CISTER
Research Centre in
Real-Time & Embedded
Computing Systems

Conference Paper

Safe Parallelism: Compiler Analysis Techniques for Ada and OpenMP

Sara Royuela
Xavier Martorell
Eduardo Quiñones
Luis Miguel Pinho*

*CISTER Research Centre

CISTER-TR-180505

2018/06/18

Safe Parallelism: Compiler Analysis Techniques for Ada and OpenMP

Sara Royuela, Xavier Martorell, Eduardo Quiñones, Luis Miguel Pinho*

*CISTER Research Centre

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail: lmp@isep.ipp.pt

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

Abstract

There is a growing need to support parallel computation in Ada to cope with the performance requirements of the most advanced functionalities of safety-critical systems. In that regard, the use of parallel programming models is paramount to exploit the benefits of parallelism. Recent works motivate the use of OpenMP for being a de facto standard in high-performance computing for programming shared memory architectures. These works address two important aspects towards the introduction of OpenMP in Ada: the compatibility of the OpenMP syntax with the Ada language, and the interoperability of the OpenMP and the Ada runtimes, demonstrating that OpenMP complements and supports the structured parallelism approach of the tasklet model. This paper addresses a third fundamental aspect: functional safety from a compiler perspective. Particularly, it focuses on race conditions and considers the fine-grain and unstructured capabilities of OpenMP. Hereof, this paper presents a new compiler analysis technique that: 1) identifies potential race conditions in parallel Ada programs based on OpenMP or Ada tasks or both, and 2) provides solutions for the detected races.

Safe Parallelism: Compiler Analysis Techniques for Ada and OpenMP

Sara Royuela[1], Xavier Martorell[1], Eduardo Quinones[1], and Luis Miguel Pinho[2]

[1] Barcelona Supercomputing Center
sara.royuela,xavier.martorell,eduardo.quinones@bsc.es
[2] CISTER/INESC-TEC, ISEP, Polytechnic Institute of Porto lmp@isep.ipp.pt

Abstract. There is a growing need to support parallel computation in Ada to cope with the performance requirements of the most advanced functionalities of safety-critical systems. In that regard, the use of parallel programming models is paramount to exploit the benefits of parallelism. Recent works motivate the use of OpenMP for being a de facto standard in high-performance computing for programming shared memory architectures. These works address two important aspects towards the introduction of OpenMP in Ada: the compatibility of the OpenMP syntax with the Ada language, and the interoperability of the OpenMP and the Ada runtimes, demonstrating that OpenMP complements and supports the structured parallelism approach of the tasklet model.
This paper addresses a third fundamental aspect: functional safety from a compiler perspective. Particularly, it focuses on race conditions and considers the fine-grain and unstructured capabilities of OpenMP. Hereof, this paper presents a new compiler analysis technique that: 1) identifies potential race conditions in parallel Ada programs based on OpenMP or Ada tasks or both, and 2) provides solutions for the detected races.

1 Introduction

The parallel computation paradigm has irrupted in all computing domains, including safety-critical systems, to cope with the increasing need of higher levels of performance to implement advanced functionalities (e.g. autonomous driving [1]). Despite the clear benefits of parallel computation, it also introduces hazards regarding safety and reliability, crucial concepts for critical systems.

This trend has also arrived to Ada [2], a language used in safety-critical and high-security domains, and designed to keep safeness. Two main (and complementary) research lines are tackling the extension of Ada to support parallelism: a) the simple yet powerful *tasklet* model [3–7] that, based on a fully strict fork-join model, is able to exploit structured parallelism on shared memory architectures, and b) the incorporation of OpenMP into Ada, to efficiently exploit structured and unstructured parallelism [8, 9] (Section 3 provides more details of the two approaches). This paper is framed in the latter case.

OpenMP [10] has become a de facto standard for shared-memory systems as a result of being successfully used for decades in high-performance computing

(HPC). The model has recently gained much attention in the embedded field as it addresses key issues for such systems: a) the coupling of a main processor to one or more accelerators, b) the tasking model, capable of expressing fine-grain and highly-dynamic parallelism, and c) its time predictability properties.

Current works have addressed two fundamental pillars towards the adoption of OpenMP into Ada: 1) the compatibility of the OpenMP syntax with the Ada language [8], and 2) the compatibility of the OpenMP and the Ada execution models [9]. The former proves that OpenMP provides an equivalent and compatible interface to that of the tasklet model, guaranteeing the same safety features. The latter analyses the interoperability between the OpenMP and the Ada runtimes with a threefold objective: a) fulfill both specifications without jeopardizing safety, b) use OpenMP as an implementation of the tasklet model, and c) incorporate OpenMP directives in Ada programs. This paper focuses on the third fundamental and yet unaddressed pillar: *ensure functional safety in the presence of parallel computation from a compiler perspective.*

The most frequent errors in parallel computation are deadlocks, race conditions and starvation, among others. Most of these errors can effectively be identified and solved using compiler techniques. Furthermore, compilers may always take a conservative approach overreacting when the solution is not decidable at compile time. In this regard, recent works propose to extend the Ada [4] and the OpenMP [11] specifications to include new aspects and directives, respectively, to address race conditions and deadlocks (among other issues) when whole-program analysis is not possible.

In particular, this paper focuses on race conditions because this kind of error is closely related with the exploitation of the fine-grained and unstructured parallel capabilities of the OpenMP tasking model. In this context, this paper advances one step towards safe parallel execution in Ada by proposing a new compiler analysis technique that: 1) allows identifying race conditions that can potentially appear in Ada programs parallelized with both OpenMP and Ada tasks, and 2) provides solutions for the detected races. The specific contributions of the paper are the following:

1. A control flow graph that represents the semantics of Ada and OpenMP, and allows the analysis of a program combining, or not, such languages.
2. The adaptation of OpenMP compiler analysis techniques developed for sequential languages (C, C++ and Fortran) to the Ada concurrent language.
3. A compiler method based on techniques for enhancing the programmability of OpenMP, that: 1) detects race conditions in Ada programs using or not OpenMP, and 2) provides users with directions to solve the errors.

2 Background

2.1 The Ada concurrent model

The Ada concurrency model is based on the notion of *task*, a unit of concurrency that represents an independent thread of control. All, the tasks and the

mechanisms for inter-task communication and synchronization, are introduced at language level in order to allow building safer programs. As an illustration, Ada 95 [12] introduced *protected objects* to allow controlling how data is accessed, thus eliminating race conditions.

Additionally, in 1997, Burns et al. introduced the Ravenscar profile [13], a subset of the Ada programming language that allows high integrity applications to be analyzed for their timing properties by pursuing three main goals: 1) ensuring predictable execution, 2) simplifying the runtime support, and 3) eliminating constructs with high overhead. The limitations imposed by the Ravenscar profile have an inevitable impact in the complexity of correctness analyses, e.g. tasks can only communicate through shared objects (tasks entries are not allowed, so the *rendezvous* mechanism cannot be used), tasks are assumed to be non-terminating, and tasks and protected objects cannot be dynamically allocated.

Along the same lines, SPARK [14], a language that subsets Ada to enable the formal verification of programs, eliminates race conditions by forcing any global object referenced from a task to be marked as `Part_Of` that task, or be a *synchronized* object[3] [15].

2.2 The OpenMP tasking model

The tasking model appears in OpenMP 3.0 from the need of productively implementing certain types of parallelism: unbounded loops, recursion, unstructured parallelism, etc. It is based on the notion of *task*[4], a specific instance of executable code and its data environment, generated when a thread encounters certain language construct (e.g. `task`, `taskloop` and `parallel`). Other constructs, such as `taskwait` and `depend`, allow for tasks synchronization. The runtime system is responsible of creating and executing the tasks, which can be executed immediately after creation, or deferred. This depends on two factors: 1) task scheduling constraints (e.g. dependencies with other tasks described in the `depend` clauses), and 2) thread availability.

The uncertainty introduced by the tasking model regarding when the tasks are executed represents a challenge with respect to determining which portions of code are concurrent. Furthermore, the relaxed-consistency memory model of OpenMP (allowing `private`, `firstprivate`, `lastprivate` and `shared` attributes), and the way data-sharing attributes may be defined[5] add extra complexity for the user, reducing the programmability, and increasing the possibilities of introducing errors.

---

[3] Spark considers the following *synchronized* objects: protected objects, *atomic* objects (all accesses are atomic), and *suspension* objects (a kind of private semaphore).

[4] The term *task* in OpenMP is not related to Ada tasks. OpenMP tasks are lightweight parts of the code that can be executed in parallel by worker threads. In that regard, OpenMP tasks are very similar to Ada tasklets [16].

[5] OpenMP allows three ways to determine the data-sharing attributes: predetermined, implicitly determined, and explicitly determined. The first two kinds are defined by several rules in the specification, the latter requires explicit definition by the user.

3 Related work

In the last years, several works are leading the introduction of fine-grain parallelism in Ada. This is so due to the increasing demand of computational capabilities of the systems using such a programming language. There are two main approaches: 1) the implementation of a parallel model built in the Ada core language, named *tasklet* model [3–7], and 2) the introduction of OpenMP in pure Ada applications [8,9]. The latter is gaining attention lately due to several reasons: a) OpenMP is a mature parallel programming model, under continuous revision by an expert and experienced committee, b) OpenMP is flexible yet robust, allowing the definition of both structured and unstructured parallelism, as well as the use of heterogeneous architectures, and c) most compiler (e.g. GNU, Intel) and chip vendors in HPC (e.g. Intel, ARM, PowerPC, etc.) and the real-time domain (e.g. Kalray MPPA, TI Keystone II) support OpenMP.

In this context, different works have already explored the safety requirements necessary for OpenMP to be used in safety-critical environments, and they point to two main directions: time predictability and functional safety. About the former, the OpenMP tasking model has been proven to be analyzable regarding its time properties [17–20], thus valid to ensure that deadlines can be fulfilled. About the latter, different studies conclude that including some modifications in the OpenMP specification, as well as implementing some guidelines in OpenMP frameworks (including both the compiler and the runtime), may enable OpenMP programs to meet the correctness requirements of a safety-critical system [8,11].

This paper focuses on how the compiler can address functional safety. In this context, several compiler analysis techniques exist to check OpenMP programs for diverse errors, mainly deadlocks and race conditions. Among the former, Kroenig et al. developed a technique for detecting deadlocks in C/Pthreads programs [21] that can easily be applied to OpenMP because Pthreads mutexes (e.g. `pthread_mutex_lock`) are comparable to OpenMP locking routines (e.g. `omp_set_lock`). Among the latter, Ma et al. created a tool for detecting race conditions in OpenMP programs with a fixed number of threads [22], and Basupalli et al. developed a robust technique for detecting race conditions in OpenMP programs using affine constructs [23]. Finally, Royuela et al. developed a series of algorithms focused on the OpenMP tasking model to find incoherences in data-sharing and dependence clauses, as well as race conditions [24].

On another level, several methodologies exist to analyze Ada concurrent programs. These include two important aspects: 1) the representation used to describe the concurrent semantics of Ada programs, and 2) the technique used to implement analysis on top of a given representation. Regarding the former, the most common representations used for Ada analytics are Petri nets [25], control flow graphs [26], and different forms of task graphs such as program reachability graphs [27], real-time task digraphs [28] and system dependence nets [29]. Concerning the latter, most analysis techniques for Ada are based on model

checking[6], which allows the automatic verification of a system's correctness. In this sense, Faria et al. developed ATOS [30], a tool that automatically extracts a SPIN model [31] from an Ada program, as well as a set of desirable properties from a specification annotated by the user in the program, inspired by the SPARK annotation language. Resembling ATOS, GNATprove [32] is a formal verification tool for Ada, based on the GNAT compiler [33] and Meyer's *design by contract* paradigm [34]. These contracts must be explicitly stated by programmers as preconditions and postconditions for functions and procedures, and loop invariants, all in the syntax of Ada 2012.

4 Motivation

The Ada Reference Manual [35] distinguishes three kinds of errors: 1) those that can be detected at compile time, 2) those that can be detected at run time, and 3) those that do not need to be detected. The nature of Ada is to prevent users from making errors, providing a series of mechanisms for data synchronization and mutual exclusion, among others. Still, it is the responsibility of the programmers to use these mechanisms in order to avoid errors such as race conditions and deadlocks. Section 3 introduces some state-of-the-art techniques for correctness checking. On the one hand, model checking based techniques are very mature, although their usefulness depends on contracts that are also written by programmers, hence are liable to have errors. On the other hand, techniques based on petri-nets or reachability graphs mostly tackle deadlocks, because these representations do not describe data flow information, but states. Hence, there is a lack of static techniques for data race detection in Ada programs.

OpenMP also provides mechanisms for data synchronization and mutual exclusion, but the correct use of these mechanisms relies on the programmer. This is stated in the specification, when it says that *"application developers are responsible for correctly using the OpenMP API to produce a conforming program[7]"*. Still, many static and dynamic techniques have been developed for OpenMP correctness checking to enhance productivity in parallel programming, as we introduce in Section 3. Two of them are particularly interesting to us because, although developed to enhance the programmability of OpenMP, they are also useful to detect race conditions. The first technique, named *auto-scope*, automatically defines the scope of the variables in a task construct (i.e. the data-sharing clauses) [36], and the second technique, named *auto-deps*, discovers the dependencies among tasks (i.e. dependence clauses) [37]. If whole program analysis is possible, the only limitation of the algorithms concerns the use of third-party

[6] *Model checking* mechanisms allow exhaustively and automatically checking a given model regarding a given specification. Typically, hardware or software components are checked against safety requirements such as the absence of deadlocks and other critical states that can cause a system to crash.

[7] An OpenMP *conforming* program is that which follows all rules and restrictions of the OpenMP specification.

libraries which code is not visible. Anyhow, the algorithms are sound and, when a variable cannot be automatically determined, it is reported to the user.

Overall, despite the specification of both Ada and OpenMP do not require correctness checking mechanisms to ensure programs are free from errors, including those is fundamental to increase productivity in parallel programming. In that regard, we note a lack of mechanisms for detecting race conditions in Ada, which is particularly important in case of safety environments to ensure a correct operation of the system. This paper considers the algorithms developed for OpenMP and propose the adaptation of these to handle Ada semantics. With this, we are able to detect race conditions in pure Ada programs and in mixed Ada/OpenMP programs as well.

The work uses for this paper the Ada Ravenscar profile, due to its simpler concurrency model. This restriction is not related to the safety of the analysis, which is independent from the model, but to the complexity of the control flow graph that needs to be extracted and analysed. Section 5.4 provides information on how the approach extends to less restrictive models, being the goal that the approach is used with full Ada.

5 Proposal: compiler analysis for mixed Ada and OpenMP tasks

This section explains our proposal to solve race conditions in mixed Ada and OpenMP programs. It is structured as follows: first we present the singularities of Ada/OpenMP programs, then we show how we represent Ada/OpenMP programs, next we introduce the algorithm used to detect race conditions in such programs, and finally we show the results of applying the algorithm to a particular test case. For illustration purposes, we use the Ada application *Ravenscar*, defined in Section 7 of the Ada Ravenscar Profile Guide [38] as test case. The system modeled in this application includes a periodic process (*Regular_Producer*) that handles offers for a variable amount of workload (*Small_Whetstone*). When the requested workload exceeds a given threshold (*Due_Activation*), the excess load is processed by a sporadic process (*On_Call_Producer*). Additionally, interrupts may appear at any point (*External_Event_Server*), and different priorities are used to ensure preference among the different tasks.

Fig. 1 shows the HRT-HOOD[8] representation of the *Ravenscar* application. There, red dashed boxes represent tasks, blue dotted boxes represent packages with functions and procedures, and yellow double-lined boxes represent protected objects with entries and procedures. The *Ravenscar* code illustrates the expressiveness of the Ravenscar profile, for it includes several features of Ada that are of our interest: protected objects, other shared data, synchronous and asynchronous synchronizations, etc.

To exemplify how the analysis handles the two levels of parallelism (Ada coarse grain tasks and OpenMP fine grain tasks), we have introduced an OpenMP

[8] Hard Real-Time Hierarchical Object-Oriented Design (HRT-HOOD) is an object-based structured design method for hard real-time systems [39].

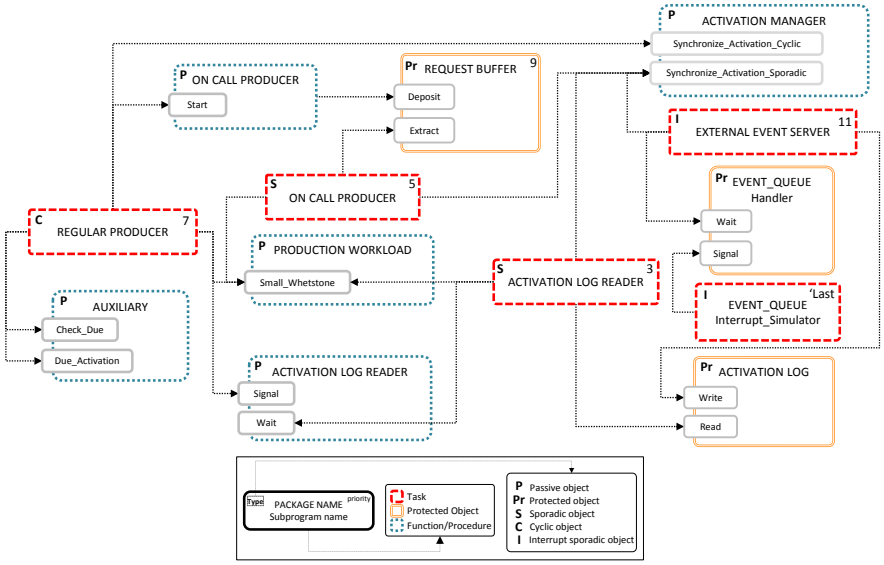

Fig. 1: HRT-HOOD representation of the *Ravenscar* application.

computation in the *Small_Whetstone* procedure, which turns into the entry point of a sensor fusion operation. This new functionality is described in Fig. 2 using the syntax proposed in Ada to use OpenMP [8]. There, the `parallel` construct initiates parallel execution by creating a team of threads. Then, the `single` construct indicates that only one thread will execute the inner statements. Finally, the `taskloop` construct indicates that the iterations of the most outer loop are split into chunks that can be executed in parallel by the threads in the current team using OpenMP tasks. In this implementation, the parameter of the *Small_Whetstone* procedure indicates the operation to carry out: 1 means reading sensor A, 2 means reading sensor B, and 3 means fusing the two sensors by adding up its values. Sensor A is read periodically from *Regular_Producer*, sensor B is read sporadically from *On_Call_Producer*, and the fusion is performed sporadically from *Activation_Log_Reader*.

5.1 Mixing Ada and OpenMP

As introduced previously, pure Ada programs define concurrency by means of tasks, while OpenMP creates parallelism by means of the `parallel` construct, and distributes it by means of worksharing and tasking constructs. When both languages are used together, concurrency may be defined at multiple levels: between Ada tasks, between OpenMP tasks, and between Ada and OpenMP tasks. Table 1 summarizes our approach to resolve race conditions in each case.

Ada protected objects are a robust and lightweight mechanism for mutual exclusion and data synchronization. For this reason, they are to be used whenever possible to solve race conditions, i.e. when race conditions occur between

```
1 package body Production_Workload is
2   type Dim is range 1 .. 512;
3   type M is array (Dim, Dim) of Float;
4   M_A, M_B, M_C: M;
5
6   procedure Read_Sensor_A is begin
7     pragma OMP (parallel);
8     pragma OMP (single);
9     pragma OMP (taskloop);
10    for I in Dim loop
11      for J in Dim loop
12        M_A(I,J) := sensor(1, I, J);
13      end loop;
14    end loop;
15  end Read_Sensor_A;
16
17  procedure Read_Sensor_B is begin
18    pragma OMP (parallel);
19    pragma OMP (single);
20    pragma OMP (taskloop);
21    for I in Dim loop
22      for J in Dim loop
23        M_B(I,J) := sensor(2, I, J);
24      end loop;
25    end loop;
26  end Read_Sensor_B;
27
28  procedure Fuse_Sensors is
29  begin
30    pragma OMP (parallel);
31    pragma OMP (single);
32    pragma OMP (taskloop);
33    for I in Dim loop
34      for J in Dim loop
35        M_C(I,J) := M_A(I,J) + M_B(I,J);
36      end loop;
37    end loop;
38  end Fuse_Sensors;
39
40  procedure Small_Whetstone
41            (Workload: Positive) is
42  begin
43    case Workload is
44      when 1 => Read_Sensor_A;
45      when 2 => Read_Sensor_B;
46      when 3 => Fuse_Sensors;
47      when others => null;
48    end case;
49  end Small_Whetstone;
50
51 end Production_Workload;
```

Fig. 2: OpenMP code inserted in the *Production_Workload* package of the *Ravenscar* application.

Race condition between		Solution
Ada tasks		Ada mechanisms: protected object
Ada and OpenMP tasks		
OpenMP tasks	different binding regions	
	same binding region	OpenMP mechanisms: * Synchronization constructs and clauses: `taskwait`, `barrier`, `depend` * Mutual exclusion constructs: `critical`, `atomic` * Data-sharing attributes: `private`, `firstprivate`, `lastprivate`

Table 1: Solutions for race conditions in an Ada/OpenMP application.

Ada tasks, between Ada and OpenMP tasks, and between OpenMP tasks that belong to different binding regions[9]. The last case is particularly interesting because in C/C++/Fortran OpenMP[10] programs, tasks belonging to different binding regions cannot be concurrent unless there are nested parallel regions. Tasks in such situation cannot be synchronized, and only data synchronization is available via the flush operation, a highly unrecommended mechanism when safety is essential due to the difficulty of analyzing its behavior. The extra layer

[9] In OpenMP, the *binding region* is the enclosing region that determines the execution context. The binding region of a task is the innermost enclosing parallel region.

[10] The OpenMP API is an specification for defining parallelism in C, C++ and Fortran programs.

of concurrency introduced by Ada unlocks this scenario, hence only protected objects are safe enough to synchronize data. Finally, to exploit the flexibility of OpenMP, race conditions between OpenMP tasks that belong to the same binding region are to be solved using OpenMP mechanisms: mutual exclusion constructs (i.e. `atomic` and `critical` constructs), synchronization constructs (e.g. `taskwait` and `barrier`), synchronization clauses (i.e. `depend`) and data-sharing clauses (e.g. `private`, `firstprivate` and `lastprivate`).

5.2 Representation of an Ada/OpenMP program

As introduced in Section 3, several representations allow expressing the semantics of an Ada program (e.g. reachability graphs, Petri nets, control flow graphs, etc.). However, some representations are not suitable for our purpose, for instance Petri nets and reachability graphs, because these express states whereas data flow information is hidden. Furthermore, these representations have other limitations such as the state explosion problem, and the inability of representing recursive programs. Hence, to represent the behavior of an Ada/OpenMP program we use the classic control flow graph (CFG) representation extended to support Ada concurrency and OpenMP parallelism. Our graph draws from the parallel control flow graph for C/C++ and OpenMP/OmpSs [40] developed by Royuela et al. [24], and the control flow graph for Ada developed by Fechete et al. [26].

To ease the reading we show the CFGs of the original *Ravenscar* application and the new OpenMP code separately, in Fig. 3 and Fig. 4 respectively (the complete CFG of the Ada code is displayed in Appendix A). The CFG of the original *Ravenscar* code shows the code executed at elaboration time (top of the figure), and the Ada code run during the execution of the program (rest of the figure). Each partial CFG represents a task (*Regular_Producer*, *On_Call_Producer* and *Activation_Log_Reader*). The special nodes *En* and *Ex* express the entry and the exit points of each task, and the OpenMP code is pointed with dashed-dotted purple lines. Finally, the turquoise square boxes at the bottom represent some significant shared data, and the edges relating this boxes to the CFG nodes symbolize the type of access to the data: read (dotted dark red), write (solid yellow) and read/write (dashed green).

Regarding the OpenMP code, it is independent from the Ada code because the data structures being used are different. However, it is important to note that the OpenMP parallel tasks are inherently concurrent because they are called from within different Ada tasks, which are in turn concurrent.

Definition 1. *A* block of concurrency*, or* concurrent block*, is a set of portions of code that may execute in parallel.*

Since the application meets the Ravenscar profile, the CFG is particularly simple because all tasks are created at library level, meaning that they start executing at the beginning of the program (after elaboration) and terminate when the program ends (task allocators, task termination and abortion, and task hierarchies, among others, are not allowed). Hence, there are only two blocks of

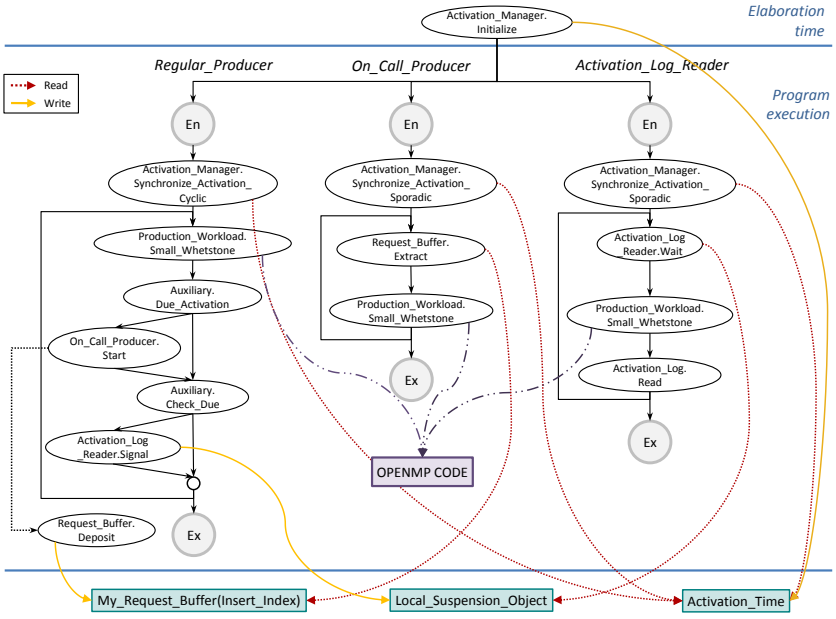

Fig. 3: Simplified CFG of the *Ravenscar* application.

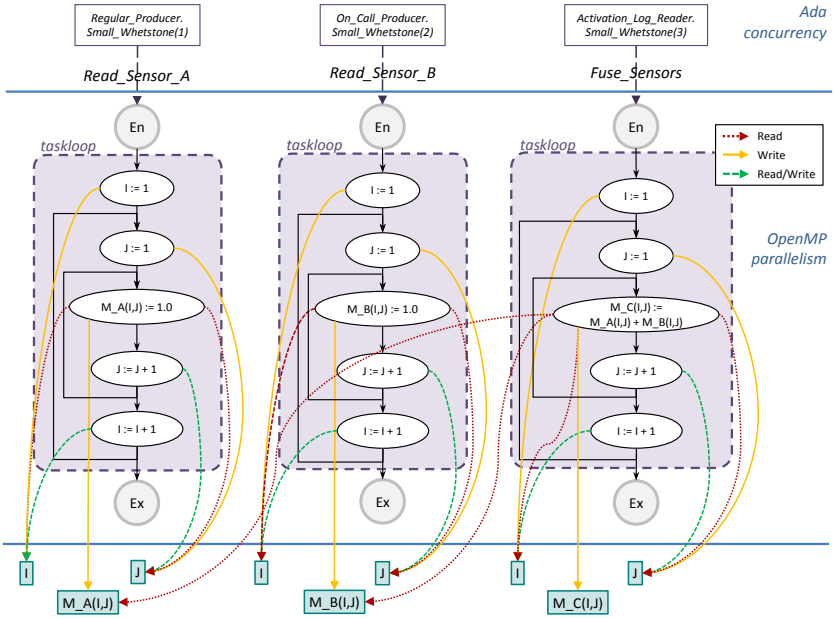

Fig. 4: CFG of the OpenMP code introduced in the *Small_Whetstone* procedure.

concurrency (split by blue lines in the CFG) that correspond to the code executed during elaboration, and the rest of the code.

5.3 Correctness analysis

Inspired by the algorithms presented in the scope of OpenMP to automatically determine the data-scoping attributes [36] and the dependence clauses [37] of an OpenMP task, we present an algorithm able to find data-race conditions in Ada concurrent programs, containing or not OpenMP tasks. The high-level description of the algorithm is outlined in Listing 1.

Algorithm 1 High-level description of the race detection algorithm.

1. Build the inter-procedural CFG of the program.
2. Recognize the blocks of concurrency (in a Ravenscar application this is as simple as splitting the elaboration code and the rest of the code).
3. For each concurrent block, look for concurrent accesses to shared data, where at least one of the accesses is a write. If that occurs:
   (a) If all accesses to the shared data are within OpenMP tasks that belong to the same binding region, then:
      – If the operations are commutative [41], then protect the accesses with an `atomic` or a `critical` construct.
      – Otherwise, there are two approaches:
         * Use full synchronizations: insert a `taskwait` or `barrier` construct between the two accesses.
         * Use partial synchronizations: follow the algorithm to automatically determine the dependence clauses of an OpenMP tasks [37].
   (b) Otherwise, propose to wrap the shared data in a protected object.

Applying the two first steps of the algorithm to our test case results in the CFGs presented in Section 5.2. All Ada and OpenMP tasks correspond to the same block of concurrency, hence potential race conditions may occur among all Ada and OpenMP tasks. However, since OpenMP and Ada tasks manage different share data, we can treat them separately.

Applying the third step on the original *Ravenscar* code reveals that: a) *Activation_Time* is not in a race condition because the read and the write accesses are in different concurrent blocks, b) *Local_Suspension_Object* is not in a race condition because the operations performed on it are atomic with respect to each other, as the standard says, and c) *My_Request_Buffer(Insert_Index)* is not in a race condition because this object is part of the protected object *Request_Buffer*. The results of the algorithm on the original *Ravenscar* application successfully found that the code contains no race conditions.

Regarding the analysis of the OpenMP code note that the OpenMP data-sharing rules indicate that there is a private copy of the induction variable of the taskloop for each thread. As a result, applying the third step of the algorithm on the OpenMP code reveals that accesses to variables I and J are not in a race

condition. On the other hand, accesses to the matrices *M_A* and *M_B* are in a race condition because the write access to *M_A* and *M_B* from *Read_Sensor_A* and *Read_Sensor_B* respectively collide with the read access to both variables from *Fuse_Sensor*. The results of the algorithm indicate the use of partial synchronizations in the form of task dependence clauses, which are shown in Fig. 5.

```
1  procedure Read_Sensor_A is begin
2    pragma OMP (parallel);
3    pragma OMP (single);
4    pragma OMP (taskloop, depend=>in, M_A(0:Dim,0:Dim));
5    ...
6  end Read_Sensor_A;
7
8  procedure Read_Sensor_B is
9  begin
10   pragma OMP (parallel);
11   pragma OMP (single);
12   pragma OMP (taskloop, depend=>in, M_B(0:Dim,0:Dim));
13   ...
14 end Read_Sensor_B;
15
16 procedure Fuse_Sensors is
17 begin
18     pragma OMP (parallel);
19     pragma OMP (single);
20     pragma OMP (taskloop, depend=>in, M_A(0:Dim,0:Dim), M_B(0:Dim,0:Dim),
21                           depend=>out, M_C(0:Dim,0:Dim));
22     ...
23 end Fuse_Sensors;
```

Fig. 5: Snippet of the OpenMP code inserted in the *Production_Workload* package of the *Ravenscar* application including the dependence clauses proposed by the correctness analysis.

5.4 Safe parallelism beyond the Ravenscar profile

This work currently assumes a restricted model, where Ada applications follow the Ravenscar profile [38], and considering only the sharing of variables declared in the same scope. This restriction is not related to the approach per se, but to the complexity of the CFG as well as the program code visibility required for the analysis. Hence, to support the full Ada concurrency model, the CFG must be extended as to include further edges between tasks (e.g. master dependencies, task termination, rendezvous, etc.). These edges must be taken into account to determine the concurrency blocks (considering when tasks come to life and terminate), and also to tune the accuracy of the results of the race condition algorithm (considering when data is actually accessed, if possible). The compiler approach in this analysis must always be conservative in the sense that false positives are acceptable, but false negatives are inadmissible.

Another important consideration is the introduction of full program analysis to allow the algorithm addressing the data sharing of variables declared in any scope. In this sense, we consider the proposals for both Ada [4] and OpenMP [11] to cope with this limitation, both consisting in annotations added to APIs of

those applications which are to be used as third-party libraries. The Ada annotations include the aspects `Global` and `Potentially_Blocking` to resolve race conditions and deadlocks respectively, and the OpenMP annotations include the directives `globals` and `usage` to resolve race conditions and illegal nesting[11] (including nested regions that can cause deadlocks).

6 Conclusions

This paper provides one step further in the work to enable OpenMP fine-grained parallelism in Ada, by addressing the safety of the code in the presence of parallel computation. For this, the paper proposes compiler analysis techniques that can identify potential race conditions in Ada, both considering Ada tasks and parallel OpenMP code. These techniques are built on top of three components: a) a graph representation that includes both control- and data-flow dependencies of concurrent and parallel code, b) an adaptation of existent compiler techniques developed for sequential languages to consider Ada tasks, and c) compiler methods that detect data races and guide the programmer in solving them.

Together with previous works, this paper provides a solution to enable the use of the OpenMP fine-grained tasking model, which can be used together with, or supporting, the existent Ada parallel tasklet model.

7 Acknowledgments

This work was supported by the Spanish Ministry of Science and Innovation under contract TIN2015-65316-P, and by the FCT (Portuguese Foundation for Science and Technology) within the CISTER Research Unit (CEC/04234).

References

1. NVIDIA: Automotive. https://www.nvidia.com/en-us/self-driving-cars (2017)
2. Liu, S., Tang, J., Zhang, Z., Gaudiot, J.L.: Adacore automotive. (2018)
3. Pinho, L.M., Moore, B., Michell, S.: Parallelism in Ada: status and prospects. In George, L., Vardanega, T., eds.: 19th Ada-Europe International Conference on Reliable Software Technologies, Springer (2014) 91–106
4. Taft, S.T., Moore, B., Pinho, L.M., Michell, S.: Safe parallel programming in Ada with language extensions. ACM SIGAda Ada Letters 34(3) (2014) 87–96
5. Pinho, L.M., Moore, B., Michell, S., Taft, S.T.: An Execution Model for Fine-Grained Parallelism in Ada. In de la Puente, J.A., Vardanega, T., eds.: 20th Ada-Europe International Conference on Reliable Software Technologies, Springer (June 2015) 196–211
6. Pinho, L.M., Moore, B., Michell, S., Taft, S.T.: Real-time fine-grained parallelism in ada. ACM SIGAda Ada Letters 35(1) (2015) 46–58

[11] The OpenMP specification (Section 2.17 [10]) defines a series of rules that determine which constructs cannot be nested within each other.

7. Taft, T., Moore, B., Pinho, L.M., Michell, S.: Reduction of Parallel Computation in the Parallel Model for Ada. ACM SIGAda Ada Letters 36(1) (2016) 9–24
8. Royuela, S., Martorell, X., Quiñones, E., Pinho, L.M.: OpenMP Tasking Model for Ada: Safety and Correctness. In Blieberger, J., Bader, M., eds.: 22nd Ada-Europe International Conference on Reliable Software Technologies, Springer (June 2017) 184–200
9. Royuela, S., Pinho, L.M., Quiñones, E.: Converging Safety and High-performance Domains: Integrating OpenMP into Ada. In de Supinski, B.R., L., O.S., Terboven, C., Chapman, B.M., Müller, M.S., eds.: Design, Automation & Test in Europe Conference & Exhibition, IEEE (March 2018)
10. Board, O.A.R.: OpenMP Application Programming Interface 4.5. http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf (2015)
11. Royuela, S., Duran, A., Serrano, M.A., Quiñones, E., Martorell, X.: A Functional Safety OpenMP* for Critical Real-Time Embedded Systems. In de Supinski, B.R., L., O.S., Terboven, C., Chapman, B.M., Müller, M.S., eds.: Scaling OpenMP for Exascale Performance and Portability – 13th International Workshop on OpenMP, Springer (September 2017) 231–245
12. Ada Resource Association: Ada 95 Reference Manual. ISO/IEC 8652:1995(E) with COR.1. http://www.adaic.org/resources/add_content/standards/95lrm/RM.pdf (2000)
13. Burns, A., Dobbing, B., Romanski, G.: The Ravenscar tasking profile for high integrity real-time programs. In Asplund, L., ed.: 3rd Ada-Europe International Conference on Reliable Software Technologies, Springer (June 1998) 263–275
14. Barnes, J.G.P.: High integrity software: the spark approach to safety and security: sample chapters. Pearson Education (2003)
15. Taft, S.T., Schanda, F., Moy, Y.: High-Integrity Multitasking in SPARK: Static Detection of Data Races and Locking Cycles. In Babiceanu, R., Waeselynck, H., Paul, R.A., Cukic, B., Xu, J., eds.: 17th International Symposium on High Assurance Systems Engineering, IEEE (2016) 238–239
16. Michell, S., Moore, B., Pinho, L.M.: Tasklettes – a fine grained parallelism for Ada on multicores. In Keller, H.B., Erhard, P., Dencker, P., Klenk, H., eds.: 18th Ada-Europe International Conference on Reliable Software Technologies, Springer (June 2013) 17–34
17. Serrano, M.A., Melani, A., Vargas, R., Marongiu, A., Bertogna, M., Quiñones, E.: Timing characterization of OpenMP4 tasking model. In Iyer, R., Garg, S., eds.: International Conference on Compilers, Architecture and Synthesis for Embedded Systems, IEEE Press (October 2015) 157–166
18. Serrano, M.A., Melani, A., Bertogna, M., Quiñones, E.: Response-time analysis of DAG tasks under fixed priority scheduling with limited preemptions. In: Design, Automation & Test in Europe Conference & Exhibition, IEEE (March 2016) 1066–1071
19. Melani, A., Serrano, M.A., Bertogna, M., Cerutti, I., Quiñones, E., Buttazzo, G.: A static scheduling approach to enable safety-critical OpenMP applications. In: 22nd Asia and South Pacific Design Automation Conference, IEEE (January 2017) 659–665
20. Sun, J., Guan, N., Wang, Y., He, Q., Yi, W.: Scheduling and analysis of real-time openmp task systems with tied tasks. In: Proceedings of Real-Time Systems Symposium. (2017)
21. Kroening, D., Poetzl, D., Schrammel, P., Wachter, B.: Sound static deadlock analysis for C/Pthreads. In Lo, D., Apel, S., Khurshid, S., eds.: 31st International Conference on Automated Software Engineering, IEEE (September 2016) 379–390

22. Ma, H., Diersen, S.R., Wang, L., Liao, C., Quinlan, D., Yang, Z.: Symbolic analysis of concurrency errors in openmp programs. In Bilof, R., ed.: 42nd International Conference on Parallel Processing, IEEE (October 2013) 510–516
23. Basupalli, V., Yuki, T., Rajopadhye, S., Morvan, A., Derrien, S., Quinton, P., Wonnacott, D.: ompVerify: polyhedral analysis for the OpenMP programmer, Springer (June 2011) 37–53
24. Royuela, S., Ferrer, R., Caballero, D., Martorell, X.: Compiler analysis for OpenMP tasks correctness. In: 12th International Conference on Computing Frontiers, ACM (May 2015) 7
25. Evangelista, S., Kaiser, C., Pradat-Peyre, J.F., Rousseau, P.: Quasar: a new tool for concurrent Ada programs analysis. In: Ada-Europe, Springer (2003) 168–181
26. Fechete, R., Kienesberger, G.: A Framework for CFG-Based Static Program Analysis of Ada Programs. In Kordon, F., Vardanega, T., eds.: 13th Ada-Europe International Conference on Reliable Software Technologies, Springer (June 2008) 130–143
27. Qi, X., Xu, B.: An approach to slicing concurrent Ada programs based on program reachability graphs. International Journal of Computer Science and Network Security 6(1) (2005) 29–37
28. Mohaqeqi, M., Abdullah, J., Guan, N., Yi, W.: Schedulability analysis of synchronous digraph real-time tasks. In O'Conne, L., ed.: 28th Euromicro Conference on Real-Time Systems, IEEE (July 2016) 176–186
29. Wang, B., Gao, H., Cheng, J.: Definition-Use Net and System Dependence Net generators for Ada 2012 programs and their applications. Ada User Journal 38(1) (2017) 37–55
30. Faria, J.M., Martins, J., Pinto, J.S.: An Approach to Model Checking Ada Programs. In Brorsson, M., Pinho, L.M., eds.: 17th Ada-Europe International Conference on Reliable Software Technologies, Springer (June 2012) 105–118
31. Holzmann, G.J.: The model checker SPIN. IEEE Transactions on software engineering 23(5) (May 1997) 279–295
32. Project Hi-Lite: GNATprove. http://www.open-do.org/projects/hi-lite/gnatprove (2017)
33. GNU: GNAT. https://www.gnu.org/software/gnat (2016)
34. Meyer, Bertrand: Object-oriented software construction. Volume 2. Prentice hall New York (1988)
35. Ada Resource Association: Ada Reference Manual, ISO/IEC 8652:2012(E). http://archive.adaic.com/standards/83lrm/html (2012)
36. Royuela, S., Duran, A., Liao, C., Quinlan, D.J.: Auto-scoping for OpenMP Tasks. In Chapman, B.M., Massaioli, F., S., M.M., Rorro, M., eds.: OpenMP in a Heterogeneous World – 8th International Workshop on OpenMP, Springer (June 2012) 29–43
37. Royuela, S., Duran, A., Martorell, X.: Compiler automatic discovery of ompss task dependencies. In Kasahara, H., Kimura, K., eds.: International Workshop on Languages and Compilers for Parallel Computing, Springer (September 2012) 234–248
38. Burns, A., Dobbing, B., Vardanega, T.: Guide for the use of the Ada Ravenscar Profile in high integrity systems. ACM SIGAda Ada Letters 24(2) (2004) 1–74
39. Burns, A., Wellings, A.J.: HRT-HOOD: A structured design method for hard real-time systems. Real-Time Systems 6(1) (Jan 1994) 73–114
40. Duran, A., Ayguadé, E., Badia, R.M., Labarta, J., Martinell, L., Martorell, X., Planas, J.: Ompss: a proposal for programming heterogeneous multi-core architectures. Parallel Processing Letters 21(02) (2011) 173–193

41. Lippe, E., van Oosterom, N.: Operation-based merging. In: Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments. SDE 5, New York, NY, USA, ACM (1992) 78–87

A Complete CFG of the Ravenscar application

This appendix includes the complete CFG of the Ada code used to illustrate the proposal of this paper, extracted from the Ada Ravenscar Profile Guide [38].

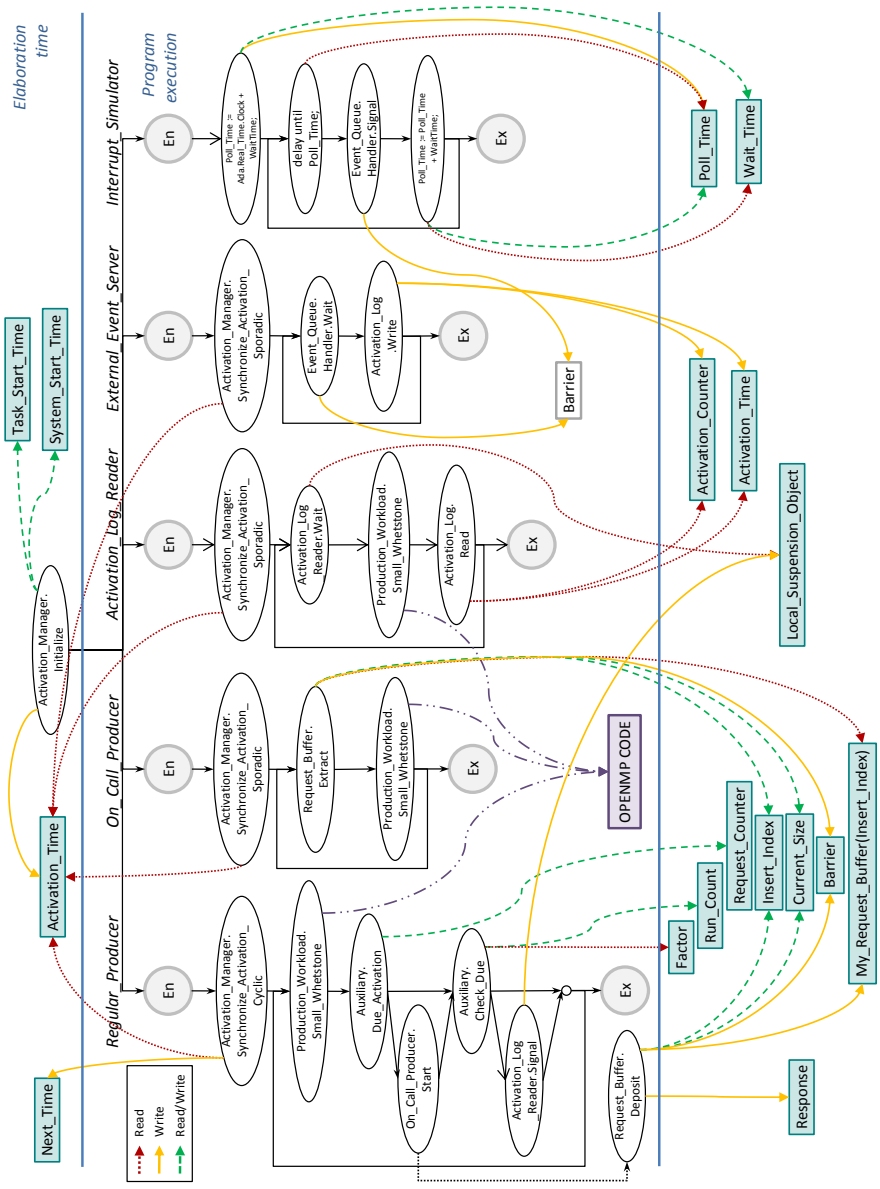

Fig. 6: Control flow graph of the *Ravenscar* application defined in Section 7 of the Ada Ravenscar Profile Guide, containing accesses to shared data.