelism manager client defines the work. The Engage is called
    -- once 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.
-- The Requester is the clients id for the worker that requested
-- the work. Item is returned as the clients id for the worker that
-- will accept the work. This routine is
-- intended 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 an engaged work plan has
-- completed, 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 Offer_Work (Pool : in out Task_Pool_Interface;
  Plan : aliased in out Work_Plan; 
  Item : Plan_Index) is abstract;
-- Allows a work plan to request workers from the task pool. The Work
-- plan is offered to the task pool, which 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 application client code

procedure Offer_Work_To_Group (Pool         : in out Task_Pool_Interface;
  Plan         : aliased in out Work_Plan; 
  Worker_Count : Positive_Worker_Count)
is abstract;
-- Allows a client to request a group of workers all at once

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;
  Requester : Plan_Index;
  Item : out Plan_Index) is null;
-- Lets the client determine what its next worker id will be, before the work is offered
-- to a worker. The task pool calls back to the client via the Starting primitive to allow
-- the client to determine the next worker id safely from within a protected operation.

procedure Finished_Work (Pool : in out Task_Pool_Interface;
  Worker : Pool_Index;
  Plan : aliased in out Work_Plan;
  Item : Plan_Index) is null;
-- Lets the client indicate when a worker has completed its work.
-- This calls back to the client via the Completing primitive to allow the client to
-- perform any clean up or finalization safely from within a protected operation.
end Ada.Parallel.Ravenscar_Task_Pools;

6.1 A Potential Ravenscar Task Pool

The following shows a possible design for a Ravenscar task pool. One of the differences to note is that the parameters to the task pool are provided via generic formal parameters instead of via discriminants to the task pool object. This is largely needed because of the static nature of the declarations needed to construct the pool. Otherwise, the interface is quite straightforward.

pragma Profile (Ravenscar);

with System.Storage_Elements;
with System.Multiprocessors; use System;

generic
  Storage_Size : System.Storage_Elements.Storage_Count := Default_Worker_Storage_Size;
The following example illustrates use and declaration of this task pool in an application for a target with 4 cores.

```ada
package Parallel_Ravenscar_Configuration is
  package Ravenscar_Task_Pools is new
    (Storage_Size => Ada.Parallel.Default_Worker_Storage_Size,
     Worker_Priority => Ada.Parallel.Default_Worker_Priority,
     Number_Of_Workers => 4);
  Worker1 : aliased Ravenscar_Task_Pools.Worker (Core => 1);
  Worker2 : aliased Ravenscar_Task_Pools.Worker (Core => 2);
  Worker3 : aliased Ravenscar_Task_Pools.Worker (Core => 3);
  Worker4 : aliased Ravenscar_Task_Pools.Worker (Core => 4);
  Workers : aliased Ravenscar_Task_Pools.Worker (Core => 1 .. Number_Of_Workers);
  Worker_Pool : aliased Ravenscar_Task_Pools.Task_Pool (Workers => Workers'Access);
end Parallel_Ravenscar_Configuration;
```

### 6.2 A Potential Ravenscar Parallelism Manager

While the interface to Ravenscar task pools has diverged from the general task pool interface, these interfaces are strictly between the Task Pool Manager and the Parallelism Manager, which are provided by compiler vendors or third party library writers. Fortunately, the interface between the application writer and the parallelism manager is common between Ravenscar and non-Ravenscar. The following shows a specification for a Work-Seeking Parallel loop that performs integer reductions. The intermediate instantiations for the Reduction and the Loop manager can be shared between Ravenscar and Non-Ravenscar parallelism managers, and have been shown in section 5.

```ada
pragma Profile (Ravenscar);
with Ada.Parallel.Ravenscar_Task_Pools;
generic
package Ada.Parallel.Functional_Reduction.Loops.Ravenscar_Work_Seeking is
  type Work_Seeking_Manager
    (Workers : not null access Ravenscar_Task_Pools.Task_Pool_Interface'Class;
     Chunk_Size : Natural) is limited new Parallelism_Manager with null record;

not overriding function Create
  (Workers : not null access Ravenscar_Task_Pools.Task_Pool_Interface'Class;
     Chunk_Size : Natural := Default_Chunk_Size) return Work_Seeking_Manager;

overriding procedure Execute_Parallel_Loop
  (Manager : Work_Seeking_Manager;
   From : Iteration_Index_Type := Iteration_Index_Type'First;
   ... Implementation Defined
end Parallel_Ravenscar_Configuration;
```
To : Iteration_Index_Type := Iteration_Index_Type'Last;
Worker_Count : Worker_Count_Type := Default_Worker_Count;
Process : not null access procedure (Start, Finish : Iteration_Index_Type;
Item : in out Result_Type);
Result : in out Result_Type);


with Ada.Synchronous_Task_Control; use Ada;

package body Ada.Parallel.Functional_Reduction.Loops.Ravenscar_Work_Seeking is

function Create (Workers : aliased in out Ravenscar_Task_Pools.Task_Pool_Interface'Class;
Chunk_Size : Natural := Default_Chunk_Size)
return Work_Seeking_Manager is
begin
return Manager : Work_Seeking_Manager (Workers'Access,
Chunk_Size)
do
null;
end return;
end Create;

procedure Execute_Parallel_Loop (Manager : Work_Seeking_Manager;
From : Iteration_Index_Type := Iteration_Index_Type'First;
To : Iteration_Index_Type := Iteration_Index_Type'Last;
Worker_Count : Worker_Count_Type := Default_Worker_Count;
Process : not null access procedure (Start, Finish : Iteration_Index_Type;
Item : in out Result_Type);
Result : in out Result_Type)
is

Iterations : constant Positive :=
Positive'Base (Iteration_Index_Type'Pos (To) - Iteration_Index_Type'Pos (From)) + 1;
...

type Work_Seeking_Plan is limited
new Ravenscar_Task_Pools.Work_Plan with
record
Completed : Synchronous_Task_Control.Suspension_Object;
...
end record;

overriding procedure Engage (Plan : in out Work_Seeking_Plan;
Worker : Ravenscar_Task_Pools.Pool_Index;
Item : Ravenscar_Task_Pools.Plan_Index);

overriding procedure Starting (Plan : in out Work_Seeking_Plan;
Requester : Ravenscar_Task_Pools.Plan_Index;
Item : out Ravenscar_Task_Pools.Plan_Index);

overriding procedure Completing (Plan : in out Work_Seeking_Plan;
Item : Ravenscar_Task_Pools.Plan_Index);
...

procedure Completing (Plan : in out Work_Seeking_Plan;
Item : Ravenscar_Task_Pools.Plan_Index) is
begin
...
-- Now do the real reduction
Reducing_List.Reduce
(Container => Reduction_List,
Item => Initial_Work (Item).Value,
Position => Reducing_List.To_Cursor
(Worker => Worker_Count_Type (Item)));
...
if Plan.Outstanding_Workers = 0 then
  Synchronous_Task_Control.Set_True (Plan.Completed);
...
end if;
end Completing;
...
Work_Plan : aliased Work_Seeking_Plan;

procedure Engage
(Plan : in out Work_Seeking_Plan;
Worker : Ravenscar_Task_Pools.Pool_Index;
Item : Ravenscar_Task_Pools.Plan_Index)
is
...
begin
  Work_Loop : loop
  Process (Chunk_Start, Chunk_Finish, Item);
  ...
  Manager.Workers.Next_Worker_Id
  (Plan => Work_Plan,
   Requester => Item,
   Item => Work_Id);
  ...
  Manager.Workers.Offer_Work
  (Work_Plan,
   Work_Id);
  ...
  end loop Work_Loop;

-- Calls to Task Manager, which calls Plan to indicate worker has completed.
Manager.Workers.Finished_Work (Worker,
Work_Plan,
Item); exception
when E : others =>
  Exception_Info (Item).Exception_Raised := True;
  Ada.Exceptions.Save_Occurrence
  (Target => Exception_Info (Item).Saved_Exception,
   Source => E);
  Manager.Workers.Finished_Work (Worker,
  Work_Plan,
  Item);
end Engage;

procedure Starting
(Plan : in out Work_Seeking_Plan;
Requester : Ravenscar_Task_Pools.Plan_Index;
Item : out Ravenscar_Task_Pools.Plan_Index) is
begin
...  
end Starting;
Reduction_Result : Result_Type;
begin -- Execute Parallel Loop
...
Synchronous_Task_Control:Set_False (Work_Plan.Completed);
...
-- Get the workers needed to do the work from the task pool
Manager.Workers.Offer_Work_To_Group
(Plan => Work_Plan,
   Worker_Count => Effective_Workers);
Synchronous_Task_Control:Suspend_Until_True (Work_Plan.Completed);
...
Result := Reducer (Result, Reduction_Result);
if Work_Plan.Exception_Raised then
   Exceptions.Reraise_Occurrence
      (X => Exception_Info (Work_Plan.Saved_Exception).Saved_Exception);
end if;
end Execute_Parallel_Loop;
end Parallel.Functional_Reduction.Loops.Ravenscar_Work_Seeking;

7 Further Issues

7.1 Dispatching Domains
For non-Ravenscar task pools, it is desirable to specify the dispatching domain associated with the workers in a task pool. There are many possible schemes. For example, a set of cores might be dedicated to acceleration of tasks, and could have its own dispatching domain. Dispatching domains need to be created statically before program startup. The Bounded Task Pool example shows how a task pool could be implemented to support the assignment of a task pool object to a particular dispatching domain. Within a dispatching domain, it may be desirable to dynamically manipulate the affinities of the workers. This dynamic manipulation, if needed, is better left to the parallelism manager objects. For example, it may be desirable to limit migration of the workers to a subset of the cores in the associated dispatching domain, or it may be desirable to specify the precise affinity of each worker task and completely disallow processor migration altogether.

7.2 CPU Affinities
Currently Ada provides a mechanism to set the processor affinity of a task to a specific CPU, or to all CPU's within a dispatching domain. There is no means however to set the processor affinity to a subset of the CPU's in a dispatching domain, where the number of CPU's is greater than 1 but less than the number of CPU's in the dispatching domain, even though modern OS's (eg., Linux, Windows) provide this capability. Having this ability would allow for more flexibility and newer scheduling approaches. An AI for Ada 2012 is currently in the works [12] that provides a more flexible means to define non-contiguous sets of CPU's, called a CPU_Set. IRTAW should consider whether a new call should be provided to set the affinity of a task to the CPU's associated with a CPU_Set.

e.g.

   procedure Set_CPU (Set : CPU_Set;

7.3 Priorities
Priorities are still crucial in real time systems that maintain a contiguous map from the real-time priorities down to the idle task(s). The risks of priority inversion increases when multiple dispatching domains are introduced.

There are several approaches to addressing this issue. One approach is to ensure that every task pool only manages tasks of
the same priority. Applications that use the task pool would need to ensure that the priority of the task matched that of the task pool. Such a relationship can be enforced by the use of Ada 2012 Preconditions. Another approach is to provide a task pool that can provide workers for application tasks of different priorities, but to require that statically the number of workers in the task pool is sufficient to service the worst-case worker loads. Such a determination could be made by static analysis.

Another issue is that there is no mechanism in Ravenscar to determine the priority of a task because the package Ada.Dynamic_Priorities is banned from use. While it makes sense to ban the ability to set the priority in Ravenscar, it would be useful to at least determine the priority of a task in some static fashion. This could be useful for preconditions such as shown for the private specification of the Offer_Work subprogram in section 5.2, to ensure that the user of task pool has a priority within an expected range. This issue was recently discussed informally by the ARG, as an issue that should be passed on to IRTAW to consider.

7.4 Hard Real-Time and Task Pools

We acknowledge the concern of the real-time community about the risk of priority inversion and the timing issues surrounding parallel access to resources. We believe that these issues are solvable with a very careful creation of dispatching domains. In another paper [13], we are addressing these issues.

8 Conclusions

Between this paper and the Ada Europe paper [7], a complete mechanism has been proposed to introduce a powerful capability into to Ada to have a fine-grained parallelism mechanism that facilitates the writing of parallel applications in a more natural manner. This paper shows how the mechanism can be extended to support user-supported task pools, and to extend that to supporting real time systems.

References

On real-time partitioned multicore systems

Juan Zamorano
jzamora@datsi.fi.upm.es

Juan A. de la Puente
jpuente@dit.upm.es

Universidad Politécnica de Madrid (UPM), Spain

Abstract

Partitioning is a common approach to developing mixed-criticality systems, where partitions are isolated from each other both in the temporal and the spatial domain in order to prevent low-criticality subsystems from compromising other subsystems with high level of criticality in case of misbehaviour. The advent of many-core processors, on the other hand, opens the way to highly parallel systems in which all partitions can be allocated to dedicated processor cores. This trend will simplify processor scheduling, although other issues such as mutual interference in the temporal domain may arise as a consequence of memory and device sharing. The paper describes an architecture for multi-core partitioned systems including critical subsystems built with the Ada Ravenscar profile. Some implementation issues are discussed, and experience on implementing the ORK kernel on the XtratuM partitioning hypervisor is presented.

1 Introduction

Mixed-criticality systems are composed of several subsystems, some of which may have a high level of criticality, possibly requiring certification with respect to some domain-specific standard, while some others are not so critical and can be developed using a less demanding verification and validation (V&V) process. Since certification is generally carried out at system level, unless specific arrangements are made all the system components must be certified to the highest criticality level in the system. This is often unfeasible, as lower criticality subsystems may include COTS components that are not amenable to a strict V&V process, and almost always unpractical, as the cost of certification may be unacceptably high.

Although other approaches are possible (see e.g. [3]), partitioning is a common approach to overcoming this kind of problem. It is based on temporal and spatial isolation (TSI). Subsystems with different levels of criticality are built in such a way that the temporal behaviour of a subsystem does not affect that of other subsystems (temporal separation), and its use of memory is limited to its own memory space, without invading other subsystems’ spaces (spatial separation). In this way, critical subsystems can be certified independently of non-critical subsystems, as a possible misbehaviour in the latter cannot compromise the properties of critical parts of the system.

This principle can be implemented in different ways. A radical approach is physical separation, by implementing critical and non-critical subsystems on different hardware platforms, possibly communicating by means of well defined links. However, as more and more powerful processors have become available, there is a trend towards implementing mixed-criticality systems on a single computer platform. The concept of Integrated Modular Avionics (IMA) [1] is a well-known example of this approach that has developed in the aeronautic field. The term partitioned systems is commonly applied to this kind of system.

*This work has been partially funded by the Spanish Government, project HI-PARTES (TIN2011-28567-C03-01), and by the European Commission FP7 programme, project MultiPARTES (IST 287702).
Partitioned systems consist of a number of partitions, running on a shared computer platform. Each partition hosts a different subsystem with a given criticality level. Partitions are isolated from each other both in the temporal and in the spatial domain, although some means of inter-partition communication are usually provided. A separation kernel takes care of implementing temporal and spatial separation by scheduling the execution of partitions and providing separated memory spaces for them. In this way, only the most critical partitions, together with the separation kernel, are required to undergo the certification process.

Virtualization provides a means to divide a single hardware platform into a number of virtual machines, each with a set of virtual resources that are mapped to the available physical resources. This technique can be used to implement partitioning, by making each virtual machine a separate partition, possibly with a different operating system depending on the criticality requirements of the subsystems, or applications, running on it (figure 1). Although there are different approaches to virtualization, it is generally accepted that the best approach for real-time embedded systems is based on the use of a hypervisor or virtual machine monitor [16].

In the recent years we have applied these ideas to real-time systems written in Ada with the ORK+ kernel and the XtratuM hypervisor [9, 10, 21]. The resulting ORK+/XtratuM platform is currently being used to develop some pilot real-time systems, with promising results. The platform currently runs on monoprocessor computers, but as multicore processors are becoming more and more common in embedded systems (see e.g. European Comission [11]), the need for extending the concepts of partitioned systems to this kind of system has arisen as a natural extension. In the rest of this paper we analyse the implications of using multicore processors to run partitioned systems, and explore some ideas about the implementation of mixed-criticality real-time systems in Ada on top of such kind of architecture.

2 Multicore processors

In the following we shall assume a multicore processor architecture, where a number of identical processor cores are packaged together in a single chip. All cores share a common memory accessed through a computer bus, and caches may be local or global to all the cores. Cache coherence is ensured by hardware. Other hardware resources, such as timers or bus interfaces, may also be shared.

This kind of architecture is supported by the current Ada standard [2] by means of the packages in the System.Multiprocessors hierarchy. A task may be assigned to a dispatching domain, i.e. a set of
processors on which a task may run, and can also be restricted to executing on a particular processor (CPU). In this way, an Ada program can control very precisely the processor on which its tasks run, thus enabling a predictable execution for concurrent and real-time programs [4, 18].

From the real-time systems point of view, multicore architectures pose some important questions that are not fully solved so far:

- Task scheduling methods and resource control protocols have a high complexity and sometimes exhibit unexpected behaviour [8, 12]. Using a fully portioned approach, with every task running always on the same processor, is often used together with FPPS\(^1\) or EDF\(^2\) within each processor in order to enable an analysable temporal behaviour.

- Cache interference and bus contention introduce additional unpredictability and complexity to execution-time analysis [13, 20].

- Shared hardware resources may also add uncertainty to the run-time behaviour of multiprocessor systems.

A version of the Ravenscar profile for multiprocessors using Ada 2012 constructs to provide fully partitioned scheduling has been proposed by Ruiz [17].

### 3 Partitioned multicore systems

Mixed-criticality systems can be implemented as partitioned systems on top of a multicore platform. Figure 2 shows a partitioned architecture built with a modified version of the XtratuM hypervisor\(^3\) that has been taken as a basis for the HI-PARTES\(^4\) and MultiPARTES\(^5\) projects.

As in the monoprocessor architecture, the hypervisor provides an interface with a set of virtual machines that are used to hold partitions, possibly with different operating systems and different criticality levels. However, on a multicore platform each partition can run on one or more virtual processors. Several virtual processors can be mapped to a single physical processor, multiplexing its execution time among all the virtual processors, as is the rule in monoprocessor systems. However, provided there are enough cores, each virtual processor can be mapped to a single physical processor core, thus providing physical parallelism to the execution of partitions. Mixed schemes are also possible.

Indeed, as multicore architectures evolve towards a growing number of processor cores, a fully parallel approach is getting most interesting for mixed-criticality systems. Having one or more physical cores dedicated to each partition simplifies scheduling (see below) and eases temporal separation. However, some problems remain that have to be analysed. The main ones are predictable temporal behaviour, virtualization of hardware devices other than processor cores, and interrupt handling. The next paragraphs discuss more in detail these and other topics.

**Processor allocation.** In order to enable predictable execution, a static assignment of partitions (or virtual processors) to physical cores is assumed.

---

\(^1\)Fixed-priority pre-emptive scheduling.

\(^2\)Earliest-deadline first.

\(^3\)See [www.xtratum.org](http://www.xtratum.org) for more details.

\(^4\)www.dit.upm.es/rts/projects/hipartes

\(^5\)www.multipartes.eu
Scheduling. Scheduling in partitioned systems is commonly done in a hierarchical way. A *global scheduler* multiplexes processor time among partitions, whereas a *local scheduler*, usually part of the partition OS or kernel, rules the execution of application processes and tasks within each partition. Although different approaches are possible (see e.g. [14, 15, 19]), we assume here a static, ARINC-like global scheduling method whenever global scheduling is necessary [7].

However, if there are at least as many cores as partitions, as it is likely to become common in the near future, a simpler approach is possible. If each partition is allocated one or more virtual processors that are mapped one-to-one to physical processor cores, there is no need for global scheduling of processor time. All the partitions run in parallel, and only a local scheduler is needed in each partition in order to allocate processor time to the processes or tasks in the partition. Ideally the performance of physically separated systems could be achieved, with a significant reduction in cost. Nevertheless life is not so simple, as partitions still share memory and other hardware resources such as timers and input-output devices.

Temporal separation. Temporal separation can be achieved in a simple way by physical parallelism. If partitions run on separate processors, the temporal behaviour of a partition is independent from other partitions. However, the calculation of WCET must take into account the interference induced by other partitions accessing shared memory or devices. Whenever possible, virtual devices should be assigned dedicated hardware devices in order to reduce such kind of interference.

Spatial separation. Separation of memory spaces between partitions can be enforced by using hardware memory management units (MMU), as in monoprocessor systems.
**Interrupt handling.** Interrupt sources can be statically assigned to cores by using the programmable interrupt controller (PIC). Notice that input-output registers in the interrupting device can be allocated to the partition to which the core is assigned. In this way, the device can be handled by the partition software without the need for using hypercalls to manage interrupts or to access registers. Special cases are the programmable interrupt timers, that can be statically assigned to partitions so that partition timing services can be implemented efficiently.

**4 Implementing the Ravenscar profile on partitioned multicore systems**

The Ravenscar profile was originally designed to achieve predictable tasking in Ada real-time systems [5]. It has found wide acceptance in industry and academy, and has been incorporated into the Ada standard since the 2005 amendment. Since it was developed for monoprocessor platforms, its use in multiprocessor and partitioned systems has to be analysed and, where necessary, implementation mechanisms have to be devised.

Ruiz [17] discussed the extension of the profile to multiprocessor systems. Fully-partitioned fixed-priority pre-emptive scheduling is proposed in order to enable the temporal behaviour to be analysed, in spite of the reduced worst-case schedulable utilization that can arise. The CPU aspect can be used to statically allocate a task to a processor. This proposal was implemented in a version of GNAT for LEON computers [6]. Protected objects used by tasks on different processors were implemented using fair locks (i.e., locks with a bounded blocking time), and interrupt handler affinities are set by the startup routine as there is no high-level mechanism to do it in Ada.

Zamorano et al. [21] considered the extension of Ada real-time services to partitioned systems, and developed an implementation of the Ravenscar profile on top of the XtratuM hypervisor [9].

A natural extension based on these two approaches can then be proposed in order to implement the Ravenscar profile on partitioned multicore systems. Its basic elements are:

- Partitions are statically allocated to one or more virtual processors, which are mapped one-to-one to physical processors.
- Each partition has an isolated memory space. Isolation is managed by the hypervisor using the underlying MMU hardware.
- Ravenscar partitions are always active and run a single Ada program compiled with the Ravenscar profile. Tasks within the program are statically allocated to processors and are scheduled with a fixed-priority pre-emptive method, as prescribed by the profile.
- Protected operations are accessed using a fair lock, and executed in the processor of the calling task.
- Input-output devices allocated to the partition are not shared with other partitions.
- Interrupt handlers are statically allocated to processors by the hypervisor.
- Dedicated hardware timers are allocated to the partition in order to provide timing services.

An open issue is whether enforcing the Ravenscar profile in a partition is enough to guarantee a fully predictable temporal behaviour. Interference from cache and memory access from other partitions, interrupt handling, and other factors, may influence the execution time of the tasks running in a Ravenscar partition. Another important question is what level of certification can be achieved in a given partition. We are working on a pilot implementation using ORK+ and XtratuM on a multicore LEON3 computer, which we expect will contribute to provide answers to these and other questions.
5 Conclusions and future work

The trend towards using multicore processors in embedded systems, and the need to run applications with mixed criticality levels on the same computer platform lead naturally to the concept of partitioned multicore systems. The mid-term scenario is one where there are more processor cores than partitions, thus making unnecessary the use of partition scheduling methods such as the kind of static scheduling used in ARINC-653 and other architectures.

We have analysed the implications of such architecture for developing real-time systems using Ada and the Ravenscar profile, and have proposed an implementation scheme based on the assumption that at least one processor core can be exclusively allocated to every partition. Although there are other possible approaches to building partitioned many-core systems, we believe that our proposal is simple to implement and ensures a high level of temporal predictability to critical partitions.

We are working on a pilot implementation of such a system, and we expect to derive additional knowledge from it, including metrics and an assessment of the criticality levels that can be successfully guaranteed with our approach.

Acknowledgments

The authors wish to acknowledge the fruitful collaboration with the members of the Hi-PARTES and MultiPARTES teams.

References


Ada and Many-core Platforms

Luis Miguel Pinho\(^1\)  Stephen Michell\(^2\)  Brad Moore\(^3\)

\(^1\) CISTER/INESC-TEC, ISEP, Polytechnic Institute of Porto, Portugal, lmp@isep.ipp.pt
\(^2\) Maurya Software Inc, Canada, stephen.michell@maurya.on.ca
\(^3\) General Dynamics, Canada, brad.moore@gdcanada.com

Abstract

The new many-core platforms make it difficult for the programmer to provide efficient mapping of Ada programs onto the underlying non-uniform hardware. This paper discusses how existing Ada concepts such as Dispatching Domains and the Distributed Systems Annex can be extended or changed to better support the hierarchical and heterogeneous nature of these platforms.

1 Introduction

Processor manufacturers are increasingly providing higher performance by increasing the number of cores in a chip as a result of the consequences of Moore’s Law and power dissipation. This widespread trend, usually referred as the “the multi-core revolution”, is now even more challenging as many-core chips become more prevalent. Many-core chips have higher number of cores, tens to hundreds, typically interconnected by true networks-on-chip (NoC).

Examples of this trend include the Tilera Tile CPUs [1] (Tile64Pro features 64 cores), Intel Single Chip Cloud Computer (SCC) [2] (an experimental processor with 48 cores), Intel Many Integrated Core (MIC) Architecture [3] (Xeon Phi features 60 cores), STMicroelectronics P2012 [4] (prototypes are available with 69 cores), Kalray’s Multi-Purpose Processor Array (MPPA) [5] (up to 1024 cores – current version is 256 cores) or the Adapteva Epiphany with up to 4096 cores and currently with 1024 cores [6]. These many-core architectures allow the concentration of multiple applications into the same processor, the maximization of hardware utilization, the reduction of cost, size, weight, and power requirements, and improved application performance by exploiting parallelism at the application level.

Although we are still in an experimental phase where multiple architectures are being proposed making it difficult to have an all-enclosing model, some architectural trends can be observed. Some of these architectures do allow using the current model of Ada (applications are provided with a coherent global address space, and scheduling is global or partitioned), but there are architectures which defeat the existent model, either because

- memory models are not flat, or
- cores communicate through messages, or
- scheduling is hierarchical.

Issue (i) has already been discussed at the Ada Real-Time Workshop [7], although no decision was reached [8]. We believe that this discussion should be resumed, and that it should address how models such as the Partitioned Global Address Space (PGAS) [9] or Hardware Locality (hwloc) [10] could be used. The programming model should allow for a more visible mapping of the memory architecture at the programming level.

Furthermore, although Ada abstracts programmers away from the low-level architecture of the hardware, the different topologies and costs of communication inside a many-core chip (and non cache-coherent architectures) introduce efficiency concerns which need to be taken into consideration. It is nevertheless true that the correct level of abstraction is difficult to achieve, as it is shown by the difficulty in determining the correct way to handle sets of CPUs (see AI12-0033 [11]).
This paper discusses the many-core challenges and examines ways to map Ada programs onto non-uniform underlying hardware using techniques such as hierarchical scheduling domains, the use of partitions and ways to extend shared passive partitions to improve efficient inter-partition communication.

The paper is structured as follows. Section 2 presents the challenges which are put forward by many-core platforms, and their impact in the software world. Then Section 3 presents the Ada dispatching model and discusses the proposal of hierarchical scheduling domains, while Section 4 discusses the use of the Ada Distributed Systems Annex for many-core systems. Finally, Section 5 provides some conclusions.

2 The challenge of many-core processors

The everlasting requirements for more functionality, performance and flexibility, together with Moore’s law, and the undefeated (for now) power wall, have first led to the multi-core and now to the (too) many-core platform. As noted in the introduction, examples include the Tile64Pro, P2012, MPPA and Epiphany, in the embedded domain, and the SCC and now Xeon Phi in the high-performance computing domain (both domains are presenting very similar requirements and solutions). Furthermore, many-core chips are also being integrated with general-purpose multi-core processors to provide accelerator coprocessors.

Nevertheless, all of this power does come with a few important challenges. First, the many-different, non-uniform and heterogeneous architectures require new parallel programming models and runtimes. Multi-cores introduced the need for the contradictory requirements of sharing resources and isolating tasks. Many-cores further introduce the problem that there will be more cores than actual tasks to execute, so programs will require work parallelization techniques in order to effectively utilize the available cores.

Examples such as OpenMP [12], Cilk [13], OpenCL [14], or OmpSs [15], have been considered, and solutions have been proposed for Ada [16,17]. Developers will need to adapt to these many-core components or they will need to be content with stagnating execution speeds and reduced functionalities, or being relegated to niche markets using obsolete hardware components.

Even though the pressure is to move to parallelization at any cost, it is still necessary to provide efficient and predictable behaviour. And for that, programmers must be able to control how programs are structured and mapped into the underlying hardware structure, even though this structure is changing and evolving.

The first obvious changes are on core interconnection and memory hierarchy. As the number of cores increases, traditional solutions, such as buses or caches, become bottlenecks due to the contention on simultaneous accesses [18]. Cache coherency is being challenged [19], although solutions are being proposed that can scale to dozens of cores [20], and some chips provide (software-based) solutions. Buses do not scale, as contention becomes a major bottleneck and the paradigm is shifting to networks-on-chip (NoC). Furthermore, platforms can be homogenous, with either symmetric multiprocessing or asymmetric multiprocessing, or heterogeneous, with different core types.

One of the observed base trends is to divide the cores in clusters or tiles. Inside a cluster/tile, a traditional bus can be used, with clusters/tiles interconnected by a network switch to neighbour tiles, to form a 2D-mesh. The NoC also serves as a communication channel between the tiles and off-chip memory and I/O.

Caches can be: private to the cluster/tile and either coherent at that level or globally coherent in the chip; or not coherent at any level. Examples also exist of architectures where this coherency can be programmed between (dynamic) sets of tiles (e.g., Tilera).

One of the first examples of non-cache coherent approaches was provided by the Intel SCC [2], a chip containing 24 tiles connected with a 4×6 2D-mesh, with each tile having 2 Pentium cores and a shared message passing buffer (MPB). However, currently Intel is proposing the MIC architecture that, although integrating the main features of previous Intel many-cores implementations, is different in several aspects. One of the MIC instantiations, the Xeon Phi [3], integrates 60 (Pentium-based) cores connected through a dual ring-bus with coherent cache. It is being considered both for stand-alone many-core applications, and as an accelerator co-processor under the control of a Xeon multi-core.

The Tilera [1] architecture is also a tiled architecture but is different, as each tile is in a single core with its own cache. Tiles are interconnected by several parallel NoCs (iMesh), with configurable routers. It is possible to divide the tiles into separate domains with separate cache-coherent areas, or bring all tiles together in the same domain...
(always in a single address space). Epiphany is similar, although with both local memory and shared distributed memory.

A recent proposal by STMicroelectronics is the Platform 2012 (P2012) [4], which is being proposed also as an accelerator architecture. It is also based on multiple processor clusters connected with a NoC, with each cluster having up to 16 (STHORM) cores with no cache coherence. The clusters are controlled by 2 host ARM processor(s), in the same chip. A similar non cache coherent 16 core per cluster architecture is provided by the Kalray MPPA [5], whose current chip has a total of 256 (work) cores, plus a few dozens more for I/O and interconnect.

In terms of software architecture, it is possible to find solutions with a single operating system (OS) for all of the cores, or separate solutions. The Tilera, the Xeon Phi and the P2012 allow for both a single OS for the multiple clusters/tiles or independent OSes in each cluster. The Xeon Phi and the P2012 also allow for the same OS as the host processors when being used as accelerators.

The experimental SCC also led to an experimental OS (Barrelfish [21]) that proposes a multi-kernel approach in which every core runs a simple kernel with all kernels working together to maintain a coherent global state. The MPPA has a similar structure; each core has a simple runtime, but within a cluster the runtimes work together. Communication with the other clusters is through an IPC-like mechanism. MPPA provides a distributed memory space, with each cluster having its separate memory area.

These multiple different architectures question if Ada’s memory, concurrency, and partition models are able to support all variants. Even if not providing architecture specific mechanisms (impossible with the current diversity and experimentation), it is still necessary to give programmers the basic tools that enable structuring of the application into clusters of processors, and both shared-data and message-passing approaches of inter-core/cluster communication. This is the subject of this paper, which discusses the use of Ada Dispatching Domains, and the role of the Distributed Systems Annex.

3 Dispatching Models

3.1 Ada 2012 Multi-core Dispatching Model

After a proposal by Burns and Wellings [22] of language extensions for supporting multi-processors, and discussion in the International Real-Time Ada Workshop [23], a multi-processor affinity model was integrated in Ada 2012 (details can be found in [24]).

This is based on the notion of a multi-core dispatching model that provides non-overlapping ranges of CPUs, with a different scheduler provided in each domain. Ada 2012 defines packages for handling the set of CPUs available to the program, and the creation of the domains. CPU ranges are defined in:

```ada
package System.Multiprocessors is
  pragma Preelaborate(Multiprocessors);

  type CPU_Range is range 0 ..
      implementation-defined;
  constant Not_A_Specific_CPU: CPU_Range := 0;
  subtype CPU is CPU_Range
      range 1 .. CPU_Range'Last;

  function Number_Of_CPUs return CPU;
end System.Multiprocessors;
```

and the dispatching model is defined in a child package System.Multiprocessors.Dispatching_Domains:

```ada
package System.Multiprocessors.Dispatching_Domains is
  dispatching_Domain_Error: exception;

  type Dispatching_Domain<> is
      limited private;
  constant Dispatching_Domain;
end System.Multiprocessors.Dispatching_Domains;
```
function Create(First, Last: CPU)
return Dispatching_Domain;
-- ...

function Get_Dispatching_Domain(
    T: Task_Id := Current_Task)
return Dispatching_Domain;

procedure Assign_Task(
    Domain: in out Dispatching_Domain;
    CPU: in CPU_Range := Not_A_Specific_CPU;
    T: in Task_Id := Current_Task);

procedure Set_CPU(CPU: in CPU_Range;
    T: in Task_Id := Current_Task);

function Get_CPU(
    T: in Task_Id := Current_Task)
return CPU_Range;

procedure Delay_Until_And_Set_CPU(
    Delay_Until_Time: in Time;
    CPU: in CPU_Range);

private ...
end System.MultiprocessorsDispatching_Domains;

CPUs are grouped into these dispatching domains, and can only be a member of one domain. All CPUs are initially in a System_Dispatching_Domain. In order to provide a more flexible handling of CPUs, currently CPU sets are being proposed to allow for domains to be configured with disjoint CPUs [11].

The current model implies:

- Dispatching domains are static sets/ranges that can only be setup during elaboration before the main program begins executing.
- A task can only be assigned once to a dispatching domain, and only if it currently belongs to the system dispatching domain.
- The System_Dispatching_Domain cannot be empty, so at least one CPU cannot be allocated to another domain.
- If not set explicitly, a new task inherits the domain of its activating task.
- Once a task belongs to a dispatching domain, its execution inside the domain can be configured with Set_CPU to set the affinity to a specific CPU dynamically (within that dispatching domain) or with Not_A_Specific_CPU to allow migration between all CPU's in the domain.

### 3.2 A more flexible dispatching model

The Ada dispatching model is flexible as it allows the implementation of multiple different scheduling approaches such as global scheduling (a single domain), partitioned, clustered-global, or even task splitting [16]. In fact, this model aligns very well with cluster-based scheduling approaches, which have been shown to improve the percentage of task sets that can be found schedulable [25].

It supports systems where many-core clusters are provided as an accelerator mechanism, if a single operating system is used. The System Dispatching Domain can reside in the host cores, and an accelerator Dispatching domain is created in the many-cores (potentially using the task pools and the model of [16,17].

Nevertheless, it is a static model, which does not allow the dynamic changing of the allocation of tasks, or the selection of domains during execution.

A first proposal is to allow tasks to dynamically change their domain, providing better support for load balancing, fault tolerance, or mode changes.
We therefore propose that the restriction which currently exists that prevents tasks from migrating between domains [26, Annex D1.6.1, Paragraph 26/3] be removed. Under this proposal, tasks could not be holding a protected lock when a dispatching domain change is made, so a migration would need to be done after the task leaves any abort-deferred region.

Note that the programmer can produce the same behaviour by:

- Using the notion of a migration manager which needs to execute in the System Dispatching Domain – this manager will be responsible for the creation of a new task to be migrated.
- Every time a task wishes to migrate, it explicitly requests to the migration manager that a new task is created (of the same type) and after creation assigned to the new dispatching domain.
- The programmer then explicitly handles the copying of information from the original task to the new task, after which the original task can terminate.

However, placing the responsibility for explicit handling of the migration in the hands of the programmer creates more fragile programs and more chances for errors. It would be simpler for the compiler and runtime to generate the required code for migrating the task and its associated state (and potentially code) from core to core.

A second proposal is a more flexible hierarchical model.

Current real-time hierarchical models are mostly based on static assignment of tasks to clusters (dispatching domains), which are then dynamically scheduled within the domain. However, with the advent of many-cores, it is expected that more flexible schemes will be proposed, such as decentralized agreement protocols [21,27].

Even if tasks are allowed to migrate between domains, these decentralized agreement protocols would need to be managed explicitly by the programmer, and potentially with code running in all domains. By creating the notion of higher-level allocation domains it would be possible to define domain-level allocation mechanisms.

The proposal is therefore to create the notion of hierarchical domains, where:

- Tasks could not be dispatched at non-leaf domains, as this would introduce conflicting issues. Multiple dispatchers cannot be scheduling the same task. There also cannot be multiple dispatchers for a domain.
- Non-leaf domains would be exclusively allocation domains, and not dispatching domains. Allocation domains would only allocate tasks to dispatching (or maybe other hierarchically lower allocation) domains.
- Allocation domains can only allocate tasks to a directly nested allocation or dispatching domain.
- These allocation domains would have ready queues and priorities, but would not dispatch the tasks to CPUs. The priority of a task could be the same in the different domains (if applicable), or could be an explicit parameter in the allocation.
- A newly created task could be assigned to the dispatching domain of the activating task, or not assigned to a dispatching domain at all, and instead placed in the top level Allocation Domain.
- A migrating task between dispatching domains could be first forced to be reassigned to the enclosing allocation domain from its dispatching domain, and then reassigned from the allocation domain to one of its child dispatching domains, in the case that the direct migration of a task between two schedulers is problematic.
- Allocation-domain schedulers could be implemented through an agreement-based solution between domain schedulers.

In this way, we could specify a new type in the package Dispatching_Domains (in the same package as the Dispatching_Domain type so that implementation can see the internals of both):

```ada
type Allocation_Domain (<>)
  is
  limited private;

System_Allocation_Domain :
  constant Allocation_Domain;

function Create return Allocation_Domain;
```
procedure Add (  
    A_Domain : in out Allocation_Domain;  
    D_Domain: Dispatching_Domain);  

function Get_Allocation_Domain (  
    D_Domain: Dispatching_Domain)  
  return Allocation_Domain;  

-- and even  
function Get_Allocation_Domain (  
    A_Domain: Allocation_Domain)  
  return Allocation_Domain;  

procedure Allocate_Task (  
    A_Domain : in out Allocation_Domain;  
    T: Ada.Task_Identification.Task_Id :=  
    Ada.Task_Identification.Current_Task);  

4 Distribution and many-cores

4.1 The Ada Distributed Systems Annex

Although the Distributed Systems Annex (DSA) [26, Annex E] of Ada exists to support building Ada applications which span several nodes in a distributed system, the emergence of many-core platforms require that this model be considered in the construction of Ada applications which span several cores, and in particular multiple (heterogeneous) clusters.

In Ada, the concept of a partition exists (in the core language) to denote the independent parts of the application which work together to provide the complete functionality: "a partition is a program or part of a program that can be invoked from outside the Ada implementation" [26, Section 10.2]. Partitions can be active, which can have tasks and execute application code, and passive, which do not have processing capability and serve for storing data objects.

Active partitions communicate through the use of remote types or remote subprograms, which are implemented as remote procedure calls for both subprogram calls and distributed objects. These underlying communication mechanisms are hidden by compiler generated stubs that satisfy the interface of the Partition Communication System (PCS) defined in the standard. The standard also states that it is possible for a compiler vendor to provide an implementation defined interface to the Partition Communication System that differs from the one described in the standard [26, Annex E.5, Paragraph 27.1/3]. Although the implementation details are not defined by the standard, the model allows partitions to be aborted and restarted (reloaded) during the execution of a distributed program [26, Annex E.1, Paragraph 13].

Passive partitions cannot have tasks or protected objects with entries, since they are supposed to reside in nodes without processing abilities.

4.2 Using partitions for many-core systems

As well as the dispatching model, the distributed model of Ada can be considered for the case of many-core platforms. In fact, present day many-cores are distributed systems ¹.

If we apply the DSA model to such a system, processor clusters would each get their own (active) partition (or more than one per cluster). In particular, in the case of heterogeneous many-cores, partitioning would be at least by core type. Passive partitions can be used to represent the shared off-chip memory, if it can be mapped to different active partitions. Communication between clusters (usually through some message passing mechanism) would be made via shared passive partitions, Remote Types and Remote Call Interface DSA library units.

This proposed use of the DSA could also be extended for single-core partitions that do not share caches, and need to communicate through message passing.

¹ There is not much difference between a P2012 chip, integrating two ARM cores connected to a cluster of 16 STHORM processors by a NoC, and a Xeon based system with two boards in a rack, one with a Xeon dual-core and the other with a 60-core Xeon Phi.
Nevertheless, several issues still need to be considered. First, the current DSA defines that communication between partitions is through remote subprogram calls. Even if the PCS is not used, this may not be the most appropriate mechanism (e.g. for the case of the SCC, messages are exchanged through data placed in the Message Buffer, which can be made coherent in all cores through software [28]). This could also be made optional. Hardware is quickly changing so portability should abstract from hardware implementation issues.

Another issue is that many-core applications can either be intended to be completely isolated, or interacting for a common goal. It is in this latter case that using partitions to construct an application makes sense. In this case, it is expected that applications interact, and probably more often than in a distributed system. In particular, partitions would be used to represent heterogeneous components that work together: e.g. a host partition and an accelerator partition.

In this case of heterogeneous partitioning, it is expected that synchronization is required. It can be possible to imagine a situation where data is being shared between the host and the accelerator partitions, in a passive partition. The accelerator tasks will potentially be blocked waiting for this data to be available. The solution for this would be for the program in the host partition to place the data in the shared partition, and then making a remote call to the accelerator partition in order to release the blocked task.

Another approach would be to allow protected objects in passive partitions to have entries. This would allow one active partition to release tasks in another partition just by making the data available (this is the general model of protected objects). In this case, tasks from only one active partition would be allowed to call the entries of the protected objects.

Obviously an underlying mechanism would need to exist, and the protected entry in the shared passive partition would need to have a controlling entity (a shadow object with the respective queues). One choice is for each shared passive partition to have a controlling active partition. Another choice is to have each object in such a partition to have a controlling active partition. Tasks from other active partitions would only be able to call protected procedures and functions. This would be either statically defined (and configuration-time or compile-time checked) or an exception could be raised if a task called an entry of a partition not being controlled by the task’s own partition.

The mechanism by which the task waiting in the PO would be released could be implementation defined, but it could be through inter-core interrupts (useful when remote calls are more costly than inter-core interrupts). A necessary restriction is that the proxy model for released tasks (where the releaser can execute code on behalf of the task being released) cannot be applied since the caller may be from a different partition and have no access beyond the simple release signal to synchronization primitives on the controlling partition for that protected object.

The use of the shared active partitions which has been proposed in [29] is not applicable, as shared active are required to be placed in processing nodes, and data replicated in all of the nodes. Furthermore, this would required a tighter interrelation between active partitions, since POs would be replicated across different partitions, therefore priorities and queue policy would need to be the same, preventing the use of different models or runtimes.

The final issue is if the name of the annex should be changed, or a new annex created. If this proposal can be made acceptable to the level of the Ada Community, a ISO/IEC JTC1/SC22/WG9 technical specification could be issued on the use of the Ada DSA for many-core processors.

5 Conclusions

The current trend of chip manufacturers to provide processors with increasing number of cores imposes new challenges to software development, calling for new parallel programming models and runtimes.

Ada abstracts programmers from the hardware architecture but this abstraction may introduce so much inefficiency as to render the program ineffective. The diverse architectures being currently researched exacerbate the situation and create situations where programmers must be able to specify the mapping of applications into the underlying hardware structure in order to effectively use the underlying resources.

This paper discusses how to map Ada programs onto many-core platforms, and in particular hierarchical scheduling domains and the use of partitions. Both partitions and (hierarchical) domains support application development for many-core processors with different structural representations.
Partitions denote parts of the application which are intended to be separate, either because there are different instruction sets (and even programming models), different address spaces, the need for different runtimes, or just for being able to start/restart in different clusters. Within each partition, multiple dispatching domains can be created, when the programmer intends to divide the cores into different clusters (for instance to apply semi-partitioned or clustered real-time scheduling, or even separating the application domain from the task-pool domain).

Hierarchical allocation domains would be used when the programmer wishes to perform some form of hierarchical scheduling or allocation.

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) and AVIACC (ref. FCOMP-01-0124-FEDER-020486) projects.

References


Incorporating the Deadline Floor Protocol in Ada

Mario Aldea\textsuperscript{1}, Alan Burns\textsuperscript{2}, Marina Gutiérrez\textsuperscript{1} and Michael González Harbour\textsuperscript{1}

\textsuperscript{1}Universidad de Cantabria \hspace{1em} \{aldeam, gutierrezlm, mgh\}@unican.es
\textsuperscript{2}University of York \hspace{1em} alan.burns@york.ac.uk

Abstract

The Ada 2005 standard introduced “Earliest Deadline First” (EDF) as one of the supported dispatching policies. The standard specifies the “Stack Resource Protocol” (SRP) as the protocol for resource sharing among EDF tasks. During the time the SRP has been in the standard it has shown to be a relatively complex protocol. Recently, a new protocol has been proposed for resource sharing in EDF. This new protocol, called “Deadline Floor inheritance Protocol” (DFP), is simpler and more efficient than SRP while keeping all its good properties. In this paper we briefly describe both protocols and compare them from the complexity point of view. In light of its simplicity, we propose to change the language standard to include DFP instead of SRP. Some alternative modifications of the Ada Reference Manual are pointed out in order to include DFP in the most straightforward way.

1. Introduction

The Ada 2005 standard introduced “Earliest Deadline First” (EDF) as one of the supported dispatching policies and the “Stack Resource Policy” (SRP) was specified as the protocol for resource sharing among EDF tasks. SRP is a complex protocol as has been shown by the initial difficulties with its specification and implementation. Recently, Alan Burns has defined a new protocol for resource sharing in EDF [3]. This new protocol, called “Deadline Floor Protocol” (DFP), is simpler and more efficient than SRP while keeping all its good properties.

MaRTE OS [6] [7] is a real-time operating system developed by the Computers and Real-time group at the University of Cantabria. Most of its code is written in Ada with some C and assembler parts. It provides the applications with a POSIX/C interface to be used for concurrent C/C++ applications. From the Ada point of view, the POSIX version of the GNAT runtime system has been adapted to run on top of the POSIX/C interface of MaRTE OS. MaRTE OS provides support for most of the new real-time functionalities defined in the Ada 2012 standard; in particular it supports EDF dispatching (including SRP) [8].

In [9], we have implemented the DFP in the kernel of MaRTE OS and defined a POSIX-like interface to manage the specific parameters required for this protocol. We compared the performance of SRP and DFP and found that DFP outperforms SRP in both less implementation complexity and smaller runtime overheads.

In this paper we extend the performance analysis of DFP vs SRP to the Ada context. Then, we propose some changes that would be needed to incorporate the DFP into the Ada language, as a replacement for SRP. We analyze the required changes to the protected object locking policies and to the EDF task dispatching policy itself. We describe different alternatives for their discussion at the Real-Time Ada Workshop.

The paper is organized as follows. Section 2 briefly describes both resource sharing protocols: SRP and DFP. In Section 3 we give an overview of the EDF dispatching policy defined in Ada 2012 together with the specification of the SRP for protected objects. Section 4 describes the implementation of the new DFP in MaRTE OS, followed in Section 5 by a comparison of both resource sharing protocols in the context of the Ada specification. In Section 6 we propose some alternatives to incorporate the DFP to the Ada language, while Section 7 contains a discussion of the interactions between the new policy and the other policies defined in Ada. Finally, Section 8 gives our conclusions.

2. Resource sharing policies

For Ada tasks scheduled under fixed priorities, i.e., with the Fifo Within Priorities task dispatching policy, the Ceiling Locking locking policy is used to manage access to shared resources through protected objects. When EDF was added in the Ada 2005 version of the language, tasks scheduled under the EDF Across Priorities task dispatching policy also used the Ceiling Locking policy with a set of scheduling rules that made the scheduling of tasks using protected
objects equivalent to the Stack Resource Protocol (SRP). We will now review the SRP rules and introduce the Deadline Floor Protocol (DFP).

2.1. The SRP algorithm

Baker presents in [1] the Stack Resource Policy (SRP) for bounding priority inversion when accessing resources in real-time systems scheduled under EDF. SRP is a generalization of the Priority Inheritance Protocol (PIP) [2] and the Priority Ceiling Protocol (PCP) [2]. On a single processor, SRP presents the good properties of PCP: mutual exclusion ensured by the protocol itself (without needing an actual lock), deadlock avoidance, and at most a single blocking effect from any task with a longer relative deadline. These properties hold as long as there is no suspension while holding a lock.

With SRP, each task is assigned a number called the “preemption level” that correlates inversely to its relative deadline: the shorter the deadline the higher the preemption level. Shared resources are also assigned a preemption level that is the highest of the preemption levels of all the tasks that may use that resource.

The use of SRP imposes a new rule to the base EDF scheduling: “a thread can only become ready for execution if its preemption level is strictly higher than the preemption levels of the resources currently locked in the system”. This rule is the cause of most of the SRP complexity as will be discussed later on.

2.2. Deadline Floor inheritance Protocol

Recently, Burns introduced a new protocol for resource sharing in EDF, called Deadline Floor Protocol (DFP) [3]. The DFP has all the key properties of SRP, specially causing at most a single blocking effect from any task with a longer relative deadline, which leads to the same worst-case blocking in both protocols. In an EDF-scheduled system, DFP is structurally equivalent to PCP in a system scheduled under fixed priorities.

Under DFP every resource has a relative deadline equal to the shortest relative deadline of any task that uses it. The relative deadline of a resource is called “deadline floor”, making it clear the symmetry with the “priority ceiling” defined for the resources in PCP.

The key idea of the DFP is that the absolute deadline of a task could be temporarily shortened while accessing a resource. Given a task with absolute deadline \(d\) that accesses a resource with deadline floor \(D\) at time \(t\), its absolute deadline is (potentially) reduced according to \(d = \min\{d, t+D\}\) while holding the resource.

DFP does not add any new rule to the EDF scheduling, thus its leads to simpler and more efficient implementations than SRP as it will be shown later on.

3. Earliest Deadline First Dispatching and SRP in Ada 2012

Support for the Earliest Deadline First (EDF) dispatching is in the language from the Ada 2005 standard. The definition of EDF has been maintained unchanged in Ada 2012 (apart from the replacement of “pragmas” for the new Ada 2012 “aspects”).

EDF dispatching is defined in the Ada Reference Manual (ARM) D.2.6 where a policy identifier (EDF_Across_Priorities) is defined along with the language-defined library package Ada.Dispatching.EDF. This package provides operations to manage the absolute deadlines of the tasks.

```ada
with Ada.Real_Time;
with Ada.Task_Identification;
package Ada.Dispatching.EDF is
    subtype Deadline is Ada.Real_Time.Time;
    procedure Set_Deadline
        (D : in Deadline;
         T : in Ada.Task_Identification.Task_Id :=
          Ada.Task_Identification.Current_Task);
    procedure Delay_Until_And_Set_Deadline (Deadline_Time : in Ada.Real_Time.Time;
                                           Deadline_Offset : in Ada.Real_Time.Time_Span);
    function Get_Deadline (T : Ada.Task_Identification.Task_Id :=
end Ada.Dispatching.EDF;
```
The most complex part of the EDF dispatching definition in Ada is the integration of the base Ada dispatching model (based on fixed priorities for the tasks and priority ceilings for the protected objects) with the SRP rules and the “preemption level” concept. EDF is defined to work in a given band of priority levels, which may cover the whole range of system priorities, or a specific interval. The Reference Manual defines a clever integration of preemption levels and priorities: preemption levels of tasks and protected objects are mapped to priorities in the EDF priority band.

```ada
task T with Priority => 20;
-- if priority 20 is in an EDF range it represents the task’s preemption level

protected Object with Priority => 24 is ...
-- if priority 24 is in an EDF range it represents the protected object’s preemption level
```

The sources of priority inheritance are redefined for EDF tasks. The RM defines that, by default, the active priority of an EDF task is the lowest priority in its EDF priority band. The task will inherit priorities as any other Ada task, in particular, when an EDF task executes a protected operation it will inherit the priority (preemption level) of the protected object. But, for EDF tasks, the RM defines a third source of priority inheritance: “the highest priority P, if any, less than the base priority of T such that one or more tasks are executing within a protected object with ceiling priority P and task T has an earlier deadline than all such tasks; and furthermore T has an earlier deadline than all other tasks on ready queues with priorities in the given EDF_Across_Priorities range that are strictly less than P”. This latter rule added to the base EDF scheduling is required to obey the SRP algorithm and is the major source of the complexity of this policy.

An example of the ordering of the EDF tasks is shown in Figure 1. Figure 1 (a) shows the schedule of three EDF tasks T_a, T_b and T_c with relative deadlines d_a=9, d_b=6 and d_c=3 and preemption levels p1_a=1, p1_b=2 and p1_c=3. There is a protected object P0_1 with preemption level p1=2 that is accessed by T_a at t=1.

Figure 1 (b) shows the configuration of the ready queue at t=4. At that point in time T_a has inherited the preemption level of P0_1 and, consequently it is placed in the queue of priority 2. T_b is executing at the lowest priority in the range since it does not inherit T_a’s priority because, although it has a more urgent deadline than T_a, its preemption level is not strictly higher than T_a’s active priority as required by the third priority inheritance rule of SRP. T_c has inherited T_a’s priority because it verifies the conditions imposed by the third priority inheritance rule of SRP: it has a shorter deadline that any task at priorities 2 and below, and its preemption level is strictly higher than T_a’s priority.

The Ada dispatching model suggests a direct implementation of the ready queue as an array of queues, one for each priority level (Figure 1 (b)). Using this implementation of the ready queue and the rules for the EDF dispatching, the algorithm to find the right place for a newly activated EDF task T in the ready queue would be the one shown in Figure 2.

This section has shown the inherent complexity of the SRP. It is important to notice that this complexity is not due to the particular implementation of the SRP used in Ada (based on priority queues). Other alternative implementations, such as the one used in MaRTE OS that is based on only one queue [9], suffer from this complexity. Further evidence of this inherent
complexity of the SRP was the mistake in the original definition of SRP rules in Ada 2005 [4] and the original erroneous implementation in MaRTE OS [5].

There is one drawback that is specific to the Ada definition of SRP: the limited number of distinct preemption levels. The number of distinct preemption levels that can be used for tasks in an EDF priority range is the size of the range minus one. In a system with few priority levels or in a narrow EDF range this limitation could jeopardize the schedulability of the system by causing more blocking than is necessary. It could be argued that the implementation could provide more priority levels, but priority levels are expensive because they affect the size and performance of many of the run-time data structures such as the ready queues and the entry queues.

4. DFP implementation in MaRTE OS

Support for EDF and SRP was included in the kernel of MaRTE OS some years ago [8]. The operating system provides a POSIX-like interface to these services that is used by the adapted GNAT run-time system used by MaRTE OS to provide support to the EDF dispatching functionality defined in the Ada Reference Manual.

Recently, we have implemented the DFP in the MaRTE OS kernel. Implementing DFP is quite straightforward since it does not impose any additional scheduling rule to an EDF scheduled system, so the ordering criterion of the ready queue remains the same as in any other EDF system. Consequently, the ready queue in EDF & DFP is much simpler than the array of queues defined in Ada for EDF & SRP. It is enough with a single queue where tasks are ordered according to their absolute deadlines (shorter deadlines first).

The only specific parameter required by the DFP protocol is the deadline floor that, in the MaRTE OS implementation, has been added as a new mutex parameter (MaRTE uses POSIX mutexes as the base primitive for exclusive access to shared resources).

Our implementation of DFP has been simplified by adding the limitation to supporting just nested but not arbitrarily overlapped critical sections. This limitation has no effects for Ada, which does not allow overlapped critical sections. This simplification eases the implementation of the mutex lock and unlock operations. In the case of the lock operation, the old absolute deadline of the task is stored in a field of the mutex and the task is assigned a (possibly) new absolute deadline according to the deadline floor of the mutex. In the unlock operation the task’s deadline is returned to the value stored during the lock operation and then the task has to be reordered in the ready queue according to its (possibly) longer deadline, which could lead to a context switch. Note that the possible deadline change in the lock operation cannot lead to a context switch since it could only shorten the deadline of the running task which, by definition, is the task with the earliest deadline in the system.

Figure 2. Algorithm to find the right place for a newly activated task in the ready queue with SRP

procedure Add_To_Ready_Queue (Task) is
begin
  Prio_Max := Lower_Priority_In_EDF_Range;
  for Prio in Lower_Priority_In_EDF_Range+1 .. T_Preemption_Level-1 loop
    if not Queue(Prio).Empty then
      if T.Deadline < Queue(Prio).Head.Deadline then
        Prio_Max := Prio;
      else
        exit;
      end if;
    end if;
  end loop;
  if Prio_Max = Lower_Priority_In_EDF_Range then
    Queue(Lower_Priority_In_EDF_Range).Add_In_Deadline_Order(T);
  else
    Queue(Prio_Max).Add_Head(T);
  end if;
end Add_To_Ready_Queue;
A small drawback of the DFP is the need for reading the clock in the lock operation. However, the time consumed by this operation is not an impediment for EDF & DFP to be more efficient than EDF & SRP as we will see in the next section.

Further details of the implementation of DFP in MaRTE OS can be found in [9].

5. Comparative analysis

The MaRTE OS implementation of SRP does not use an array of queues as the ARM suggests. Instead, it uses a single queue at the expense of adding complexity to the function used to order tasks in the queue. Of course the MaRTE OS implementation is functionally equivalent to the SRP algorithm defined by Ada.

The work in [9] presents a comparison of the DFP and SRP implementations in MaRTE OS. The results obtained show that the DFP implementation is simpler and much more efficient than the implementation of SRP. However, since the MaRTE OS implementation of SRP is different from the model described in the Ada standard, it could be argued that an implementation according to the Ada model could be more efficient than the one used in MaRTE. To analyze that model we have done an experiment creating the data structures used for both the multiple-queue implementation described by the ARM for EDF & SRP, and the single binary heap ready queue used for EDF & DFP in [9]. Such structures are shown in Figure 3.

The doubly-linked list is a linked data structure that consists of a set of sequentially linked nodes, each containing a ready task in this case. In addition, each node contains two links that are references to the previous and to the next node in the sequence of nodes. Removing and inserting the head of the list are operations that can be performed in constant time.

The binary heap [10] is a heap data structure created using a binary tree, with two additional constraints: all levels of the tree, except possibly the last one, i.e. the deepest one, are fully filled and each node is less than or equal to each of its children. The binary heap is an efficient implementation of an ordered queue: insert and remove operations take logarithmic time while peeking the head takes constant time.

Figure 3-b shows that in the implementation of the ready queue for EDF & SRP, the lower priority level of the EDF range can be implemented with a binary heap, because only the absolute deadlines are used to order the tasks at this priority level, which only has tasks that do not hold shared resources. For the other priority levels we use doubly-linked lists because elements are only added, removed or read at the head of the list.

Table 1 gives the pseudocode of the parts of the lock and unlock operations that are specific to the DFP and SRP implementations. Procedure Reorder() places the task in its correct position according to its new scheduling parameters. Procedure Reorder_And_Dispatch() calls Reorder() and, in the case the most urgent task had changed, it performs a context switch.

Let us consider the simpler case of the lock and unlock operations: a task, with no previous resources held, locks and immediately unlocks a mutex. Locking an SRP mutex involves a call to Reorder(), which dequeues the task from the head of the heap and adds it to the head of the doubly-linked list of the ceiling priority of the mutex. Unlocking the SRP mutex involves a call to Reorder_And_Dispatch() which removes the task from the head of the doubly-linked list and enqueues it into the binary heap. Note that now the task could possibly not be the most urgent task, because other tasks could have been activated in the meanwhile, implying the need for a context switch.
For a DFP mutex the sequence of calls is the following: locking the mutex requires reading the clock but no reordering since the task continues to be the most urgent one; unlocking the mutex involves a reordering of the task in the queue, since the end of the deadline inheritance could make the task less urgent than some other task activated while the mutex was being held.

To summarize, on the one hand, the higher overhead operations needed to lock and unlock an SRP mutex are the queue and dequeue operations in the binary heap. On the other hand, locking and unlocking a DFP mutex requires reading the clock and a requeue into the heap. In our tests we have found that reading the clock is faster than a requeue operation in a heap larger than just two elements. Figure 4 shows the execution times measured for a lock operation followed by an unlock as a function of the number of tasks under a worst-case situation, in a Pentium III at 800MHz. Figure 4a shows the performance comparison when using the heavier &ORFN function included in package $GD5HDOWLPH, while Figure 4b shows the same comparison but using an internal MaRTE OS clock. This latter comparison is more accurate, because the internal implementation of the mutex operations would use internal OS resources. In consequence, the DFP outperforms the SRP even in the context of the Ada task dispatching model.

### Alternatives to incorporate the DFP into the Ada Reference Manual

In order to include the DFP protocol in the Ada language the most decisive issue is how to deal with the current definition of the EDF dispatching. We can think of several alternatives:

a. Replace the current definition of EDF_Across_Priorities (RM D.2.6) by a new (simpler) definition that uses DFP instead of SRP.

b. Add a new dispatching policy (EDF_With_Deadline_Floor) to define the EDF & DFP dispatching and keep the current definition of EDF_Across_Priorities in the standard but declare it obsolescent.

c. Add a new dispatching policy (EDF_With_Deadline_Floor) to define the EDF & DFP dispatching and keep the definition of EDF_Across_Priorities in the standard. An implementation could choose to implement both, one or none of these two dispatching policies.

### Table 1 Lock and unlock operations for DFP and SRP

<table>
<thead>
<tr>
<th></th>
<th>DFP</th>
<th>SRP</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock</td>
<td>procedure Task_Locks_Mutex (Task, Mutex) is</td>
<td>procedure Task_Locks_Mutex (Mutex) is</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>Mutex.Owner_Deadline := Task.Deadline; Heir_Deadline := Clock +</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>Mutex.Deadlinefloor; if Task.Deadline &gt; Heir_Deadline then</td>
<td>Task.Num_Mutex_Owned ++; Mutex.Owner_Preemption_Level :=</td>
</tr>
<tr>
<td></td>
<td>Task.Deadline := Heir_Deadline;</td>
<td>Task.Preemption_Level;</td>
</tr>
<tr>
<td></td>
<td>end if;</td>
<td>if Task.Preemption_Level &lt;</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>Mutex.Owner_Preemption_Level =</td>
</tr>
<tr>
<td></td>
<td>end Task_Locks_Mutex;</td>
<td>Task.Preemption_Level :=</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reorder (Task);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>end if;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>end Task_Locks_Mutex;</td>
</tr>
</tbody>
</table>

| unlock| procedure Task_Unlocks_Mutex (Task, Mutex) is                      | procedure Task_Unlocks_Mutex (Task, Mutex) is                     |
|-------| ...                                                                | ...                                                                |
|       | ...                                                                | ...                                                                |
|       | Task.Deadline := Mutex.Owner_Deadline; Reorder_and_Dispatch (Task);| Task.Num_Mutex_Owned --;                                           |
|       | ...                                                                | Task.Preemption_Level :=                                           |
|       | end Task_Unlocks_Mutex;                                             | Mutex.Owner_Preemption_Level;                                      |
|       |                                                                     | Reorder_and_Dispatch (Task);                                       |
|       |                                                                     | ...                                                                |
|       |                                                                     | end Task_Unlocks_Mutex;                                             |
The proposed new dispatching policy identifier (EDF_With_Deadline_Floor) is intended to be used in the Task_Dispatching_Policy and the Priority_Specific_Dispatching pragmas:

```
pragma Task_Dispatching_Policy (EDF_With_Deadline_Floor);
pragma Priority_Specific_Dispatching (EDF_With_Deadline_Floor,
                  first_priority, last_priority);
```

Pragma Priority_Specific_Dispatching specifies the task dispatching policy for the specified range of priorities. Since EDF_With_Deadline_Floor dispatching only requires one priority level it could be considered an error to provide a range with more than one priority levels. However, as discussed in Subsection 7.1, for backwards compatibility of previous applications and also for analogy with the other dispatching policies, ranges of any size should be allowed. In this latter case the ARM should state that the active priority of the tasks in the EDF_With_Deadline_Floor band would be collapsed to the lower level of that band.

Another issue is related to the Ceiling_Locking policy. Locking policies are applied to the whole partition using the Locking_Policy pragma, so it could be argued that the identifier name Ceiling_Locking is inappropriate since its use for a partition implies that a protocol different from the “Ceiling Locking” is going to be used in EDF_With_Deadline_Floor priority ranges. On the other hand, it could be considered that for EDF scheduling the DFP is the equivalent to the “Ceiling Locking” concept and then the identifier would be appropriate.

More relevant than the identifier name are the deep modifications that would be required in the definition of the ceiling locking policy (RM D.3). Currently this policy is only defined in terms of priorities and, with the incorporation of the DFP, it would be necessary to define it also in terms of deadlines.

A new aspect is required to assign deadline floors to the protected objects:

```
protected Object with Deadline_Floor => Ada.Real_Time.Milliseconds(24) is ...
```

If a new aspect is not deemed necessary, it would be possible to reuse the Relative_Deadline aspect, much in the way the priority ceiling of the PCP protected objects is named Priority instead of Priority_Ceiling.

From Ada 95 the language allows the dynamic change of the base priority of a task with the operations in the Ada.Dynamic_Priorities package and in Ada 2005 the picture was completed with the definition of the Priority attribute that allows applications to dynamically change the ceiling priority of a protected object. In the same way, for reconfigurable systems that use the DFP protocol we would need primitives to change the relative deadline of tasks and protected objects.

---

**Figure 4.** Performance comparison of DFP vs SRP

The performance comparison of DFP vs SRP is shown in Figure 4. The graphs illustrate the performance using Ada.Realtime.Clock and an internal OS clock.
objects. In the case of the tasks, a new package Ada.Dispatching.EDF.Dynamic_Relative_Deadlines should be provided:

```ada
with Ada.Real_Time;
with Ada.Task_Identification;
package Ada.Dispatching.EDF.Dynamic_Relative_Deadlines is

    procedure Set_Relative_Deadline
        (D : in Real_Time.Time_Span;
         T : in Ada.Task_Identification.Task_Id :=
           Ada.Task_Identification.Current_Task);

    function Get_Relative_Deadline
        (T : Ada.Task_Identification.Task_Id :=
        return Real_Time.Time_Span;

end Ada.Dispatching.EDF.Dynamic_Relative_Deadlines;
```

Currently the relative deadline is only used for setting the first absolute deadline after the task's activation, so it could be argued that this facility is not needed. However, the DFP specification should require that there are no violations of the deadline floor protocol by checking that tasks only use protected objects with a deadline floor that is not larger than the relative deadline of the task. If the deadline floors are allowed to change dynamically, so should the task relative deadlines.

To dynamically change the deadline floor of a protected object a new Deadline attribute should be provided. This new attribute would behave very much like the Priority attribute:

```ada
protected body PO is

    procedure Change_Relative_Deadline (D: in Real_Time.Time_Span) is
        begin
            -- PO'Deadline has old value here
            PO'Deadline := D;
            -- PO'Deadline has new value here
        end Change_Relative_Deadline;

    ...  

end PO;
```

Detecting ceiling violations could require a new check: when the absolute deadline of an EDF task is changed the implementation could check that the new absolute deadline is not sooner than the current time plus the relative deadline minus the real-time clock jitter.

Another check that might be necessary is detecting incoherent behaviors when changing the relative deadline. If the relative deadline is changed to a value that is greater than the interval from now until the absolute deadline there is an interval during which the absolute deadline is sooner than anticipated by the new relative deadline, which might imply glitches with some degree of priority inversion during the mode change. The main implication of this problem is that the protocol might not be able to ensure mutual exclusion just by itself, and an actual lock might be needed.

7. Integration of DFP in the current Ada dispatching model

Two main issues have to be solved related to the integration of the new EDF_With_Deadline_Floor dispatching with the other policies defined in the standard:

- Backwards compatibility: it is desirable that old EDF applications would need minor changes in their code (or no changes at all) to run with the new EDF & DFP dispatching policy.
- Interaction with the other dispatching policies: in applications with priority-specific dispatching policies it is usual that tasks with different policies interact using protected objects. Effects of this interaction should be well defined in the standard and should not lead to priority inversions nor deadlocks.
7.1. Backwards compatibility

The DFP rules allow an EDF application written for Ada 2005 to execute with no changes for the new EDF & DFP policy. The only difference would be the possible change of dispatching policy identifier in the configuration pragma Task_Dispersing_Policy or Priority_Specific_Dispatching. For this purpose, the new EDF_With_Deadline_Floor dispatching policy should be allowed to have priority bands with multiple levels, although the language would specify that those priorities would be collapsed to the lowest priority in the EDF band when calculating active priorities of tasks.

Assigning deadline floors to the protected objects would be desirable but not required since the default deadline for a protected object would be Ada.Real.Time.Time_Span_Zero, in the same way that the default ceiling priority for a protected object is System.Priority’Last (RM D.3 11/3). Of course assigning deadline floors to protected objects would be very important to improve system schedulability.

7.2. Interaction between EDF_With_Deadline_Floor and FIFO_Within_Priorities tasks

The first situation to consider is when the EDF range is at a higher priority level than the FIFO_Within_Priorities range. In such situation a FIFO task could use protected objects in the EDF range to interact with EDF tasks. The FIFO task would inherit both the priority ceiling and the deadline floor of the protected object.

Deadline floors of protected objects in the EDF priority band are assigned according to the deadlines of the EDF tasks that access them. To avoid a deadline floor violation when they are used by a FIFO task, it is enough to assume that the relative deadline of the FIFO tasks is infinite (Ada.Real.Time.Time_Span_Last). Consequently, a FIFO task accessing an EDF protected object would inherit its deadline floor and could only be preempted by EDF tasks with shorter relative deadlines. This behavior is the expected since, by definition, those tasks with shorter relative deadlines are not going to access the protected object.

In the opposite situation, when the EDF range is below the FIFO_Within_Priorities range, only the basic ceiling locking policy rules go into action: the EDF task that accesses a protected object in the FIFO range inherits the priority ceiling of the protected object and, consequently, can only be preempted by tasks with higher priorities.

7.3. Interaction between EDF_With_Deadline_Floor and Round_Robin_Within_Priorities tasks

It is exactly the same situation as for the FIFO_Within_Priorities policy since the round robin tasks behave like FIFO tasks while executing a protected action (the quantum does not expire until the protected action finishes).

7.4. Interaction between two EDF_With_Deadline_Floor tasks in different priority ranges

Although the utility of having more than one EDF priority range could be arguable, the language must be complete and should take into account that possibility.

It is the user’s responsibility to avoid deadline floor violations when a task from the lower priority range uses a protected object in the higher priority range. The deadline floors assigned to the protected objects should consider the deadlines of all the tasks that access them in both priority ranges.

7.5. Interaction between EDF_With_Deadline_Floor and EDF_Across_Priorities tasks

In the assumption that both versions of EDF are allowed to coexist in the same system, synchronization among tasks in different EDF priority ranges would also work according to the expected properties of the protocols.

In the case that the EDF_With_Deadline_Floor is the higher priority range, the only issue to be taken into account is the correct assignment of deadline floors as discussed in Subection 7.4. If the EDF_Across_Priorities range is the higher priority range, no extra considerations are required since the ceiling locking policy is enough to ensure the correct behavior of the system.

8. Conclusions

The Deadline Floor inheritance Protocol (DFP) has been proposed as an alternative to the Stack Resource Protocol defined in Ada. In this paper we have shown that the DFP is simpler to understand, describe, and implement. DFP also has better performance than SRP. We have also discussed different alternatives for including the protocol in Ada. As a conclu-
sion we can state that the DFP is a good alternative to SRP in Ada and its standardization should be addressed in a future version of the language.

Acknowledgments
This work has been funded in part by the Spanish Government and FEDER funds under grant TIN2011-28567-C03-02 (HI-PARTES).

References
https://hdl.handle.net/10902/1509
Locking Policies for Multiprocessor Ada

A. Burns and A.J. Wellings
Department of Computer Science,
University of York,
YO10 5GH, UK

Abstract

Lock-based resource sharing protocols for single processor systems are well understood and supported in programming languages such as Ada. In contrast, multiprocessor resource sharing protocols are less well developed with no agreed best practice. In this paper we consider what the next version of Ada should support. Two proposals are considered, one requiring a minor change to the current language, another requiring a more substantial change.

1 Introduction

One of the significant extensions made to the definition of Ada during the Ada 2012 revision process was the support provided for programming multi-tasking systems for execution on multiprocessor platforms. The new language features, however, did not provide complete support for this paradigm. In particular, the need to support the efficient use of protected objects accessed by tasks executing in parallel was not really addressed. This lack of support is not really surprising as appropriate locking policies for such shared object have not yet been identified. Although there has been some success in determining the necessary support for scheduling [7], the issue of how best to support multiprocessor lock-based resource control protocols is far from clear. New results are emerging; indeed, there is a plethora of proposed schemes (see [10] for a review of the literature in this area, and the work of Brandenburg [4]); a consensus is, unfortunately, far from being realised.

In this paper we consider two possible policies, one that can be achieved with only minor modifications to Ada, the other requiring a new locking policy to be defined in the language. We concentrate on fixed priority scheduling.

2 Ada 2012 Provision

Ada 2012 allows a collection of processors (CPUs) to be partitioned into a distinct set of dispatching domains, (DDs). A task runs in only one DD. Within a DD, a task can be either bound to a single CPU, or be allowed to execute and migrate over the entire DD. In this way, the two classic ways of scheduling tasks (partitioned and global) can be supported as well as a number of semi-partitioned schemes [5].

Real programs share data; in Ada this means having concurrent access to protected objects (POs). With a single processor, an effective sharing scheme is delivered by the pragma Locking_Policy(Ceiling_Locking) which assigns a ceiling priority to each PO. Whenever a task enters a PO its priority is raised to the ceiling. As a result, blocking time is minimised and deadlocks are prevented (where blocking time is the time a task has to wait until a lower priority task has completed some action). As indicated above, a similar well-accepted scheme for multiprocessor platforms and parallel (not just concurrent) access does not yet exist.
The only support that Ada 2012 gives to concurrency control for POs in multiprocessor platforms is to allow Locking Policy (Ceiling Locking) to mean that a task will spin at the ceiling priority if immediate access to the PO is not possible (as another task on a different CPU is currently executing with the PO lock).

As partitioned systems provide the greatest challenge for the definition of effective locking policies, we restrict our considerations in this paper to programs that consist of a single dispatching domain onto which all tasks are statically allocated. We note that this is the recommended structure for the Ravenscar protocol when implemented on a multiprocessor architecture.

3 Blocking Time

As noted in the previous section, an effective locking scheme must bound the impact on schedulability. With a static task allocation model, an ideal scheme would imply that for any task $\tau_i$ the progress it makes towards completing before its deadline should only be negatively impacted by tasks of a higher priority running on the same processor. Putting this another way, a task should not be blocked by lower priority tasks on the same processor or by any task on another processor.

Unfortunately this ideal cannot be fully met if resources are to be shared under mutual exclusion. So if task $\tau_i$ shares a protected object ($PO_j$) with task $\tau_k$ then, if $\tau_k$ has a lower priority or is allocated to another CPU, blocking will occur. If all POs are only accessed by tasks on the same processor then the priority ceiling protocol [15] (as supported by Locking Policy (Ceiling Locking)) gives a minimum blocking time. Where more than one processor is involved, two issues arise; assume $PO_j$ is being used by $\tau_k$, and $\tau_i$ wishes to gain access:

- should $\tau_i$ continue to spin while waiting to gain access, or should it be suspended? and
- as $\tau_k$ can be locally preempted, by a local high priority task (higher priority than the ceiling of $PO_j$), $\tau_i$ can be held up for a relatively long interval of time.

There is considerable evidence (see [10, 4]) that spinning is the most appropriate approach to use unless protected operations are exceedingly long (which most guidelines argue against). Moreover, suspension brings with it another scheduling issue, as the state 'suspended waiting to gain access to a protected object' is not currently part of the tasking model within Ada. Therefore, it seems appropriate to use spinning as Ada currently recommends. However, to bound the time that it takes to gain access (using spinning) it must be possible to impose some form of queue on the waiting tasks (for example, a FIFO queue – see discussion below).

To deal with the second issue we consider two schemes: Non-Preemptive Locking and a Multiprocessor Resource Sharing Protocol which we name MrsP. These are now considered.

4 Non-Preemptive Locking

If all the POs are executed non-preemptively then, obviously, a task cannot be preempted while holding a PO lock. It follows that it will hold the lock for the duration of the protected action (which we will represent by $a_j$ for $PO_j$). If there is FIFO spinning then the maximum time a task can wait to gain access is $(m_j - 1)a_j$ where $m_j$ is the number of processors on which is allocated a task that uses $PO_j$. This represents external blocking time for the task. Its total execution time for using the PO will be $(m_j - 1)a_j + a_j$ which is just $m_ja_j$.

So for task $\tau_i$ it has external blocking of $(m_j - 1)a_j$ for each resource it uses, and for any local task $\tau_k$ with priority higher than $\tau_i$ it has local blocking of $m_ja_j$. As non-preemption is an extreme form of ceiling
locking then arbitrary task $\tau_i$ will have a single local blocking term equal to the maximum value of $m_ja_j$ over all tasks $\tau_s$ with priority lower than $\tau_i$.

To implement this approach in Ada 2020, a minor variation to the current locking policy is needed. As well as spinning, a FIFO queue is required. This could be a refinement to the definition of `Locking_Policy (Ceiling_Locking)` or could be a new pragma. Non-preemption locking can be achieved with current Ada by just assigning the ceiling priority of all POs to be higher than the priority of all tasks. Alternatively a new explicit locking policy, `Locking_Policy (Non_Preemptive_Locking)`, could be defined.

The disadvantage of the non-preemption approach is that the high priority tasks (those that typically have the shortest deadlines) suffer significant blocking (both local and external). The amount of blocking is bounded but may be too much to guarantee that deadlines will never be missed. If guarantees can be made (blocking is manageable) then there is no better protocol than non-preemption. The next protocol to be defined is for use when non-preemption is not adequate.

We note that once true parallel execution is enabled then failures such an deadlock are possible. Pure non-preemption can lead to deadlock if resource use is nested (i.e one protected object calls another). A common deadlock prevention scheme is to statically order resources and to only allow access to resources with an order number greater than that of any currently held resource. Although this may be considered a restriction on the expressive power of the computational model, it is more expressive than those protocols that ban nested usage or require a group lock – that has the same impact as non-nesting. Within Ada the use of exceptions for ‘out of order’ requests allow robust programs to be developed. We have investigated the use of nested but ordered locks for general multiprocessor resource control protocols elsewhere [10] (though not for non-preemption or MrsP). The adoption of ordered locking for the protocols discussed in this paper is straightforward, though not covered further here.

5 Multiprocessor Resource Sharing Protocol - MrsP

There are a number of multiprocessor priority ceiling protocols discussed in the literature, for example [13, 14, 12, 16, 9, 1, 2, 8, 11, 3]. MrsP has the distinctive property that it deals fairly with both high and low priority tasks. In this section we first define a general model for MrsP, then an Ada-specific version is defined. Finally, an implementation scheme is described.

5.1 The MrsP Protocol

If we return to the single processor notion that all activities associated with protected object $PO_j$ occur at the ceiling priority of $PO_j$, then tasks of high priority are unaffected by the existence of the PO. This is clearly not true for the non-preemptive protocol. But if spinning occurs at the ceiling level then legal preemption of the lock-holding task leads to excessive blocking (as described earlier). MrsP (defined in a non-Ada paper [6]) eliminates this excess by using a ‘helping scheme’ [16].

We define here the MrsP protocol for non-nested PO usage (it is easily extended to nested usage). The basic aspects of the protocol are:

1. All POs are assigned a set of ceiling priorities, one per processor (for those processors that have tasks that use the resource); for processor $p_k$ it is the maximum priority of all tasks allocated to $p_k$ that use the resource.

2. An access request on any PO results in the priority of the task being immediately raised to the local ceiling for the PO.

3. Accesses to a PO are dealt with in a FIFO order.
4. While waiting to gain access to the resource, and while actually using the resource, the task continues
to be active and executes (possible spinning) with priority equal to the local ceiling of the PO.

5. Any task waiting to gain access to a resource must be capable of undertaking the associated computa-
tion on behalf of any other waiting task.

6. This cooperating task must undertake the outstanding requests in the original FIFO order.

So all action takes place at the ceiling priority level (hence higher priority tasks are not impacted). If there
is no local preemptions then a task will wait at most \((m_j - 1)a_j\) to gain access (as described earlier). But
if there is a local preemption then the waiting tasks must take over the work. In the worst case task \(\tau_i\) when
accessing \(PO_j\) will execute the protected actions on behalf of the other tasks that want to use \(PO_j\) (but are
preempted). So rather than spin unproductively the task executes the protected actions on behalf of other
task ahead of it in the queue. As long as at least one task is free to execute then progress will be made. And
if all tasks are locally preempted then it is acceptable for no progress to be made as all involved processors
are executing tasks with higher priorities.

5.2 Ada-Specific MrsP

The general definition of MrsP required each protected object (shared resource) to have a ceiling priority
per processor. Although such a protocol could be defined for Ada, here we give a more constrained protocol
that is closer to current Ada. The first rule of the protocol (defined above) is replaced by:

1. All POs are assigned a ceiling priority, it is the maximum priority of all tasks that use the resource\(^1\).

In other words, this is the current definition of the priority ceiling for a protected object. With this definition
it remains the case that all actions take place at the ceiling priority level (hence higher priority tasks are not
impacted), but the actual value of the ceiling might be higher than it would be with the more expressive
protocol.

So, for example, if a set of tasks \((\tau_1 \ldots \tau_6); each with priority \(p_i\), with \(p_6 > p_5\) etc.) are statically
allocated to two CPUs in the following way: \(\tau_2, \tau_4\) and \(\tau_6\) to CPU(1) and \(\tau_1, \tau_3\) and \(\tau_5\) to CPU(2). Assume
two POs (\(PO_x\) and \(PO_y\)). Let \(\tau_1\) and \(\tau_4\) access \(PO_x\); and \(\tau_2\) and \(\tau_4\) access \(PO_y\). Both POs therefore have
the priority ceiling \(p_4\) (one is used by tasks on different CPUs, the other is used by co-located tasks).

Let the execution time of the protected actions be \(A\). Each task can be analysed in the following way:

\(\tau_1\) Interference from \(\tau_3\) and \(\tau_5\); no blocking, use of \(PO_x\) costs \(2A\).

\(\tau_2\) Interference from \(\tau_4\) (including \(2A\) from its use of \(PO_x\)) and \(\tau_6\); no blocking, use of \(PO_y\) costs \(A\).

\(\tau_3\) Interference from \(\tau_5\); blocking, via \(PO_x\), of \(2A\).

\(\tau_4\) Interference from \(\tau_6\); blocking, via \(PO_y\), of \(A\); use of \(PO_x\) costs \(2A\).

\(\tau_5\) No interference or blocking; ceiling priority of the POs \((p_4)\) less then \(p_5\).

\(\tau_6\) No interference or blocking; ceiling priority of the POs \((p_4)\) less then \(p_6\).

If the full MrsP protocol had been used then \(\tau_3\) would not suffer blocking as \(\tau_1\) would spin at priority \(p_1\)
(not \(p_4\)). Note also that if non-preemption was used then \(\tau_5\) and \(\tau_6\) would also suffer blocking.

\(^1\)This requires all tasks to be assigned priorities that are globally consistent even though the task set is partitioned amongst the
available processors. So, for example, deadline monotonic priorities assignment could be applied to the complete task set before it
is partitioned.
5.3 Implementation of Ada-Specific MrsP

In the context of Ada the notion of one task executing the code of a protected action on behalf of another task (that has been released from an entry barrier) is part of the standard implementation model. Hence the requirements for MrsP are not exceptional.

Ada provides for tasks to migrate between CPUs in the same dispatching domain. Here we assume that tasks are statically allocated to CPUs apart from when they migrate to implement MrsP. Hence all the functionality required (and explained in the following pseudo code) would be supported in an Ada 2012 compliant run-time support system.

The basis of the implementation scheme is to dynamically associate a set of affinities with each PO\(^2\). An attempt to lock a PO adds a CPU to the PO’s affinities. While accessing the PO, the task inherits the PO’s affinities and hence it can execute on any CPU that has a waiting (spinning) task. The following pseudo code outlines a possible implementation; in this code R is the PO, t is the accessing task and p is the CPU from which the lock request is being made (i.e. t is executing on p). The code also keeps track, in R(t), of the id of the task that currently holds the lock (if there is one, otherwise R(t) is null). When the request to lock R is made, the protocol raises the priority of the t to the local ceiling and sets the affinity of the PO to include its own affinity. If the PO is already locked, it sets the current lock-holder’s affinity to be that of the resource and spins at the next available position in the FIFO queue. The task continues to spin until it is released (by the action of some other task calling Unlock).

\[
\text{Lock (R, t, p) ->} \\
\text{raise priority of t to local ceiling of R} \\
\text{Affinities(R) := Affinities(R) + p} \\
\text{if already locked} \\
\text{get current resource user R(t)} \\
\text{Affinities(R(t)) := Affinities(R)} \\
\text{obtain FIFO lock on R and spin} \\
\text{else} \\
\text{Affinities(t) := Affinities(R)} \\
\text{end if} \\
\text{set current lock holder to self} \\
\text{raise priority of t by 1} \\
\text{-- use R} \\
\]

\[
\text{Unlock (R, t, p) ->} \\
\text{Affinities(R) := Affinities(R) - p} \\
\text{Release next task in FIFO queue (if there is one)} \\
\text{Affinities(t) := p} \\
\text{lower priority of t to its base value} \\
\]

\text{It should be noted that all code is run at the ceiling priority of the resource, and hence the protocol does not contain any non-preemptive sections.}

The use of ‘+1’ in assigning the task’s priority means that the task holding the PO lock can migrate to any processor on which there is a spinning task, and then preempt that spinning task. Obviously no other task can be allocated this priority. And to be clear, the spinning task must be waiting to gain access the the same PO.

We define this new locking policy: Locking_Policy(MRSP_Locking).

\(^2\)A task’s affinity is the set of processor on which it is allowed to execute.
6 Global Scheduling

The MrsP approach has been defined for fully partitioned single dispatching domain (DD) systems. Here we consider how to generalise the protocol. Within a single DD if some of the involved tasks are globally scheduled (their affinities are equal the full set of CPUs) then the protocol is easily implemented as progress will always be made unless there are $N$ higher priority tasks runnable (for $N$ CPUs).

If there is a combination of fixed and global tasks (within the same DD) then the partitioned tasks must inherit the full set of CPUs when it is accessing a PO shared with one or more ‘global’ tasks.

Between DDs, the situation is less clear. What is the right means of communicating between tasks within different DDs? Perhaps only non-locking protocols should be used. It would seem inappropriate to use MrsP. One of the key properties of the Ada model, in its use of DDs, is that tasks can never migrate between DDs. The use of migration for MrsP would seem to prevent its use. One is therefore left with non-preemptive locking for POs that are shared between DDs.

7 Conclusions

To complete the definition of Ada for use on multiprocessor platforms it is necessary to define appropriate locking policies for protected objects (PO). Currently, the language recommends spinning at the ceiling priority of the PO. We argue here that, at a minimum, a queuing discipline needs to be added to this policy. And that, at least, FIFO queuing should be supported.

With FIFO spinning, non-preemptive locking (if it can deliver a schedulable system) is an acceptable policy. It is easy and efficient to implement. However, on a single processor system, non-preemption is not considered sufficient (hence the use of ceiling locking). On a multiprocessor, blocking is inevitably increased and it is therefore unlikely that non-preemption can form the basis of a general purpose solution.

The MrsP policy uses controlled migration to ensure that external and internal blocking is limited (to its minimum value). High priority tasks with short deadlines will not suffer blocking from POs with lower ceiling priorities. Clearly an implementation is required to evaluate the overheads with the policy. We recommend that such an implementation is attempted, and if successful the policy be included in Ada 2020.

References


Lock-Free Protected Types for Real-Time Ada

Geert Bosch
AdaCore
104 Fifth Avenue
New York, NY 10011
bosch@adacore.com

Abstract

The Ravenscar profile defined by Ada prevents deadlock and starvation of conforming Ada programs on monoprocessor systems, allowing for better reasoning about real-time behavior. While defined in terms of mutual exclusion, we show Ada's protected types are general enough to allow an enhanced compiler to automatically generate appropriate lock-free synchronization code even on multiprocessors for some basic datastructures. In the context of real-time systems this allows for a more deterministic real-time response and improves the ability to statically analyze generated code.

1 Introduction

By using appropriate scheduling policies, Ada programs using protected types on a monoprocessor system can ensure freedom of deadlock and bound priority inversion, by simply not yielding the processor or performing other potentially blocking operations while in a critical section. This property unfortunately does not extend to multiprocessors. The locks typically used for implementing protected types on such systems, can easily lead to deadlock: consider two tasks simultaneously accessing the same two resources, but in different order. Each will hold access to one, while waiting for the other to become available.

Modern multiprocessors invariably provide read-modify-write instructions or more advanced memory transactions that avoid the need for locking in many situations. The question is how to take advantage of these primitives without giving up the abstraction, flexibility and robustness of protected objects. The use of atomic primitives, memory barriers or transactional memory are implementation details after all, that should not appear in actual user code [1].

We will show that Ada’s protected objects constitute an excellent abstraction for both atomic primitives and transactional memory, allowing a compiler to automatically choose the best implementation for each context. In addition we will suggest two new Ada 2012 aspects that will allow the programmer to specify restrictions on the use of shared data that allow efficient implementation and provable absence of deadlock and unbounded priority inversion for Ada programs using the Ravenscar profile on a multiprocessor real-time system.

2 Protected Types

Ada’s protected types are based on the concept of monitors that originated in Concurrent Pascal [2]. A protected object is an instance of a protected type and contains one or more shared variables, together with a number of subprograms to query or update the data. Their semantics are specified in terms of acquiring an execution resource, apparently implying the use of locks, at least on multiprocessors. Whenever two synchronized objects call on each other on such systems, there is a possibility for priority inversion or even
deadlock. All tasks using a specific lock have to cooperate and establish a global protocol to avoid being cancelled or blocked due to preemption or further locking operations, while holding a lock.

While solving these problems in their entirety just by improving the language implementation is not currently feasible, a much restricted use of those synchronized objects that may be accessed concurrently from multiple processors, together with existing scheduling-based solutions for synchronized objects accessed from a single processor, allows compilers to automatically determine a suitable lock-free implementation when possible.

2.1 Lock-Free Protected Objects

This section will show how an enhanced implementation of the Ada language can provide an effective way to make certain data structures lock-free or wait-free, potentially even avoiding any locking or atomic primitives at all, or automatically falling back to atomic primitives or locks where required and acceptable. Protected objects with entries, that specify barriers, will be considered separately in 4.1.

Protected components are always private and can only be accessed through calls on protected procedures, which may perform updates, and functions, which only read. Together protected procedures and functions are referred to as protected actions.

Protected functions may execute concurrently with each other, but a protected procedure call is defined as acquiring the execution resources associated with a protected object for exclusive read-write access. In addition, two protected actions on the same protected object establish a sequential relationship between those actions, if at least one is a call on a protected procedure.

Protected actions may not execute delay statements, entry calls or other potentially blocking operations.
A task trying to access a contended protected object is not considered to be blocked however, but is instead allowed to consume execution resources while waiting its turn. This allows both for the use of spinlocks, and efficient implementation of protected objects using speculative execution and transactional memory. See figure 1 for possible implementations.

### 2.2 Mutual Exclusion through Scheduling

While for large general-purpose multiprocessor systems with many independent threads of execution, it is important to allow migration of tasks between processors to balance the load and maximize throughput, for small real-time systems it is often more appropriate to partition a multiprocessor system and statically assign tasks to specific processors. In particular, in context of Ada’s Ravenscar profile, a very efficient lock-free implementation of protected operations is possible when it is known a protected object is only accessed by tasks running on a specific processor.

The Ravenscar profile requires the `FIFO_Within_Priorities` scheduling profile as defined by Ada, where tasks can only be pre-empted by higher-priority tasks, as well as the `Ceiling_Locking` locking policy. With this policy, every protected object $P$ has a priority ceiling associated with it. Tasks with a higher priority than that of $P$ cannot call on $P$, while threads with a lower priority temporarily raise their priority to the ceiling priority.

While one task is executing a protected action on $P$, other tasks with a priority of at most $P$ will not be able to execute until the protected action has completed. Thus, mutual exclusion is achieved by acquiring the execution resource (processor), without any need for the use of locking or atomic primitives. As a result this priority ceiling protocol guarantees freedom of unbounded priority inversion or deadlock [5].

Whereas the requirement of single-processor access is trivially true on monoprocessors, on multiprocessors this condition cannot easily be checked at compile time. Though Ada already has a `CPU` aspect for tasks, assigning a task to a specific processor, there is no such aspect for protected objects. Adding such an attribute would allow avoiding locks on protected objects that are known to not be accessed by different processors.

### 3 Transactional Implementation of Protected Actions

This section examines the ramifications of using a transactional approach for implementing protected actions in a lock-free manner on multiprocessors, and the conditions under which this new technique meets the requirements of the language specification. Then we present our initial limited implementation, which even with all its restrictions has more capabilities than a fixed library of atomic primitives can have, while avoiding the problems inherent to a library-based approach.

#### 3.1 Transactional Memory

A memory transaction is a finite sequence of operations on a shared memory, executed by a single task, that appears to execute both serially – not interleaved with operations from other tasks – and atomically. Each transaction makes a sequence of tentative changes to shared memory. When the transaction completes, it either commits, making all changes visible to all tasks instantaneously, or aborts. In this last case, all changes are discarded and never become visible to any other task [4]. Two transactions conflict if they cannot both be committed without potentially violating the serializability or atomicity requirements. In this case, when one transaction commits, the other one is doomed, as it will have to be aborted eventually.

The equivalence of a memory transaction and a protected action is based on the following assumptions:

- Memory transactions are bounded in duration
A successful memory transaction is equivalent to a single atomic read-modify-write instruction, and therefore a successful spinlock acquisition.

An aborted memory transaction is equivalent to a single atomic read and therefore an unsuccessful spinlock acquisition.

Increasing execution priority to some ceiling, executing a single atomic instruction and restoring execution priority is equivalent to executing the atomic instruction at the original priority.

While an application cannot in general detect the difference between speculatively executing the body of the protected operation or trying to acquire a spin lock, the maximum amount of time spent in a doomed transaction attempt must be bounded for this to be true. Similarly, a transaction may not execute any instructions that are observable by another task or have an external effect. Though these two limitations are the only fundamental restrictions on using memory transactions, practical implementations will have further limitations in both supported operations and transaction size.

### 3.2 Timing Implications of a Transactional Implementation

Protected components, such as Count in the example from figure 3, can only be accessed by protected actions of that type, so it is possible for the compiler to analyze all code that may access the protected components and determine if all protected operations can be implemented as transactions. For real-time systems this may include determining the worst-case execution time $T_{try}$ for a transaction attempt in order to bound overhead due to executing a doomed transaction. Based on this analysis, the compiler can automatically choose between available implementations. Callers don’t have to know the distinction and can make a regular function or procedure call in any case.

As our initial goal was to provide a better alternative to a fixed library of atomic primitives on the widest range of systems possible, including real-time embedded systems, we have decided to follow a minimalistic approach to memory transactions based on compare-and-swap or, equivalently, load-link/store-conditional primitives, that can be supported by the large majority of current multiprocessor hardware. We require that memory transactions only access a single location in shared memory: a specific protected component. The atomic compare-and-swap compares the current value of a given memory location with an expected value, and if equal replaces it with the desired value computed by the body of the protected action. This allows a transaction to be implemented as shown in figure 2.

As it is common for protected procedures to only conditionally update a component, the compare-and-swap is omitted if both the expected and desired values are the same. In this case the procedure effectively has the same semantics as a protected function and its execution will be wait-free. This last property allows protected actions to implement objects with an infinite consensus number, as defined by Herlihy in [3].

Table 1 shows how the transactional implementation of protected action leaves one case where timing may not be bounded. A given CPU performing an update will attempt to commit at a minimum rate of $1/T_{trans}$, and will succeed if there is no successful update by another CPU for $T_{trans}$ seconds. When the
rate of successful updates approaches this minimum, maximum wait times for updates may be unbounded. Readers are never affected, and do not block writers. A simple restriction to ensure fairness in a system with $N$ writers is for the time between updates to be at least $NT_{\text{trans}}$. This will bound the worst case time to perform an update to that period.

### 3.3 Restrictions on Protected Actions

For the initial implementation of lock-free protected objects, a number of constraints have been adopted:

1. To bound the time a speculatively executing transaction keeps executing after a conflicting transaction has been committed, the following Ada constructs are forbidden in lock-free protected actions:
   - loop, goto and procedure call statements
   - calls to non-static functions
   - quantified expressions

2. To restrict accesses to shared memory to a single location, a protected action can only refer to a single component, the size of which must not exceed the maximum size for which lock-free atomic primitives are supported. The component may itself be a composite variable such as a record or array. Additionally, the following are not allowed:
   - access to non-local variables (constants are fine)
   - non-elementary parameters
   - dereferences of access values (pointers)
   - address clauses
   - imported or exported entities
   - allocators (new)
3. To prevent instructions with external effect, volatile variables are not allowed.

None of the above restrictions are fundamental, and future implementations using transactional memory may eliminate most or all of them, though for real-time systems worst-case execution time of a failed transaction must be limited. Any objects not satisfying these constraints will use one of the other approaches given in figure 1. Given this list of restrictions, it is probably worth to mention that the handling and raising of exceptions is fine. In the procedure case, a raise statement still requires the transaction to be committed first. If the commit fails, the entire execution has to be redone as usual. Only when the commit succeeds will the raise statement be processed.

For exceptions raised as a result of language defined checks, such as for overflow in integer arithmetic, Ada explicitly allows for raising an exception anywhere in the execution of the action, so no commit is necessary. From the user’s point of view, such rollback is indeed better than leaving the protected object in a potentially inconsistent state.

3.4 Recommendations for Ada’s Ravenscar Profile

In order to be able to better reason about the behavior of parallel programs conforming to the Ravenscar Profile, two new aspects and a new restriction are proposed for the Ada language. For a protected type declaration or single protected declaration, the following representation aspects may be specified:

- **Lock_Free.** The type of aspect Lock_Free is Boolean
- **CPU.** The aspect CPU is an expression, which shall be of type System.Multiprocessors.CPU_Range

The expression specified for the CPU aspect of a protected declaration is evaluated for each protected object. The CPU value is then associated with the protected object whose task declaration specifies the aspect. Program_Error is raised when calling a protected operation on a protected object associated with a different CPU than that of the current task. It is a bounded error if the protected object has an associated CPU while the current task has not.

A lock-free protected type is one for which the aspect Lock_Free is True. A lock-free protected object is one for which the aspect Lock_Free is True, or any object of a lock-free type. Operations on a lock-free protected object are indivisible. This guarantees that there is no possibility of deadlock involving only such operations. It is illegal to specify the Lock_Free aspect for an object or type for which the compiler cannot guarantee indivisible updates.

A new restriction No_Locking specifies that each protected object that is not lock-free, will be associated with the CPU of the task creating it, which will be the environment task on Ravenscar systems. This will ensure the program is free from deadlock.

4 Experimental Results

When specifying with Lock_Free or with Lock_Free => True, the current version of GNAT Pro will avoid locks, either through scheduling or by using transactions, or give a compile-time error if this is not possible. When specifying with Lock_Free => False, the implementation will use explicit locking and unlocking calls. The attribute ‘Lock_Free can be used to query whether the actual implementation is indeed lock free.

Because generally an Ada program unit may only depend on the specification of other units, and not the implementation, callers of protected operations without Lock_Free aspect must assume they could be implemented using locks. Therefore, the protected object will include space for a lock, and initialization
and finalization procedures must be called at appropriate points, even if the compiler ends up choosing a lock-free implementation. In that case the actual initialization and finalization will be null procedures, and the lock will remain unused.

By specifying the `Lock_Free` aspect in the specification, no space overhead is necessary\(^1\). This allows a protected object to be as small as a single byte. Similarly, callers will know that no initialization and finalization is required. Protected objects can even be cheaper than simple atomic variables, because accesses may not require full barriers, as they are not defined to be synchronizing with other reads.

We have implemented the minimal approach shown in figure 2 in the GNAT Pro compiler. The GCC 4.7 back end, on which this compiler is based, provides the atomic primitives as compiler intrinsics. When these are not known at compile time to be lock-free, the compiler will reject the program if the `Lock_Free` aspect was specified, or otherwise fall back on an alternative implementation, if necessary. The compiler knows the semantic properties of its intrinsics, and will avoid code motion, speculation or other optimizations where needed.

### 4.1 Protected entries

While we do not yet support lock-free protected types with entries, we have implemented the `Suspension_Object` from the Ada.Synchronous_Task_Control package using a lock-free protected object describing its state. The state object contains a `Task_Id` indicating the task that is blocked on a False condition, null if it is False without blocked task, or a special `Task_Id` value to indicate a True value. The `Set_True` and `Suspend_Until_True` operations consist of two parts: the updating of the suspension object, and either the suspension or wakeup. If it is necessary to wake up a task on another processor, a per-processor queue with `Insert` and `Remove_All` operations can be implemented in lock-free fashion. For the Ravenscar profile, where entries conditions consist of just a single boolean variable, it should be possible to implement the entry by means of a `Suspension_Object`. This still remains to be done.

### 4.2 Optimizing Opportunities

Because the lock-free code generation for protected objects does not require any library calls, and generally requires just a few instructions, all code can be inlined at the call sites. This means that synchronization instructions can be optimized in a similar fashion to regular computational code. In particular, consecutive memory barriers may be eliminated when the only instructions between them are known to not access potentially shared memory.

### 4.3 The Simple Counter

Looking at the example of the simple counter, it is worth mentioning that the lock-free compilation of this code includes all required Ada semantics, including overflow checking. A call to `Increment` that would cause an overflow situation does not lead to erroneous behavior or silent wrap-around, but will raise a `Constraint_Error` exception, and leave the object in its unmodified state with the maximum possible value of `Count`.

When it can be proven that overflows cannot happen, or when the programmer requests these checks to be suppressed, the compiler may assume overflow cannot happen and optimize the code to an atomic increment, if such is provided by the target hardware. We have not implemented this optimization yet, but manual replacement of the generated code shows about a 15% improvement. Even using an unsafe, regular non-atomic increment instruction does not create a large speedup. Performance analysis shows that this is due to the cost of cache misses, which cannot be avoided.

\(^1\)To be honest: if a non-static ceiling priority is specified, this still needs to be stored with the object.
protected Counter is
    procedure Increment;
    function Get return Natural;
private
    Count : Natural := 0;
end Counter;

protected body Counter is
    procedure Increment is
        begin
            Count := Count + 1;
        end Increment;

    function Get return Natural is
        begin
            return Count;
        end Get;
end Counter;

Figure 3: Protected object implementing a simple counter

<table>
<thead>
<tr>
<th>Lock_Free aspect</th>
<th>False</th>
<th>True</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protected object size (bytes)</td>
<td>120</td>
<td>4</td>
</tr>
<tr>
<td>Mean time per update (ns)</td>
<td>132</td>
<td>24</td>
</tr>
</tbody>
</table>

For testing we use 4 tasks, each with its own protected counter. Of the increments, 90% are on the task’s own counter, and 10% on the next task’s counter. This means that there is a moderate amount of contention. All testing was done on a dual socket server with two Intel Xeon 5680 CPUs, each of which has 6 cores.

5 Conclusion

By taking advantage of the abstract interface provided by Ada’s protected types, it is possible to translate Ada’s protected objects without requiring traditional locking using a variety of techniques, from scheduling protocols to atomic primitives and memory transactions. Even with the large set of restrictions in our initial implementation, lock-free protected objects are more flexible and powerful than a fixed library of atomic primitives can be. The existing Ravenscar profile, together with a few minor extensions, allows creating multiprocessor real-time systems that avoid deadlock and bound priority inversion.

6 Acknowledgements

Many thanks to Vincent Pucci, who did amazing work on the implementon of lock-free protected types in the GNAT compiler. I’d also like to thank the anonymous reviewers whose helpful comments allowed me to improve this paper.
References


Programming Simple Reactive Systems in Ada: Premature Program Termination

A.J. Wellings, A. Burns, A.L.C. Cavalcanti and N.K. Singh
Department of Computer Science, University of York
Heslington, York YO10 5GH, UK
(andy.wellings, alan.burns, ana.cavalcanti, neeraj.singh}@york.ac.uk

Abstract

Reactive systems are systems that respond to stimuli from the environment within the time constraints imposed by the environment. This paper identifies an ease-of-use issue with Ada for developing small reactive systems. The problem is that Ada defines program termination solely in terms of whether all tasks have terminated. There are, however, some advantages in adopting a purely interrupt-driven design in the implementation of small reactive systems. With such programs, there are no tasks other than the environment task, which typically terminates when it finishes executing the main program. We argue that this is not the expected behaviour. To avoid this unexpected premature program termination, this paper proposes changes to the program termination conditions in the language so that the environment task of an active partition terminates when (1) all its dependent tasks have terminated, (2) the partition has no active timing events, and (3) no handlers are attached to interrupts that are to be serviced by the partition. However, this would be a non-backward compatible change, and some programs that currently terminate would not terminate with the new rules if they still have attached interrupt handlers or outstanding timing events.

1 Introduction

Reactive systems respond to stimuli from the environment within the time constraints imposed by the environment [3]. Hence they are typically event-triggered systems where the absence of an expected event can also be considered as an event itself (a timeout event). In this short paper, we illustrate an ease-of-use problem when programming small reactive systems in Ada.

Simple designs of a reactive system are based on (centralised) state information that is manipulated by atomic operations. In these cases, the kernel of the system is typically implemented as a deterministic automaton. Given a set of available inputs (or events), the automaton selects a transition that can be performed and executes the associated code implementing an atomic opera-
tion; it is then ready for its next reaction – this is illustrated in Figure 1. More complex reactive systems may be structured hierarchically and allow parallel or distributed execution of operations. Esterel [2] is a language that targets the implementation of potentially complex reactive systems. Here, however, we are concerned with single processor systems that are usually embedded. Our motivating example has been that of a cardiac pacemaker [4].

2 Reactive Systems and Ada

In embedded Ada applications, interaction with the environment is via interrupt handling and the various timeout mechanisms supported by the language. With systems for which environmental events must be polled, Ada supports abstractions that allow periodic tasks to be implemented, or periodic timing events to be generated. Here, we propose to use a combination of the Ada interrupt handling facility and the Ada.Real_Time.Timing_Events package to implement deterministic automata, which in turn can be used to support simple reactive applications. Hence, we are interested in applications for which there are no tasks, other than the environment task that executes the main program.

Our proposed approach is to encapsulate all the procedures for handling interrupts and timing events within the same library-level protected object, as illustrated in Figure 2. The advantages of this approach (for small systems) are:

- All event-handling procedures execute atomically with respect to one another – this is guaranteed by the language’s semantics for protected objects.

- The worst-case execution time needed to respond to each event is well defined – there can be no preemptions (hence, no cache refills during execution, etc) and the code to be executed is clearly identifiable from the procedure that handles the event.

- The response time for each event is predictable – there can be no preemptions and protected procedures cannot self suspend for any reason.

There are, perhaps, two disadvantages of this implementation model.
1. Interrupts that are unable to be delivered are assumed to be queued by the hardware – this is usually the case and certainly required for a reactive system that must keep up with the environment.

2. It is not possible to control the order of handling when multiple events are available – this is inevitable when interrupts come in from various sources; typically the hardware priority of the device will determine the order in which the processor processes the outstanding interrupts. Of course, if events are being polled for, the order of polling will allow some control. More significantly, there is no order specified for the servicing of timing events that occur at the same time.

These disadvantages may not be relevant for (many) applications. In the next section, we consider a significant case study where these issues do not pose a serious problem.

3 An Illustrative Example

In this section we summarise an Ada implementation of our motivating example. A full description of the Cardiac Pacemaker is given in [4] along with details of the implementation in Safety Critical Java and Ravenscar Ada.

A pacemaker system is a small electronic device that helps the heart to maintain a regular beat. The conventional pacemakers serve two major functions: pacing and sensing. The pacemaker’s actuator paces by the delivery of a short, intense electrical pulse into the heart. The pacemaker sensor uses the same electrode to detect the intrinsic activity of the heart. So, the pacemaker’s pacing and sensing activities are dependent on the behavior of the heart.

Essentially, the pacemaker monitors (senses) the intrinsic activity of the heart and in the absence of such activity forces the heart to beat by the delivery of an electric current (pacing). The sensing and pacing activities can be performed in both chambers of the heart (atrial and ventricle). The
control requirements can be complex, and it is beyond the scope of this paper to consider them in
detail (see [4]).

Figure 3 depicts the scenarios for sensing and pacing activities. The Ventriculoatrial Interval
(VAI) is the maximum time the pacemaker should wait after sensing ventricle activity (either in-
trinsic or paced) for some indication of intrinsic activity in the atrium. If none is present, the
pacemaker should pace in the atrial chamber. The Atrioventricular Interval (AVI) is the maximum
time the pacemaker should wait after sensing atrial activity (either intrinsic or paced) for some
indication of intrinsic activity in the ventricles. If none is present then the pacemaker should pace
in the ventricle chamber. After every pace in the ventricle chamber, there is some sensed activity
in the atrial, but this is not true intrinsic heart activity and should be ignored. The Postventricular
Atrial Refractory Period (PVARP) indicates the length of time during which such activity should
be ignored. Sensed atrial activity is called a P wave, and sensed ventricular activity is called a QRS
complex. A T wave follows a QRS complex and represents the recovery of the ventricles. A P
wave is sensed when the amplitude of the signal is greater than a threshold $P_{th}$ for $T_p$ time units.
Similarly, a QRS complex is detected when the amplitude of the signal is greater than a threshold
$QRS_{th}$ for $T_{QRS}$ time units. For safety, it is imperative to ensure that pulsing does not occur during
a T wave.

There are four possible scenarios for pacing and sensing activities, which are given in Figure 3.

- **Scenario A** – shows a situation in which the pacemaker paces after standard time intervals
  (VAI and AVI) in both chambers. This is the reaction when no intrinsic heart activity is
detected.

- **Scenario B** – shows a situation in which the pacemaker paces in the atrial chamber after
  VAI, while the ventricular pacing is inhibited due to a sensing of intrinsic activity from the
  ventricle.

- **Scenario C** – shows a situation in which intrinsic atria activity is sensed, pacing is inhibited in
  the atrial chamber, but occurs in the ventricular chamber after AVI (due to a lack of intrinsic
  ventricular activity).

- **Scenario D** – represents the case where both pacing activities are inhibited due to a sensing
  of intrinsic activities in both chambers.

Table 1 summarises the main timing requirements for a particular pacemaker, and Figure 4 illus-
trates the basic requirements that must be met. The required pacing activities are not regular enough
to be controlled by a periodic activity. They are essentially aperiodic and time-triggered depending
on the presence or absence of intrinsic heart activity. There are no interrupts generated other than
those needed to support Ada’s timing events.

There are clearly several software architectures that could be adopted. Here we use Ada’s timing
events to control both the sensing and pacing activities as this eliminates the needs for tasks and,
therefore, reduces the size of the Ada run-time support needed. There are two main timing events:
the first (Watchdog) is used to detect the absence of intrinsics activities and to control the pacing
Figure 3. Example Pacing Scenarios

Time goes left to right, and a flat line indicates no heart activity. A spike above the lines indicates intrinsic activity and a spike below the line indicates activity as a result of the action of the pacemaker. A rounded spike indicates activity in the atrial and a sharp spike indicates activity in the ventricle.
Time Intervals | Purpose | Time in milliseconds
---|---|---
Length of a P wave ($T_P$) | Time during which intrinsic atrium activity must be sensed | 110
Duration of pulse ($T_{pulse}$) | the time for which the pulse current must be maintained | 1
Length of a QRS complex ($T_{QRS}$) | Time during which intrinsic ventricular activity must be sensed | 100
Atrioventricular interval (AVI) | Provides time for ventricle to fill following an atrial contraction | 150
Ventriculoatrial interval (VAI) | Ensures an atrial pulse following a ventricular pulse | 850
Postventricular atrial refractory (PVARP) | Ensures atrium doesn’t falsely sense ventricular activity | 350
Mode Switching Interval (MSI) | Time between two atrial events used to change modes | 500

Table 1. Example Timing Intervals in a Single Heart Beat

![Diagram](image)

Figure 4. The Required Pacing Cycle
current, and the second (Sensor Readings) is used to initiate the reading of sensors. A single protected object (Timer) is used for encapsulating the handlers for these timing events.

The details of the Ada approach are shown in Figure 5. The algorithm follows closely that given in Figure 4 which informally defines the requirements. The protected object code is given below. The full code for the application can be found at http://www.cs.york.ac.uk/circus/hijac/case.html.

```ada
with System; use System;
package Timers is
type Sensor is (Atrium, Ventricle);
pattern Timer is
  pragma Priority(Priority’Last);
  procedure Atrium_Pace_On(E : in out Timing_Event);
  procedure Atrium_Pace_Off(E : in out Timing_Event);
```

**Figure 5. Cardiac Pacemaker Architecture in Ada**
procedure Ventricle_Pace_On(E : in out Timing_Event);
procedure Ventricle_Pace_Off(E : in out Timing_Event);
procedure PVARP_Countdown(E : in out Timing_Event);
procedure Sensor_Read(E : in out Timing_Event);

private
  Reading : Sensor := Atrium;
  IntrinsicV_Sensed : Boolean := False;
end Timer;

General_Timeouts : Timing_Event;
Sensor_Readings : Timing_Event;
end Timers;

-- various with and use clauses

package body Timers is

protected body Timer is

procedure Atrium_Pace_On(E : in out Timing_Event) is
begin
  Pace_A_On; -- turns pace current on
  Set_Handler(General_Timeouts, Clock+Pulse_Duration, Atrium_Pace_Off’Access);
end;

procedure Atrium_Pace_Off(E : in out Timing_Event) is
begin
  Pace_A_Off; -- turns pace current off
  Set_Handler(General_Timeouts, Clock+AVI, Ventricle_Pace_On’Access);
  Reading := Ventricle;
end;

procedure Ventricle_Pace_On(E : in out Timing_Event) is
begin
  Pace_V_On; -- turns pace current on
  if IntrinsicV_Sensed then
    IntrinsicV_Sensed := False;
  end if;
  Cancel_Handler(Sensor_Readings, Set);
  pragma assert(Set);
  Set_Handler(General_Timeouts, Clock+Pulse_Duration, Ventricle_Pace_Off’Access);
end;

procedure Ventricle_Pace_Off(E : in out Timing_Event) is
begin
  Pace_V_Off; -- turns pace current off
  Set_Handler(General_Timeouts, Clock+PVARP, PVARP_Countdown’Access);
  Reading := Atrium;
end;

procedure PVARP_Countdown(E : in out Timing_Event) is
res : Float;
begin
res := Read_Atrium_Data; -- measure intrinsic activity
if res > 0.3 then
  -- Intrinsic activity sensed in atrium;
  Set_Handler(General_Timeouts, Clock+AVI, Ventricle_Pace_On’Access);
end if;
end;
procedure Sensor_Read(E : in out Timing_Event) is
res : Float;
begin
if Reading = Atrium then
res := Read_Atrium_Data;
if res > 0.3 then
-- Intrinsic activity sensed in Atrium
Set_Handler(General_Timeouts, Clock+AVI, Ventricle_Pace_On'Access);
Reading := Ventricle;
end if;
Set_Handler(Timers.Sensor_Readings, Sensor_Period, Timer.Sensor_Read'Access);
else
-- reading ventricle
res := Read_Ventricle_Data;
if res >= 0.9 then
-- Intrinsic activity sensed in ventricle;
Set_Handler(General_Timeouts, Clock+PVARP, PVARP_Countdown'Access);
Reading := Atrium;
IntrinsicV_Sensed := True;
else
IntrinsicV_Sensed := False;
end if;
end if;
end Sensor_Read;
end Timer;
end Timers;

The main program sets up the first timing event. From that point on, every timing event handler will set up at least one other timing event.

-- with clauses omitted
procedure Pacemaker is --DDR
begin
Set_Handler(Timers.Sensor_Readings, Clock+PVARP, Timer.Sensor_Read'Access); -- sets initial event
end Pacemaker;

The resulting program design is efficient, with handlers only executing when control is needed. (We observe, however, that there is a problem with this solution, which we discuss in the next section.) The handlers for timing events in Ada are called from the Ada run-time clock interrupt handling code. Hence, no Ada tasks are actually required. As all code is run at interrupt-level, it is imperative to keep the handling code as simple and as short as possible so that the computation can be completed before another timing event needs to be set. The Atrium_Pace_On, Atrium_Pace_Off, Ventricle_Pace_Off and PVARP_Timeout handlers are mutually exclusive events. Hence the requirements are for:
- **Atrium_Pace_On** to be completed before the pulse duration (Tpulse) has expired (1 millisecond – see Table 1) when **Atrium_Pace_Off** needs to be called. The code in the handler is simply one actuator operation and the setting of one timing event.

- **Atrium_Pace_Off** to be completed within safety margins for the pulse duration – and before the AVI (150 milliseconds). The code in the handler is simply one actuator operation and the setting of one timing event.

- **Ventricle_Pace_On** to be completed before the pulse duration (Tpulse) has expired (1 millisecond – see Table 1) when **Ventricle_Pace_Off** needs to be called. The code in the handler is simply one actuator operation and the setting of one timing event and the canceling of another.

- **Ventricle_Pace_Off** to be completed within safety margins for the pulse duration – and before the PVARP duration (350 milliseconds). The code in the handler is simply one actuator operation and the setting of one timing event.

- **PVARP_Timeout** to be completed before the VAI duration (850 milliseconds). The maximum code in the handler is simply one sensor operation and the setting of two timing events.

The sensor reading handlers are similarly simple, mainly consisting of one sensor reading operation and at most two settings of timing events. The asynchronous relationship between the watchdog and sensor-reading timing events means that it is possible for one event to want to fire whilst the other event is being handled. Hence, the response time of each handler must include an interference time equal to the maximum handling timing of the other event.

### 4 The Premature Termination of Simple Reactive Programs in Ada

The implementation of our motivating example, as presented so far, does not work. This is because the program terminates as soon as the main program finishes executing; there are no tasks to keep the program alive. Hence, we had to add a delay in the main procedure to keep the program alive, as illustrated below. (The alternative is to introduce a low priority idle task, or to call Suspend_Until_True on a suspension object.)

```ada
-- with clauses omitted
procedure Pacemaker is --DDDR begin
  Set_Handler(Timers.Sensor_Readings, Clock+PVARP, Timer.Sensor_Read'Access); -- sets initial event
delay until Ada.Real_Time.Time_Last;
end Pacemaker;
```

This is clearly not very elegant. The reason is that the Ada programming language specification defines the following.

The execution of a program consists of the execution of a set of partitions. Further details are implementation defined. The execution of a partition starts with the execution
of its environment task, ends when the environment task terminates, and includes the executions of all tasks of the partition. The execution of the (implicit) task body of the environment task acts as a master for all other tasks created as part of the execution of the partition. When the environment task completes (normally or abnormally), it waits for the termination of all such tasks, and then finalizes any remaining objects of the partition. (Ada Reference Manual Section 10.2 Program Execution).

On a single processor system, a program is considered to be a single active partition that terminates when its environment task terminates. Hence, when the environment task finishes executing the main program, it checks to see if there are any active tasks. If there are not, the program terminates.

**Modifying Ada Program Termination Rules**

In order to ensure that unexpected terminations of programs do not occur, the Ada language specification would need to be changed so that:

The environment task of an active partition terminates when all its dependent tasks have terminated *and* the partition has no active timing events *and* there are no handlers attached to interrupts that are to be serviced by the partition.

This would require all interrupt handlers to be attached dynamically rather than via the static use of the `Attach_Handler` pragma. Alternatively, the language could be modified to allow dynamic detaching of a static handler. For the Ravenscar profile, where programs are not meant to terminate, an additional pragma could be used to indicate that there are no tasks in the program.

With this new rule, our implementation described in the previous section would be correct. However, such a change would not formally be backward compatible. Some programs that currently terminate would not terminate with the new rules, if they still have outstanding timing events or attached interrupt handlers. Arguably, a program that terminates with an outstanding timing event has a dormant fault – it is almost the equivalent of mixing the terminate and a delay alternatives in a select statement (only at the program level rather than the task level). For interrupt handling, the current rules generate a race condition between the interrupts being handled and the program terminating. The new rules, however, could lead to the case where handlers are attached but all interrupts have been disabled. This would be equivalent of writing a program that deadlocks.

### 5 Conclusions

This paper has identified an ease-of-use issue with Ada for developing small reactive systems. The issue is that Ada defines program termination solely in terms of whether all tasks (application and environment) have terminated. There are some advantages in implementing small reactive systems as being interrupt driven, be they timing interrupts or other device interrupts. With such programs, there are no tasks other than the environment task, which typically terminates when it
finishes executing the main program. This is not the expected behaviour. While the work-arounds are simple, they are a little inelegant.

To avoid this unexpected premature program termination, it is necessary to change the program termination conditions in the language so that the environment task of an active partition terminates when all its dependent tasks have terminated and the partition has no active timing events and no handlers that are attached to interrupts that are to be serviced by the partition. It is interesting to note that the initial version of the Real-Time Specification for Java had a similar problem with the way it specified program termination[1]. There, all asynchronous event handlers attached to environment-generated events (happenings in the RTSJ terminology) were treated as daemon Java threads. This resulted in purely reactive programs suffering from premature termination.

The paper has also illustrated that for time-driven reactive programs, the order of servicing timing events is undefined when more than one event is due at the same time. Most implementation probably service them in a FIFO order. There may be some merit in allowing a priority to be assigned to each event. Also, allowing periodic timing events to be specified would remove the need to continually reset them.

Acknowledgements

The authors are grateful to Johan Nielsen for pointing up the backward incompatibility of our proposals, and for suggesting the use of suspension objects as another work-around.

References


Execution time timers for interrupt handling

Kristoffer Nyborg Gregertsen  
Department of Applied Cybernetics  
SINTEF ICT, Trondheim, Norway  
kristoffer.gregertsen@sintef.no

Amund Skavhaug  
Department of Engineering Cybernetics  
NTNU, Trondheim, Norway  
amund@itk.ntnu.no

Abstract

This paper argues that the addition of interrupt timers follows naturally by execution time measurement for interrupt handling introduced with Ada 2012, and that full execution time control for interrupts allows safety against unexpected interrupt rates that could not be achieved in an efficient and easy manner otherwise. Hence, it is argued that this feature should be considered for the next revision of Ada.

1 Introduction

Scheduling analysis relies on the worst-case execution time (WCET) of tasks being known. However, finding this WCET by timing analysis may be very hard and costly, and performance enhancing techniques such as pipelines and caches make it even harder [16]. Another approach is execution time control. It combines a mechanism that measures the total time a task has been executed and calls a handler when a specified timeout value is reached, and a scheduling policy taking advantage of this feature [15], to achieve online control of task execution time instead of relying exclusively on the offline guarantees provided by timing analysis. Execution time control also allows advanced scheduling features such as execution time servers for soft sporadic tasks [1].

Ada has supported execution time control for tasks since the 2005 revision of the language standard [8]. Common for most implementations was that they charged the running task the execution time of interrupt handlers. This causes inaccuracy in the execution time measurement of tasks, and was raised as an issue when the Ada 2005 real-time features were evaluated [14]. Furthermore, as interrupt handling has higher priority than normal task execution, deadlines may be lost in the case of an unexpected high rate of interrupts, either due to a design or analysis fault, or an error in hardware generating interrupt bursts. The lack of proper protection against unexpectedly high rates of interrupt occurrences, motivated the development of execution time control for interrupt handling similar to that for tasks.

When Ada 2005 execution time control was developed for the AVR32 version of the GNAT Ravenscar bare-board run-time environment (GNATforAVR32) [3], separate execution time measurement and timers for each interrupt priority was included [5]. This improved the accuracy of execution time measurement and allowed execution time control for interrupts. At IRTAW-14, these new features were proposed added to the next revision of Ada [4]. Another proposal presented at the workshop measured the combined execution time of interrupt handling [11]. Following the recommendations of the workshop [10,13], the new Ada 2012 standard [9] now includes execution time measurement for all interrupt handling combined and for separate interrupts. These new features were implemented on GNATforAVR32 in addition to non-standard timers for separate interrupts [6]. To show the usefulness of these timers the object-oriented real-time framework was extended with execution time servers for interrupts [6]. Low additional overhead for interrupt handling is important for these features to be acceptable in real-time systems. Therefore the Time Management Unit (TMU), a hardware timer specialized for execution time control was designed [7], and later implemented.
for the AVR32 architecture [12]. Testing with GNATforAVR32 showed that the TMU reduced the overhead of execution time control significantly [7].

This paper is organized as follows: Section 2 describes the additions needed to the Ada 2012 standard for supporting interrupt timers; Section 3 gives a summary of the two implementations for GNATforAVR32 and test results for these; Section 4 describes how the object-oriented real-time framework can be extended to use interrupt timers, and an example application is given; Section 5 discusses the usefulness of interrupt timer features, the API changes, its possible implementation and overhead; and finally Section 6 gives the conclusions.

2 API for interrupt timers

The package Ada.Execution_Time defines the type CPU_Time and the function Clock for execution time measurement of tasks [9]. The execution time of a task is defined as the time spent by the system executing that task, including the time spent executing run-time or system services on its behalf [9]. With Ada 2012 the ability to account for the total or separate execution time of interrupts handlers was introduced. The additions for this feature are shown in Listing 1. Here the function Clock_For_Interrupts returns the total execution time of interrupt handlers since system start-up. The child package Interrupts is new for Ada 2012, and has a function Clock that returns the execution time spent handling the given Interrupt_Id since start-up. The initialization procedure for timers checks if the object is of the tagged type Interrupt_Timer, and uses a different internal clock depending on this. Interrupt timers are used in the exact same way as task timers.

3 Implementation on GNATforAVR32

Two implementations of execution time control exists for GNATforAVR32: a standard implementation using the COUNT / COMPARE registers of the AVR32 [6], and a hardware accelerated implementation using the
Listing 2: Proposed interrupt timer specification

package Ada.Execution_Time.Interrupts.Timers is


private
...
end Ada.Execution_Time.Interrupts.Timers;

Time Management Unit (TMU) [7]. The two implementations share the base design, and only differ in the low-level code interfacing to the hardware timers.

This design takes advantage of the similarities between the real-time clock (RTC) and execution time clocks (ETCs), by having a single implementation of these two clocks in the internal timing package of the run-time environment. This package represents time as a 64-bit modular integer and defines public routines for clock and alarm operations. It also has procedures used internally by the run-time environment for changing the active execution time clock. After initialization there are two active clocks: the RTC that is always active and the ETC that points to the clock of the running thread, that of the interrupt being handled or to the idle clock. The ETC is changed as a result of a context switch, interrupt handling, or system idling. Interrupt clocks are activated by a call from the low-level interrupt handler prior to calling the interrupt handler as shown in Figure 1, and the previous ETC is stored on a stack. After the interrupt has been handled the interrupted clock is popped from the stack and reactivated. Interrupts can be nested.

Each clock has a queue of pending alarms managed as a linked list and sorted in ascending order after timeout. To avoid the special condition of an empty queue, there is a sentinel alarm with timeout at Time'Last that is always present at the end of the queue. The hardware timer is reprogrammed if the first alarm is changed as a result of an alarm being set or cancelled. The alarm type is used both for timing events and execution time timers, and is also used internally for task wake-up.

3.1 Standard implementation

The 32-bit COUNT / COMPARE system registers of the AVR32 are used both for the RTC and execution time clocks in the implementation without TMU. The COUNT register is reset to zero at system start-up and is incremented by one every CPU clock cycle. The COMPARE interrupt is triggered when COUNT equals COMPARE. The use of the hardware timer is tick-less and does not require a periodic clock overflow interrupt. Instead COUNT is reset when the ETC is changed, and the base time of the RTC and the old ETC is incremented with the previous COUNT. By doing this the same hardware timer may be used for both the RTC and the ETC. The elapsed time of a clock \( t \) since the epoch is computed from the base time \( b \) and the COUNT register value:

\[
  t = \begin{cases} 
    b + \text{COUNT} & \text{if clock is active} \\
    b & \text{else}
  \end{cases}
\]

The COMPARE register is adjusted after updating ETC and after the first alarm of an active clock is changed. The COMPARE value is the shortest remaining time until timeout \( T \) for the RTC and ETC. However, this value is never greater than the maximal COMPARE value \( C_M \) to avoid COUNT from overflowing. The interrupt will thus always be pending when COUNT is greater than this maximal value, and COUNT is reset when it is handled, preventing overflow.
The COMPARE interrupt handler has the highest interrupt priority. This handler removes and clears all expired alarms for the RTC and the previous ETC, and call their alarm handlers. At this point the active ETC is that of the COMPARE interrupt itself, for which no alarms are allowed, so only the interrupt ETC on top of the stack or the RTC may be the cause of the interrupt. As all alarms are handled, there is no need to check which of these caused the interrupt.

### 3.2 TMU implementation

The TMU is designed as a memory-mapped device accessible through a high-speed bus. It has a 64-bit COUNT register that is incremented on every positive edge of the clock signal. If $\text{COUNT} \geq \text{COMPARE}$ an interrupt signal is asserted. In order to atomically swap a new set of $\text{COUNT} / \text{COMPARE}$ values with the current, two swap registers are provided. The registers are swapped when the final word of the swap registers is written, and the previous values of $\text{COUNT}$ and $\text{COMPARE}$ can be read back. The swap registers allow for simple and efficient change of execution time clocks.

Some changes to the standard implementation were needed to use two hardware timers for the different clock types. The function for reading the elapsed time is updated to use the TMU if the given clock is the ETC. and the procedure for updating COMPARE updates the correct hardware timer if the given clock is active. The COMPARE interrupt handler now only handles alarms for the RTC, and a new interrupt handler for the TMU is added, handling ETC alarms. Also the context switch routine now changes the active ETC directly, using only a few AVR32 assembler instructions due to the careful design of the TMU memory interface.

### 3.3 Test results

Test results are given for the implementation with and without the TMU [6, 7], to show the interrupt handler overhead caused by implementing execution time control for interrupts as seen in Figure 1. Testing of the TMU was done by simulation as it has not yet been included in a produced UC3 chip. However, since the syntheziable RTL code of the UC3 was used the results are the same as if obtained on hardware.

The test program uses the 16-bit Timer / Counter (TC) peripheral unit to generate interrupts at regular intervals each time its counter is reset. The counter value is read by the interrupt handler and stored in memory. This provides a good estimate of the time from the interrupt line is asserted to the interrupt handler is called. Due to the deterministic nature of the UC3 microcontroller and the simplicity of the test program, all recorded samples were of the exact same value for both implementations. As seen from Table 1, the overhead caused by execution time control is 120 clock cycles for the standard implementation and 90 cycles with the TMU. The reduction when using the TMU is 30 cycles or 25% of the overhead.
Table 1: Measured interrupt overhead.

<table>
<thead>
<tr>
<th>ETC implementation</th>
<th>Overhead in CPU cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard</td>
<td>324</td>
</tr>
<tr>
<td>With TMU</td>
<td>294</td>
</tr>
<tr>
<td>No ETC</td>
<td>204</td>
</tr>
</tbody>
</table>

Listing 3: Definition of interrupt controller

```ada
package Interrupt Controllers is

  type Interrupt Controller is limited interface;

  procedure Enable
  (C : in out Interrupt Controller;
   I : Interrupt_ID) is abstract;

  procedure Disable
  (C : in out Interrupt Controller;
   I : Interrupt_ID) is abstract;

  function Supported
  (C : Interrupt_Controller;
   I : Interrupt_ID) return Boolean is abstract;

  type Any Interrupt Controller is access all Interrupt_Controller'Class;

  Unsupported Interrupt : exception;

end Interrupt Controllers;
```

4 Use of interrupt timers

To ease development of real-time applications an object-oriented framework has been developed by several contributors in the Ada community [2]. By integrating the Interrupt_Timer into this framework, it is also possible to control the execution time spent on interrupt handling and thereby prevent deadlines being lost due to bursts of interrupts. The framework components related to interrupt handling can be separated into three parts: (1) the interface Interrupt Controller used to control hardware interrupt generation; (2) the protected interface Interrupt_Server used to control the execution time spent handling a given Interrupt_ID in accordance with some policy; (3) the protected interrupt handlers, the framework provides the release mechanism Sporadic Interrupt to release tasks as a result of an interrupt.

4.1 Interrupt controller

Interrupts cause the normal execution of tasks to be paused and a handler to be executed, either as a result of an asynchronous hardware interrupt line being asserted or a synchronous software interrupt being triggered. An interrupt is said to occur, and an occurrence is pending in the time between its generation and its delivery to the system, in the form of the appropriate handler being called. It depends on the hardware whether a generated interrupt occurrence is lost if another of the same type is already pending. Hardware interrupts may be generated by components of the computer system such as peripheral units, or by external sources. Often the computer architecture has an interrupt controller that multiplexes and groups interrupt lines, and
Listing 4: Interrupt server interface

```ada
package Interrupt_Servers is

  type Interrupt_Server_Parameters is tagged record
    Controller : Any_Interrupt_Controller;
    Budget : Time_Span;
  end record;

  type Interrupt_Server is protected interface;

  procedure Initialize (S : in out Interrupt_Server) is abstract;

  type Any_Interrupt_Server is access all Interrupt_Server;

end Interrupt_Servers;
```

triggers the interrupt handling on the processor. There may also be several interrupt levels, where interrupts may be interrupted by others of a higher level. With the exception of non-maskable interrupts, the delivery of interrupts may be blocked by the use of masks. Whether a blocked interrupt remains pending or is lost depends on the architecture and hardware.

The interface Interrupt_Controller is defined as shown in Listing 3. The interface will typically be implemented by a peripheral driver. Depending on the peripheral it may control one or more interrupts. Use of the interface is very straight-forward: Enable enables the generation of given Interrupt_ID and Disable disables it. The function Supported indicates if the controller supports the interrupt, if other operations of a controller is called with an unsupported interrupt the Unsupported_Interrupt exception will be raised.

### 4.2 Interrupt servers

The interface Interrupt_Server shown in Listing 4, uses Interrupt_Controller to control the execution time spent handling a given interrupt according to a policy by enabling and disabling its generation. The tagged type Interrupt_Server_Parameters is used to pass the controller and the execution time budget to implementations of the interface.

The protected object Deferrable_Interrupt_Server shown in Listing 5, implements this interface following the deferrable server policy. This allows us to model the execution time spent handling the given interrupt as a periodic task with a given period and budget. The type Deferrable_Server_Parameters defines the additional parameters needed by the server, in this case the replenishing period of the execution time budget. Notice that the Interrupt_ID is given as a separate discriminant, this is needed to declare the timer statically in the protected object. Internally the deferrable server has a timing event used to call the procedure Replenish periodically with the period given as parameter. The procedure sets the execution time budget for the interrupt using the interrupt timer, and enables the interrupt if necessary. The first call to Replenish is at the system epoch, and will enable the generation of the interrupt. The procedure Overrun is called when the execution time budget is exceeded and disables the generation of the interrupt.

### 4.3 Example application

Our example application has a real-time task implemented by a tagged type inheriting Periodic_Task of the real-time framework. The task has period 10 ms and a 5 ms budget, and we use the periodic release
Listing 5: Deferrable interrupt server

```ada
package Interrupt_Servers.Deferrable is

  type Deferrable_Server_Parameters is new Interrupt_Server_Parameters with
    record
      Period : Time_Span;
    end record;

  protected type Deferrable_Interrupt_Server (I : Interrupt_ID;
    Param : access Deferrable_Server_Parameters) is new Interrupt_Server with

    procedure Initialize;

    pragma Priority (Any_Priority'Last);

private

    procedure Replenish (Event : in out Timing_Event);
    procedure Overrun (TM : in out Timer);

    Replenish_Event : Timing_Event;
    Execution_Timer : Interrupt_Timer (I);
    Next : Time;
    Disabled : Boolean := True;

end Deferrable_Interrupt_Server;

end Interrupt_Servers.Deferrable;
```

mechanism with overrun and deadline miss detection. For each release the task simply busy waits 75% of its budget. In addition the application receives data from the PC through the USART line. The tagged type USART_Controller implements Interrupt_Controller and is used to setup, enable and disable the RX interrupt of the USART. A protected object with the USART interrupt handler counts the number of characters received.

The configured baud rate of the USART line is a far higher than what the system can handle using interrupts. However, the intended usage is that characters are typed one-by-one to the serial line by the user, and therefore will be limited to a few characters per second. Since we do not fully trust this limitation to be respected, a deferrable interrupt server is included to control the execution time spent handling receive USART interrupt. We let the server have a replenishing period of 10 ms, the same period as the real-time task, and a budget of 1 ms. Hence, the total utilization is 60% which is known to be safe using RMA. The parts of the application related to interrupt handling are shown in Listing 6.

In order to test the interrupt execution time control the USART line was flooded with data. It was observed that the USART interrupt is disabled when the budget is exceeded and re-enabled when it is replenished. During the test the real-time task did not miss any deadline. However, only 40% of the characters sent were successfully received by the system. This loss could be prevented by using USART hardware flow control or buffering, but we want to keep the example application simple. As expected the real-time task misses all its deadlines during the burst when the interrupt server is removed from the system.
Listing 6: Usage of interrupt server

```ada
package body Test is

    USART : aliased USART_Controller (USART_1_Address);

    Param : aliased constant Deferrable_Server_Parameters :=
        (Controller => USART'Access,
        Budget => Milliseconds (1),
        Period => Milliseconds (10));

    USART_Server : Deferrable_Interrupt_Server (USART_1, Param'Access);

protected
    RX_Counter is
        pragma Interrupt_Priority (USART_1_Priority);

        function Get_Count return Natural;

private
    procedure Increment;

        pragma Attach_Handler (Increment, USART_1);

        Count : Natural := 0;

    end RX_Counter;

protected body RX_Counter is ...
end RX_Counter;

begin
    USART.Initialize;
    USART_Server.Initialize;
end Test;
```

5 Discussion

5.1 The usefulness of interrupt timers

The interrupt timer is not a part of the Ada 2012 standard but should in the authors opinion be added to
the next revision for the following reasons. First it provides execution time control for interrupts similar to
that for tasks. If we measure the execution time for interrupts it should also be controllable by means such as
the framework extensions described in this paper. This is important as the execution time for interrupt handling may be very hard to predict, given that interrupts often are generated by external hardware not
directly controllable by the system. Alternatives to interrupt timers are to count the number of interrupts and disable the interrupt if the count gets to high, or to poll the execution time of the interrupt after the handler is called and disable the interrupt if the budget is exceeded. These solutions are less precise and also less efficient than using interrupt timers.

As seen from the brief description of the AVR32forGNAT implementation of execution time control, there is no additional code needed for interrupt timers as the same clock and alarm type is used for all timing functionality. This may not be done for all other implementations, but it seems reasonable to assume that at least task and interrupt clocks are implemented with the same type. The described standard implementation has a modest overhead caused by execution time control before calling interrupt handlers, and when using the TMU this overhead is smaller still. The additional overhead seems justified by the additional security provided by interrupt timers. For other hardware the of overhead of reprogramming the hardware timer before calling the interrupt handler may be higher, in which case it could be hard to implement interrupt timers. Therefore it would be beneficial to see implementations of this feature for other run-time environment and
hardware combinations.

5.2 Additions to the API

As seen from Listing 2 the new package for interrupt timers simply defines a tagged type `Interrupt_Timer` that inherits `Timer` and its operations. It is assumed that implementations of execution time control for interrupts will use the same underlying type for task and interrupt timers, as is the case for the GNATforAVR32 implementation. Therefore no additional code other than a check for the tagged type in the initialization code for timers is needed.

One thing to remember is that an `Interrupt_Timer` can be viewed as an ordinary `Timer` with a null task id. This can be a source of errors as task timers are not allowed to have a null task id in Ada 2012. An alternative is to define an abstract tagged `Root_Timer` type that both `Timer` and `Interrupt_Timer` inherits. The solution is perhaps clearer but also requires more changes to the standard library.

5.3 Framework extensions

Execution time control for interrupts is facilitated by extending the object-oriented real-time framework to provide execution time servers for interrupts. While the execution time servers for tasks control the execution time for a group of tasks released sporadically, the interrupt server controls the execution time spent invoking one interrupt handler sporadically many times. The object-oriented nature of the framework allows us to create servers suitable for different needs. The deferrable server assumes that it is acceptable to ignore interrupts for a while. Another scheme is to reconfigure the system into fail-safe mode in the case of overruns. Since there is no way to cancel the interrupt being handled in Ada, the budget for the deferrable server has to allow for an overrun of one additional handler invocation.

5.4 Example application

For the example application a low rate of interrupts is assumed but cannot be guaranteed. Errors may cause the system to handle more interrupts than budgeted for in the real-time analysis, which again could cause task deadlines to be missed and thereby system failure. The presented extensions to the real-time framework provides an easy way to protect our example application against these situations. By using the deferrable interrupt server of the real-time framework, we can easily set a budget for the interrupt so that our real-time task is guaranteed sufficient execution time to meet its deadline. No deadlines were lost due to burst of interrupts when the application was tested with the deferrable server, while several deadlines were lost during the burst when the server was not used. This gives a good indication that the deferrable interrupt server works as intended.

6 Conclusion

Execution time measurement for interrupt handling is not enough to efficiently protect against unexpectedly high interrupt rates. Therefore non-standard interrupt timers following the same pattern as task timers were added to Ada 2012. It has been shown that interrupt timers can have a low implementation cost and cause only a modest run-time overhead, and their usefulness was demonstated by extending the object-oriented real-time framework with execution time servers for interrupt handling. The example application experienced no deadline misses due to interrupt bursts when using the implemented deferrable interrupt server, something that could not be achieved easily and efficiently without interrupt timers. In the authors opinion these are good reasons to consider interrupt timers for the next revision of Ada. Before standardization of
the feature is decided upon more implementations should be developed for other run-time environments and hardware architectures.

References


Deferred and Atomic Setting of Scheduling Attributes for Ada

Sergio Sáez, Jorge Real, and Alfons Crespo
{ssaez|jorge}@disca.upv.es
Universitat Politècnica de València, Spain

Abstract

Deferred setting of scheduling attributes refers to a single operation that sets a new value for a scheduling attribute of a task at some future time. Although deferred setting of scheduling attributes is possible in Ada 2012, it is in a rather limited way: only deadline or CPU can be changed deferredly, either at a specified time or when the task is released from a suspension object. And only one of those two attributes at a time. Other scheduling attributes such as priority cannot have deferred setting by means of a single operation. This would be a convenient feature to have for schemes such as job partitioning, task splitting, or mode changes. Another issue is the absence of operations for atomically changing several parameters at a time, which would avoid scheduling issues specially on multiprocessors.

In this paper we explore a proposal aimed at correcting these two drawbacks. On one hand, we want to be able to change more attributes, not only deadlines, deferredly or immediately. On the other hand, we want to atomically change (now or later) a set of attributes, thereby avoiding scheduling artifacts that arise from sequentially changing several attributes, specially when the CPU is one of them. Rather than providing a number of library operations for postponing the setting of a variety of scheduling attributes, we propose to encapsulate the scheduling attributes of each task in a single tagged type that can be extended with more attributes for specific applications if needed.

1 Introduction

Scheduling attributes refer to task attributes such as priority, CPU affinity, period, deadline, etc. Deferred setting of scheduling attributes refer to the ability for the programmer to specify that one or more scheduling attributes of a task need be set to a new value but not immediately, but at some point in time in the future, such as the next activation time of the task. One example of deferred setting of a scheduling attribute for a task is the procedure Delay Until Set Deadline from the Ada 2005 standard package Ada.Dispatching.EDF. Similarly, sporadic tasks can also have their deadline changed upon their next activation if they use a suspension object for activation control. This can be achieved by means of Suspend Until True And Set Deadline from package Ada.Synchronous_Task_Control.EDF.

At the 15th IRTAW, Mario Aldea, chairman of the workshop, summarized the discussion initiated around this topic as follows [1]:

A presentation was made on the existing limitations of the current model of setting attributes (priority, deadline and affinity) that can cause undesirable effects when trying to change several of them simultaneously for the same task. There was some discussion about whether these changes could be performed atomically from inside a protected operation. The conclusion was that this is not a valid
approach when changing other task’s attributes. The group sentiment was that a mechanism is required to allow deferred attribute setting for the next dispatching point of a task. Two alternative implementations of the aforementioned mechanism were discussed: using an attributes object or using a set of procedures. It was agreed that this issue needs further investigation, modeling and trial implementations.

From the two alternative implementations mentioned at the end of this quote, we want to propose a model that uses the first approach and encloses the setting of several scheduling parameters in a single operation that can be executed atomically. By doing so, undesirable artifacts are avoided at run time, specially on multiprocessor platforms. A single container object for all relevant scheduling attributes has also advantages if it is tagged, as we will show.

The rest of this paper is structured as follows. Section 2 describes the problem context. Section 3 defines our proposal. Section 4 discusses implementation issues related to the proposal and that need be solved. A use example is given in Section 5. Finally, Section 6 concludes the paper.

2 Context and problem description

The problems we are trying to solve with this proposal were already described in [5]. For convenience, we give here a brief description. The following actions are sources for scheduling decisions and also for potential scheduling issues:

1. Changing a single task’s scheduling parameter.
2. Changing a single scheduling parameter in the future.
3. Changing a set of scheduling parameters, either now or in the future.

By scheduling parameters we mean task parameters that have an impact on how the system schedules that task, including priority, deadline and CPU, and possibly other user-defined attributes.

The first case is solved in Ada by delaying the actual effect of the parameter change until the task’s next dispatching point. The second case involves a parameter change plus the execution of a delay until statement. Ada allows the deferred setting of scheduling attributes for the cases of deadline and CPU by means of subprograms Delay Until And Set Deadline and Delay Until And Set CPU. The third case, however, cannot be cleanly solved in Ada. The case is particularly problematic in the context of multiprocessor platforms, when the CPU attribute is one of the parameters to be changed. In other words, tasks or jobs need to (dynamically) migrate to a different processor. This is the case of multi-moded systems and also in multiprocessor scheduling approaches such as job partitioning and task splitting [2, 3], to give some examples.

Figures 1 and 2 (reproduced from [5]) show how a task can miss its deadline when it tries to simultaneously change both its priority and its target CPU. In both scenarios, the task \( T_0 \) migrates from one CPU to another, but uses a different priority in the target CPU. The expressions \( TX/PY \) in these figures denote task \( X/priority \ Y \).

In Figure 1, task \( T_0 \) misses its deadline while executing the Set_Priority + Set_CPU sequence. The expected execution would be that task \( T_0 \) migrates to CPU1 and preempts \( T_2 \), while \( T_1 \) becomes the highest priority task in CPU0, as shown in the expected execution side of the figure. However, right after \( T_0 \) changes its priority, the task \( T_1 \) has the highest priority on CPU0, preempts \( T_0 \) and therefore impedes it to execute the Set_CPU statement until it is too late (after deadline \( D_0 \)). Figure 2 shows a different situation where the incorrect behavior is caused by the sequence Set_CPU + Set_Priority. In this second case, \( T_0 \) migrates to CPU1 with the wrong priority, hence it can not preempt \( T_2 \) and is dispatched too late to meet its deadline. The only solution to these situations is to provide a mechanism to simultaneously change the priority and the target CPU. Similar issues arise under multiprocessor EDF dispatching with respect to Set_Deadline and Set_CPU.
Although the scenarios shown in Figures 1 and 2 can be solved by encapsulating both Set_CPU and Set_Priority within a protected operation, this cannot be done when the change of priority or deadline, and target CPU is combined with a delay until statement. As shown in the two following code examples, no correct sequence of code can be found using the current multiprocessor support in Ada. Note that these sequences of code are natural ways to implement job partitioning schemes, for setting the CPU where the next job is going to be executed before the current job finishes; and also task splitting, for resetting the original CPU at the end of the job.

```ada
loop
  -- Task code
  ...
  Next_Time := Next_Time + Period;
  Set_Deadline(Next_Time + Relative_Deadline);
  Delay_Until_And_Set_CPU(Next_Time, Next_CPU);
  -- Similar to scenario with Set_Priority + Set_CPU sequence
end loop;

loop
  -- Task code
  ...
  Next_Time := Next_Time + Period;
  Set_CPU(Next_CPU);
  Delay_Until_And_Set_Deadline(Next_Time, Relative_Deadline);
  -- Similar to scenario with Set_CPU + Set_Priority sequence
end loop;
```
3 Proposal

The main ideas behind our proposal are:

- An object of a tagged type contains the relevant scheduling parameters (or attributes) for any given task. Let’s call this type \texttt{Sched\_Params}. In principle, \texttt{Sched\_Params} contains only the CPU and priority of the task.

- The first natural extension to \texttt{Sched\_Params} is to add a field for representing the deadline of a task. This is useful only for tasks scheduled under deadline-based policies such as EDF, hence we propose it as an extension to the root type.

- The \texttt{Sched\_Params} type can also be extended by the user with other parameters that are relevant for a particular application. For example urgency level, offsets, capacity in server tasks, etc.

- The following operations are possible over \texttt{Sched\_Params} objects:
  
  \textbf{Set\_Attribute} \hspace{1em} Sets the new value for a given attribute in the \texttt{Sched\_Params} object. The \texttt{Attribute} part of this setter refers to the attributes priority or CPU for the root type. Derived types may define setters for additional parameters, such as deadline for the \texttt{Sched\_Params} of an EDF task (see later).

  \textbf{Get\_Attribute} \hspace{1em} Obtains the current value of a given attribute from the \texttt{Sched\_Params} object.

  \textbf{Apply\_Sched\_Params} \hspace{1em} As the name indicates, makes the scheduling parameters effective \textit{immediately} and atomically. This procedure can be applied to the currently executing task or to another given task.

  \textbf{Delay\_Until\_And\_Apply\_Sched\_Params} \hspace{1em} This is to delay the task until a given time and atomically apply the set of scheduling parameters defined in a \texttt{Sched\_Params} object.

  \textbf{Suspend\_Until\_True\_And\_Apply\_Sched\_Params} \hspace{1em} For symmetry with the existing procedure \texttt{Suspend\_Until\_True\_And\_Set\_Deadline}, in \texttt{Ada.Synchronous\_Task\_Control.EDF}.

The following listings give the profiles and location (new Ada library packages) for the proposed functionalities. We will first consider the type \texttt{Sched\_Params}, a tagged record that holds a minimal set of scheduling attributes, and can be extended with more attributes if needed. In listing 1 we propose a new library package \texttt{Ada.Scheduling\_Parameters} for the definition of this type.

\begin{verbatim}
Listing 1. Definition of the root type Sched\_Params


package Ada.Scheduling_Parameters is

    type Sched_Params is tagged private;

    procedure Set_Priority (SP: in out Sched_Params; Prio: Any_Priority);
    function Get_Priority (SP: Sched_Params) return Any_Priority;

    procedure Set_CPU (SP : in out Sched_Params; CPU_Nr: CPU_Range);
    function Get_CPU (SP : Sched_Params) return CPU_Range;

end Ada.Scheduling_Parameters;
\end{verbatim}
procedure Apply_Sched_Params (SP: Sched_Params; T_Id: Task_Id := Current_Task);
procedure Delay_Until_And_Apply_Sched_Params (SP: Sched_Params;
Delay_Until_Time: Time);

private
type Sched_Params is
  record
    Prio: Any_Priority := Default_Priority;
    CPU_Nr: CPU_Range := Not_A_Specific_CPU;
  end record;
end Ada.Scheduling_Parameters;

A first extension to the root type Sched_Params will include a deadline parameter, useful for EDF tasks. In listing 2 we propose a new child package of Ada.Scheduling_Parameters to include the new type Sched_Params_EDF, derived from Sched_Params, and setter and getter subprograms for the new deadline parameter. The package also provides new procedures to apply these extended scheduling parameters to EDF tasks. Note that we also include a new scheduling parameter At_Time that we explain below.

Listing 2. Extension of root scheduling parameters for EDF

with Ada.Real_Time; use Ada.Real_Time;

package Ada.Scheduling_Parameters.EDF is

type Sched_Params_EDF is new Sched_Params with private;

procedure Set_Deadline (SP: in out Sched_Params_EDF; D: Time_Span);
function Get_Deadline (SP: Sched_Params_EDF) return Time_Span;

procedure Set_At_Time (SP: in out Sched_Params_EDF; At_Time: Time);
function Get_At_Time (SP: Sched_Params_EDF) return Time;

procedure Apply_Sched_Params (SP: Sched_Params_EDF; T_Id: Task_Id := Current_Task);
procedure Delay_Until_And_Apply_Sched_Params (SP: Sched_Params_EDF;
Delay_Until_Time: Time);

private
type Sched_Params_EDF is new Sched_Params with
  record
    Relative_Deadline: Time_Span := Time_Span_Last;
    At_Time: Time := Time_Last;
  end record;
end Ada.Scheduling_Parameters.EDF;

We propose the deadline parameter to be of the type Time_Span. The semantics of Delay_Until_And_Apply_Sched_Params would be that the new absolute deadline is set for the time Delay_Until_Time plus the relative deadline given in the Sched_Params_EDF object. The absolute deadline is not so obvious in the case of Apply_Sched_Params. It could be the result of adding the relative deadline to the real-time clock value during the execution of Apply_Sched_Params. But that clock value is uncertain. We therefore propose to
include the additional scheduling parameter \texttt{At\_Time}, an absolute time taken as the reference to calculate the next absolute deadline for the task.

We considered the possibility of using such an absolute time reference as an additional, third parameter passed to \texttt{Apply\_Sched\_Params}. What makes that approach unattractive is that the profile for the primitive \texttt{Apply\_Sched\_Params} defined in package \texttt{Ada.Scheduling\_Parameters} would no longer be valid for all cases, since that primitive does not include such parameter. Note that the functionality provided by such \texttt{At\_Time} attribute is not achievable with \texttt{Delay\_Until\_And\_Apply\_Sched\_Params}. For example, we may want to promote a task by shortening its absolute deadline after a certain time of the task’s execution. Hence the task cannot delay until a certain time and then shorten its deadline, since it needs to be executing code meanwhile.

Finally, a link between Ada’s synchronous task control and scheduling parameters would be useful for sporadic tasks whose activation is regulated by a suspension object. This is in line with the existing subprogram \texttt{Suspend\_Until\_True\_And\_Set\_Deadline}, which is limited to setting only the deadline for the next reactivation of a task waiting on a suspension object. In listing 3 we propose a child package \texttt{Ada.Synchronous\_Task\_Control.Scheduling\_Parameters} to contain the new functionality\(^1\).

\textbf{Listing 3. Synchronous task control and scheduling attributes}

\begin{verbatim}
with Ada.Scheduling_Parameters; use Ada.Scheduling_Parameters;
package Ada.Synchronous_Task_Control.Scheduling_Parameters is
    procedure Suspend_Until_True_And_Apply_Sched_Params (  
        S : in out Suspension_Object; -- See new type proposed in section 4.3  
        SP: Sched_Params'Class);
end Ada.Synchronous_Task_Control.Scheduling_Parameters;
\end{verbatim}

Note that \texttt{SP}, the \texttt{Sched\_Params} parameter for this procedure, is class-wide. Hence it can dispatch to root-type, EDF-extended or user-extended scheduling parameters.

\section{Implementation issues}

The operations presented above, \texttt{Apply\_Sched\_Params} and \texttt{Delay\_Until\_And\_Apply\_Sched\_Params}, allow the application to atomically change several task scheduling parameters. The underlying Operating System (OS) has to provide specific support in order to allow the Ada Run-Time Support to implement these operations adequately. Although the required behaviour within the operating system kernel is simple, as it will be shown bellow, this support is not present in any POSIX-like operating system (to the best of our knowledge) including those that add non-portable extensions, such as the Linux kernel.

Any change in one scheduling parameter implies that the operating system removes the implied thread from the current run queue and inserts it again in a (possibly different) run queue in a different position. As one or more system run queues are modified, the system scheduler has to be invoked to determine the new highest priority thread. Furthermore, if the CPU of a thread is changed, then some kind of \textit{Inter-Processor Interrupt} (IPI) has to be sent to inform the affected CPU or CPUs that they have to execute the scheduler. Therefore, the change of a scheduling parameter has to be considered always a thread dispatching point.

If an application needs to change several scheduling parameters of a task at the same time, e.g. its priority and CPU, it has to invoke several system calls to change them. For example, if the underlying OS is the Linux kernel, the application has to invoke \texttt{sched\_setparam} and \texttt{sched\_setaffinity} system calls. Each of these system calls is a thread dispatching point, since they may imply changes in the system run queues. The scheduling artifacts presented in section 2 are due to the sequential execution of these system calls and their corresponding thread dispatching points. Each time the application changes a single scheduling parameter, the scheduler can

\footnote{For clarity, we are using the existing type \texttt{Suspension\_Object} in listing 3. In section 4.3 we will justify why we are proposing a new type of suspension object that implements modification of scheduling parameters.}
dispatch a different thread in one or more CPUs, and therefore, the thread that is changing the parameters can lose the CPU or the thread with the new scheduling parameter can temporarily disturb other running threads. This undesired behaviour could be avoided if the scheduling parameters were changed atomically.

The actions the OS kernel has to perform to support the simultaneous modification of several scheduling parameters are relatively simple:

1. Remove the thread from the run queue where it is currently located.
2. Change all the scheduling parameters specified by the application.
3. Insert the thread in a new task queue according to the new set of scheduling parameters. This queue could be a new priority queue in a new CPU or it could be the timer or mutex queue if the thread has to be suspended.

The main implementation issue is how to offer this kernel functionality to the application. We’ll now explore the POSIX case and see what extensions would be needed.

4.1 POSIX support and proposed extensions

Although the POSIX standard does not provide support to simultaneously changing several scheduling parameters on a running process or thread (other than scheduling policy and priority using the `sched_setscheduler` system call), it provides a similar functionality for establishing all the scheduling parameters for the creation of a new thread. This functionality is offered through the structure `pthread_attr_t` and the following C functions that allow the application to specify the full set of thread attributes before creating it with `pthread_create`.

Listing 4. Thread attributes manipulation functions

```c
pthread_attr_init/destroy // initialize and destroy thread attributes object
pthread_attr_set/getdetachstate // set/get detach state attribute in thread attributes object
pthread_attr_set/getinheritsched // set/get inherit scheduler attribute in thread attributes object
pthread_attr_set/getschedparam // set/get scheduling parameter attributes in thread attributes object
pthread_attr_set/getschedpolicy // set/get scheduling policy attribute in thread attributes object
... // Other attributes not directly related with scheduling
```

In the case of the Linux kernel, a small set of non-portable extensions exist that support setting and getting the CPU affinity of a thread. In addition, Linux also provides the non portable function `pthread_getattr_np` for retrieving the current attributes of an already created thread.

Listing 5. Linux specific non portable extension to thread attributes

```c
pthread_attr_set/getaffinity_np // set/get CPU affinity attribute in thread attributes object
pthread_getattr_np // get attributes of created thread
```

We propose to extend this API with the corresponding `pthread_setattr_np` function to allow the application to specify various scheduling attributes that have to be simultaneously applied to an already created thread. However, in order to support the operations that imply a possible suspension of the thread, i.e. Delay Until - And Apply Sched Params, it is required that the new API offers the possibility of deferring the setting of the attributes until the thread wakes up, either from a delay statement or from a suspension object. We propose the following specification, from the two alternatives given in [4]:
The next section assumes the existence of these non portable functions to show how the Ada Run-Time Support could implement the main operations of the new Sched.Params type.

4.2 Implementation of Sched.Params operations

Based on these new OS functionalities and taking the source code GNAT GPL 2012 as a reference, the implementation of the new proposed operations could be as follows:

```ada
procedure Apply_Sched_Params (SP: Sched_Params; T_Id: Task_Id:= Current_Task) is
    Attributes: aliased pthread_attr_t;
    Result: Interfaces.C.int;
begin
    -- Retrieve the current thread attributes
    Get_Task_Attributes (Attributes'Access, T_Id);
    -- Modify the task attributes
    Set_Attr_Priority(Attributes'Access, SP.Prio);
    Set_Attr_CPU(Attributes'Access, SP.CPU_Nr);
    -- Set the new thread attributes immediately
    Result := pthread_setattr_np (T_Id.Common.LL.Thread, Attributes'Access);
    pragma Assert (Result = 0);
end Apply_Sched_Params;

procedure Delay_Until_And_Apply_Sched_Params
    (SP: Sched_Params;
     Delay_Until_Time: Ada.Real_Time.Time)
is
    Attributes : aliased pthread_attr_t;
    Result : Interfaces.C.int;
begin
    -- Retrieve the current thread attributes
    Get_Task_Attributes (Attributes'Access, T_Id);
    -- Modify the task attributes
    Set_Attr_Priority(Attributes'Access, SP.Prio);
    Set_Attr_CPU(Attributes'Access, SP.CPU_Nr);
    -- Take note of the new thread attributes to be applied on thread wakeup
    Result:= pthread_setattr_on_wakeup_np (Attributes'Access);
    pragma Assert (Result = 0);
    delay until Delay_Until_Time; -- New attributes take effect on wakeup
end Delay_Until_And_Apply_Sched_Params;
```

In order to simplify the implementation, it is supposed that the Ada run-time system will provide procedures to retrieve and manipulate the Attributes type. In the current GNAT GPL 2012, the Attributes type is an opaque type that is manipulated using the POSIX interface only. The procedures used above (i.e. Get_Task_Attributes, Set_Attr_Priority and Set_Attr_CPU), will use these existing POSIX functions and the thread information maintained by the Ada run-time system, to prepare the Attributes object. This object represents the thread scheduling parameters at operating system level.
4.3 Implementation of Suspension Objects

The implementation of Suspend_Until_True_And_Apply_Sched_Params needs a different approach to Delay_Until_And_Set_Sched_Params. In this second case, it is clear in advance when the task will be awakened (at the specified absolute time) and hence have its new parameters applied. But in the case of using a suspension object, the task calling the suspension operation may either go through immediately (if the object’s state is True) or it may have to wait for another task to set the object state to True. In the first case, the attributes need be changed as part of the call to the suspension operation; whereas in the second case, it is the call to Set_True that has the effect of enforcing the new scheduling parameters. So we need to store the task identification and new scheduling parameters to apply them at the proper time.

We therefore propose a new type of suspension object (Suspension_Object_With_Sched_Params) for sporadic tasks that use deferred setting of scheduling attributes. This new type contains, as part of its internal state, two fields to store the scheduling parameters (SP) and the task identification (T_Id).

```ada
type Suspension_Object_With_Sched_Params is record
  -- Boolean that indicates whether the object is open
  State : Boolean;
pragma Atomic (State);

  -- Flag showing if there is a task already suspended on this object
  Waiting : Boolean;

  -- Protection for ensuring mutual exclusion on the Suspension_Object
  L : aliased System.OS_Interface.pthread_mutex_t;

  -- Condition variable used to queue threads until condition is signaled
  CV : aliased System.OS_Interface.pthread_cond_t;

  -- Task suspended within the Suspension Object
  T_Id : Task_Id;

  -- Scheduling Parameters to be applied to the suspended task
  SP : access all Sched_Params’Class;
end record;
```

When a sporadic task invokes Suspend_Until_True_And_Apply_Sched_Params with a new set of scheduling parameters, if the suspension object state is true, the scheduling parameters are applied immediately within the suspension object. Then the sporadic task continues with its next activation using the new scheduling parameters.

If the suspension object state is false, the sporadic task will be suspended until the state becomes true. In this case, Suspend_Until_True stores the task identifier of the sporadic task and scheduling parameters for its next activation. The task that invokes the Set_True procedure will apply the new scheduling parameters to the sporadic task before signaling the conditional variable within the suspension object, and therefore, before waking up the sporadic task. When the sporadic task wakes up, it already has its new scheduling parameters.

The Suspension_Object_With_Sched_Params type will also offer the Suspend_Until_True operation, that allows a task to be suspended until the suspension object state becomes true, but without modifying its scheduling parameters.

Based on the source code from GNAT GPL 2012, the new suspension object operations could be implemented as follows:
procedure Suspend_Until_True_And_Apply_Sched_Params
(S : in out Suspension_Object_With_Sched_Params;
SP : access all Sched_Params’Class)
is
Result : Interfaces.C.int;
begin
SSL.Abort_Defer.all;
Result := pthread_mutex_lock (S.L’Access);
pragma Assert (Result = 0);
if S.Waiting then
 Result := pthread_mutex_unlock (S.L’Access);
pragma Assert (Result = 0);
SSL.Abort_Undefer.all;
raise Program_Error;
else
 if S.State then
 S.State := False;
 SP.Apply_Sched_Params;
 else
 S.Waiting := True;
 S.T_Id := Current_Task;
 S.SP := SP;
 loop
 Result := pthread_cond_wait (S.CV’Access, S.L’Access);
pragma Assert (Result = 0 or else Result = EINTR);
exit when not S.Waiting;
end loop;
end if;
Result := pthread_mutex_unlock (S.L’Access);
pragma Assert (Result = 0);
SSL.Abort_Undefer.all;
end if;
end Suspend_Until_True;

procedure Set_True (S : in out Suspension_Object_With_Sched_Params)
is
Result : Interfaces.C.int;
begin
SSL.Abort_Defer.all;
Result := pthread_mutex_lock (S.L’Access);
pragma Assert (Result = 0);
if S.Waiting then
 S.Waiting := False;
 S.State := False;
if S.SP /= null then
 S.SP.Apply_Sched_Params(S.T_Id);
end if;
 Result := pthread_cond_signal (S.CV’Access);
pragma Assert (Result = 0);
else
 S.State := True;
end if;
Result := pthread_mutex_unlock (S.L’Access);
pragma Assert (Result = 0);
SSL.Abort_Undefer.all;
end Set_True;
5 Use example

This section shows a brief example where this new functionality is used to implement a task subject to job partitioning. With this scheduling scheme, a periodic task could decide to use a different CPU and priority for each job (i.e., each activation of the task). In the example below, this design decision is represented by a cyclic plan of scheduling parameters, `Params_List`. At the end of each job execution, the task retrieves the next set of scheduling parameters from its plan, and calls `Delay_Until_And_Apply_Sched_Params`. This allows the task to change the scheduling parameters for its next job atomically and avoids the scheduling artifacts mentioned in section 2.

Listing 7. Periodic task with job partitioning based on delay until

```ada
task body Periodic_With_Job_Partitioning is
  type List_Range is mod N;
  -- List of scheduling parameters, decided at design time
  Params_List: array (List_Range) of Sched_Params := (...);
  Params_Iter: List_Range := List_Range’First;
  Next_Params: Sched_Params;
  Period: Time_Span := ...;
begin
  Task_Initialize;

  -- First job parameters
  Next_Params := Param_List(Param_Iter);
  -- Scheduling parameters for the first activation
  Next_Params.Apply_Sched_Params;
  loop
    Task_Main_Loop_Actions;

    -- Next job preparation
    Params_Iter := Params_Iter’Succ;
    Next_Params := Params_List(Params_Iter);
    Next_Release := Next_Release + Period;

    -- Suspends the task until the next job activation
    Delay_Until_And_Apply_Sched_Params(Next_Params, Next_Release);
    -- Next job will wake up with the next scheduling parameters applied
  end loop;
end Periodic_With_Job_Partitioning;
```

6 Conclusion

The deferred, atomic setting of a set of scheduling attributes is a useful feature that is currently absent in Ada. It provides a clear semantics and avoids scheduling artifacts and wrong effects derived from sequentially applying one attribute after another, specially when the underlying hardware is a multiprocessor platform. In this paper we have proposed changes in the direction of including this feature in Ada. All changes proposed are additions to the standard library, with no modification proposed to any other part of the language. The changes are also user-extensible since they are based on the use of tagged types. Perhaps the
A proposed major change is a new type of suspension object to give support to deferred, atomic setting of attributes of sporadic tasks. The fact that a sporadic task may have its parameters changed either immediately upon calling $\text{Suspend\_Until\_True}$ (when the object’s state is True) or deferredly when another task calls $\text{Set\_True}$ (in case the object state was False when the task called $\text{Suspend\_Until\_True}$) makes it necessary to provide a different type of suspension object, augmented with the capability of setting the waiting task’s parameters.

These proposed extensions are mainly directed towards multiprocessor platforms, since the intended semantics is feasibly implementable on single-processors in Ada. However, some single-processor scheduling approaches could benefit from the changes proposed here, if only aesthetically (e.g., dual-priority scheduling, existing schemes for control tasks structured as Initial-Mandatory-Optional-Final, etc).

We want to finally note a gracious side effect of this proposal. With the proposed set of procedures, there would be strictly no need to use the existing procedure $\text{Delay\_Until\_And\_Set\_CPU}$ from package $\text{System.Multi-processors.Dispatching\_Domains}$. The nice effect is that, if that procedure did not exist, then there would be no dependence with $\text{Ada.Real\_Time}$, and therefore $\text{System.Multi-processors.Dispatching\_Domains}$ could be preelaborable. But, unfortunately, changing the standard for this reason would introduce backward incompatibility.

References


An extended Ravenscar profile for execution time control

Kristoffer Nyborg Gregertsen
Department of Applied Cybernetics
SINTEF ICT, Trondheim, Norway
kristoffer.gregertsen@sintef.no

Abstract
This position paper argues that an “extended Ravenscar” profile supporting execution time control should be specified. The new profile should add sufficient tasking features to handle execution time overruns, while keeping most of the Ravenscar properties such as a static task set and efficient run-time environments.

1 Background
The Ravenscar profile is a restricted sub-set of the Ada tasking model [12]. It is designed to provide a static and deterministic tasking environment allowing static scheduling analysis [1]. This makes it very well suited for high-integrity fields such as aviation, defense and space, that historically used to rely on the cyclic-executive to achieve deterministic and analyzable systems. Another positive effect of the Ravenscar profile is that its restricted tasking model allows small and efficient run-time environment implementations.

Static scheduling analysis for the Ravenscar profile, using established techniques such as rate monotonic or response time analysis, requires that the worst-case execution time (WCET) of the tasks is known in advance. The WCET is computed by applying timing analysis on the task, either by static analysis of the source code and the compiled executable using an abstract model of the computer architecture, or by measuring the execution time of the task or parts of the task when executed on the targeted hardware [15]. Finding the WCET may be hard for all but the most trivial tasks as a large space of input data and initial conditions need to be considered. Furthermore, timing analysis is made magnitudes harder by performance enhancing techniques such as multi-level cache, deep pipelines with shared execution units, and more. These techniques introduce timing anomalies that can be counter-intuitive and hard to predict [15]. Another challenge is multi-core computer architectures, where the different cores can affect each other through use of the coherent cache hierarchy and other shared resources. While state-of-the-art techniques and tools in timing analysis allows these problems to be solved [15], they are available for a limited set of computer architectures and are often costly and timing consuming, limiting their use to the above mentioned high-integrity fields with large budgets and long development times.

It is also important to remember that for safety to be guaranteed by the timing analysis, the computed WCET has to be an upper limit for the real WCET. The overestimation of the WCET for state-of-the-art timing analysis tools is reported to be in the range of 30-50% [15]. Also the real WCET will often be very pessimistic compared to the average execution time as it assumes that many performance enhancing techniques fail. This means that if a correctly computed WCET is used for the scheduling analysis, it can be guaranteed that no tasks will ever miss their deadline, but the systems computational resources will be wasted if there are no background tasks to use the remaining CPU time. This can lead to computer systems that are over-dimensioned compared to their actual load.
An alternative approach to the static analysis is to apply *execution time control* to achieve online control of task execution time. Execution time control combines a *mechanism* that measures the total time a task has been executed and calls a handler when a timeout value is reached, and a scheduling *policy* that takes advantage of this mechanism to prevent deadline misses in the case of overruns [14]. In practice this means that simple timing analysis techniques can be applied, for instance based on measuring, and that the safety will be provided by the execution time control instead of an guaranteed bound for the WCET. This can both reduce the development time, and also allow better usage of the computational resources. Execution time control also allows advanced features such as execution time servers for soft sporadic tasks [2], and state-of-the-art scheduling techniques for improved utilization of multi-processors.

The Ravenscar profile provides a limited and easy-to-understand set of tasking features sufficient for most real-time systems, while requiring only a limited run-time environment that can fit on small embedded computer systems [8]. This makes it appealing for a wider range of embedded system projects than just those in the traditional safety-critical fields. However, the Ravenscar profile does not allow execution time control by banning execution time timers. The reason for this may be that the deterministic nature of Ravenscar does not allow the tasking mechanisms needed for overrun handling. Instead, applications would have to rely on polling [9] or reconfiguration [5] to handle overruns. This motivates the definition of an “extended Ravenscar” profile supporting execution time control.

In the following there is a brief description of some overrun handling techniques and the Ravenscar profile, before a discussion of what features could be included in an extended profile to facilitate execution time control. Finally a conclusion is given.

## 2 Overrun handling policies

Different overrun handling policies exist to prevent deadline miss and system failure in the case of a task execution time overrun [6]:

- **Handled**: The overrun is recorded and the task allowed to continue executing. This may be used for testing, for critical tasks that must be allowed to finish their work, or in cases where an occasional overrun is acceptable.

- **Stopped**: The task instance is stopped in the case of an overrun by means such as the select-then-abort statement, and is not repeated. The task starts executing normally the next time it is released.

- **Imprecise**: The task consists of a mandatory part that is usually short, and an optional part that refines the result. The optional part is aborted in case of an overrun. This scheme allows fixed priority for tasks executing algorithms where predicting the WCET is hard.

- **Lowered**: The task is lowered to background priority in the case of an overrun to avoid deadline miss for tasks with lower priority. The task may finish if there is sufficient CPU resources. The task priority is restored at the next release.

Another alternative is to reconfigure the system into a safe-state when an execution time overrun is detected. This is already possible with the tasking features provided by the Ravenscar profile given that execution time timers are provided [5].

## 3 The Ravenscar profile

The Ravenscar profile is equivalent to the set of pragmas shown in Listing 1 [12]. As seen it specifies FIFO within priorities and ceiling locking as scheduling policy. The restrictions limit the tasking features allowed...
Listing 1: Ravenscar definition

```ada
pragma Task_Dispatching_Policy (FIFO_Within_Priorities);
pragma Locking_Policy (Ceiling_Locking);
pragma Detect_Blocking;
pragma Restrictions (
  No_Abort_Statements,
  No_Dynamic_Attachment,
  No_Dynamic_Priorities,
  No_Implicit_Heap_Allocations,
  No_Local_Protected_Objects,
  No_Local_Timing_Events,
  No_Protected_Type_Allocators,
  No_Relative_Delay,
  No_Requeue_Statements,
  No_Select_Statements,
  No_Specific_Termination_Handlers,
  No_Task_Allocators,
  No_Task_Hierarchy,
  No_Task_Termination,
  Simple_Barriers,
  Max_Entry_Queue_Length => 1,
  Max_Protected_Entries => 1,
  Max_Task_Entries => 0,
  No_Dependence => Ada.Asynchronous_Task_Control,
  No_Dependence => Ada.Calendar,
  No_Dependence => Ada.Execution_Time.Group_Budgets,
  No_Dependence => Ada.Execution_Time.Timers,
  No_Dependence => Ada.Task_Attributes,
  No_Dependence => System.Multiprocessors.Dispatching_Domains);
```

to create a static and deterministic environment. The restrictions most relevant for this paper are those that limit task overrun handling:

- **No execution time timers** means that it is not possible to be notified about task overruns in the first place.

- **No dynamic priorities** means that the base priority of tasks cannot be reduced.

- **No abort statement** means that a task cannot be aborted.

- **No asynchronous task control** means that a task cannot be held until later resumed.

- **No select statements** means that the current task instance cannot be aborted by the select-then-abort construct.

Also, the restriction **no group budgets** means that execution time servers cannot be used. Therefore sporadic tasks must have lower priority than the hard tasks, increasing their average response time.

4 Possible extensions to Ravenscar

This sections discusses what restrictions of the Ravenscar profile should be lifted for the extended profile in order to allow execution time control, and what the possible implications of doing so are.
4.1 Timers

In order for execution time control to be feasible the restriction forbidding execution time timers must be lifted, as this mechanism is needed to handle task overruns efficiently. The only alternative is to poll the tasks execution time clock, which is not desirable. The ability to limit the number of timers for a given task should be used, setting this limit to at most one timer per task [5]. This will allow for more efficient runtime environment implementations and also reduce the possible usage pattern complexity of this feature. By allowing one timer for each task the handled overrun handling policy is possible. If interrupt timers are added to Ada, as suggested in another paper by the author [7], they should also be allowed in this new profile to protect against unexpected high interrupt rates. Timers and group budgets are also needed for implementing execution time servers.

The implementation cost and run-time overhead of timers is moderate, and can be further reduced by using specialized hardware timers [11]. The feature has already been included in otherwise Ravenscar compliant run-time environments [5, 10, 13]. Group budgets comes at an additional cost, as another check has to be made when reprogramming the timers. However, this cost is expected to be small and acceptable to allow execution time server for groups of tasks.

4.2 Dynamic priority

Dynamic priority allows a tasks base priority to be changed at run-time. This allows the lowered overrun handling policy, and can also be used for execution time servers such as the deferrable and sporadic servers. It can also be used for dual priority scheduling schemes for increased processor utilization [4]. In this the dynamic priority feature is very powerful and flexible. However, this flexibility is a two-edged sword and could be misused reducing the determinism of the system. A more restricted alternative would be to not allow dynamic priorities, but define a new package named Ada.Task_Backgrounding or similar that sets tasks to a defined background priority, and allows them to be restored to their static base priority later. This would allow the same benefits for execution time control without adding the same possibility of misuse.

4.3 Asynchronous task control and abort

Asynchronous control allows a task to be halted by reducing its priority to below the idle task. When the task is resumed it will continue from where it was halted. This is useful for execution time servers when dynamic priority is not supported, but it is not as good for the lowered overrun handling policy as the task is not allowed to continue if sufficient resources are available. When the task is resumed at its next release it would have to execute the remaining parts of the previous instance and the current instance.

However, asynchronous control can be used for halting a task forever in the case of overrun, and releasing an alternative simpler task that is known to require less execution time. A task could also be aborted for this reconfiguration scheme, but the benefits of aborting a task instead of halting it forever are limited in an environment that does not allow dynamic task creation, as the tasks memory resources cannot be reclaimed by the system. Therefore the abort statement is not deemed needed. Also the Ravenscar profile forbids task termination.

An alternative approach for this reconfiguration scheme could be to define a new package that allows tasks to be reset to their initial state. The task would query its controller (i.e. the protected object with its overrun handler) to see if was reset and what configuration it should enter. In this way a task could be reset and start executing with a simpler algorithm in the case of overrun. This reconfiguration would allow reuse of the tasks resources saving valuable memory.
Table 1: Summary of possible extensions.

<table>
<thead>
<tr>
<th>Ravenscar restriction</th>
<th>In extended profile?</th>
</tr>
</thead>
<tbody>
<tr>
<td>No timers</td>
<td>Yes, must have</td>
</tr>
<tr>
<td>No asynchronous task control</td>
<td>Yes, should have</td>
</tr>
<tr>
<td>No dynamic priorities</td>
<td>Maybe</td>
</tr>
<tr>
<td>No select statements</td>
<td>Maybe</td>
</tr>
<tr>
<td>No abort statement</td>
<td>No, not needed</td>
</tr>
</tbody>
</table>

4.4 The select statement

The asynchronous transfer of control (ATC) using the select statement allows a block of code to be aborted in the case of a trigger event. This is a prerequisite for the stopped overrun handling policy as it is used to abort the current task instance, and the imprecise policy to abort the optional part. This feature is used for overrun handling in the object-oriented real-time framework [3]. The only alternative for stopping the task instance is the task itself polling a flag set by the overrun handler [9]. Polling is not an elegant or efficient solution, so ATC would be a valuable addition for execution time control.

However, this feature is expected to come at a rather high implementation cost for the run-time environment, and it can no longer be assumed that blocks of code will run until completion. Also, by allowing select statements the timed entry call would also be allowed unless explicitly forbidden by a new restriction pragma. The selective accept statement is not relevant, as tasks are not allowed to have entries by the Ravenscar profile. In essence, ATC seems to be one of the most useful constructs for execution time control, but is also the one that probably comes with the highest implementation cost for the run-time environment.

5 Conclusion

The main recommendation of this paper is that an extended Ravenscar profile with support for execution time control should be defined. Table 1 summarizes the opinions of the author regarding these extensions. An extended profile would allow embedded projects that cannot perform a full timing analysis to have most of the benefits of the Ravenscar profile, while allowing them to be protected against the consequences of execution time overruns by execution time control, and also have useful real-time constructs such as execution time servers for sporadic tasks. Such an extended Ravenscar profile could open up new fields for Ada in embedded systems outside of its traditional stronghold in high-integrity fields.

5.1 Further work

Whether to include dynamic priorities and asynchronous transfer of control is left as an open question as seen in Table 1. Dynamic priorities are not expected to have a large additional implementation cost and overhead. Asynchronous transfer of control will probably have a higher implementation cost and overhead but is needed for the stopped and imprecise policies. Further discussion and example implementations are needed to evaluate the cost/benefit ratio of these features.

References


Session Summary: Parallel and Multicore Systems
Chair: Luis Miguel Pinho
Rapporteurs: Stephen Michell and Brad Moore

1 Introduction
The majority of the session was based on the position papers submitted by Michell, Moore and Pinho. An addition paper[1] had also been distributed to the workshop participants as necessary reading to understand the papers submitted to the workshop, as it deals with the general model of fine-grained concurrency proposed by the authors. The second part of the session was mostly based on the position paper submitted by Zamorano and de la Puente.

2 Discussions – Fine-grained Parallelism for Ada
Papers:
Burns - “Parallel Ada – A Requirement for Ada 2020” [1]

Miguel Pinho opened the discussion in the morning, and laid out the format of the discussion. Alan Burns had proposed to not have a separate discussion on his position paper, saying that it served as a placeholder to generate discussion, but that the material that it covered was also present in the other papers.

Stephen Michell then continued to present the basic model proposed for supporting augmenting Ada to support parallel computation models. The motivation for a parallel solution in Ada is two-fold, in response to changes in computer chip architectures currently available, as well as future directions. The first important change to note, is that Moore's law no longer applies. We can no longer rely on faster CPU clock speeds to absorb increasing complexity and demands of computer applications. The second factor is related and has to do with how chip manufacturers are responding to practical limits in CPU clock speed, by increasing the number of cores in the computer chip.

The term Parallelism OPPortunity (POP) was introduced which was invented for the IRTAW papers to represent the locations in the programmer's code that are suitable for parallel execution.

The goal for the general model is to allow for POP's to be explicitly identified in the programmer's code. To illustrate the use and need for POP's, the example of a parallel loop was given, which seemed like a good choice given that loops are very prevalent in application code, and that applying a divide and conquer strategy is perhaps easier to understand than perhaps a recursive subprogram example.

There was significant discussion about the first example. Some attendees had the impression that the work being presented only dealt with parallelism of control flow artifacts, such as loops, as illustrated below:

```ada
for I in 1 .. N
   with parallel, chunk_size => X
   loop
      F(i)
   end loop;
```

Discussion followed about the wisdom of giving any directive further than with parallel for the programmers to control the details of how parallelism is configured, executed and potentially mapped in the runtime. Programmers may not provide the correct specification of detailed controls, and as hardware changes over time, some argued that it is better to let the compiler have the control on these inputs. The counter argument was raised that in real-time systems there is a need for the programmer to specify such...
control to directly specify the behaviour, which is required for behaviour analysis and timing behaviour analysis. In other cases, the default performance parameters may be suboptimal for a particular problem, and the programmer may need to squeeze out extra performance by tweaking the controls. This could be the case in particular when code is being written for a very specific target hardware platform.

Questions were raised about the memory model of the proposal. The general model is that it supports a shared memory system, with cache coherency, with uniform access to memory, within a single partition. At the same time the desire was not to restrict the model if at all possible, to other possibilities. Underlying memory buses and memory organization, however, mean that there can be orders of magnitude difference in accessing any particular memory location from various cpu's, and issues such as cached memory and cache flushes can cause wildly varying access times, and possibly inconsistent views of shared data.

It was emphasized that the view of a partition as a shared memory model is pretty ingrained in Ada.

The presenter explained some of the terminology associated with parallelism, in particular, the term Reduce, is described as a special subprogram needed to combine results from multiple workers into a single overall result. In addition, the term Identity value is described as a value that when applied as one of the arguments to the reducer function produces the identical result. This terminology is commonly used in fine-grained parallelism approaches.

It follows that the syntax of the proposal can be implemented entirely through the use of the addition of special parallelism aspects, although one of the other language syntax changes would involve adding the ability to specify aspects on a loop, in order to support parallel loops to connect the programmers code to the backend parallelism model.

The additional controls that could be specified for the fine-grained parallelism were presented, with the idea that defaults were always provided (or selected by the implementation based upon the number of cores and memory layout specifics. Examples of controls that a programmer might wish to specify include:

- the reduction function;
- identity value;
- parallelism strategy (eg. work-stealing, work-seeking, work-sharing);
- chunk size;
- worker count;
- ceiling priority;
- affinity;
- worker task storage size; and
- task pool size and behaviours (such as dynamic or static).

As an example of a reducing loop, the example of a loop that calculated the sum of integers from 1 to N was given, where Sum is a variable declared in a global scope outside the loop. Ordinarily computing the sum in parallel would cause problems due to concurrent access to the Sum variable, but this can be avoided if each worker computes a local Sum value for each worker task, which is then combined (Reduced) into a single value by the time all the workers have completed their work.

There was significant of discussion about needing a definition for the unit of parallelism, and to define the semantics of a Tasklette, and indeed whether Tasklette is even an appropriate name for the concept. Alternate names suggested were Strand, and Fibre. The difficulty that participants had with Tasklette was that name is very close to Task, which seems to imply that one should be able to have attributes, execution time accounting, and blocking on such creations, which was antithetic to what participants wanted. No decision was taken, so this summary uses the term tasklette to stay consistent with the workshop papers. The reader is invited to substitute strand or fibre as they choose.

Andy Wellings presented a glimpse of the model of the Multicore Association “Multicore Programming Practices” and in particular the model of differentiated control level parallelism from data level parallelism. Although the presentation started off with discussing data-level parallelism constructs

---

1 The literature calls Task-level parallelism, but we use the term control-level parallelism to make it clear that we are talking about the parallelism of control structures, not task-based parallelism.
such as parallel loops and parallel recursion, the group felt that the case for control-level parallelism was more important and relevant for discussion in a real time context. Miguel and Steve point out that the proposal is about providing some basic building blocks for parallelism, which included both control-level parallelism and data-level parallelism. Nonetheless, the group expressed interest in focusing on control-level parallelism, which was the subject for the remaining part of the discussion.

Some participants objected to the “bottom up” approach taken by the authors. There was a discussion that programs often take a top-down design of the software (such as an object/method view of the world and disassemble or refine these objects as needed), and that parallelism models should be developed from the application models.

A request was made to discuss the parallelism model without discussing the underlying implementation. This was agreed in general and the rest of the morning's discussion largely stayed away from the underlying implementation model.

A discussion was held about how exceptions in the proposal were handled. The authors agreed that exceptions were not explicitly discussed in their papers, but stated that exceptions could take the following form:

- An exception raised in a tasklette is returned to the tasklette parent and the tasklette ceases to exist.
- Any other exception raised in another tasklette would detect that an exception had already been raised in this POP and the tasklette ceases to exist.
- Any tasklettes that have not commenced execution of their portion will not be started, even if the values that they process would have executed in the sequential model.
- When the parent resumes execution from the end of the POP, it does so in an exception handler following the standard Ada model.
- The semantics of parallel exception handling will be different from the sequential model, but it was noted that, in Ada, one cannot rely on any data values being updated in a construct that is the subject of an exception.

A belief was expressed that parallel loop operations seem to be always on an array. This led to a discussion of the characteristics of loop POP's. The most obvious loops that can be parallelized are for loops that span a predetermined (i.e. before the execution of the first pass of the loop) count of iterations. These match very closely with arrays so it is natural to give simple examples over arrays.

It was pointed out that the example of finding out if a number is prime, is an example where a loop could be used in parallel, without any association with an array.

It is also possible to parallelize loops that do not have a known stopping point, but the most efficient parallelization techniques may create a significant amount of execution that must be discarded once the actual exit condition is calculated (i.e. all iterations corresponding to execution beyond the exit condition).

A discussion item was raised on the possibility of having fine grained and coarse grained parallelism within same programming domain? The presenter responded that the model being presented accommodates both simultaneously.

A point of view was given that perhaps all of the needed functionality could be provided through libraries, i.e. no new language syntax. The presenter responded that libraries alone (i.e. with no supporting language syntax) almost always require the programmer to rewrite the algorithm to take advantage of the libraries and that this often makes reading and maintaining the algorithm problematic. The authors also pointed out that the model not only provides what a set of libraries would provide, but also gives the user the ability to plug in or provide the functionality to handle more challenging environments, such as uneven memory systems, real time systems and even hard real time systems.

It was agreed by the workshop that they needed to understand what other languages were doing in this domain. Miguel presented the parallelism proposals for other languages such as C, C++, C#, including the Cilk and Cilk Plus functionality for C and C++, Open MP, Thread Building Blocks (TBB) and a little on ParaSail.

It was noted that Cilk Plus uses a strict fork-join model. Strict means that a Cilk task\textsuperscript{2} cannot jump\footnote{The C++ usage of the term task to refer to what Moore, Pinho and Michell call tasklettes is a source of confusion.}
into the middle of a parallel computation, and no execution can proceed beyond the end of the POP until all tasklettes have completed. Cilk Plus has an explicit Cilk_Sync that should be used before any variables written by the tasklettes are consumed, but that there is an implicit sync before the block or function containing the POP returns. It was explained that the Ada model being proposed contains only implicit synchronizations and that it must occur before the result of the POP is consumed.

One advantage of the Cilk Plus fork-join model is that, if you remove the Cilk_Spawn, Cilk_For and the Cilk_Sync commands, the program executes completely sequentially. For the proposals for Ada, removal of the with Parallel aspect results in the normal sequential execution.

Another advantage of the strict fork-join model is that the strands have full visibility into the stack of the task that contains the POP, with the knowledge that the stack frame cannot be finalized until after all strands have finished. Models that use futures must create explicit return objects for the POP to deliver results into (likely on the heap) which can then be consumed at the explicit discretion of the programmer.

The issue of functions without side effects was raised, i.e. no in out or out parameters, no aliased parameters, and no access types passed as parameters, unless there is a mechanism to show that such actuals are not written to during the execution of the strands. The issue of pure functions was discussed, but no conclusions were reached.

Discussions were held about whether or not tasklettes should be named entities within Ada. There was interest that explicit algorithms could be created that used such named entities. The presenters explained that there is a clear separation between concurrency, which is captured by tasks, and parallelism, which is the transformation of the sequential code so that it could be executed by as many execution resources as are needed at the time. After significant discussion, it was agreed that tasklettes need not be named entities.

The issue of the language Parasail generated further discussion. Parasail permits all constructs that are not explicitly made sequential to be executed in parallel with other parallel statements or constructs. Loops can be executed in parallel, unless designated forward or reverse. The workshop considered if

```ada
for I in reverse 1 .. N
   with parallel loop
     ...
end loop;
```

meant that the loop must be executed sequentially for Ada. It was noted that, since Ada already had the reverse keyword, one could not automatically enforce a rule that all such loops must be sequential. It was also noted that there were viable parallel algorithms for such cases, meaning that the use of such keywords to signify directed sequential behaviour would likely not work.

A discussion was held about whether or not parallel code should be executed explicitly by library routines, such as Paraffin. It was pointed out by the presenters that the library mechanism did not provide automatic transformation of POP code. It takes significant rewrite of the sequential code to fit it into the library call mechanism, and the code becomes more fragile, more difficult to read and more difficult to maintain when using libraries. Syntax provided by the presenters, on the other hand, becomes aspects of the POP structures that provide direction to the compiler in how to map the sequential code for parallel execution.

A subtopic of the discussion of tasklettes, was what happens to exceptions that are raised inside of tasklettes. Since tasklettes represent simply a parallel execution of the parent task, the exception must be delivered back to parent at the point of synchronization. If multiple exceptions are raised by tasklettes, all but one exception are discarded. Following Ada's exception semantics, it is irrelevant what tasklette instance captured the exception, because you cannot rely upon any state that was being changed when an exception occurred.

Another issue discussed was how much support that compilers can give to programmers in identifying code that cannot be successfully parallelized by the compiler. This could be because of data dependencies between tasklettes, aliasing of parameters, non associativity of operations, etc. It was noted that in other languages that compilers are not required to make such checks, but with Ada's stricter language rules it

---

*Therefore, when talking about the C++ usage, the term Cilk task or a C++ task is used.*
may be possible to have more language support to at least detect and report parallelization errors.

REAL TIME PROPERTIES OF TASKLETTES

As the discussion moved towards the real-time aspects of the model, the workshop began to focus on what properties of tasklettes were needed in the semantic model.

Many participants saw tasklettes as exclusively a run-to-completion model, where tasklettes could only execute code to the synchronization point, and should not block, i.e. call barriers, suspension objects, entries, delays or file IO. This notion is at odds with what competing languages are doing, as C++ examples show many tasklettes (tasks in C++) performing HTML-based calls over the internet, which certainly is a blocking operation. It also is at odds with the notion of higher priority tasklettes interrupting lower priority tasklettes. It was noted that one of the reasons why those non-Ada models of parallel computations went other ways than run-to-completion may be due to their missing concurrency in the original language, hence causing the need to address concurrency and parallelism in the same entity space.

There are, however, reasons for wanting a run-to-completion model for tasklettes that is derived from performance considerations of massively parallel machines. Effectively, processors with dozens or hundreds of cores cannot maintain a strict cache coherence between all cores, and although they can construct a model of shared completely shared memory, the reality is that the time to access any given address in the system may vary by orders of magnitude between different cores, and cache flushes may have dramatic adverse effects on neighbouring but independent variables. One way to mitigate such effects is to copy all relevant code and data needed for an algorithm to a worker task (or worker CPU), have it execute the algorithm, then copy back the results when finished.

Another issue supporting the no blocking approach is that such blocking involves the scheduler that manages tasks, but anonymous tasklettes do not have task control blocks, hence may not be schedulable. Even if each tasklette is executed by a worker task as proposed by the presenters, it is an open issue whether the blocking of a tasklette would result in a block of the carrying worker task, or if that task or if that worker task would simply pick up another tasklette for execution. There is an obvious impact in analysability, depending what approach is taken.

The issue was not resolved, but there was brief mention made that such blocking behaviour could be selectable by an aspect.

In the same vein, significant discussion was held about what the runtime should return if a call was made to Current_Task, or to get or set task attributes within a POP, resulting in tasklettes making such calls. There was some opinion that in such cases, tasklettes should act as if it was the parent task making the call, for example returning the Parent’s Task_ID for Current_Task. Since no polls were taken on these subjects, it remains open.

Another issue discussed was whether or not tasklettes could be aborted. Since tasklettes cannot be named in the program, there is no way to explicitly abort a tasklette.

NESTED PARALLELISM

There were discussions as to whether or not tasklettes could spawn other tasklettes. The issue of recursive subprograms or subprograms being executed by a tasklette and containing a POP shows clearly that tasklettes must be able to spawn more tasklettes.

EXPLICIT PROGRAMMER CONTROL

There was a discussion about the need for explicit programmer control of the various factors that impact the performance of parallelism, but also the explicit needs of real-time systems. Some of the issues that programmers may need to control include

1. Data locality
2. Aliasing of data
3. Reuse of already-calculated objects
4. Calculation deadlines of the parent task
5. Derived calculation deadlines of POP's
6. Blocking or non-blocking of tasklettes

Some that implement compilers and runtimes raised the issue that many times programmers try to control an algorithm but often hinder the implementation's ability to manage all of the issues effectively. This is especially true when the same code can be executed on widely varying underlying hardware. The opinion was expressed that programmers should give high-level guidance to implementations on the management issues and leave it to the implementation to perform the actual layout and management.

Those that build real-time systems raised the issue that regulators will not permit them to “trust the implementation”. They work in an environment where they must be able to account for all behaviours produced by the program and the implementation; hence must be able to specify and control such behaviours.

It was generally agreed that the management of POP's needs to support multiple modes of control. The three identified were:

- The compiler decides everything as much as possible
- The programmer provides general guidance; and
- The programmer provides explicit control of how the POP is implemented.

There was some support that the aspect mechanism provided by the presenters had many of the characteristics needed, and that more discussion of the individual aspects of the proposal was required, but is still considered an open issue.

3. **Other parallel architecture issues**

Paper: Juan Antonio de la Puente and Juan Zamorano - “On Real Time Partitioned Multicore Systems”

The authors presented their position that there are ways that high criticality systems and low criticality systems can reside on the same system. The implementation requires that all levels of criticality be separated into their own partitions. These partitions are separated from each other in time (partition scheduler), and by memory space (MMU). Individual partitions are scheduled locally.

Various approaches have been used in prototyping such systems. One approach is to place partitions onto virtual cores, and to map the virtual cores using the Hypervisor virtual machine system. Physical processors were statically scheduled, with predictable scheduling within each partition.

The potential difficulties with this approach are memory access contention and maintaining cache coherence.

The presenters have implemented a demonstration system on a single board using an Intel processor and a Leon 32 processor sharing common memory and running Hypervisor. They also analyzed Ada 2012 with respect to mixed criticality systems, and report that there are no new language features needed (beyond those available in Ada 2012) in Ada for such systems.

As systems move to many-core systems, proposals have been made to place a single Ravenscar task on each core and analyzing the system using that paradigm. In high criticality systems, however, there is deep concern that bus contention, memory contention, and cache coherence issues make timing analysis and behaviour reasoning unreliable.

Significant issues remain in designing and implementing such systems. Communication between partitions is a concern, in that safety-related partitions must not rely on data from low criticality partitions, and security-related partitions cannot pass secure data to less secure systems. Similarly, the possibility that
the individual MMU's can be compromised, or that shared buses can be overloaded by the low criticality systems are significant concerns. There was also discussion on approaches where high and low criticality code were executed inside the same partition (an example was presented), but it was felt that the correct model should be to separate criticalities in different partitions.

One of the issues raised was if the inter-partition communication model of Ada is appropriate for these types of systems. It was felt that other models (such as publish-subscribe based) could also be interesting. It was agreed that other inter-partition models would be a reasonable future direction for workshop submissions.

These systems are being investigated, but for now multiprocessor mixed criticality multicore systems are not possible. For now all high criticality systems disable all but a single cpu in their systems.

It was recommended that IRTAW follow this thread as it progresses. Of interest is what the aviation community is doing, as well as the automotive industry.


Miguel Pinho led the discussion, raising the idea that partitions could also be units of concurrency or parallelism. It was questioned whether the Ada single memory-space / few task model was really capable of describing where technology was moving with thousands of processors, possibly with non-uniform instruction sets, and non-uniform memory structures.

A discussion was held that there is a model of Ada partitions as units of concurrency, which could possibly be extended to units of parallelism, but that the current restrictions on partitions make using partitions in this way less efficient. It was agreed that the remote procedure call mechanisms are heavy-weight for communicating between tasklettes, and the shared passive partition model prevents the usual communication models between partners in a communication. The solutions proposed by the authors were discussed, but no consensus was reached in this session.

4. Conclusions

The following summarize the agreements reached at the workshop about the applicability of fine-grained parallelism to Ada programs. It would be useful to have a syntax and a semantic model for control-oriented parallelism, and such a model could be based on the notion of an unit of potential parallelism. In such a model:

1. Tasklette need support of some schedulable entity that gains cores for execution.
2. Tasklette/Strand do not have identities and do not have their own existence
3. Any attributes or invocations such as Current_Task could be the Creator task
4. Their executing time is not accounted
5. The underlying entity that executes a tasklette may be a task, but may be other constructs.
6. The creator task should not block but should continue executing, usually by executing one or more tasklettes and execution time accounting is done only for the parent task. This could lead to busy waiting just because of execution time accountancy
7. The model should be a strict fork-join model. The entity that created tasklettes may needs to wait for their completion. This could be a busy wait to satisfy execution time accounting.
8. In the priority model, tasklettes inherit the priority of the task and may be executed non-preemptively. It was noted that issues associated with processor affinities and dispatching domains must be revisited.
9. Exceptions could be treated in the same way that Cilk is treating them – the first exception is flagged to be raised in the parent and others are discarded. This may create different behaviour from a sequential program.
10. The nominal units for parallelization are:
   • subprogram calls, including in expressions
   • for loops
   • Ada whole operations, such as assignment of aggregates
     but we need syntax to address conflicts, such as overlapping ranges

REFERENCES

Ada Europe 2013
Session Summary: Locking Protocols

Chair: Alan Burns
Rapporteur: Andy Wellings

1 Introduction

The session considered two main issues: the introduction of the deadline floor locking protocol into a future version of Ada and multiprocessor locking policies.

2 The Deadline Floor Protocol

Ada 2005 introduced EDF scheduling across priority bands. A version of Baker’s Stack Resource Control Protocol (called the Preemption Level Control Protocol) was also introduced so that ceiling priorities could be used within an EDF context. However, the Preemption Level Control Protocol is complex and the position paper by Aldea, Burns, Gutierrez and Gonzalez Harbour entitled “Incorporating the Deadline Floor Protocol in Ada” has proposed an alternative protocol that is conceptually much simpler and easier to implement.

Alan Burns introduced the protocol and explained its main motivations and features. The protocol is targeted at single processor system and the discussion was held within this context. The protocol requires each protected object to have a related deadline associated with it. This deadline is the minimum (floor) relative deadline of all the tasks that use that protected object. Proper setting of the floors ensures that each task gets only a single block and mutual exclusion is guaranteed by the protocol itself.

Several issues were raised in the discussion and these are summarised below.

- The impact of release jitter on the correctness of the protocol. Michael Gonzalez Harbour explained that care had to be taken when tasks could be subject to release jitter as this could result in the delayed execution of a shorter deadline tasks that then could preempt a longer deadline tasks while it was active in the protected object. It was, therefore, necessary to use the values of Deadline – Jitter for each task rather than its simple deadline. Failure to do this would invalidate the protocol, and mutual exclusion would not be
guaranteed by the protocol itself. Hence, for safety it is also necessary to provide a mutex lock to control protected object access. It was noted, that a similar problem occurs with jitter and the priority ceiling protocol. However, in that case, more priority inversion results instead of the breaking of mutual exclusion. It was also noted that it was possible to optimize the lock so that it was a single bit that indicate that the protected object was occupied. Any attempt to access an occupied protected object would result in an exception being raised.

- The meaning of an inherited deadline. In a real-time system there are usually consequences that must be managed if a task misses its deadline. With the deadline floor protocol, a task may inherit a deadline, which will be shorter than its application-defined deadline. The workshop discussed the consequences of a task missing its inherited deadline. It was agreed that inherited deadlines were required to control scheduling and missing them had no repercussions for the application tasks. For example, the default floor for a protected object is `Time_Span.First`, and hence it is quite possible that an absolute deadline computed using this floor value is missed. Consequently, the workshop recommended that, similar to priorities, that there should be a notion of base and active deadline. The programmer would have no visibility of the active deadline of a task. Any application-level deadline detection mechanisms involves its base rather than its active deadline.

- Protected objects shared between EDF-scheduled and priority-scheduled tasks. In order to fit into the Ada framework for scheduling mixed systems, it is necessary to allow some protected objects to have both a priority ceiling and a deadline floor. The rules are simple, if the ceiling of the protected object is a FIFO-within priority level, the task’s active deadline is not updated while executing within the protected object (i.e. there is no need to have a deadline floor). If the ceiling priority is an EDF-within priority level, the task’s active deadline is updated (i.e. it does need a floor). Nested protected object across levels require further consideration.

- Dynamic changes to the base deadline. It was noted that asynchronous changes to the base deadline of a task does not result in the recalculation of any active deadline associated with the task. Also a new optional check could be specified when using `Delay_Until_And_Set_Deadline` to ensure that the new deadline is longer than or equal to now plus the relative deadline of the tasks (as set by the pragma `Relative_Deadline`).

- Deadlines and other inheritance points in Ada. For completeness, the work-
shop agreed that in principle a server task should run with an active deadline which is the shortest of its own deadline and the deadline of the calling tasks during a rendezvous between two tasks. Similarly, deadline inheritance should occur during task activation.

Following the above discussion, the workshop agreed that the deadline floor protocol would be a useful addition to Ada and that the Preemption Level Control Protocol should be made obsolete. This could be achieved with a new dispatching policy and/or new locking policy.

3 Multiprocessor Issues

The issue of how to integrate appropriate policies for accessing protected objects in multiprocessor system (into the Ada language) is still largely unresolved. The Ada reference manual suggests that tasks busy-wait for a lock but does not specify any priority or queuing policy associated with this. There were two papers submitted to the workshop on this topic. One considered a new lock-based approach (“Locking Policies for Multiprocessor Ada” by Burns and Wellings). The other considered a lock-free approach (“Lock-Free Protected Types for Real-Time Ada” by Bosch). The workshop discussed both approaches but felt they were both were not yet mature enough to warrant suggested language changes at this time. For the Burns-Wellings paper, further experiments and evaluation were needed including a prototype Ada implementation.

Much of the discussion on the lock-free approach focused on the restrictions that had to be placed on the application code so that updates to the protected data could be achieved by a single machine instruction. This was compared to an approach of having library-supported atomic operations on primitive types (e.g. operations on atomic integers). The main advantage of using protected objects was that the application got to define its own atomic regions rather than having predefined operations. The workshop felt the approach was promising but wanted to see more detailed definitions of the restrictions (and how they would be checked) and whether other forms of lock-free approaches and algorithms were possible.
Session Summary – Improvements to Ada

Chair: Tullio Vardanega
Rapporteur: Rod White

1. Introduction

This session was introduced by Tullio who outlined the papers being considered and how they might lead to improvements in the Ada language and some of the potential challenges they posed.

Three position papers were discussed in this session:


ii. Execution time timers for interrupt handling, Kristoffer Nyborg Gregersen and Amund Skavhaug

iii. Deferred Setting of Scheduling Attributes for Periodic and Sporadic Tasks, Sergio Sáez, Jorge Real and Alfons Crespo

2. Programming Simple Reactive Systems in Ada

The paper covers the use of Ada to develop simple reactive, deterministic automata, and the issues of termination of non-tasking programs. Paper identifies two main issues:

- Queuing of interrupts and the difficulty of determining the ordering of multiple events, and more fundamentally
- Program termination – the issue that prevents the simple reactive model from working.

The proposal to the workshop was that the termination semantics for Ada should be changed to be defined thus:

The environment task should terminate when all of its dependent tasks have terminated, and the partition has:

- No active timers, and
- No handlers attached to interrupts that are serviced by the partition.

(Proposed changes in italics)

It was noted in the paper that if the termination semantics are changed as suggested it will break backwards compatibility as it is currently possible for programs to terminate with timers and attached interrupts.

In the case of the active timers there was a consensus that the termination in their presence is probably an incorrect, and possibly unintended, behaviour. The interrupt issue is slightly less clear, it can be addressed by either handlers being attached and detached dynamically, or by permitting statically attached handler to be detached dynamically – possibly a somewhat counter intuitive concept. The paper also recognised that the problem can be overcome within the context of the current language facilities – a
kludge is possible: the main procedure can either perform either a `delay until Time’last` or a wait on a suspension object that is never set true.

Two possible approaches were initially suggested that would solve the problem without impacting backwards compatibility.

- An indication via a pragma (or aspect) that the environment thread was not to terminate, or
- The ability to control termination – for cases where termination is required.

In this discussion only single processor programs were considered, restricting the discussion to task-free programs – inclusion of multiple processors and tasks would add further complexity.

Whilst the termination in the presence of attached interrupts was not seen as a major issue there was a general consensus that termination in the presence of active timing events was incorrect – as these had been programmed, and if they were not needed then they should be explicitly cancelled by the application.

There was some concern over the need to check for outstanding timing events – when and where should this be done? There was another concern regarding the pattern whereby timing events are programmed to give a periodic behaviour; this pattern would never terminate, but explicit cancellation could address this.

It was noted that the problem has its origin in the change to the interrupt handling model that occurred between Ada 83 and Ada 95 – in Ada 83 interrupts were handled directly by tasks – hence there was no problem with interrupt handlers being left attached after the tasks had terminated. This change in the way interrupts are addressed by the language has been one of the biggest issues in the migration of applications from Ada 83 to Ada 95.

The group concluded that this was not a pressing issue given the simple work-arounds that exist and that there was little merit in making language changes in this area.

It was also agreed that the termination issue should be noted in the assessment of concurrency vulnerabilities.

### 3. Execution time timers for interrupt handling

Ada 2012 introduced execution time clocks for interrupt handlers – the proposal made in the paper was that Ada should be extended to provide execution time timers for interrupt handlers.

Identified issues with interrupts include:

- Hard to predict their rate of arrival;
- Hardware faults can result in bursts;
- In Ada 2012 it is only possible to measure the execution time of interrupt handlers (using the `Clocks defined in Ada.Execution_Time.Interrupts`);
- Interrupt timers can be efficient with respect to the alternative of polling the time to determine when it has been exceeded;
- There is also a related issue with timing events where the facilities are even more limited; here, unlike interrupts, it is neither possible to measure the execution time, nor to set an execution time timer.
The proposal was for there to be a timer for each Interrupt_Id but not one for the overall time consumed by all interrupts (Ada 2012 also supports the concept of a single execution time clock for all interrupts). A prototype of such a solution has been implemented in the GNAT compiler for the AVR32 processor.

Whilst the paper viewed this as an extension to Ravenscar it was noted that it would fall outside of Ravenscar as execution time timers were not in the Ravenscar profile owing to the lack of an effective model of use that would fit the spirit of the profile.

It was proposed that the timer type should be a derived type of the task timer, but the group felt that this was inappropriate/incorrect as the as here was a mismatch between the two forms: the task timer contains a Task_Id whilst the interrupt timer required an Interrupt_Id. It was suggested (and agreed) that the best approach would be to define a now root type for timers that could be specialised for the specified of the task and interrupt timers; this approach would then allow for the inclusion of timers for timing events in a similar manner.

In general it was felt that interrupt handler code should be straightforward and serial, and hence of limited and bounded duration, this in turn led to the concern that there might be significant overheads due to the facility that might detract from this position. This led to the question: are we really only interested in the total interrupt count and rate of arrival rather than the CPU time consumed? It was noted that there is probably more of an interest in providing timers for timing events as these are firmly in the application domain, the one where timers are more widely considered to be useful.

A major concern expressed quite widely was the potential cost/overhead of the feature. The authors explained the advantages of hardware support to provide timers, but this clearly was not going to be a universal solution. There was a concern regarding the overhead of accessing the hardware clock, which for some modern processors is seen as being potentially significant.

Fundamentally the group agreed that the goal must be to retain predictable behaviour.

It was felt that, with the inclusion of the timers for timing events, this was a useful facility that would be of use now; the inclusion of counters was seen as being a useful addition. There was general support for the basic idea if not the detail – given we already have half the facility (clocks for interrupts) it seems sensible to provide this kind of extension.

A number of issues were noted that had to be worked on to give a more coherent solution.

- The way in which the deferrable server would work was not entirely clear and a more complete description was required;
- The type model needs to be reworked to make the types for timers in general coherent;
- The model should be extended to also include timers for timing events;
- It is important that any implementation can ensure that its support for this feature results in zero overhead for any application that does not make use of the feature.

Given these issues are adequately addressed interrupt timers could be a feature for inclusion in a future revision of the language.

4. Deferred Setting of Scheduling Attributes for Periodic and Sporadic Tasks

Over the past two IRTAWs the issue of setting multiple scheduling attributes simultaneously has been noted as a topic of some interest and importance.
This paper is a follow on from the previous IRTAW where the issue of setting the various attributes of a task atomically had been seen as being an issue – the current model in Ada 2012 allows only for the setting of a single attribute at a time (except for period and deadline). In outline the paper proposes a new type to capture a set of scheduling attributes, an instance of which is associated with each individual task, which can be passed to the underlying kernel in a single call, hence facilitating their simultaneous, atomic setting.

The Ada code is relatively straightforward:

- A simple extendable type holding the attributes for the task appropriate to its dispatching regime and the execution platform, e.g. priority, affinity for FIFO_Within_Priorities dispatching, and extending to include relative and absolute deadlines where EDF dispatching is used;
- Some helper subprograms to set/get the various attributes in a the local copy of the attributes object; and
- A pair of subprograms to commit/recover the current attributes to the underlying OS.

The OS can be source of much of the problem, and in the case of general purpose operating systems the provision of appropriate OS support for the Ada tasking model, and its semantics and attributes is the hardest part to solve.

The proposal includes two basic options with respect to setting the attributes of a task: setting them immediately, and setting them and suspending for them to apply at the next release. In both cases issues were raised regarding exactly how these might work. In the first case there was the point that setting could not be immediate if the caller was in a protected operation – the application would have to be deferred until after the protected operation had been completed. In the second case, that where the task becomes suspended, two significant points were raised:

- Firstly, does the suspension take place in the context of the new or the old attributes? This leads to a number of more detailed considerations such as: where the affinity is changed in the attributes is the task suspended on the original, or new processor?

- Secondly: what should the behaviour be for zero and negative delay values? The Ada behaviour is not necessarily the same as that of operating system interfaces such as POSIX where these cases may not result in a dispatching point.

It was agreed that the suspend form of the operation could only be applied to the current task (i.e. itself) and thus the Task_Id parameter was redundant – this principle was not extended to the immediate form of the operation.

Given the complexity, an alternative approach was tentatively suggested. Why not replace the suspension by a timing event that sets the attribute in its protected operation? The fact that it is a PO will ensure atomicity of the attribute change, but it was noted that this is not necessarily the case where the affinity is changed. From this there was some discussion as to whether affinity is particularly difficult and should be treated as a special case – no specific conclusion emerged from this discussion.

There was some concern about where this facility would feature in the ARM. It was agreed that it would be in an Annex, probably Annex D, and that its implementation would have to be all or nothing at the level of the individual child package. Thus it would be an optional feature.
It was noted that the parameters must be scheduling parameters, the “At_Time” field was viewed as being a helper, for the EDF extension the absolute and relative deadlines should be discrete fields in the record.

In summary:

- The possibilities are not well tied down – there is a high degree of operating system dependence in the current proposal.
- The facility is highly dependent on the underlying OS for its support – if the OS does not support the concept of task attributes in a way that is compatible with the Ada model then simply don’t support the facility.
- Experimental changes need to be made to the Linux kernel to facilitate the feature – results should be reported at the next IRTAW. (It was felt that it would be easier to make the change to Linux than to get POSIX changed for a feature that is essentially needed for real-time operation – the POSIX real-time community is seen as being less active than that of Linux).
- In terms of the code, the unnecessary references to Task_Id should be removed.
- Attributes must be true scheduling parameters – not “helpers” – thus for EDF dispatching both relative and absolute deadlines should be captured;
- The feature should be developed for inclusion in Annex D.
Session Summary: Open Issues

Chair: Jorge Real
Rapporteur: Juan Antonio de la Puente

1 Introduction

Most of the session was focused on discussing the opportunity to define a new Ada profile by adding execution-time control mechanisms to the Ravenscar profile. The basis for the proposal was the position papers by Gregertsen. Related work includes the paper by Gregertsen and Skavhaug [2] on execution-time control mechanisms.

The session started by the chair recalling a statement from a proposal presented at IRTAW-15 [1]:

*To make it worthwhile to define a new profile, there must be*
- a clear application need,
- a computational model that reflects this need,
- and an implementation strategy that leads to a run-time footprint significantly smaller than that needed by the full language.

The above criteria were considered meaningful by the group.

2 An extended Ravenscar profile

Kristoffer Gregertsen presented his proposal of an extended Ravenscar profile with execution-time control mechanisms. The main motivation is to overcome the limitations of the Ravenscar profile with respect to real-time fault tolerance. The features that could be included in the new profile are:

- execution-time timers;
- group budgets;
- asynchronous task control;
- dynamic priorities;
- asynchronous transfer of control;
- abort statement.

Execution-time timers and group budgets are proposed as run-time mechanisms for detecting overruns. Asynchronous task control and dynamic priorities can be used to lower the priority of a faulty task, thus reducing its impact on the system, and asynchronous transfer of control and abort can provide further support for this purpose.

There was a vivid discussion on the proposal. A basic consideration is the wish to keep the run-time system efficient and small, in order to facilitate certification when required. Robert Dewar made a point
that adding a profile would not be too complex for compiler builders, but adding new restrictions might be. There was general agreement that abort and ATC are the most complex features to implement, whereas the rest would not pose so much of a problem.

Another topic is the possible uses of the extended profile. The Ravenscar profile forces a static environment that enables schedulability analysis to be carried out in critical systems, and was originally conceived as a replacement for cyclic executives that were dominant at the time. On the other hand, an extended profile may add flexibility for other possible uses. Geert Bosch commented that Ravenscar is too limited for some users, while Rod White observed that some non-critical applications use the Ravenscar runtime because it is small and simple. Amund Skavhaug stressed the interest of the extended profile in education, where it could be used in small student projects.

The discussion went on by considering some specific details of the proposal. Dynamic priorities and asynchronous task control were considered as mechanisms for dealing with faulty tasks. Tullio Vardanega pointed out three possible policies after a deadline overrun:

- the faulty task can be made non-eligible for running;
- if it can still do some useful work, it can be allowed to run at a low priority;
- it can be restarted, or a mode changed can be triggered.

There was consensus that asynchronous task control is a complex issue that can be difficult to implement in a reduced runtime system.

3 Conclusions

The proposal of defining a new profile that adds flexibility and run-time control mechanisms to Ravenscar while keeping a reduced size and complexity seems interesting and the group agrees that it deserves further investigation. Especially asynchronous control and dynamic priorities have to be studied in detail in order to find all the possible implementation issues. Further work is also needed on the definition of useful fault recovery policies.

References


REUSABLE SOFTWARE COMPONENTS

Trudy Levine
Fairleigh Dickinson University
Teaneck, NJ 07666
levine@fdu.edu
http://alpha.fdu.edu/~levine/reuse_course/columns

This column contains a listing of reusable software components, begun for Ada Letters in 1990. All information is obtained directly from parties affiliated with web sites hosting Ada components or from the sites themselves. As always, no recommendations or guarantees are implied. We appreciate comments, corrections, and suggestions from our readers.

Hard copies of Ada Letters (with green covers) going back to 1988 are available for reuse upon request.

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Ada-Belgium

One of the aims of the Ada-Belgium organization is to disseminate Ada-related information. So, in addition to the organization of seminars, workshops, etc., and the management of two mailing lists, it also has set up an Ada site that enables everyone interested to consult and download a large variety of Ada software and documents using a server in Belgium.

Key items include:

* Conferences and events for the International Ada Community
* Ada job announcements, in or close to Belgium
* A disk copy of the last version of the Ada and Software Engineering Library (ASE2, a 2 disk CD-ROM set).
* A complete archive of the last public GNAT distribution that uses the GNAT Modified General Public License (3.15p).
* A directory with Free Ada Software provided by Belgian Ada users, including Rob Veenker’s instructions for using native Ada application on an Android device:

The Ada-Belgium archive is primarily intended for the Belgian Ada community, but anyone interested in Ada is welcome to use it.


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Ada Class Library


Ada Core

AdaCore provides open source tools and expertise for the development of mission-critical, safety-critical, and security-critical software. AdaCore’s flagship products are the GNAT Pro and SPARK Pro development environments and the CodePeer automatic code reviewer and validator. The GNAT technology is the first to support all four ISO standards of the Ada programming language - Ada 83, Ada 95, Ada 2005, as well as Ada 2012. GNAT Pro also comes with Frontline Support (provided by the developers of the toolset) and expert Ada consulting.

The GNAT technology includes:
- GNAT Programming Studio /GNATbench – Plug-In for Eclipse IDEs.
- GNAT Pro High-Integrity Family of products supporting safety and security standards such as DO-178B and MILS Support for over 70 native and cross platforms including Unix, Linux, Windows, .NET, the JVM, bare boards, VxWorks 5/6/653/MILS, LynxOS, and PikeOS

Add-on technologies:
- GNATemulator - for fast target emulation on the host.
- GNATcoverage - for source and object code coverage.
- Traceability Study - for DO-178B/C level A source-to-object code traceability.

Standalone additional technology includes:
- CodePeer - automatic code review and robustness validation.
- SPARK Pro – code verification, based on information-flow analysis and theorem-proving.
- CodePeer - automatic code review and robustness validation.

The GNAT Academic Program (GAP) was created to help bring Ada to the forefront of university study. It includes a comprehensive toolset and support packages designed to give educators the tools they need to teach Ada.

Free Software developers and students can download GNAT GPL from
http://libre.adacore.com/libre (updated 2012)
http://www.adacore.com/academia
or contact: gap-contact@adacore.com

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Ada Europe


See: http://www.ada-europe.org

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Ada IC

The Ada Information Clearinghouse has been providing free information about Ada and software engineering since 1990. Sponsored by the Ada Resource Association <http://www.adaresource.com>, a consortium of Ada tool vendors and developers, the AdaIC maintains close contact with the Ada community in order to obtain the latest information on a variety of topics. Several blogs are maintained to continue conversion on Ada topics.


The Ada-wide search engine indexes all known Ada content (more than 76,000 pages according to Randy Brukhartd’s last count). General search engines, such as Google, have too many references to the term “Ada” to make them practical for the purposes of the Ada community.

Please send any news you have on Ada to <news@adaic.org>. The Ada News of the AdaIC sometimes transmits press releases about the programming language to about 500 technical journalists and editors, as well as citing it on the AdaIC Website, as a free service to its users.

A comprehensive collection of Ada articles, reports, textbooks, videos, and CD-ROMS is available for browsing on-line through the AdaIC website. Users may access older components at the Virtual Library: <http://archive.adaic.com>

Reusable software components are available at <http://www.adaic.org/ada-resources/tools-libraries/> (updated May 2013)

AJPO

The Ada Joint Project Office was closed on October 1998. For information on the AJPO see http://sw-eng.falls-church.va.us/ajpofaq.html
http://sw-eng.falls-church.va.us/ajpo_databases/products_tools1.html

Adalog

Adalog offers Ada utilities, Ada components, and Adapplets. These can be freely used and modified for any purpose, under the GMGPL license. Functions include Protection, Debugging, and OS_Services, among others.

The site also contains Adasubst/Adadeep programs that are useful utilities for rearranging Ada programs, and AdaControl, a powerful utility for checking and enforcing style and coding rules.

AdaControl is a free (GMGPL) tool that detects the use of various kinds of constructs in Ada programs. Its first goal is to control proper usage of style or programming rules, but it can also be used as a powerful tool to search for use (or non-use) of various forms of programming styles or design patterns. Searched elements range from very simple, like the occurrence of certain entities, declarations, or statements, to very sophisticated, like verifying that certain programming patterns are being obeyed. The next issue of AdaControl will deal with a number of Ada2005/2012 features. Since it is GMGPL, all of its parts can be reused for any purpose.
These programs are built on top of ASIS and include valuable packages providing higher level queries for ASIS (package Thick_Queries). For example, look for the function called “Full_Name_ Image,” which returns the unique name of any Identifier.

In addition, there is sc_timer, the Session Chair universal clock, which is very useful to those who have to chair a session, and a demo of GTK-Ada.

SEE:  http://www.adalog.fr/  (site updated 2012)
Ada components available at http://www.adalog.fr/compo1.htm

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

AdaPower

AdaPower.com is a repository of Ada information, links to resources, source code examples and packages for reuse. AdaPower.com can be divided into the following sections:

Articles and Links
   Articles and Links to Ada Related Topics, Ada learning materials, and people in the Ada on-line community.

The Ada Source Code Treasury
   Source code examples of using Ada and Ada related bindings and tools for both beginner and advanced students of Ada.

Packages for Reuse
   An extensive repository of categorically arranged packages for download and links to packages and libraries available for reuse on the internet at

see:  http://www.adapower.com/  (Site updated 2012)

AdaPower's website has been entirely redone in Ada, using GRAW, a rapid agile web development framework, to maintain the repository of Ada information, links to resources, source code examples, and packages for reuse.

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Ada Structured Library Version 1.4

Ada Structured Library is a set of general containers and utilities. The library is licensed under the same license as GNAT (see GNU and FSF, below), which is GPL, but is modified to allow inclusion into a program without bringing the whole program under the GPL.

The utilities include some things lacking in Ada95, including:
   * Abstract I/O - allows the I/O user and the I/O to be decoupled, so you can do file I/O, socket I/O, serial I/O, etc. by changing the I/O object the user references. Includes many functions of Ada.Text_IO.
   * Calendar - Full-featured time and calendar manipulation.
   * Telnet - A general telnet library implemented over sockets.
   * Command processor - Does string tokenizing and command processing over Abstract I/O.
   * A set of general-purpose containers, including Lists, Vectors, Trees, Graphs, and a Btreee, with lots of options.

See:  http://adasl.sourceforge.net/
       http://sourceforge.net/projects/adasl

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Ada-Switzerland

Ada-Switzerland is an association that promotes the use of the Ada programming language. In particular, it maintains links to resources and projects of the Ada Programming Language. See: http://www.ada-switzerland.ch/ (updated 2013) http://www.ada-switzerland.ch/resources.aspx

Booch Components

The Ada 95 Booch Components began in late 1994 when David Weller ported Grady Booch's C++ components to Ada95. They have since been taken over by Simon Wright and Martin Krischik, and at this time, include implementations of bags, collections, dequeues, graphs, lists, maps, queues, rings, sets, stacks, and trees. These include definite and indefinite types, bounded and unbounded implementations, and dynamic and static storage allocations. Filtering and sorting operations are supported.

The Containers are compatible with Ada 95, Ada 2005, and Ada 2012 in GNAT 2012 mode. Backward compatibility is retained.
http://sourceforge.net/projects/booch95/
http://booch95.sourceforge.net/documentation.html#the-containers
http://sourceforge.net/projects/booch95/develop

Charles

Charles is a container and algorithms library for Ada, modeled on the C++ STL. Sequence containers (vectors, deques, and lists) store unordered elements, inserted at specified positions. Associative containers (sets, maps, multisets, and multimaps) order elements according to a key associated with each element; both sorted (tree-based) and hashed containers are provided. A separate iterator type associated with each container is used to visit container items and to effect direct modification of elements. Charles is flexible and efficient, and its design has been guided by the philosophy that a library should stay out of the programmer's way.

The Ada 2005 AI-302 reference implementation is located in the ai302 subdirectory: http://charles.tigris.org/source/browse/charles/src/ai302/
See: http://charles.tigris.org (Site updated 2005)

FREE SOFTWARE FOUNDATION

The Free Software Foundation is dedicated to eliminating restrictions on people's right to use, copy, modify, and redistribute computer programs. It promotes the development and use of free software and its documentation in all areas using computers. Specifically, it is maintaining a complete, integrated software system named "GNU". ("GNU" is pronounced "guh-new" and stands for "GNU's Not Unix").

The word "free" in "Free Software Foundation" refers to freedom, not price. You may or may not pay money to get GNU software, but regardless you have specific freedoms once you get it: the freedom to copy a program and give it away to your friends and co-workers; and the freedom to change a program as you wish, by having full access to source code. You can study the source and learn how such programs are written. You may then be able to port it, improve it, and share your changes with others. If you redistribute GNU software you may charge a distribution fee or give it away. For the Free Software Definition, see: www.gnu.org/philosophy/free-sw.html
What is Copyleft?

The simplest way to make a program free is to put it in the public domain, uncopyrighted. But this permits proprietary modifications, denying others the freedom to use and freely redistribute improvements; it is contrary to the intent of increasing the total amount of free software. To prevent this, copyleft uses copyrights in a novel manner. Typically copyrights take away freedoms; copyleft preserves them. It is a legal instrument that requires those who pass on programs to include the rights to use, modify, and redistribute the code; the code and rights become legally inseparable.

The copyleft used by the GNU Project is made from the combination of a regular copyright notice and the "GNU General Public License." (www.gnu.org/copyleft/gpl.html) GPL is a copying license which basically says that you have the aforementioned freedoms. An alternate form, the "GNU Lesser General Public License" applies particularly to certain GNU libraries. This license permits linking the libraries into proprietary executables under certain conditions.

See www.gnu.org/copyleft/copyleft.html
www.gnu.org/licenses/licenses.html

GNAT is listed in the Free Software Directory, which catalogs useful free software that runs under free operating systems, particularly the GNU operating system and its GNU/Linux variants. The GNAT Technology includes the implementation of the ASIS standard (Ada Semantic Interface Specification), GtkAda to build portable and efficient GUIs in Ada, AWS (Ada Web Server) the framework to develop Web-based applications in Ada, the XML/Ada library to process XML streams in Ada, GLADE to develop distributed applications following the Ada Distributed Systems Annex standards, and PolyORB to develop distributed applications following the CORBA standard.

The GNAT GPL 2012 Edition, which is available free of charge from libre.adacore.com/, is licensed for Free Software development under the terms and conditions of the GNU General Public License (GPL).

For more information visit the following links:

GNAT Pro: www.adacore.com/gnatpro/
http://directory.fsf.org/wiki/GNAT (site updated 2012)

Free Software Foundation, Inc.
51 Franklin Street, Fifth Floor
Boston, MA 02110-1301
+1 617 542 5942 x 23
+1 617 542 2652 (fax)
email: info@fsf.org
See: http://www.fsf.org (Site updated 2013)

http://www.gnu.org

GNAVI

GNAVI, the GNU Ada Visual Interface, is the open source alternative to visual software development languages like Delphi and Visual Basic. In addition to being fully Open Source under the GPL, the language foundation of GNAVI is the international standard of engineering, Ada. GNAVI for Windows offers comparable features to Delphi and Visual Basic, including use of Active X controls and the ability to interface with .NET and Java.

http://www.gnavi.org
Kazakov Objects

Dmitry Kazakov maintains a web site of free Ada components. The license is GM GPL, where appropriate. The library conforms to both Ada 95 and Ada 2005 language standards and includes:

1. Objects and handles (smart pointers)
2. Persistency
3. Sets and maps
4. Unbounded arrays
5. Unbounded arrays of pointers
6. Stacks
7. Pools
8. Doubly-linked networks
9. Graphs
10. Lock-free structures
11. Locking synchronization primitives
12. Parsers
13. Cryptography
14. Numerics
15. Miscellany
16. Networking
17. Packages
18. Installation
19. Changes log

See:  www.dmitry-kazakov.de/ada/components.htm
www.download25.com/simple-components-for-ada-download.html (site updated 2013)

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Leake Components

Stephen Leake maintains the following Ada components:
Auto_Text_IO: automatically generates Text_IO packages for Ada packages
<http://stephe-leake.org/ada/auto_text_io.html>
Stephe's Ada Library: another entry in the Standard Ada Library sweepstakes
A large part of SAL provides math operations for kinematics and dynamics of masses in 3
dimensional space. Cartesian vectors, quaternions, orthonormal rotation matrices, moments of inertia,
forces, acceleration, velocity are supported, in 3 and 6 degrees of freedom (translation and rotation).
This library has been used for both robotics and satellite simulation.
<http://stephe-leake.org/ada/sal.html>
Emacs Ada mode: indentation, navigation, interface to GNAT tools for Emacs,
<http://stephe-leake.org/emacs/ada-mode/emacs-ada-mode.html>
An info version of the Ada 2005 and
OpenToken: LALR parser generator
<http://stephe-leake.org/ada/opentoken.html>
http://stephe-leake.org/

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

NBAda

NBAda is a library of lock-free data structure and algorithms for Ada. NBAda is released under the
GNU general public license version 2 or later.
Development tree for the structures is at <http://gitorious.org/nbada>
See:  http://www.gidenstam.org/Ada/Non-Blocking

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Matreshka

Matreshka is an Ada framework to help develop information systems. It includes:

- League --- a rich set of reusable core components to develop Ada applications. Its main purpose is to provide a high level abstraction tool for localization, internationalization and globalization of applications, as well as a portable interface to different operating systems. It contains many other useful features, among them advanced calendrical calculations, regular expressions, and JSON support to process and generate data in JSON format.

- XML processor --- provides the capability of manipulating XML streams and documents; including:
  - SAX reader to read XML streams and documents; it supports both XML1.0 (Fifth Edition), XML1.1 (Second Edition), Namespaces in XML and XML Base specifications.
  - SAX writer to generate XML streams and documents from applications.
  - XML Catalogs resolver

- Web framework
  - The FastCGI module assists with developing server side applications completely in Ada and using them with standard HTTP servers.
  - The SOAP module provides implementation of SOAP 1.2 protocol specification and assists in developing Web Services in Ada. This module includes implementation of standard security services:
  - WSDL to Ada translator

- SQL database access provides a simple generic API for accessing SQL databases. Supported databases include MYSQL, Oracle, PostgreSQL, SQLite 3, and Interbase/Firebird

- Ada Modeling Framework provides implementation of OMG’s Meta Object Facility (MOF) written completely in Ada. Extension modules are provided to assist in the analysis and modification of
  - UML models and their extensions: MOF Extensions, OCL models, UML Testing Profile
  - Diagram Definition

http://forge.adu-ru.org/matreshka (site updated 2013)
Commercial support of Matreshka is provided by QtAda

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

USAFA

Professor Martin Carlisle at the US Air Force Academy continues to develop free software for use by the computer science community. Although previously known for tools specifically for Ada programmers (in particular A#, AdaGIDE, and RAPID), his more recent developments have targeted the computer science education and computer security audiences. The newest tools, RAPTOR and IRONSIDES, have Ada inside and are developed using AdaGIDE, GNAT, SPARK Ada and A#. RAPTOR is a flowchart-based programming environment useful for teaching introductory computer science and is taught in at least 22 countries.

IRONSIDES is an authoritative DNS server implemented in SPARK Ada using formal methods to prove the absence of many major categories of security vulnerabilities. (last updated 2013)

See:  http://ironsides.martincarlisle.com
http://raptor.martincarlisle.com
http://adagide.martincarlisle.com
http://www.martincarlisle.com/ada_stuff.html
http://asharp.martincarlisle.com
http://rapid.martincarlisle.com

CONTACT:  Martin C. Carlisle, Professor of Computer Science, US Air Force Academy
carlislem@acm.org
High integrity software must not only meet correctness and performance criteria but also satisfy stringent safety and/or security demands, typically entailing certification against a relevant standard.

A significant factor affecting whether and how such requirements are met is the chosen language technology and its supporting tools: not just the programming language(s) but also languages for expressing specifications, program properties, domain models, and other attributes of the software or overall system.

HILT 2013 will provide a forum for experts from academia/research, industry, and government to present their latest findings in designing, implementing, and using language technology for high integrity software.


---

### FEATURED SPEAKERS

**Model Checking: Past, Present, and Future**

EDMUND M. CLARKE  
Carnegie Mellon University  
Electrical and Computer Engineering (ECE)

**Formal Methods: An Industrial Perspective**

JEANNETTE WING  
Microsoft Research

**Building Confidence in System Behavior**

JOHN GOODENOUGH  
Carnegie Mellon University  
Software Engineering Institute (SEI)

**Up and Out: Scaling Formal Analysis Using Model-Based Engineering**

MICHAEL WHALEN  
University of Minnesota

---

### CORPORATE SPONSORS

**PLATINUM LEVEL**

**Microsoft Research**

**GOLD LEVEL**

**VEROCIEL**

**SILVER LEVEL**

**Ellidiss Software**

**BASIC LEVEL**

**LDRA**

**MathWorks**
**TECHNICAL PROGRAM / November 12–14**

**TUESDAY**

9:00 AM–10:30 AM  
Greetings  
SIGAda and Conference Officers  

Keynote Address  
Edmund Clarke, CMU/ECE  
Model Checking: Past, Present, and Future  

10:30 AM–11:00 AM Break / Exhibits  

11:00 AM–12:30 PM  
Panel on Underlying Formal Verification Technologies  
Topics to be covered: Model Checking, SAT Solvers and SMT Solvers, Static Analysis and Abstract Interpretation, Coq-Based Proofs  

Sponsor Presentation  
12:30 PM–2:00 PM Break / Exhibits  

2:00 PM–3:30 PM  
Formal Verification Toolsets  
J. Hendrix  
SAW: The Software Analysis Workbench  
A. Hawthorn  
Optimizing Development and Verification Effort with SPARK 2014  
Z. Zhang  
Towards the Formalization of SPARK 2014 Semantics with Explicit Run-time Checks Using Coq  

3:30 PM–4:00 PM Break / Exhibits  

4:00 PM–5:30 PM  
High-Integrity Parallel Programming  
Panel on Safe, Efficient Parallel Programming  
Topics to be covered: Real-Time Programming on Accelerator Many-Core Processors, Bringing Parallel Programming to the SPARK Verifiable Subset of Ada, Deadlock Detection for Ada 2012  

Sponsor Presentation  
5:30 PM–6:00 PM Break  

6:00 PM–10:00 PM  
Social Event / Dinner  

**WEDNESDAY**

9:00 AM–10:30 AM  
Announcements  
SIGAda Awards  
Ricky E. Sward, Past SIGAda Chair  

Invited Talk  
Michael Whalen, University of Minnesota  
Up and Out: Scaling Formal Analysis Using Model-Based Engineering  

10:30 AM–11:00 AM Break / Exhibits  

11:00 AM–12:30 PM  
Model-Based Integration and Code Generation  
D. Ward  
An Approach to Integration of Complex Systems: The SAVI Virtual Integration Process (industrial presentation)  
M. Beeby  
Using Autocode Generators for Avionics Systems and Maintaining Compliance to DO-178 and DO-331 (industrial presentation)  

Industrial Presentation  
12:30 PM–2:00 PM Break / Exhibits  

2:00 PM–3:30 PM  
Keynote Address  
John Goodenough, CMU/SEI  
Building Confidence in System Behavior  

3:30 PM–4:00 PM Break  

4:00 PM–5:30 PM  
Architecture-Level Design Languages and Compositional Verification  
A. Murugesan  
Compositional Verification of a Medical Device System  
B. Larson  
Illustrating the AADL Error Modeling Annex (v. 2) Using a Simple Safety-Critical Medical Device  

Sponsor Presentation  
5:30 PM–7:00 PM Break  

7:00 PM–10:00 PM  
Workshops / Birds-of-a-Feather Sessions  

**THURSDAY**

9:00 AM–10:30 AM  
Announcements  
Best Paper and Student Paper Awards  
Tucker Taft, HILT 2013 Program Chair  

Keynote Address  
Jeannette Wing, Microsoft Research  
Formal Methods: An Industrial Perspective  

10:30 AM–11:00 AM Break  

11:00 AM–12:00 PM  
Panel on Approaches to Software Safety and Security  
Topics to be covered: Secure Coding, Static Analysis, Formal Verification, Automatic vs. Interactive Program Verification  

12:00 PM–12:30 PM  
Announcements  
(Ada-Europe 2014, SIGAda 2014)  
Closing Remarks and Conference Adjournment  

To register online, and for more information and updates, visit  
www.sigada.org/conf/hilt2013  

---  

PRE-CONFERENCE TUTORIALS / November 10–11

**SUNDAY**

SA1—Morning / 9:00 AM–12:30 PM  
Ed Colbert / Absolute Software  
Ada 2012 Part 1  

SA2—Morning / 9:00 AM–12:30 PM  
Tucker Taft / AdaCore  
Proving Safety of Parallel/Multi-Threaded Programs  

SP1—Afternoon / 2:00 PM–5:30 PM  
Ed Colbert / Absolute Software  
Ada 2012 Part 2  

SP2—Afternoon / 2:00 PM–5:30 PM  
Ethan K. Jackson / Microsoft Research  
Formula 2.0: A Language for Formal Specifications; A Tool for Automated Analysis  

---  

Photo of Pittsburgh by User:Derek.cashman courtesy of Wikimedia Commons
VENUE / HOTEL

HILT 2013 will be held at the Wyndham Pittsburgh University Center, www.tinyurl.com/HILT2013-hotel.

The Wyndham Pittsburgh University Center has reserved a block of rooms for the HILT 2013 conference. The conference rate is $119 for single, double, triple, or quadruple occupancy rooms. This includes complimentary wireless Internet in all the guest rooms and free parking for guests. State and local taxes will be added per night. All reservations must be guaranteed by credit card. Please also visit www.sigada.org/conf/hilt2013/hotel-rates.html and www.acm.org/sig_volunteer_info/whyhotel.htm for additional details. For directions from Pittsburgh International Airport (PIT) as well as information about transportation to and from the airport via Super Shuttle, Taxi Services, and Bus Service, please visit www.pitairport.com, or www.pitairport.com/public_transportation for public transport options.

SPONSORS / EXHIBITORS

HILT 2013 will include vendor participation, featuring presentations on their products and services during main sessions. For specific information, please contact the Exhibits Chair, Greg Gicca, gicca@verocel.com.

GRANTS TO EDUCATORS

As in past years, SIGAda is offering grants to educators to attend the conference. Grants cover the registration and tutorial fees; members of the GNAT Academic Program may be eligible for travel funds from AdaCore. Apply by e-mail, no later than October 14, 2013. Grant program details are available from the conference website or Professor Michael B. Feldman, mfeldman@gwu.edu.

WORKSHOPS / BIRDS-OF-A-FEATHER

To propose a focused workshop or informal Birds-of-a-Feather session related to the conference theme, please contact the Workshops Chair, John W. McCormick, mccormick@cs.uni.edu.

REGISTRATION FEES

<table>
<thead>
<tr>
<th>Conference (Full)</th>
<th>Conference (One Day)</th>
<th>Tutorial (Full Day)</th>
<th>Tutorial (Half Day)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member of ACM, SIGAda, or cooperating organization:</td>
<td>Member of ACM, SIGAda, or cooperating organization:</td>
<td>Member of ACM, SIGAda, or cooperating organization:</td>
<td>Member of ACM, SIGAda, or cooperating organization:</td>
</tr>
<tr>
<td>$575 early / $725 after Oct. 21</td>
<td>$325 early / $325 after Oct. 21</td>
<td>$310 early / $370 after Oct. 21</td>
<td>$155 early / $185 after Oct. 21</td>
</tr>
<tr>
<td>Non-members:</td>
<td>Non-members:</td>
<td>Non-members:</td>
<td>Non-members:</td>
</tr>
<tr>
<td>$875 early / $975 after Oct. 21</td>
<td>$325 early / $325 after Oct. 21</td>
<td>$420 early / $470 after Oct. 21</td>
<td>$210 early / $235 after Oct. 21</td>
</tr>
<tr>
<td>Full-time Student: $50</td>
<td>Full-time Student: $25</td>
<td>Full-time Student: $30</td>
<td>Full-time Student: $15</td>
</tr>
</tbody>
</table>

For early registration rates, register online by October 21 at http://sigada.org/conf/hilt2013/register/index.html

CONFERENCE TEAM

Conference Chair / Local Arrangements Chair
Jeff Boleng, Software Engineering Institute / JLBoleng@SEI.CMU.edu

Program Chair / Proceedings Chair
Tucker Taft, AdaCore / taft@adacore.com

Treasurer
Ricky E. Sward, The MITRE Corporation / rsward@mitre.org

Workshops Chair / Tutorials Chair
John W. McCormick, University of Northern Iowa / mccormick@cs.uni.edu

Webmaster
Clyde Roby, Institute for Defense Analyses / clyderoby@acm.org

Exhibits and Sponsorships Chair
Greg Gicca, Verocel / gicca@verocel.com

Registration Chair / Academic Community Liaison
Michael B. Feldman, George Washington University (Ret.) / mfeldman@gwu.edu

Publicity Chair
Alok Srivastava, TASC Inc. / alok.srivastava@tasc.com

Logo Designer
Weston Pan, Raytheon Space and Airborne Systems

SIGAda Officers

Chair
David Cook, Stephen F. Austin State University / cookda@sfasu.edu

Vice Chair
Tucker Taft, AdaCore / taft@adacore.com

Secretary / Treasurer
Clyde Roby, Institute for Defense Analyses / clyderoby@acm.org

International Representative
Dirk Craeynest, KU Leuven, Department of Computer Science / dirk.craeynest@cs.kuleuven.be

Past Chair
Ricky E. Sward, The MITRE Corporation / rsward@mitre.org

ACM Ada Letters Editor
Alok Srivastava, TASC Inc. / alok.srivastava@tasc.com

Ada Letters, August 2013

143

Volume XXXIII, Number 2
Come to HILT 2013 and discover the latest developments in language technology for safe, secure, and reliable software.

Listen to and meet world-renowned experts in the field, see how industry is converting research into practical experience, and learn both the challenges confronting high-integrity software and the solutions available to address them.

REGISTER ONLINE BY OCTOBER 21 FOR THE LOWEST REGISTRATION RATES
Call for Papers

19\textsuperscript{th} International Conference on Reliable Software Technologies – Ada-Europe 2014

23-27 June 2014, Paris, France

http://www.adaeurope.org/conference2014

General Information

The 19\textsuperscript{th} International Conference on Reliable Software Technologies – Ada-Europe 2014 will take place in Paris, France. As per its traditional style, the conference will span a full week, including, from Tuesday to Thursday, three days of parallel scientific, technical and industrial programs, along with tutorials and workshops on Monday and Friday.

Schedule

<table>
<thead>
<tr>
<th>Date</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 December 2013</td>
<td>Submission of regular papers, tutorial and workshop proposals</td>
</tr>
<tr>
<td>19 January 2014</td>
<td>Submission of industrial presentation proposals</td>
</tr>
<tr>
<td>16 February 2014</td>
<td>Notification of acceptance to all authors</td>
</tr>
<tr>
<td>16 March 2014</td>
<td>Camera-ready version of regular papers required</td>
</tr>
<tr>
<td>18 May 2014</td>
<td>Industrial presentations, tutorial and workshop material required</td>
</tr>
</tbody>
</table>

Topics

The conference has over the years become a leading international forum for providers, practitioners and researchers in reliable software technologies. The conference presentations will illustrate current work in the theory and practice of the design, development and maintenance of long-lived, high-quality software systems for a challenging variety of application domains. The program will allow ample time for keynotes, Q&A sessions and discussions, and social events. Participants include practitioners and researchers representing industry, academia and government organizations active in the promotion and development of reliable software technologies.

Topics of interest to this edition of the conference include but are not limited to:

- **Multicore and Manycore Programming**: Predictable Programming Approaches for Multicore and Manycore Systems, Parallel Programming Models, Scheduling Analysis Techniques.
- **Theory and Practice of High-Integrity Systems**: Challenges from Mixed-Criticality Systems; Medium to Large-Scale Distribution, Fault Tolerance, Security, Reliability, Trust and Safety, Languages Vulnerabilities.
- **Software Architectures**: Design Patterns, Frameworks, Architecture-Centred Development, Component-based Design and Development.
- **Software Quality**: Quality Management and Assurance, Risk Analysis, Program Analysis, Verification, Validation, Testing of Software Systems.
- **Mainstream and Emerging Applications**: Manufacturing, Robotics, Avionics, Space, Health Care, Transportation, Cloud Environments, Smart Energy systems, Serious Games, etc.
- **Experience Reports in Reliable System Development**: Case Studies and Comparative Assessments, Management Approaches, Qualitative and Quantitative Metrics.
- **Experiences with Ada and its Future**: Reviews of the Ada 2012 new language features, implementation and use issues, positioning in the market and in the software engineering curriculum, lessons learned on Ada Education and Training Activities with bearing on any of the conference topics.
Program Committee

Mario Aldea, Universidad de Cantabria, Spain
Ted Baker, US National Science Foundation, USA
Johann Blieberger, Technische Universität Wien, Austria
Bernd Burgstaller, Yonsei University, Korea
Maryline Chetto, University of Nantes, France
Liliana Cucu, INRIA, France
Christian Fraboulet, ENSEEIHT, France
Laurent George, ECE Paris, France
Xavier Grave, CNRS, France
Emmanuel Grolleau, ENSMA, France
Jérôme Hugues, ISAE, France
Albert Llemosí, Universitat de les Illes Balears, Spain
Kristina Lundqvist, Mälardalen University, Sweden
Francesco Mazzanti, ISTI-CNR, Italy
John McCormick, University of Northern Iowa, USA
Stephen Michell, Maurya Software, Canada
Laurent Pautet, Telecom ParisTech, France
Luis Miguel Pinho, CISTER/ISEP, Portugal
Erhard Piekarek, Universität Stuttgart, Germany
Juan A. de la Puente, Universidad Politécnica de Madrid, Spain
Jorge Real, Universitat Politècnica de València, Spain
José Ruiz, AdaCore, France
Sergio Sáez, Universidad Politécnica de Valencia, Spain
Amand Skavhug, NTNU, Norway
Yves Sorel, INRIA, France
Tucker Taft, AdaCore, USA
Theodor Tempelmeier, University of Applied Sciences, Germany
Elena Troubitsyna, Abo Akademi University, Finland
Tullio Vardanega, University of Padova, Italy
Juan Zamorano, Universidad Politécnica de Madrid, Spain

Industrial Committee

Jacob Sparre Andersen, JSA Consulting, Denmark
Roger Brandt, Tellia, Sweden
Ian Broster, Rapita Systems, UK
Jorgen Bundgaard, Rambøll, DK
Dirk Craeynest, Ada-Belgium & KU Leuven, Belgium
Peter Dencker, ETAS, Germany
Ismael Lafoz, Airbus, Spain
Maria del Carmen Lomba Sorrondegui, GMV, Spain
Ahlan Marriott, White Elephant, CH
Robin Messer, Altran-Praxis, UK
Quentin Ochem, AdaCore, France
Seen Palm, Terma, Denmark
Paolo Panaroni, Intecs, Italy
Paul Parkinson, Wind River, UK
Ana Rodriguez, Silver-Atena, Spain
Jean-Pierre Rosen, Adalog, France
Alok Srivastava, TASC, USA
Clas Stellwag, Elektrobit, Germany
Jean-Loup Terrailhon, European Space Agency, Netherlands
Rod White, MBDA, UK

Call for Regular Papers

Authors of regular papers which are to undergo peer review for acceptance are invited to submit original contributions. Paper submissions shall not exceed 14 LNCS-style pages in length. Authors shall submit their work via EasyChair following the relevant link on the conference web site. The format for submission is solely PDF.

Proceedings

The conference proceedings will be published in the Lecture Notes in Computer Science (LNCS) series by Springer, and will be available at the start of the conference. The authors of accepted regular papers shall prepare camera-ready submissions in full conformance with the LNCS style, not exceeding 14 pages and strictly by March 16, 2014. For format and style guidelines authors should refer to http://www.springer.de/comp/lncs/authors.html. Failure to comply and to register for the conference by that date will prevent the paper from appearing in the proceedings.

The CORE ranking (dated 2008) has the conference in class A. The CiteSeerX Venue Impact Factor had it in the top quarter. Microsoft Academic Search has it in the top third for conferences on programming languages by number of citations in the last 10 years. The conference is listed in DBLP, SCOPUS and Web of Science Conference Proceedings Citation index, among others.

Awards

Ada-Europe will offer honorary awards for the best regular paper and the best presentation.

Call for Industrial Presentations

The conference seeks industrial presentations which deliver value and insight but may not fit the selection process for regular papers. Authors are invited to submit a presentation outline of exactly 1 page in length by January 19, 2014. Submissions shall be made via EasyChair following the relevant link on the conference web site. The Industrial Committee will review the submissions and make the selection. The authors of selected presentations shall prepare a final short abstract and submit it by May 18, 2014, aiming at a 20-minute talk. The authors of accepted presentations will be invited to submit corresponding articles for publication in the Ada User Journal, which will host the proceedings of the Industrial Program of the Conference. For any further information please contact the Industrial Chair directly.

Call for Tutorials

Tutorials should address subjects that fall within the scope of the conference and may be proposed as either half- or full-day events. Proposals should include a title, an abstract, a description of the topic, a detailed outline of the presentation, a description of the presenter’s lecturing expertise in general and with the proposed topic in particular, the proposed duration (half day or full day), the intended level of the tutorial (introductory, intermediate, or advanced), the recommended audience experience and background, and a statement of the reasons for attending. Proposals should be submitted by e-mail to the Tutorial Chair. The authors of accepted full-day tutorials will receive a complimentary conference registration as well as a fee for every paying participant in excess of 5; for half-day tutorials, these benefits will be accordingly halved. The Ada User Journal will offer space for the publication of summaries of the accepted tutorials.

Call for Workshops

Workshops on themes that fall within the conference scope may be proposed. Proposals may be submitted for half- or full-day events, to be scheduled at either end of the conference week. Workshop proposals should be submitted to the General Chair. The workshop organizer shall also commit to preparing proceedings for timely publication in the Ada User Journal.

Call for Exhibitors

The commercial exhibition will span the three days of the main conference. Vendors and providers of software products and services should contact the General Chair for information and for allowing suitable planning of the exhibition space and time.

Grants for Reduced Student Fees

A limited number of sponsored grants for reduced fees is expected to be available for students who would like to attend the conference or tutorials. Contact the General Chair for details.
ACM: KNOWLEDGE, COLLABORATION & INNOVATION IN COMPUTING

Uniting the world's computing professionals, researchers and educators to inspire dialogue, share resources and address the computing field's challenges in the 21st Century.

Association for Computing Machinery
Advancing Computing as a Science & Profession
www.acm.org/learnmore
acmqueue is guided and written by distinguished and widely known industry experts. The newly expanded site also offers more content and unique features such as planetqueue blogs by queue authors who “unlock” important content from the ACM Digital Library and provide commentary; videos; downloadable audio; roundtable discussions; plus unique acmqueue case studies.

acmqueue provides a critical perspective on current and emerging technologies by bridging the worlds of journalism and peer review journals. Its distinguished Editorial Board of experts makes sure that acmqueue's high quality content dives deep into the technical challenges and critical questions software engineers should be thinking about.

Visit today!
http://queue.acm.org/