A functional safety OpenMP* for critical real-time embedded systems

Sara Royuela[1], Alejandro Duran[2], Maria A. Serrano[1], Eduardo Quiñones[1], and Xavier Martorell[1]

[1] Barcelona Supercomputing Center
sara.royuela,maria.serrano,eduardo.quinones,xavier.martorell@bsc.es
[2] Intel Corporation Iberia alejandro.duran@intel.com

Abstract. OpenMP* has recently gained attention in the embedded domain by virtue of the augmentations implemented in the last specification. Yet, the language has a minimal impact in the embedded real-time domain mostly due to the lack of reliability and resiliency mechanisms. As a result, functional safety properties cannot be guaranteed. This paper analyses in detail the latest specification to determine whether and how the compliant OpenMP implementations can guarantee functional safety. Given the conclusions drawn from the analysis, the paper describes a set of modifications to the specification, and a set of requirements for compiler and runtime systems to qualify for safety critical environments. Through the proposed solution, OpenMP can be used in critical real-time embedded systems without compromising functional safety.

1 Introduction

There is a visible trend in the critical real-time embedded industry to adopt parallel processor architectures, with the objective of providing the performance requirements needed to support advanced functionalities, e.g. autonomous driving and unmanned aerial vehicles. Examples targeting automotive, avionics and industrial domains include the Kalray MPPA featuring a many-core fabric [7], the NVIDIA Tegra X1 featuring a multi-core and a GPU [4], or the TI Keystone II featuring a multi-core and a DSP fabric [2]. These recent advances on parallel embedded architectures are driving an interesting convergence between the high-performance and the embedded domain[1]. In this context, the use of parallel programming models is of paramount importance. To begin with, to efficiently exploit the performance opportunities of these architectures. Besides, to provide programmability and portability. All crucial to meet productivity.

OpenMP* has recently gained much attention in the real-time embedded domain owing to the augmentations of the latest specification. These address the key issues in parallel heterogeneous embedded architectures: a) the coupling of a main host processor to one or more accelerators, where highly-parallel code kernels can be offloaded for improved performance/watt; and b) the capability of expressing fine-grained, both structured and unstructured, and highly-dynamic task parallelism. As a result, OpenMP is already supported by several chip and compiler vendors targeting embedded systems such as Kalray, Texas Instruments and ARM. A fact that relaxes portability issues.

The final publication is available at Springer via http://https://link.springer.com/chapter/10.1007/978-3-319-65578-9_16

Furthermore, recent studies demonstrate that the structure and syntax of the OpenMP tasking model resembles the Directed Acyclic Graph (DAG) scheduling real-time model [35]. This enables the analysis of the timing properties for such a model. However, the analysis of the OpenMP thread-centric model in terms of timing and progress guarantees is still an open issue. Last but not least, the use of OpenMP to enable fine-grained parallelism in critical real-time suitable languages like Ada has already been proposed [31].

Overall, critical real-time embedded systems can benefit from the flexibility delivered by OpenMP. Yet, the impact of the language in such a domain is very limited. The reason is that critical real-time systems require *functional safety* guarantees, imposing the system to operate correctly in response to its inputs from both functional and timing perspectives. This paper focuses on the former. Functional safety is verified by means of safety standards as the ISO26262 [16] for automotive, the DO178C [8] for avionics or the IEC61508 [15] for industry. The use of reliability and resiliency mechanisms allow guaranteeing the correct operation of the (parallel) execution. Moreover, the complete system stack must be guaranteed, from the processor architectural perspective (e.g. multi-core processor designs ARM A57 or Infineon Aurix are safety compliant) to the operating system (e.g. PikeOS, VxWorks, Erika Enterprise are safety compliant). In that respect, OpenMP lacks the required reliability and resiliency mechanisms at both compiler and runtime levels.

§ 2 analyses in detail the latest specification of OpenMP [3] to identify the features that may entail a hazard regarding functional safety on critical real-time embedded systems. Along with the analysis, § 3 proposes changes in the specification as well as a series of implementation considerations to take into account in both compilers and runtimes. This proposal aims to eliminate non-determinism, increase efficiency and simplify the kernel of high-integrity applications, covering most issues that can prevent OpenMP from being used in a safety-critical environment.

2 OpenMP hazards for critical real-time embedded systems

The current section discusses the OpenMP specification with the aim of: a) detecting those features that can be a hazard for functional safety when used in a critical real-time embedded system, and b) proposing solutions to avoid the hazard at design, compile or run time, depending on the case.

2.1 Unspecified behavior

OpenMP defines the situations that result in an unspecified behavior as: non-conforming programs, implementation defined features and issues documented to have an unspecified behavior. The impact of each situation to the safety-critical domain, as well as the solutions we propose, are exposed below.

Non-conforming programs

The OpenMP specification defines several requirements to applications that are parallelized with OpenMP. Programs that do not follow these rules are called *non-conforming*. According to the specification, OpenMP compliant implementations are not required to verify conformity. Despite this, safety-critical environments compel frameworks to do this validation to certify functional safety.

OpenMP restrictions affect directives, clauses and the associated user code. Checking some restrictions just requires the verification of OpenMP constructions (e.g. which clauses and how many times a clause can be associated with a specific directive, for example, *at most one* `if` *clause can appear on the* `task` *directive*). However, checking some restrictions requires visibility of different parts of the application (e.g some regions cannot be nested and/or closely nested in other regions, for example, `atomic` *regions must not contain OpenMP constructs*).

Compilers must implement inter-procedural analysis to have access to the whole application. This capability has been successfully implemented in many vendors following different approaches, such as the Intel IPO [14] or the GCC LTO [12]. Nevertheless, access to the whole code is possible only for monolithic applications. This is not very common in the critical domain, where systems consist of multiple components developed by different teams, and rely on third-party libraries. In these cases, additional information may be needed. We discuss this situation and propose a solution to it in § 3. This solution is based on new directives that provide the required information. Henceforward, we assume that the information needed to perform whole program analysis is always accessible.

Implementation defined behavior

Some aspects of the implementation of an OpenMP compliant system are not fixed in the specification. These aspects are said to have an *implementation defined* behavior, and they may indeed vary between different compliant implementations. The different aspects can be grouped as follows:

1. Aspects that are naturally implementation defined, so the specification can be used in multiple architectures: definitions for *processor*, *device*, *device address* and *memory model* features.
2. Aspects that are implementation defined to allow flexibility: internal control variables (e.g.: *nthreads-var* and *def-sched-var* among others); selection, amount and distribution of threads (e.g. `single` construct); dynamic adjustment of threads; etc.
3. Aspects caused by bad information specified by the user: values out of range passed to runtime routines or environment variables (e.g. the argument passed to `omp_set_num_threads` is not a positive integer).

Aspects in groups 1 and 2 may not lead to an execution error or prevent the program from validating. This is not the case for aspects in group 3, where an implementation may decide to finish the execution if a value is not in the range it was expected to be. Besides, cases in group 2 may result in different

outcomes depending on the platform used for the execution. For example, when the `runtime` or the `auto` kinds are used in the `schedule` clause, the decision of how the iterations of a loop will be scheduled is deferred until runtime.

In the light of all that, some aspects in groups 2 and 3 are not suitable in a safety-critical environment because they are non-deterministic and may cause an undesired result. Situations such as the application aborting due to an unexpected value passed to either an environment variable or a runtime routine can be solved by defining a default value which will not cause the application to end (note that this value can be different across implementations without that affecting functional safety). Situations such as an `auto` or `runtime` value in the `schedule` clause can be solved by taking a conservative approach at compile time (i.e. if a deadlock may occur for any possible scheduling option, then the compiler will act as if that scheduling happens). Situations such as runtimes defining different default values for ICVs like *nthreads-var* do not need to be addressed, because they do not bring on any hazard regarding functional safety.

Other unspecified behavior

The rest of situations resulting in an undefined behavior are errors and need to be addressed to guarantee functional safety. These situations can be classified in three groups, depending on the moment at which they can be detected:

1. Situations that can be detected at compile time. In this case we can distinguish those that can be solved by the compiler (e.g. data-race conditions could be solved by automatically protecting accesses with a `critical` construct or synchronizing the accesses -§ 2.3 shows more details about data race management-), and those that need user intervention (e.g. compilers should abort compilation and report to the user situations such as the use of non-invariant expressions in a linear clause).
2. Situations that can be detected at run time. In this case, safety relies on programmers because the results deriving from these situations cannot be handled automatically. Thus, users are compelled to handle errors such as reduction clauses that contain accesses out of the range of an array section, or using the `omp_target_associate_ptr` routine to associate pointers that share underlying storage (§ 2.5 explores error handling techniques).
3. Situations that cannot be detected. These involve the semantics of the program, for example, a program that relies on the task execution order being determined by a priority-value. This case is further discussed in § 2.5 .

2.2 Deadlocks

OpenMP offers two ways to synchronize threads: via directives (`master` and synchronization constructs such as `critical` and `barrier`), and via runtime routines (lock routines such as `omp_set_lock` and `omp_unset_lock`). Although both mechanisms may introduce deadlocks, the latter is much more error-prone because these routines work in pairs. Furthermore, OpenMP introduces the concept of nestable locks, which differ from the regular locks in that they can be locked repeatedly by the same task without blocking.

Synchronization directives may cause deadlocks if various `critical` constructs with the same name are nested. Synchronization directives can introduce other problems as well, like enclosing a `barrier` construct in a condition that is special to a thread. Since barriers must always be encountered by all threads of a team, the previous situation will be non-conforming. Such errors can be easily caught by a compiler implementing whole program analysis.

Locking routines may cause errors in the following situations: attempt to access an *uninitialized* lock, attempt to unset a lock owned by another thread or attempt to set a simple lock that is in the *locked* state and is owned by the same task. There exist numerous techniques for deadlock detection, such as Chord [26] and Sherlock [10], that apply to different programming models. Most of the approaches pursue scalability without losing accuracy, thus effectiveness. However, safety-critical environments require soundness. In this regard, the only sound approach, to the best of our knowledge, for detecting deadlocks in C/Pthreads programs is the one developed by Kroening et al.[18]. OpenMP simple locks are comparable to Pthreads mutex, so the previous technique can be extended to OpenMP. Nestable locks have other peculiarities and it may not be possible to detect deadlocks at compile time. In such a case, they should not be permitted.

The use of untied tasks may cause deadlocks that may not exist when using tied tasks. This is because task scheduling constraints (particularly constraint #2) prevent from certain situations involving tied tasks to cause a deadlock by restricting the tasks that can be scheduled at a certain point. Based on that, using tied tasks may seem more suitable for critical real-time embedded systems. It has been, however, demonstrated that timing analysis for untied tasks is much more accurate than for tied tasks [33]. There is thus a trade-off between functional safety and predictability. For the sake of correctness, untied tasks may be disabled at compile time only when the static analysis detects that a deadlock caused by untied tasks may occur.

2.3 Race conditions

Race conditions appear in a concurrent execution when two or more threads simultaneously access the same resource and at least one of them is a write. This situation is not acceptable for a safety-critical environment since the results of the algorithm are non-deterministic. The problem of detecting data races in a program is NP-hard [27]. On account of this, a large variety of static, dynamic and hybrid data race detection techniques have been developed over the years.

On the one hand, dynamic tools extract information from the memory accesses of specific executions. Despite this, there exist algorithms capable of finding at least one race when races are present, as well as not reporting false positives [5]. On the other hand, static tools still seek a technique with no false negatives and minimal false positives. Current static tools have been proved to work properly on specific subsets of OpenMP such as having a fixed number of threads [22] or using only affine constructs [6]. A more general approach can be used to determine the regions of code that are definitely non-concurrent [21]. Although it is not an accurate solution, it does not produce false negatives, which

is paramount in the safety-critical domain. Therefore, the previously mentioned techniques can be combined to deliver conservative and fairly accurate results.

2.4 Cancellation

Until version 4.0, all OpenMP constructs based their execution model in the *Single Entry Single Exit (SESE)* principle. This means that no thread encountering an OpenMP region can jump out of the region skipping a part of it. This is no longer true after the incorporation of the cancellation constructs (i.e. `cancel` and `cancellation point`), which allow exiting parallel computation at a certain point that may not be the end of the region.

Unlike other models such as the Pthreads asynchronous cancellation, OpenMP only accepts synchronous cancellations at cancellation points. Although this eliminates resource leak risks, the technique introduces non-determinism, which is not desirable in a safety-critical environment. Due to the use of cancellation constructs, non-determinism appears in the following situations:

1. The order of execution between one thread that activates cancellation and another thread that encounters a cancellation point.
2. The final value of a reduction or lastprivate variable in a canceled construct.
3. The behavior of nested regions suitable of being canceled.

If a code is well written, case 1 may only affect performance, but the code will deliver a valid result whether cancellation occurs or not. Case 2, instead, may lead to errors if some threads have not finished their computation. Nonetheless, static analysis can verify that reduction and lastprivate variables are not used within a construct that may be subject to cancellation, or that the variables are used only when no cancellation occurs. Finally, case 3 can be solved by statically verifying that regions subject to cancellation are not nested.

Another issue arises when locks are used in regions subject to cancellation, because users are responsible for releasing those locks. Current deadlock detection techniques do not take into account the semantics of the cancellation constructs. Nonetheless, these techniques can easily be extended because the effect of a cancellation is similar to the existence of a jump out of the region.

2.5 Other features to consider

Although they do not necessarily involve a hazard, there are other issues that are worth to mention in the context of this paper. These are explained next.

Error handling

Resiliency is a crucial feature in the safety-critical domain. However, OpenMP does not prescribe how implementations must react to situations such as the runtime not being able to supply the number of threads requested, or the user passing an unexpected value to a routine. While the former is a problem caused by the runtime environment, the latter is an error produced by the user. Both

eventually become an unspecified behavior according to the specification, but they can be addressed differently. On the one hand, if the error is produced by the environment, users may want to define what recovery method needs to be executed. On the other hand, errors produced by the user are better caught at compile time or handled by the runtime (we discuss the latter in § 2.1).

Several approaches have been proposed with the aim of adding resiliency mechanisms to OpenMP. There are four different strategies for error handling [36]: exceptions, error codes, call-backs and directives. Each technique can be applied according to its features to different languages and situations. Exception based mechanisms fit well in programs exploiting the characteristics of exception-aware languages (e.g. C++, Ada) [11]. Error code based techniques are a good candidate when using a language unaware of exceptions (e.g. C, Fortran). Call-back methods have the advantage of isolating the code that is to be executed when an exception occurs, and thus enhance readability and maintainability [9]. Finally, the use of specific OpenMP directives has the advantage of being simple, although they cannot cover all situations and users cannot define an exact behavior. The latter is the only approach already adopted in the specification with the cancellation constructs (see more details in § 2.4).

A safety-critical framework supporting OpenMP will require the implementation of error-handling methodologies in order to ensure functional safety.

Nested parallelism

OpenMP allows nesting parallel regions to get better performance in cases where parallelism is not exploited at the same level. A distributed shared-memory machine with an appropriate memory hierarchy is necessary to exploit the benefits of this feature (the major HPC architectures).

The nature of critical real-time embedded systems is quite different, where both memory size and processor speed are usually constrained. Furthermore, the use of nested parallelism can be costly due to the overhead of creating multiple parallel regions, possible issues with data locality, and the risk of oversubscribing system resources. For the sake of simplicity, and considering that current embedded architectures will not leverage the use of nested parallelism, this feature could be deactivated by default.

Semantics of OpenMP

For an analysis tool, it is possible to address correctness based on how the program is written. However, addressing whether the program behaves as the user wants is another matter altogether. This said, some features of OpenMP may be considered as hazardous because their use may derive in errors involving the semantics of the program. We discuss some of them as follows:

- A program that relies on an specific order of execution of the tasks based on their priorities is non-conforming.
- When and how some expressions are to be executed is not defined in OpenMP. Some examples are: whether, in what order, or how many times any side effects of the evaluation of the `num_threads` or `if` clause expressions of a

`parallel` construct occur; and the order in which the values of a reduction are combined is unspecified. Thus, an application that relies on any ordering of the evaluation of the expressions mentioned before is non-conforming.
- The storage location specified in task dependencies must be identical or disjoint. Thus, runtimes are not forced to check whether two task instances have partially overlapping storage (which eases the runtime considerably).
- The use of flushes is highly error-prone, and makes it extremely hard to test whether the code is correct. However, the use of the `flush` operation is necessary for some cases such as the implementation of the producer-consumer pattern.

Frameworks cannot prevent users from writing senseless code. However, some of the features mentioned before could be deactivated if the level of criticality demands it. It is a matter of balance between functionality and safety. Thus, if necessary, support for task priorities and flushes could be deactivated. The case regarding side-effects could be simplified to using associative and commutative operations in reductions, and expressions without side-effects in the rest of clauses. Finally, the case regarding task dependency clauses could be solved at runtime by resuming parallel execution when a task contains non-conforming expressions in its dependency clauses (although this solution causes a serious impact in the performance of the application).

3 OpenMP support for critical real-time systems

Based on the discussion in § 2, this section exposes our proposal to enable the use of OpenMP in safety-critical environments without compromising functional safety. The proposal can be divided in two facets: different changes to the specification, and a series of compiler and runtime implementation considerations.

3.1 Changes to the specification

As we introduce in § 2.1 , whole program analysis may not be enough if the system includes multiple components developed by different teams or make use of third-party libraries implemented with OpenMP. In such a case, we propose that these components or libraries augment their API with information about the OpenMP features used in each method. As a result, compilers will be able to detect, on the one hand, illegal nesting of directives and data accessing clauses (i.e. data-sharing attributes, data mapping, data copying and reductions) and, on the other hand, data-races.

To tackle illegal nesting, we propose to add a new directive called `usage`. This directive is added to a function declaration and followed by a series of clauses. The clauses determine the features of OpenMP that are used within the function and any function in its call graph, and can cause an illegal nesting. Note that the use of this directive is a promise that a construct might be used in a possible path within the function. Overall, the clauses that can follow the pragma `usage` are one of the following:

– Directive related: `parallel`, `worksharing` (which epitomizes `single`, `for/do`, `sections` and `workshare`), `master`, `barrier`, `critical`, `ordered`, `cancel`, `distribute_construct` (which epitomizes `distribute`, `distribute simd`, distribute parallel loop and distribute parallel loop SIMD), `target_construct` (which epitomizes `target`, `target update`, `target data`, `target enter data` and `target exit data`), `teams`, `any` (which epitomizes any directive not included in the previous items).
– Clause related: `firstprivate`, `lastprivate`, `reduction`, `map`, `copyin` and `copyprivate`.

Based on the restrictions that apply to the nesting of regions (Section 2.17 of the specification [3]) and the restrictions that apply to the mentioned data accessing clauses, we extract the set of rules that define how the previous clauses are to be used. These rules are the following:

– Clauses `parallel`, `worksharing`, `master`, `barrier` and `ordered` are required when the corresponding construct is the outermost construct.
– Clauses `critical` and `target_construct` are required if there is any occurrence of the corresponding construct.
– Clause `teams` is required if the corresponding construct is orphaned.
– Clauses `cancel` and `cancellation point` are required if the corresponding constructs are not nested in their corresponding binding regions.
– Clause `any` must be specified if OpenMP is used and no previous case applies.
– Data accessing clauses are required when they apply to data that is accessible outside the application and particular constraints apply to them:
  • Clause `firstprivate` if used in a worksharing, `distribute`, `task` or `taskloop` construct not enclosed in a `parallel` or `teams` construct.
  • Clauses `lastprivate` and `reduction` if used in a worksharing not enclosed in a `parallel` construct.
  • Clauses `copyin`, `copyprivate` and `map` in any case.

To avoid data races, we propose to add a new directive called `globals`. This directive, added to a function declaration, defines which data is used within the function while it can be accessed concurrently from outside the function, thus producing a data-race. Different clauses accompany this directive: `read`, `write`, `protected_read` and `protected_write`, all accepting a list of items. While `read` and `protected_read` must be used when global data is only read, `write` and `protected_write` are required when global data is written, independently of it being read as well. The *protected* versions of these clauses must be used when the access is within an `atomic` or a `critical` construct.

Listings 1.1 and 1.2 illustrate the use of the two mentioned directives within the context of a system component that can be used without accessing its source code. The former listing contains the definition of function *foo*, which uses one of the most determining features for OpenMP to be used in parallel heterogeneous embedded architectures: the `target` construct. This function defines an asynchronous task that offloads some parallel computation to a device. The parallel computation within the device is synchronized using the `critical` construct,

and is canceled if the `cancel` directive is reached. The latter listing contains the declaration of function *foo*, augmented with the `usage` and `globals` directives. Clauses `target_construct` and `critical` associated to directive `usage` indicate that the function executes one or more `target` and `critical` constructs. A programmer and/or compiler can avoid calling function *foo* from within a `target` and a `critical` constructs, thus avoiding an illegal nesting. Note that directive `cancel` is not included because it is nested in its binding region, clauses `task` and `parallel for` are not included because no rule apply to them, and data-sharing clause `firstprivate` is not included because it does not concern to data that is visible from outside the function. Additionally, clauses `write` and `protected_write` associated to directive `globals` indicate that variables *arr[0:N-1]* and *sum* are both written, being *sum* written within a `critical` construct. A programmer and/or compiler can determine whether these variables are in a race condition without knowing the code of the function, and therefore synchronize the accesses to the variables appropriately.

Listing 1.1. Example of OpenMP function definition

```
1  void foo(float* arr, unsigned N, unsigned M,
2           float &sum, float MAX_SUM)
3  {
4    #pragma omp task shared(arr, sum) \
5                     firstprivate(N, M, MAX_SUM)
6    #pragma omp target map(tofrom:arr[0:N-1])
7    #pragma omp parallel for
8    for (int i=0; i<N; ++i) {
9      arr[i] = bar(i);
10     if (i % M == 0) {
11       #pragma omp critical
12         sum +=arr[i];
13     }
14     if (sum > MAX_SUM) {
15       #pragma omp cancel for
16     }
17   }
18 }
```

Listing 1.2. Function declaration for method in Listing 1.1 using the extensions for safety-critical OpenMP

```
1 #pragma omp usage target_construct critical \
2                 map(tofrom:arr[0:N-1])
3 #pragma omp globals write(arr[0:N-1]) protected_write(sum)
4 void foo(float* arr, unsigned N, unsigned M,
5          float &sum, float MAX_SUM);
```

Listings 1.3 and 1.4 show another example of the proposed directives. In this case, the function definition in the former listing performs the *factorial* computation parallelized using the `for` worksharing; and the function declaration in the latter listing shows the clauses required for the method to be used in a functional safe environment. Clause `any` is specified because no rule applies to directive `for`, and clause `reduction` is specified because the reduction is

used in a worksharing not enclosed in a parallel region. With this information a programmer and/or compiler can check whether the variable being reduced is shared in the parallel regions to which any of the worksharing regions bind. Analysis may also verify whether the *factorial* function is not called from within an atomic region, thus causing the program to be non-conforming. Finally, race analysis can detect whether the variable *factorial* is in a race condition by means of the clause `write`.

Listing 1.3. Factorial computation parallelized with OpenMP

```
1 void factorial(int N, int &fact)
2 {
3   fact = 1;
4   #pragma omp for reduction(*:fact)
5   for(int i=2; i <= N; ++i)
6     fact *= i;
7 }
```

Listing 1.4. Function declaration for method in Listing 1.3 using the extensions for safety-critical OpenMP

```
1 #pragma omp usage any \
2                   reduction(factorial)
3 #pragma omp globals write(factorial)
4 void factorial(int N, int &factorial);
```

3.2 Implementation considerations

Both compilers and runtime systems used within a critical real-time system must be qualified against the corresponding functional safety standard, e.g. ISO26262 for automotive or DO178C for avionics, to preserve functional safety. The following paragraphs introduce which constraints apply in our case.

Compiler contract

The development tools used for critical real-time systems need to qualify to the same integrity level[3] as the application they are helping to develop. Nonetheless, current guidelines make the qualification of development tools very difficult [17]. As an example, the standard for Software Considerations in Airborne Systems and Equipment Certification (DO-178C) [8] reads: *"Upon successful completion of verification of the software product, the compiler is considered acceptable for*

[3] The *integrity level*, also called *criticality level*, refers to the consequences of the incorrect behavior of a system. These levels are defined in different scales such as the *Safety Integrity Level* (SIL) for automotive and the *Development Assurance Level* (DAL) for avionics.

that product". As a result, sometimes compilers do not need to be qualified. Nonetheless, to gain assurance, some characteristics must be incorporated, such as being fully tested for complete coverage analysis[4], and being used in the same configuration, options, and environment as the one used to compile any other objects related to the application.

However, for an OpenMP compiler to be valid in a critical real-time environment, it must ensure the source code is compliant with the OpenMP specification. For that reason, the compiler must implement the necessary analysis techniques to allow whole program analysis. Additionally, the compiler must also include specific and sound techniques for data-race and deadlock detection, as well as the correctness analysis that allows statically detecting and fixing the unspecified behaviors commented in § 2.1.

Runtime contract

As a result of the analysis presented in § 2, we conclude that runtime libraries used in safety-critical environments shall follow some requirements to avoid unexpected aborts and fix some programmer errors. The following list is an starting point for these systems to address such undesired results:

- Runtimes should define a default value for all environment variables. This value shall be used when the value specified in the application is out of range, e.g. `OMP_NUM_THREADS` could be 1 by default, and `OMP_NESTED` could be false.
- Some clauses, such as `num_threads` and `device`, take a number as a parameter that must evaluate to a positive integer. Runtimes should define the value to be used if the expression is out of range, for example, 1.
- Other errors can be caught and fixed at runtime, e.g. different instances of the same task or sibling tasks expressing dependency clauses on list items which storage location is neither identical nor disjoint may be executed sequentially.

4 Related work

Parallel heterogeneous embedded architectures certainly require the use of parallel programming models to provide high throughput, low latency and energy-efficient solutions. Efforts to introduce OpenMP in such environments [23] reveal that OpenMP runtimes can efficiently be aware of the heterogeneity and the memory hierarchy to deliver good performance. However, all works that intend to introduce OpenMP in the embedded domain conclude that, although the language is very useful in such environments, some extensions with real-time processing and power-awareness functionalities [13] are needed.

Critical real-time embedded systems, add additional, more restrictive, constraints to those of the embedded domain. Concretely, timing guarantees and functional safety. Regarding the former, significant attempts to analyze the time

[4] *Code coverage* is a measure used to describe the amount of the source code of a program being executed when a particular test suite runs.

predictability properties of OpenMP [33] , as well as deriving response time analysis for both work-conserving dynamic and purely static schedulers [32,19,24], confirm the OpenMP tasking model as a perfectly suitable parallel pattern for safety-critical environments. In this sense, the suitability of the thread-centric model still remains unproved. Furthermore, situations such as starvation when a barrier construct is found shall be addressed.

With regard to functional safety, different works have tried to study, classify and solve mistakes commonly appearing in OpenMP applications [34] [25]. These works are very useful mostly for unexperienced programmers in order to avoid errors. Beyond the theoretical approaches, many articles propose different techniques tackling correctness in general, and OpenMP correctness in particular. § 2 introduces several techniques for detecting specific errors in concurrent programs (i.e. race conditions and dead-locks). Additionally, some techniques have been developed specifically for OpenMP to compute and verify data scoping, task dependencies and locks among others [28] [29] [30] [20].

Finally, there exist works towards the adoption of OpenMP in Ada [31], a language commonly used in safety-critical and high-security domains such as avionics and railroad systems. In Ada, concepts as safety and reliability are crucial. However, there are still some caveats about the integration of the Ada and OpenMP runtimes, because both will be mapped to the underlying threads of the operating system.

5 Conclusions and future work

OpenMP is increasingly being considered a suitable candidate to be used in critical real-time embedded systems considering its benefits: programmability, portability and efficiency, among others. However, such systems impose strict constraints to ensure functional safety in terms of functional correctness and timing predictability. This paper has focused on the former aiming to shorten the distance between OpenMP and the critical real-time domain.

In this scope, we prove that most features specified in OpenMP can be used without compromising safety, as long as compilers implement a series of analyses that can prevent errors such as dead-locks and race conditions. Indeed, analysis must involve the entire program which can be a challenging scenario. To ease this, we propose some new directives that allow whole program analysis even when third-party libraries are used. The majority of the unspecified behaviors defined in the specification can be solved at compile time either automatically by the compiler (e.g. synchronizing variables that otherwise could be accessed after their life-time has ended), or by the programmer (e.g., the use of non-invariant expressions in a linear clause). Other issues can be successfully addressed at runtime (e.g. unexpected values passed to environment variables and runtime libraries can be solved by defining default values to be used in such cases). In some cases, supporting the required level of criticality might incur more overhead than a traditional OpenMP implementation (e.g., tracking task dependencies' overlap). Last but not least, there are a series of features that can be used

erroneously if their semantics are not properly exploited (e.g. tasks priorities or flushes). We conclude that support for these features can be deactivated if the level of criticality requires so.

The small modifications that this paper proposes back up OpenMP's safety. Nonetheless, we note some lacks in the current specification, e.g. error handling techniques to improve resiliency. Hence, despite the functional safety aspect is deeply addressed in this paper, the same analysis concerning time predictability, including starvation, remains as future work. In that regard, we plan to analyze the latest specification to find out how timing analyses could be affected by the use of OpenMP.

6 Acknowledgments

This work was funded by the EU project P-SOCRATES (FP7-ICT-2013- 10) and the Spanish Ministry of Science and Innovation under contract TIN2015-65316-P.

Disclaimers

* Brands and names are the property of their respective owners.

References

1. P-SOCRATES European Project (Parallel Software Framework for Time-Critical Many-core Systems). http://p-socrates.eu
2. Texas Instruments. The 66AK2H12 Keystone II Processor
3. OpenMP Application Programming Interface. http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf (2015)
4. X1: A new era in mobile computing. NVIDIA White Paper (2015)
5. Banerjee, U., Bliss, B., Ma, Z., Petersen, P.: A theory of data race detection. In: Parallel and distributed systems: testing and debugging (2006)
6. Basupalli, V., Yuki, T., Rajopadhye, S., Morvan, A., Derrien, S., Quinton, P., Wonnacott, D.: ompVerify: polyhedral analysis for the OpenMP programmer. In: IWOMP (2011)
7. De Dinechin, B.D., Van Amstel, D., Poulhiès, M., Lager, G.: Time-critical computing on a single-chip massively parallel processor. In: DATE (2014)
8. DO, R.: 178C. Software considerations in airborne systems and equipment certification (2011)
9. Duran, A., Ferrer, R., Costa, J.J., Gonzàlez, M., Martorell, X., Ayguadé, E., Labarta, J.: A proposal for error handling in OpenMP. IJPP 35(4) (2007)
10. Eslamimehr, M., Palsberg, J.: Sherlock: scalable deadlock detection for concurrent programs. In: SIGSOFT (2014)
11. Fan, X., Mehrabi, M., Sinnen, O., Giacaman, N.: Exception Handling with OpenMP in Object-Oriented Languages. In: IWOMP (2015)
12. GNU: Link Time Optimization. https://gcc.gnu.org/onlinedocs/gccint/LTO.html (2017)
13. Hanawa, T., Sato, M., Lee, J., Imada, T., Kimura, H., Boku, T.: Evaluation of multicore processors for embedded systems by parallel benchmark program using OpenMP. In: IWOMP (2009)
14. Intel® Corporation: Interprocedural Optimization. https://software.intel.com/en-us/node/522666 (2017)
15. International Electrotechnical Comission: IEC 61508, Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems, Edition 2.0 (2009)
16. International Organization for Standardization: ISO/DIS 26262. Road Vehicles – Functional Safety (2009)
17. Kornecki, A.J.: Software Development Tools for Safety-Critical, Real-Time Systems Handbook. Office of Aviation Research and Development, FAA (2007)
18. Kroening, D., Poetzl, D., Schrammel, P., Wachter, B.: Sound static deadlock analysis for C/Pthreads. In: ASE (2016)

19. Lakshmanan, K., Kato, S., Rajkumar, R.: Scheduling parallel real-time tasks on multi-core processors. In: RTSS (2010)
20. Liao, C., Quinlan, D.J., Panas, T., De Supinski, B.R.: A ROSE-based OpenMP 3.0 research compiler supporting multiple runtime libraries. In: IWOMP (2010)
21. Lin, Y.: Static nonconcurrency analysis of OpenMP programs. In: IWOMP (2008)
22. Ma, H., Diersen, S.R., Wang, L., Liao, C., Quinlan, D., Yang, Z.: Symbolic analysis of concurrency errors in OpenMP programs. In: ICPP (2013)
23. Marongiu, A., Burgio, P., Benini, L.: Supporting OpenMP on a multi-cluster embedded MPSoC. Microprocessors and Microsystems 35(8) (2011)
24. Melani, A., Serrano, M.A., Bertogna, M., Cerutti, I., Quinones, E., Buttazzo, G.: A static scheduling approach to enable safety-critical OpenMP applications. In: ASP-DAC (2017)
25. Münchhalfen, J.F., Hilbrich, T., Protze, J., Terboven, C., Müller, M.S.: Classification of common errors in OpenMP applications. In: IWOMP (2014)
26. Naik, M., Park, C.S., Sen, K., Gay, D.: Effective static deadlock detection. In: ICSE (2009)
27. Netzer, R.H., Miller, B.P.: What are race conditions?: Some issues and formalizations. LOPLAS 1(1) (1992)
28. Royuela, S., Duran, A., Liao, C., Quinlan, D.J.: Auto-scoping for OpenMP tasks. In: IWOMP (2012)
29. Royuela, S., Duran, A., Martorell, X.: Compiler automatic discovery of ompss task dependencies. In: LCPC (2012)
30. Royuela, S., Ferrer, R., Caballero, D., Martorell, X.: Compiler analysis for OpenMP tasks correctness. In: International Conference on Computing Frontiers (2015)
31. Royuela, S., Martorell, X., Quinones, E., Pinho, L.M.: OpenMP tasking model for Ada: safety and correctness. In: AE (2017)
32. Serrano, M.A., Melani, A., Bertogna, M., Quinones, E.: Response-time analysis of DAG tasks under fixed priority scheduling with limited preemptions. In: DATE (2016)
33. Serrano, M.A., Melani, A., Vargas, R., Marongiu, A., Bertogna, M., Quinones, E.: Timing characterization of OpenMP4 tasking model. In: CASES (2015)
34. Süß, M., Leopold, C.: Common mistakes in OpenMP and how to avoid them. In: OpenMP Shared Memory Parallel Programming (2008)
35. Vargas, R., Quinones, E., Marongiu, A.: OpenMP and timing predictability: a possible union? In: DATE (2015)
36. Wong, M., Klemm, M., Duran, A., Mattson, T., Haab, G., de Supinski, B.R., Churbanov, A.: Towards an error model for OpenMP. In: IWOMP (2010)