OpenMP-RT: Native Pragma Support for Real-Time Tasks and Synchronization with LLVM under Linux

Brayden McDonald
North Carolina State University
Raleigh, United States
bwmcdon2@ncsu.edu

Frank Mueller
North Carolina State University
Raleigh, United States
fmuelle@ncsu.edu

Abstract

Increasing demands in processing power for real-time systems have been met by adding more cores to embedded architectures. However, conventional programming languages lack adequate support for parallelism with real-time constraints. To fill this gap, add-on design tools have been created, often paired with real-time operating systems, to establish temporal guarantees, but they lack native language support for expressing parallelism at a combination of coarse and fine levels. OpenMP would be a good fit to specify such parallelism as it comes with mature compiler and runtime technology for mapping concurrency to execution units via the operating system. However, OpenMP lacks real-time support for scheduling and synchronization under deadline constraints.

We analyze the suitability of existing OpenMP for coarse-grained parallelism of real-time systems and identify key points that prevent it from being used as-is. We develop a framework, OpenMP-RT, which aims to address these shortcomings by creating a real-time task construct, with a focus on `periodic` tasks, along with a robust specification for parallel real-time applications utilizing OpenMP-RT, including support for time-predictable lock-free inter-task communication across priority levels. We implement OpenMP-RT in the LLVM C compiler under OpenMP 5.1. We develop a test suite based on a variety of benchmarks with real-time constraints and demonstrate the correctness and ease of use of our OpenMP-RT implementation, as well as the performance and predictability of multiple different paradigms for inter-task communication.

CCS Concepts: • **Computer systems organization** → **Real-time system specification**; • **Theory of computation** → **Parallel computing models**.

Keywords: OpenMP, Real-Time, Scheduling

LCTES '24, June 24, 2024, Copenhagen, Denmark

ACM ISBN 979-8-4007-0616-5/24/06
https://doi.org/10.1145/3652032.3657574

ACM Reference Format:
Brayden McDonald and Frank Mueller. 2024. OpenMP-RT: Native Pragma Support for Real-Time Tasks and Synchronization with LLVM under Linux. In *Proceedings of the 25th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '24), June 24, 2024, Copenhagen, Denmark.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3652032.3657574

1 Introduction

Computer systems are a component of nearly every sufficiently complex system in the modern world. A significant number of these computer systems are so-called real-time systems, or systems subject to timeliness constraints in addition to algorithmic correctness requirements. Often, real-time applications are subject to tight timing bounds caused by external factors, where a delay in processing could have disastrous consequences, i.e., a cyberphysical system controlling industrial equipment, where failure to respond to a control signal in time could result in costly damages, injury, or even loss of life. However, as powerful processors become cheaper and more ubiquitous, we increasingly see real-time software expected to share a processor with lower priority non-real-time processes. The solution, in many cases, has been to isolate the processes as much as possible to ensure the timing guarantees for real-time processes are met.

Multi-core processors are another computing advancement that is making inroads into the area of real-time systems, driven be the drop in cost that such devices have seen as they have spread through the market [16]. However, while scheduling real-time tasks in a single-core context is a well-studied problem with known solutions, real-time scheduling on multicore processors is still an open problem, with many different frameworks and methods being actively researched.

Many frameworks for parallel processing have been created to streamline the creation of multi-threaded programs [4, 12, 20]. The OpenMP standard, with its numerous different implementations, is one of the most widely used and influential paradigms in parallel programming. Created initially for high-performance computing applications, OpenMP has since seen expanded use cases, and is now being utilized in embedded and real-time areas. This trend has been increasingly evident in recent years, as OpenMP has received numerous updates and improvements to its functionality [4, 11, 13].

A key challenge in developing real-time systems is in satisfying timing constraints as well as algorithmic correctness. Frequently, such systems are subject to tight timing bounds [6], not just in terms of speed, but *predictability* as well. Timing bounds establish predictable execution but necessitate special algorithms for scheduling the workload on the processor. In addition, the existence of such timing bounds imposes restrictions on synchronization between threads; parallel frameworks operating in this space must ensure that blocking caused by common synchronization primitives (e.g., mutual exclusion locks) does not interfere with timing guarantees of threads.

OpenMP, however, lacks a mechanism for *periodic* task execution [2]. OpenMP does allow for an expression of dependencies between tasks, represented as a directed acyclic graph, but is entirely agnostic to the timing requirements of the system. In addition, OpenMP's scheduling focuses on assigning tasks to threads (creating new threads where necessary), but leaves the scheduling of threads onto cores up to the scheduler of an operating system (OS) [1]. This limits the usefulness of OpenMP in hard real-time to cases where the OS itself can provide real-time guarantees to all threads, but the OpenMP runtime lacks real-time OS awareness. This prevents OpenMP programs from taking advantage of real-time scheduling features, e.g., even for the widely-used Linux kernel featuring real-time capabilities via the `PREEMPT_RT` patch for earliest deadline first (EDF) and static priority (SP) scheduling of threads. A complex and error-prone sequence of system calls with the correct parameters is the only way to create real-time tasks within such threads and to realize periodic releases of their jobs. Furthermore, task synchronization has to become real-time predictable, which is not the case for OpenMP runtime systems.

The need for a system that can automatically execute periodic real-time tasks in a parallel environment without requiring extensive coding is well-established. Such a system would reduce manual coding errors caused by API complexity that easily lead to errors during task execution. Our solution is to expand the existing OpenMP capabilities by introducing the "OpenMP-RT" framework. OpenMP-RT provides novel pragma clauses for `rttask` construct and several new clauses, which together introduce a model for periodic, real-time tasks to run on multicore architectures using the OpenMP API, constructs of lock-free reading and writing to/from shared data between rttasks, and a syntax for a configuration file to define the behavior of a real-time application composed of rttasks. We have created an implementation of the OpenMP-RT by enhancing the OpenMP 5.1 implementation in the LLVM compiler, including support for these features in Clang and LLVM's OpenMP runtime [8].

The key contributions of the work are:

- OpenMP-RT is proposed, which provides novel capabilities to the OpenMP specification by creating the foundation for developing periodic real-time applications supporting coarse- and fine-grained parallelism, hierarchical scheduling, and inter-task communication.
- An implementation of OpenMP-RT targeting the C/C++ languages in the LLVM compiler implementation of OpenMP 5.1 is created.
- The correctness of OpenMP-RT is assessed through an automated task generator, which incorporates task sets from multiple OpenMP benchmarks, including the EPCC and BOTS benchmarks [3, 5], as well as a comparison of different paradigms for lock-free and lock-based inter-task communication.

2 Related Work

Significant interest has been raised in utilizing the benefits of multicore architectures in real-time systems, despite the difficulties inherent to real-time scheduling in multi-core systems [7, 9, 14, 17]. In this context, a number of studies have explored the potential of OpenMP for parallel processing and specifically for in real-time systems with a focus on the assessment of timing constraints guaranteed by existing OpenMP implementations [17, 18]. However, the results of these studies indicate limitations in predictability rooted within OpenMP, which was not designed with real-time constraints in mind but was instead using a best-effort model [4]. In contrast to prior work, OpenMP-RT contributes not only novel functionality to OpenMP via new pragma and clause support but also have a well defined semantic in support of real-time requirements for predictable timing, prioritized scheduling and bounded synchronization costs. We have also provided a *full implementation and empirical evaluation* to accompany these developments, whereas prior work was constrained to conceptual studies into the addition of syntactical constructs to the OpenMP runtime [15] without implementation.

Serrano et al. [13] discuss the suitability of OpenMP for developing multi-core real-time applications and propose a number of extensions to the OpenMP specification that improve support for real-time systems. They note the relative ease of fine grained, within-task parallelism using OpenMP, with their changes focusing on support for coarse-grained parallelism. Serrano addresses the lack of recurrent tasks in OpenMP by extending the `task` construct with an `event` clause, which accepts an expression argument. A new job from the associated task is released whenever the expression in the task's associated `event` clause evaluates to true. The paper notes that there are multiple ways to implement the mechanism behind recurrent tasks, but leaves a full exploration of this implementation outside its scope. They further extend the OpenMP `task` construct with `deadline` and `priority` clauses, which specify those features of the real-time task to the scheduler. The paper describes the requirements for the OpenMP specification to incorporate a static priority or EDF scheduler. OpenMP-RT also focuses

on adding coarse-grained parallelism in real-time systems by extending OpenMP. However, OpenMP-RT incorporates an entirely new construct, `rttask`, rather than extend the functionality of the `task` construct. In addition, OpenMP-RT explicitly denotes the period in each `rttask` through the `period` clause, rather than using Serrano's event model. This frees the user from needing to manually write the code to produce periodic events and allows them to instead rely on OpenMP to enforce task periodicity. OpenMP-RT also explicitly defines how multiple tasks using different scheduling algorithms may coexist as part of the same system via hierarchical scheduling. Our work includes an implementation of our proposed extension, along with sample applications developed using this framework. The sample implementation maps real-time behavior onto the Linux scheduler, making use of real-time support given by PREEMPT_RT patch as well as static priority and EDF real-time scheduling. OpenMP-RT is, to our knowledge, the first integration of PREEMPT_RT features with OpenMP, and thus the first to provide hard real-time guarantees using OpenMP under Linux.

Sun et al. [15] analyze the timing bounds of OpenMP's `tied` tasks, which are required to execute on the same thread throughout its life cycle. They determine that the existing scheduling method used in OpenMP for `tied` tasks is not suitable for fine-grained parallelism in real-time work, as it may lead to effectively sequential execution by tying all OpenMP tasks to a single thread. To solve this, they develop a novel scheduling algorithm for `tied` tasks and derive methods of computing response time bounds for task sets using this scheduling method, assuming a one-to-one equivalence between threads and cores. Their scheduling algorithm forces a more even spread of tasks across available threads, reducing the number of cases where all tasks are assigned to a single thread and executed sequentially. OpenMP-RT differs from the Sun method in that it focuses on coarse-grained parallelism between periodic real-time tasks (we recall that the OpenMP task model is not a sufficient abstraction for periodic real-time tasks [2]), with support for fine-grained parallelism within real-time tasks. In addition, OpenMP-RT permits multiple scheduling paradigms to coexist (even including lower-priority non-real-time threads) within the same system, allowing for the development of hybrid scheduling systems.

3 Design of OpenMP-RT

The objective of this work is to provide a framework that enhances the development of multi-threaded real-time applications using OpenMP. This reduces the need for developers to program error-prone sequences of runtime API and system calls for defining real-time threads, and to eliminate the time-consuming task of creating a method for parallel real-time components to communicate in a way that minimizes contention, reduces latency, and prevents deadlocks. This framework also eliminates the need to manually manage the notion of absolute time in periodic invocations, which can lead to unwanted jitter. Developers should not have to concern themselves with low-level details, which vary between operating systems, and are distractions from focusing on the core application logic. Instead, real-time properties will be guaranteed automatically by the generated code derived from high-level specifications, OpenMP-RT pragmas, and their runtime extensions. OpenMP is a powerful programming model, but currently lacks real-time guarantees, particularly the ability to define periodic behavior for threads, or for real-time scheduling paradigms (such as EDF or static priority), to be enforced on threads. OpenMP-RT provides developers with an effective tool for easily creating parallel, time-sensitive applications, while reducing the development overhead required to achieve these goals.

Chiefly, we accomplish these goals with the inclusion of three new pragmas: the `rttask` construct, to enclose the code of a periodic real-time task, and two pragmas `rtread` and `rtwrite`, which enclose access to shared memory between parallel tasks in the application (we note that `rtread` and `rtwrite` need not be invoked solely within an `rttask` context, i.e., communication with non real time tasks is possible in a well-defined manner). Alongside this, a specification of real-time properties for all rttasks is given, along with place mappings and dependency clauses for each one, via a configuration file with a concise syntactic notation.

3.1 Design Overview

At a high level, the design of OpenMP-RT can be broken down into two distinct parts: (1) A framework governing specifications for the scheduling of real-time tasks and their assignment to cores; and (2) a framework for establishing data channels for communication between tasks, including an API (implemented as a set of pragmas) supporting both real-time and non-real time producers and consumers. Specific information on each of these components is discussed in subsequent subsections.

We define each rttask, in addition to the standard real-time properties of `period`, `deadline`, `phase`, and `wcet`, to have a non-null set of cores on which it is allowed to execute, dubbed the `places` set. Further explanation of how these properties of each rttask are passed to the compiler is detailed in Subsection 3.3.

3.2 Execution Model

OpenMP-RT's execution model supports both coarse-grained inter-task parallelism through the `rttask` pragma, as well as fine-grained intra-task parallelism (syntactically through the use of OpenMP parallel sections within an rttask context). Our execution model features a single periodic thread for each real-time task, along with a static pool of threads per priority level (see Figure 1). The size of this pool is determined at compile time, based on the upper bound of the

number of possible threads that may execute simultaneously at a given priority level. This upper bound can be determined by analyzing the configuration file defined below.

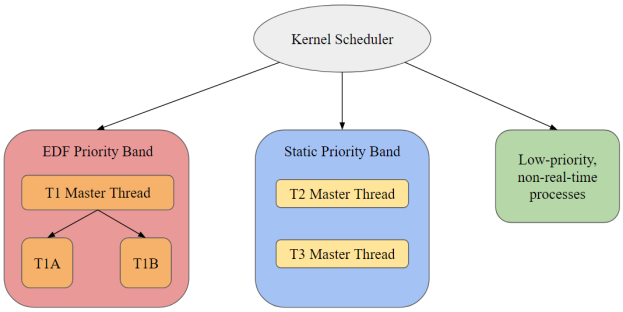

Figure 1. Hybrid scheduling system, showing the ordering between the priority bands.

Each periodic rttask thread has a defined subset of cores in which it is permitted to run. Upon entering a parallel section, the rttask will attempt to wake up a number of available threads from the pool for its associated priority required to execute the parallel section and assign a workload to each thread to execute following the fork-join model. An example of this is shown in the upper portion of Figure 2, where 2 periodic tasks in a 4-core system each encounter a parallel section requiring 2 threads. Upon section entry, each periodic task acquires a thread from the thread pool with matching priority level and assigns it to an available idle core.

We note, at this point, that it is possible for an rttask to not receive the number of threads it has requested, if, e.g., another rttask of the same priority is executing its own parallel section and has already acquired threads from the pool. This will result, in the worst case, in one rttask executing its parallel section sequentially (see example in the lower half of Figure 2). This problem can be mitigated by either incorporating pessimistic assumptions about sequential execution into worst case execution time analysis, or through isolating rttasks of the same priority to different subsets of cores, such that there will be no competition between them.

Consider the case of two rttasks, A and B (indicated as blue and red respectively in Figure 2, of the same priority running in parallel, where each contains a parallel section that requests 3 threads with a `places` set containing 3 cores. With 4 total cores available, by the pigeonhole principle, there is a minimum of 2 cores available to both A and B. In this case, the idle cores available to both tasks. In this scenario, it is possible for A to reach its parallel section first and acquire the two remaining idle cores, leaving B to execute sequentially, as seen in the lower half of Figure 2. It is equally possible for B to be first and starve A. As a result, both tasks must have their WCETs calculated as if they were sequential; a very pessimistic outcome. This can be avoided by isolating A and B such that there is minimal

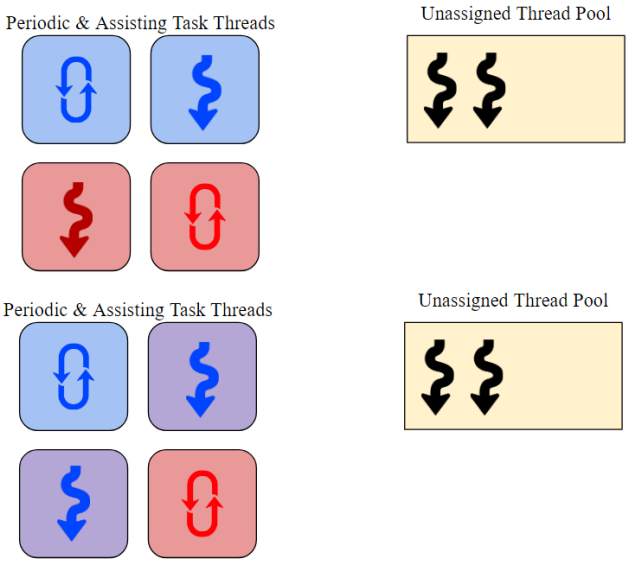

Figure 2. Periodic task threads, mapped to available threads from the thread subpool with matching priority level upon entering a parallel section. Top half: balanced load, with red & blue tasks isolated on separate pairs of cores. Lower half: potential result from red & blue tasks sharing cores: blue acquires both cores reducing red to sequential execution.

overlap in their sets of available cores. With cores divided into isolated pairs, we can guarantee 2 threads per task, a marked improvement over the previous WCET (depicted in the upper portion of Figure 2).

3.3 Real-Time Specifications via a Configuration File

The configuration file includes specifications for all rttasks, along with place mappings and dependency clauses for each one. Each rttask is associated with a name string, which identifies it within `rttask` pragmas in the application code. The configuration file is passed to the tool chain at compile time along with the source file. Passing this set of task specifications and environment variables contained within the configuration file allows for compile-time static analysis of the application task set to be performed, which will be elaborated upon below. We assume an equivalency between cores and places in this paper; such an equivalency is well-supported by the OpenMP specification.

The configuration file starts with a line defining the expected value of the OMP_PLACES environment variable. Subsequently, a second line indicates the subset of places (i.e., cores) on which any non-real-time code in the application will be permitted to run. Each subsequent line in the configuration file defines one rttask. The specification of an rttask is composed of the following parameters:

- `name` (required): Specifies the name which will identify the `rttask` pragma associated with this definition.

Listing 1. Sample configuration file.

```
ompplaces   "{0,1,2,3,4,5,6,7}"
nonrtplaces "6,7"
task name(taskbench_SP)                                    period(400)  wcet(200)            priority(30) threads(2) place(4,5)
task name(fft_SP)     depend(in: din)  depend(out: dout) period(600)  wcet(200)            priority(40) threads(1) place(4,5)
task name(fft_inv_SP) depend(in: dout) depend(out: din)  period(600)  wcet(200) phase(300) priority(40) threads(1) place(4,5)
task name(fib_1_EDF)  depend(in: N1)                     period(300)  wcet(250)                         threads(1) place(2,3)
task name(fib_2_EDF)  depend(in: N2)                     period(300)  wcet(250) phase(800)              threads(1) place(2,3)
task name(T1)                                            period(100)  wcet(20)                          threads(2) place(0,1)
task name(T2)                                            period(200)  wcet(40)  phase(50)  priority(10) threads(1) place(0,1)
task name(T3)                                            period(200)  wcet(40)  phase(100) priority(20) threads(1) place(0,1)
```

- `period` (required): Specifies the period of the rttask via a positive integer expression.
- `deadline`: Specifies the relative deadline of the rttask via a positive integer. If omitted, it defaults to the period.
- `phase`: Specifies the phase (non-negative integer) of the rttask, i.e., how long to delay the first execution. If omitted, it defaults to a zero phase and the task launches immediately upon encountering the associated `rttask` construct.
- `wcet`: Specifies the expected worst-case execution time of the rttask via a positive integer expression.
- `depend`: Establishes a task's connection to a communication channel involving shared memory. Uses a subset of the syntax for existing `depend` clause.
- `priority`: Specifies the static priority of the rttask via a positive integer. Required for static priority tasks. If omitted, the task is scheduled under EDF.
- `threads`: Specifies the number of threads requested by the task.
- `place` (required): Specifies the set of places (cores) to which this task is restricted. In the case of intra-task parallelism, (such as an rttask containing a parallel section), the each individual thread associated with the task will be restricted to this set.

In our work, the time units associated with the `period`, `deadline`, `wcet` and `phase` clauses are defined in microseconds. Minimally, any rttask scheduled under EDF must have a defined period plus WCET, and any rttask scheduled under static priority must have both its period and priority defined. Thus, inclusion or exclusion of the priority value also serves to define the scheduling behavior (i.e., either EDF or static priority) of the rttask. An example configuration file for a 7-task OpenMP-RT parallel application is presented in Listing 1. The first line indicates the value of the OMP_PLACES environment variable the application should expect at runtime, in this case an 8-core system with cores labeled 0 though 7. In the event that, at application runtime, the OMP_PLACES environment variable is different, the application will return an error and terminate. The real-time guarantees provided by OpenMP-RT cannot be guaranteed if the OMP_PLACES variable is incompatible with the specification given in the configuration file; this would invalidate core bindings for rttasks, resulting in undefined mapping of threads to cores, i.e., a problem that was not accounted for in the scheduling analysis during development. The second line provides a whitelist of cores that non-real-time processes will be restricted to. In this case, cores 6 and 7 (and thus, by extension, cores 0-5 are reserved exclusively for real-time tasks).

Each subsequent line defines one rttask. For example, line five defines the rttask `fft_inv_SP`. It has two dependencies, reading from shared variable `dout` and writing to shared variable `din` with a period of $600\mu s$, a phase delay of $300\mu s$ (half the period), and its worst case execution time has an upper bound of $200\mu s$. As it has a defined priority, it is scheduled under static priority scheduling, placing it below the `taskbench_SP` task in priority, and below all of the EDF-scheduled tasks in terms of priority. It is indicated as having only one thread (i.e., it does not feature any intra-task parallelism), and is restricted such that it may only execute on cores 4 or 5, as indicated by the `places` parameter.

3.4 The `rttask` Construct

Our framework introduces the `rttask` construct, which encloses the code for a periodic task. The `rttask` pragma takes only a single string-value clause, `name`. This is because the necessary information to compose a real-time periodic task (period, phase, deadline, scheduling behavior, priority, etc.) have already been associated with a name string within the configuration file. Therefore, merely invoking the name within the `rttask` pragma provides the compiler with all the required information for the task.

All required threads (both periodic master threads and the pools of helper threads used for intra-task parallelism) are created up front to avoid the high overhead of dynamic thread creation. Thus, execution of all rttasks is delayed until this setup is complete, with a further delay for each task as defined by its `phase` property.

The `rttask` pragma works only outside the context of other OpenMP pragmas. As rttasks use dedicated, real-time periodic threads, they are incompatible with the way that

Listing 2. Sample program using rttasks communicating over shared variables `din` and `dout` via the retry method.

```
#pragma omp rttask name(FFT_SP)
{
  float * local_cpy;
  int flg;
  #pragma omp rtread source(din) dest(local_cpy) size(10000) numtries(2) flag(flg)
  fft(local_cpy);
  #pragma omp rtwrite source(local_cpy) dest(dout) size(10000)
}
#pragma omp rttask name(FFT_inv_SP)
{
  float * local_cpy;
  #pragma omp rtread source(dout) dest(local_cpy) size(10000) numtries(2) flag(flg)
  fft(local_cpy);
  #pragma omp rtwrite source(local_cpy) dest(din) size(10000)
}
```

ordinary OpenMP parallel sections are implemented. The presence of an rttask declaration inside of a `parallel for` would result in undefined behavior. However, a subset of existing OpenMP constructs are usable *within* an rttask construct. This is how intra-task parallelism is supported in OpenMP-RT.

In particular, the OpenMP `task` and `parallel` pragmas retain the same syntax when invoked inside an rttask context. The implementation and behavior of these pragmas are slightly modified to account for OpenMP-RT's real-time guarantees. Specifically, parallel regions inside of an rttask construct will never create (or destroy) native threads. When assembling a thread team to execute a parallel region, threads must be pulled from the pre-existing pool of threads for the enclosing rttask's priority level. Similarly, threads are never destroyed upon reaching the end of a parallel section. Instead, they are returned to the pool of available threads. We note that, although the *existence* of the requested number of threads is guaranteed (as they must have already been created before any rttask is allowed to launch), availability is not. It is possible that a concurrently-executing task of the same priority and place bindings has already acquired them. Availability of threads *can* be guaranteed, however, if there is no overlap between the place bindings of tasks at a given priority level. This property may be statically checked by analyzing the configuration file. Note that the threads involved in this process do not need to inherit the priority of the rttask's main thread. They already *have* the same priority and are simply idle awaiting assignment to a job.

Within a parallel section, OpenMP-RT also supports `task` pragmas. OpenMP tasks within rttasks feature the same syntax and similar behavior to OpenMP tasks in non-real-time sections, including synchronization constructs such as `taskwait` or critical sections. As each rttask establishes its own context for parallel operations, synchronization directives (such as critical sections) only take effect with respect to that rttask. It is perfectly valid to have two threads execute critical sections simultaneously, provided they are each part of teams corresponding to different rttasks.

3.5 Inter-task Communication Framework

Communication between rttasks is handled through a lock-free framework for sharing data between rttasks, as well as between real-time and non-real-time threads. Two frameworks with different semantics are developed in this paper for comparison: (1) a retry-based framework (implemented as pragmas `rtread` and `rtwrite`), and (2) a double-buffer based one (implemented as `rtreadbuffer` and `rtwritebuffer`). Both are built upon simple atomic primitives. Lock freedom is critical for communication between non-rt and real-time threads, as allowing a non-rt thread to hold a resource relied upon by a real-time thread (and thus block that real-time thread), would violate real-time requirements. (Notice that priority inheritance is not a solution since non real-time tasks cannot inherit real-time privileges under Linux). However, there are use cases for non-real-time tasks to communicate in a two-way channel with real-time tasks. This is to support, for instance, the user interface of a self-driving vehicle. An sample application of `rtread` and `rtwrite` is featured in Listing 2.

A simpler, lock-based setup using Priority Ceiling Emulation Protocol is also provided (pragmas `rtreadlock` and `rtwritelock`), featuring per-channel mutex locks for diagnostic purposes and this work's experimental evaluation. Deadlock avoidance is ensured by creating a total order over all shared data, and ensuring that lock and unlock operations follow this order in the automatically generated code (with unlocks done in reverse). This is a sufficient condition for deadlocks to not occur [19].

The dependency clauses for each rttask defined in the configuration file provide the names of the identifiers, but also indicate the relation each rttask has to the relevant

shared data. An "in" dependency indicates that a task only reads from but never writes to that shared memory (i.e. a consumer), and an "out" dependency indicates write only without any reads (i.e., a producer).

3.5.1 Retry-based method. For communication channels under the retry-based method, each dependency variable has two associated timestamp registers in addition to the data value held. These timestamp registers are used to verify the integrity of the data, which supports single-producer, multi-consumer lock-free setups.

The `rtread` pragma accepts the following clauses:

- `source` (required): Specifies the shared memory variable to read from. Identifier must be listed as an in dependency in the configuration, or the application will not compile.
- `dest` (required): Specifies the address where the values from `source` should be written. Identifier must be private to the executing thread (thus, cannot be a task dependency), or will throw a compile-time error.
- `size` (required): Specifies the length, in bytes, of the data to be copied.
- `numtries`: Specifies number of times to attempt reading from `source` if timestamps do not match. If this many reads are attempted with no success (i.e., timestamps never match), then the no further attempts will be made to read from `source`, and the value of flag will be set to 1. If unspecified, defaults to 1.
- `flag` (required): points to a flag, which will be zero after a successful read and 1 if there was no successful read within the allotted retries.

And the `rtwrite` pragma accepts the following clauses:

- `source` (required): Specifies the local variable to read from. Must be private to the executing thread (thus, cannot be a task dependency), or will throw a compile-time error.
- `dest` (required): Specifies the shared variable the values from `source` should be written. Identifier must be listed as an out dependency in the configuration, or the application will not compile.
- `size` (required): Specifies the length, in bytes, of the data to be copied.

During an `rtwrite` to a given shared variable, the current timestamp is copied into the first timestamp field, then the data is written from source (local) to destination (shared). Once this copy is complete, the timestamp written to the first timestamp register will be copied over to the second register. This ensures that during the writing process, when the data is invalid, the timestamp registers will not match.

During an `rtread`, the value of flag is set to 1, the value of the first timestamp register is recorded, then data from the shared source is copied to the local destination. Once the copy completes, the second timestamp register is checked against the recorded first one. Upon match, a read is considered successful, data is accepted as valid, the flag is set to 0, and execution resumes on the calling thread.

Upon mismatching timestamps, the data must have been overwritten to during the time that it was being read. This invalidates the read data, and we must read again or proceed without the data (e.g., using old data from the last successful read, which is left to the user to decide). The read operation described above will be repeated until either the timestamp registers match, or the total number of reads attempted exceeds the `numtries` parameter. If this occurs, the flag value will remain as 1 when execution returns to the calling thread, indicating to the user that the read failed.

The number of retries of lock-free reads can be upper bounded by design. Consider a rttask A that can be preempted by h higher priority rttasks. If the number of higher priority jobs in the longest busy interval is b, then $b + 1$ is a safe upper bound for the number of reads required before timestamps match assuming that the time required for the reads does not exceed the smallest idle time between busy intervals. In practice, the bound may be much smaller. If rttasks are in phase, then only the number of out-of-phase rttasks with higher priority jobs in the busy interval need to be considered for b (as in-phase ones do not yield to the lock-free read of a lower priority task). In general, lock-free communication is preferable for small data as it avoids excessive lock/unlock protocol overhead whereas lock-based should be used for large data, e.g., large images or matrices. However, communication with non-real-time tasks requires lock-free protocols in OpenMP-RT. This should not be a problem if either prior data (i.e., the last image frame) is used upon an unsuccessful rtread or if partial updates to a frame are simply tolerated (e.g., line fragmentation during videos tolerated by the human eye).

3.5.2 Double-Buffer method. An alternative to lock-free data was also developed. It is implemented using pramas `rtreadbuffer` and `rtwritebuffer`. Notice that the pragma `rtreadbuffer` uses the same source/dest/size clauses as `rtread` but ignores numtries and flag as it does not rely on retrying reads. Similarly, `rtwritebuffer` has identical syntax to `rtwrite`. Only the implementation is changed.

The Double-Buffer method works by establishing two buffers for each dependency (the addresses of which are held by the producer, as indicated by the `depend(out:)` clause in the configuration file), and a single shared pointer field. Conceptually, one buffer (indicated by the shared pointer field) is considered 'readable' at a time, and the other buffer is thus 'writable'. Upon the completion of a write operation, the buffers are swapped by overwriting the shared pointer field with the address of the most-recently written to buffer. This action (performed atomically) makes the previously-writable buffer readable and the old readable buffer available to be written to.

Note that read operations under the double-buffer method have no retry requirements as there is no chance of failure. Even in the event that the producer performs a write operation while the consumer is reading, this does not invalidate the data read by the consumer, as the new data was written to a separate buffer.

We compare the performance and advantages of the Retry and Double-Buffer methods in the experimental section.

3.5.3 Lock-based benchmark method.

The lock-based communication method is included as a baseline for comparing the two lock-free communication methods. It is based on a total ordering of locks and strict adherence to this total ordering, which guarantees deadlock freedom [19].

The `rtreadlock` pragma has the same arrangement of clauses and syntax as the `rtread` pragma, except it has no need (or support) for the `numtries` and `flag` clauses, as `rtreadlock` simply waits to acquire a lock, rather than try again. The `rtwritelock` pragma has a syntax and use identical to the lock-free `rtwrite`, (i.e., source, dest, and size clauses). The only difference between the two is in behavior and implementation, as, naturally, `rtwritelock` is implemented using locks, rather than timestamp checking.

4 Implementation

This work includes an implementation of OpenMP-RT via extending the LLVM compiler's OpenMP language and runtime support. The OpenMP-RT implementation supports only the C language running on Linux with the PREEMPT_RT patch and static priority plus EDF scheduling. Further language support (e.g., C++) is an engineering effort, whereas the objective of this implementation is to provide a working prototype of OpenMP-RT as a proof-of-concept for its feasibility and to support experimental evaluations of the OpenMP-RT design using a benchmark test suite. Similarly, support was limited to the Linux kernel, as it is a widely-used, well-understood operating system that is enjoying wide adoption (for soft real-time systems and even some hard real-time systems, e.g., by Tesla, with all its current shortcomings). With the PREEMPT_RT patch and subsequent real-time extensions, Linux now provides the required OS functionality for OpenMP-RT, specifically for EDF and static priority scheduling support in higher priority bands than used by conventional non real-time tasks (threads and processes alike, know as Linux tasks). In addition, as the Linux kernel supports non-real-time tasks, this gives the implementation the opportunity to demonstrate *hybrid scheduling* support.

As seen in Section 3, OpenMP-RT provides multiple frameworks for safe data transfer between rttasks, as well as across the real-time boundary. Listing 3 features a pseudocode example of the retry-based process (see Section 3), which is executed upon reaching the `rtread` and `rtwrite` pragmas, represented there as functions.

Listing 3. Pseudocode of lock-free communication.

```
rtread(src, dest, size, numtries, flag){
  flag = 1
  i = 0;
  for(i <= numtries){
    timestamp = src.timeReg1
    copy(src.data, dest, size)
    if(timestamp == src.timeReg2) {
      flag = 0
      break
    }
  }
}
rtwrite(src, dest, size){
  dest.timeReg1 = clock_gettime()
  copy(src, dest.data, size)
  dest.timeReg2 = dest.timeReg1
}
```

5 Experimental Analysis

The primary goal of the OpenMP-RT project has been to create a foundation for developing periodic real-time applications supporting coarse- and fine-grained parallelism, hierarchical scheduling, and inter-task communication. In order to examine the effectiveness of OpenMP-RT, it was necessary to analyze a large set of applications featuring a variety of behaviors. However, manually creating the set of dozens of applications required for such testing would take an extremely long time. To this end, an automatic benchmark application generator was created, which composes a parallel OpenMP-RT application from individual benchmark tasks drawn from two OpenMP benchmark task sets: the Barcelona OpenMP Task Set (BOTS) [5], and the EPCC Microbenchmark suite [3]. Table 1 shows the experimentally-determined WCETs for each of the tasks that make up the BOTS and EPCC benchmark suites.

Experiments were conducted on a 16-core x86 machine with 8GB of DRAM at 2133 MHz under Ubuntu version 21.04 and Linux kernel 5.13 with the PREEMPT_RT patch enabled. Worst-case execution time (WCET) values were extracted experimentally for purposes of these demonstrations as the maximum observed execution time for a given task over a 1,000 run test.

5.1 Automatic Task Set Generation

The automatic benchmark application generator composes a synthetic task set by taking pre-written single tasks (in this case drawn from BOTS and EPCC benchmarks), composing these pre-written tasks together into a single application, where each benchmark represents a single rttask, and then generating a randomized OpenMP-RT configuration file to define the synthetic application.

Table 1. WCET estimation for BOTS and EPCC benchmark tasks for sequential and 2-threaded versions.

Task	WCET (μs) (sequential)	WCET (μs) (2-core)
alignment	1237	652
fft	212	117
fib	248	130
floorplan	507	262
health	957	503
NQueens	610	319
knapsack	752	411
sort	227	142
sparselu	833	456
strassen	451	273
uts	489	251
TaskBench	197	113
ArrayBench	143	82
SyncBench	205	117
SchedBench	151	96

Generated applications may incorporate exclusively single-threaded tasks (and thereby only express inter-task parallelism), or multithreaded tasks (i.e., adding intra-task parallelism), as all benchmarks from which tasks are drawn feature `parallel for` sections with configurable thread counts. For purposes of these experiments, intra-task parallelism was limited to 1-2 threads per task.

The generator further composes applications demonstrating inter-task dependencies and data sharing. Pairs of tasks sharing dependency references in their code were created, and the generator selects both of them to be included in the application.

Particular factors of the final application can be selected (such as total utilization, presence and type of inter-task communication, intra-task parallelism, number of cores to utilize, etc.) such that a set of randomized applications meeting this specification is generated. The same randomized set of applications is subsequently adjusted around that particular variable. 100 random applications were created for each of the following tests:

- Tests on increasing utilization until we observe deadline misses.
- Comparison of lock-based and lock-free data sharing.

5.2 Utilization Stress Test

In this experiment, 100 random applications were generated, initially with the stipulation that every core would have a utilization of 0.6, and that all tasks would be scheduled under EDF scheduling. The reason for requiring EDF scheduling is that, in theory, the EDF scheduler is optimal for single-core scheduling and is capable of scheduling any task set with utilization of 1 or below [10]. While this experiment involves multiple cores, restricting scheduling to EDF still limits the likelihood of a deadline miss due to an unscheduled task set being generated by the benchmark generator. No other limitations were placed, and so the set of applications includes those with and without intra-task parallelism and inter-task communication of both types or neither. In all experiments, applications were run for 60 seconds.

Table 2. Per-core utilization and number of observed deadline misses.

Utilization	deadline misses	
	OpenMP-RT(EDF)	OpenMP
0.6	0	347
0.7	0	751
0.8	0	11,374
0.9	0	54,133
0.93	0	92,806
0.95	4	107,942
0.99	5,138	162,237

For comparison, a version of each application was created using OpenMP's `task` framework (using loops & delays for periodicity) rather than the `rttask`. This was used to compare the number of observed deadline misses as follows: OpenMP tasks do not take the same real-time execution information as RT-Tasks, however, *deadline misses* can still be measured by checking the completion time of each task. Deadlines for the OpenMP tasks were assumed to be the same as those for the corresponding RT-Task.

The numbers shown below refer to the total number of observed deadline misses across all 100 applications in their 60 second runs.

As can be seen in Table 2, RT-Tasks show a complete lack of deadline misses in any applications with a per-core utilization below 0.93, avoiding the large spike in deadline misses observed with OpenMP tasks at 0.8. Inspection of the RT-Tasks with observed deadline misses at 0.95 revealed that all of them featured inter-task communication with locks. At a per-core utilization of 0.99, deadline misses are *relatively* frequent. The large number of missed deadlines for OpenMP tasks does not indicate *failed* tasks, merely tasks which complete after the specified relative deadline for the corresponding RT-Task version. This reinforces previous findings on the unsuitability of OpenMP tasks for real-time use cases [17, 18].

5.3 Comparison of Inter-Task Communication Methods

OpenMP-RT implements 3 separate frameworks for inter-task communication: (1) a simple deadlock-free locking method based on a total ordering of locks, (2) a lock-free retry based method, and (3) a lock-free double-buffer method,

as described in Section 3. We refer to these as **Lock**, **Retry**, and **DoubleBuffer**, respectively, in the following section.

A suite of experiments was devised to analyze the behaviors of these communication frameworks, specifically focusing on the overhead incurred by each method. We analyze the response time of tasks using each framework under a variety of conditions. We also pay attention to communication *across* the real-time barrier, i.e., between real-time and non-real-time tasks. For a producer/consumer layout (as all OpenMP-RT communication frameworks are defined), this gives 3 cases to study (written as producer → consumer):

- real-time → real-time (**RT→RT**)
- real-time → non real-time (**RT→nRT**)
- non real-time → real-time (**nRT→RT**)

We note the existence of non real-time → non real-time communication as another option. However, such communication, by definition, does not involve any real-time tasks, and as OpenMP-RT is a framework focused on real-time application development, it is outside the scope of this paper (and not subject to deadlines).

An application was developed containing 4 tasks on 2 available cores arranged in 2 producer/consumer pairs. Producer tasks generated two 64x64 integer matrices, which were passed to consumer tasks, then multiplied together using a single-threaded implementation of the strassen method employed in the `strassen` benchmark. All real-time tasks were scheduled using static priority scheduling, with producers and consumers occupying the same priority band. To examine the **RT→nRT** and **nRT→RT** cases, the appropriate tasks (either producers or consumers) were run in the non real-time priority band. Experiments using the **Retry** method were conducted with the `numtries` argument set to 2. We note that the **Lock** method is incompatible with communication across the non real-time barrier as having a non real-time task block a real-time task violates real-time guarantees (and would elevate non real-time tasks temporarily into higher priority bands via inheritance). This is not the case for the lock-free methods, and so these experiments are included.

Figure 3 displays the average response time of the consumer tasks across all 3 inter-task communication methods for the 3 cases involving real-time tasks. Similar results were observed for the average response time of *producer* tasks with data collected during the same runs. We omit these results due to space and similarity.

As the figure shows, the response times are similar across all cases under lower utilization. At utilizations above 1.8 (corresponding to a per-core utilization of 0.9), consumer response time climbs much more rapidly than under the lock-free methods. At a per-core utilization of 0.95 (the highest value for which we do not observe deadline misses under lock-free communication), the tasks under lock free communication had between 26% (**Retry**) and 37% (**DoubleBuffer**)

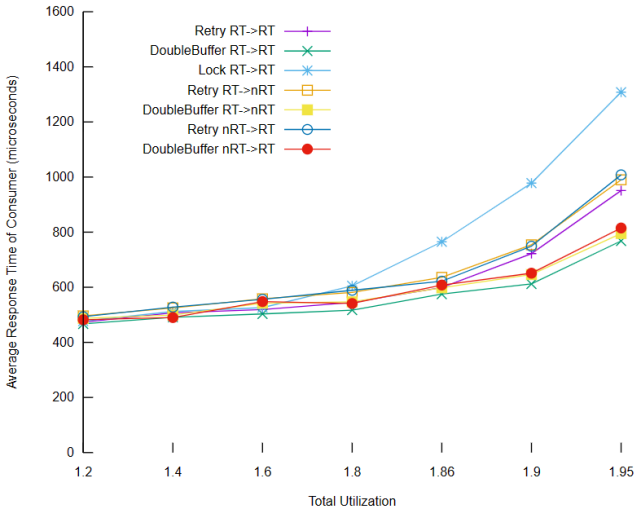

Figure 3. Average response time of a consumer vs. total utilization over 1000 runs. Notice that total utilization is across 2 cores.

lower response times than in an identical application using **Lock**. We further note that response times were not significantly different in cases where data crossed the real-time barrier in either direction.

5.4 Frequency of Re-Read for Retry-Based Communication

Another experiment was conducted to better analyze the behavior of the **Retry** method, in particular the bound on the number of retries required for a consumer to successfully read from shared data, such that its data is read repeatedly without being invalidated by the producer. As retries are prompted by the *producer* commencing a write operation before the consumer can finish reading, the frequency of retries is determined by the ratio of the periods of the producer and consumer, as well as the worst-case time spent reading the shared data.

A test application was developed consisting of 2 tasks arranged in a producer-consumer model under EDF scheduling. The producer loaded a 1MB image into shared memory, where it was then read by the consumer. This replicates the activity of a system receiving image data from a sensor or from a download. With the large transfer size and lack of other operations, nearly all of the execution time of the producer and consumer in this example are spent on accessing shared data.

Figure 4 shows the percentage of failed *initial* reads of shared data (averaged over 1000 runs) versus the ratio of the periods of the producer and consumer tasks. We denote the producer's period as P_p, and the consumer's as P_c. Data is shown for all three cases, **RT→RT**, **RT→nRT**, and **nRT→RT**, demonstrating little variation between them. We note here that the expected increase in number of failed initial reads as the ratio becomes more lopsided, i.e., when

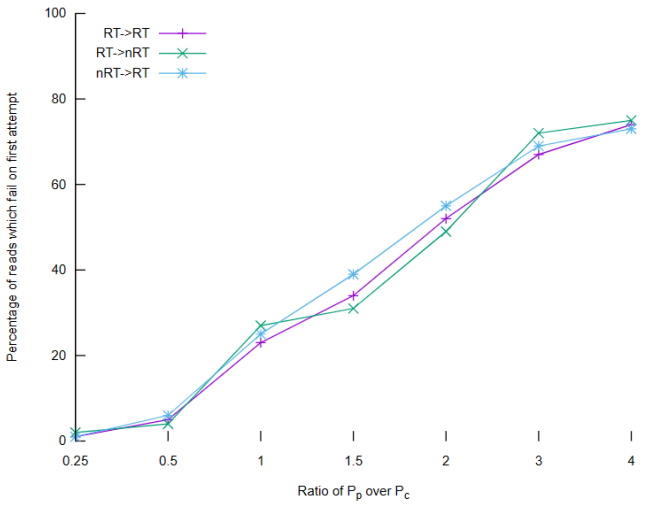

Figure 4. Failed initial reads [%] vs. ratio of P_p over P_c.

producer executes significantly more frequently than the consumer. However, this graph only depicts failed *initial* reads; with a `numtries` parameter of 2, the second read attempt *always* succeeded in these tests, which confirms that this is a safe retry bound.

6 Conclusion

We presented OpenMP-RT, a framework built upon OpenMP, targeting the development of parallel processing real-time applications, along with an implementation of this framework targeting the C language (via the LLVM compiler framework), making use of the Linux real-time scheduling features. Our implementation includes a framework for coarse-grained parallel execution of real-time periodic tasks with support for core isolation, hybrid scheduling, and multiple lock-free inter-task communication paradigms. This implementation was subsequently analyzed by using automatically-generated synthetic parallel real-time applications and tests analyzing the behavior of applications incorporating OpenMP-RT's inter-task communications.

Overall, our OpenMP-RT framework provides a simple, robust model and its implementation for developing multi-core real-time systems with particular advantages in coarse-grained parallelism between periodic tasks while still supporting fine-grained intra-task parallelism. Future work includes the expansion of the inter-task communication pragmas to cover additional methods, support for scheduling paradigms beyond EDF and static priority, and tools to assist with development, e.g., to optimize core bindings of rttasks.

Acknowledgments

This work supported in part by NSF awards CISE-17377555, CISE-1813004 and a grant by CISCO.

References

[1] Arwa Alrawais. 2021. Parallel Programming Models and Paradigms: OpenMP Analysis. In *2021 5th International Conference on Computing Methodologies and Communication (ICCMC)*. 1022–1029. https://doi.org/10.1109/ICCMC51019.2021.9418401

[2] Eduard Ayguade, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. 2009. The Design of OpenMP Tasks. *IEEE Transactions on Parallel and Distributed Systems* 20, 3 (2009), 404–418. https://doi.org/10.1109/TPDS.2008.105

[3] Mark Bull. 2002. Measuring Synchronisation and Scheduling Overheads in OpenMP. *Proceedings of First European Workshop on OpenMP* (02 2002).

[4] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. 2001. *Parallel programming in OpenMP*. Morgan kaufmann.

[5] Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. 2009. Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In *2009 International Conference on Parallel Processing*. 124–131. https://doi.org/10.1109/ICPP.2009.64

[6] David Ferry, Jing Li, Mahesh Mahadevan, Kunal Agrawal, Christopher Gill, and Chenyang Lu. 2013. A real-time scheduling service for parallel tasks. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 261–272. https://doi.org/10.1109/RTAS.2013.6531098

[7] Preeti Godabole and G.P. Bhole. 2021. Timing Analysis in Multi-Core Real Time Systems. In *2021 IEEE International Symposium on Smart Electronic Systems (iSES)*. 38–43. https://doi.org/10.1109/iSES52644.2021.00021

[8] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *International Symposium on Code Generation and Optimization*. San Jose, CA, USA, 75–88.

[9] Jing Li, Jian Jia Chen, Kunal Agrawal, Chenyang Lu, Chris Gill, and Abusayeed Saifullah. 2014. Analysis of Federated and Global Scheduling for Parallel Real-Time Tasks. In *2014 26th Euromicro Conference on Real-Time Systems*. 85–96. https://doi.org/10.1109/ECRTS.2014.23

[10] C. Liu and J. Layland. 1973. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM* 20, 1 (Jan. 1973), 46–61.

[11] Andrea Marongiu, Alessandro Capotondi, Giuseppe Tagliavini, and Luca Benini. 2013. Improving the Programmability of STHORM-Based Heterogeneous Systems with Offload-Enabled OpenMP. In *Proceedings of the First International Workshop on Many-Core Embedded Systems* (Tel-Aviv, Israel) *(MES '13)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/2489068.2489069

[12] Jiangfeng Peng, Chen Hu, and Jianqing Xi. 2009. MSI A New Parallel Programming Model. In *2009 WRI World Congress on Software Engineering*, Vol. 1. 56–60. https://doi.org/10.1109/WCSE.2009.114

[13] Maria A. Serrano, Sara Royuela, and Eduardo Quiñones. 2018. Towards an OpenMP Specification for Critical Real-Time Systems. In *Evolving OpenMP for Evolving Architectures*, Bronis R. de Supinski, Pedro Valero-Lara, Xavier Martorell, Sergi Mateo Bellido, and Jesus Labarta (Eds.). Springer International Publishing, Cham, 143–159.

[14] Lui Sha, Marco Caccamo, Renato Mancuso, Jung-Eun Kim, Man-Ki Yoon, Rodolfo Pellizzoni, Heechul Yun, Russell Kegley, Dennis Perlman, Greg Arundale, and Richard Bradford. 2016. Real-Time Computing on Multicore Processors. *Computer* 49 (09 2016), 69–77. https://doi.org/10.1109/MC.2016.271

[15] Jinghao Sun, Nan Guan, Yang Wang, Qingqiang He, and Wang Yi. 2017. Real-Time Scheduling and Analysis of OpenMP Task Systems with Tied Tasks. In *2017 IEEE Real-Time Systems Symposium (RTSS)*. 92–103. https://doi.org/10.1109/RTSS.2017.00016

[16] M. Vaidehi and T. R. Gopalakrishnan Nair. 2010. Multicore Applications in Real Time Systems. arXiv:1001.3539 [cs.SE]

[17] Roberto Vargas, Eduardo Quinones, and Andrea Marongiu. 2015. OpenMP and timing predictability: A possible union?. In *Design, Automation and Test in Europe*. 617–620. https://doi.org/10.7873/DATE.2015.0778

[18] Yang Wang, Nan Guan, Jinghao Sun, Mingsong Lv, Qingqiang He, Tianzhang He, and Wang Yi. 2017. Benchmarking OpenMP programs for real-time scheduling. In *2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. 1–10. https://doi.org/10.1109/RTCSA.2017.8046322

[19] Yin Wang, Hongwei Liao, Ahmed Nazeem, Spyros Reveliotis, Terence Kelly, Scott Mahlke, and Stephane Lafortune. 2009. Maximally permissive deadlock avoidance for multithreaded computer programs (Extended abstract). In *2009 IEEE International Conference on Automation Science and Engineering*. 37–41. https://doi.org/10.1109/COASE.2009.5234118

[20] Huabei Wu. 2008. Design-Pattern Based Parallel Programming Model and System Implementation. In *2008 4th International Conference on Wireless Communications, Networking and Mobile Computing*. 1–5. https://doi.org/10.1109/WiCom.2008.2912

Received 28 February 2024; revised 29 April 2024