# OpenMP-RT: Pragma Support for Scheduling Periodic Real-Time Tasks

Brayden McDonald[1][0009−0001−2302−4115] and Frank Mueller[2,3][0000−0002−0258−0294]

North Carolina State University, Raleigh NC 27606, USA

**Abstract.** The computational demand for real-time control systems has significantly increased, particularly with the use of machine learning, which has been met by higher core counts on embedded architectures. Mainstream programming languages support such parallelism for general purpose programming, but they lack native support for expressing parallelism under real-time constraints. While external tools and real-time operating systems offer temporal guarantees, they often fall short in providing integrated, language-level constructs for both coarse- and fine-grained parallelism. OpenMP is a natural candidate for this role, but lacks support for real-time scheduling and synchronization under deadline constraints.

This work introduces OpenMP-RT to raise a discussion for inclusion in the OpenMP standard. OpenMP-RT is a framework that extends OpenMP to support real-time periodic tasks and predictable inter-task communication. We identify the limitations of standard OpenMP for real-time applications and explore potential extensions to the OpenMP 5.1 specification to address these gaps. Our framework introduces a new task construct, `rttask`, and a configuration-driven specification model. OpenMP-RT enables time-predictable, lock-free communication across priority levels and supports both static-priority and EDF scheduling. We have implemented OpenMP-RT in the LLVM C compiler under OpenMP 5.1 and analyzed its performance in terms of deadlines and real-time synchronization.

**Keywords:** Real-Time · Parallelism · Extensions.

## 1 Introduction

Real-time systems are integral to many safety-critical and performance-sensitive applications, from industrial automation to autonomous vehicles. In addition to requirements for functional correctness, these systems must meet strict timing constraints. Periodic tasks are a foundational execution model in real-time systems, where tasks are released at regular intervals and must complete within a defined deadline. This model aligns well with many control and monitoring applications, such as sensor polling, actuator updates, and feedback loops in cyber-physical systems [1].

As computing platforms evolve, real-time software is increasingly expected to coexist with non-real-time workloads on shared hardware, particularly on multicore processors. While this shift offers benefits (e.g., in allowing easier communication between components), it also introduces challenges in ensuring predictable execution and meeting timing guarantees.

Multicore architectures have become ubiquitous as the cost of such hardware has come down and demand for compute power by software has increased [13]. However, real-time scheduling on multicore systems remains an open research problem. Traditional real-time scheduling techniques for single-core systems do not always scale well in the presence of concurrent execution and shared resources. This has led to the exploration of parallel programming frameworks that can support real-time constraints.

OpenMP is one of the most widely used frameworks for shared-memory parallel programming [10][3]. Originally designed for parallelizing loop-heavy numerical computing workloads, OpenMP has evolved to support a broader range of applications, including embedded and real-time systems [8][11]. Despite these advances, OpenMP lacks native support for periodic task execution and real-time scheduling semantics [2]. It provides no mechanisms for expressing timing constraints, and its runtime delegates thread scheduling to the operating system, which may not be real-time aware [1]. The difficulty in expressing a periodic execution model in OpenMP inhibits its usefulness in a real-time context where such execution is desirable (such as control or feedback loops). Moreover, OpenMP's synchronization primitives are not designed for real-time predictability, further limiting its applicability in hard real-time environments.

To address these challenges, we created OpenMP-RT, an extension to the OpenMP 5.1 specification that introduces real-time capabilities for periodic task execution on multicore systems. OpenMP-RT adds a new `rttask` construct and associated pragma clauses to express periodic behavior, inter-task communication, and hierarchical scheduling. Real-time task parameters, including task dependencies, are declared via a configuration file analyzed at compile time (which allows for checking schedulability, and ensures all real-time parameters are located in one place). Implementation of OpenMP-RT has been realized within the LLVM compilation framework, including modifications to both Clang and the OpenMP runtime. In contrast to prior work on OpenMP-RT [9], the objective of this paper is to spark a discussion on potential inclusion of OpenMP-RT into the OpenMP standard.

Key elements of this work are:

- The design of OpenMP-RT, a framework that extends OpenMP with constructs for periodic real-time task execution, supporting both coarse- and fine-grained parallelism, focused on a discussion on compatibility and extension with the existing OpenMP standard; and
- an LLVM-based implementation targeting C/C++ that integrates real-time task scheduling and lock-free communication between rttasks into the OpenMP runtime.

## 2 Design of OpenMP-RT

The objective of this work is to present the design and prototype of a framework that simplifies the development of multi-threaded real-time applications using OpenMP. By abstracting low-level runtime and system calls, it reduces complexity and eliminates the need for custom communication mechanisms between parallel real-time components. These mechanisms are designed to minimize contention, reduce latency, and prevent deadlocks. The framework also automates periodic real-time task timing, reducing jitter caused by manual time handling. Developers can focus on application logic, while real-time guarantees are enforced via high-level OpenMP-RT pragmas and runtime extensions. Although OpenMP supports parallelism, it lacks native real-time features such as periodic execution and scheduling policies like Earliest Deadline First (EDF) or fixed-priority scheduling [6]. While the OpenMP task construct (not to be confused with *real-time tasks*, which have different properties) does allow for a priority parameter input, such tasks are served in a best-effort method and do not invoke a real-time scheduler, even in systems where such scheduling is available. OpenMP-RT fills this gap, enabling efficient development of time-sensitive parallel applications.

OpenMP-RT introduces three pragmas: `rttask` for periodic real-time tasks, and `rtread`/`rtwrite` for shared memory access. These can be used independently of `rttask`, supporting communication between real-time and non-real-time tasks. Task properties, core placement, and dependencies are specified in a concise configuration file.

### 2.1 Design Overview

OpenMP-RT comprises two main components: (1) A scheduling and core assignment framework for real-time tasks, and (2) a communication framework for data exchange between real-time tasks. The API and associated pragmas support both real-time and non-real-time producers and consumers. Details follow in subsequent subsections.

Each `rttask` is defined with standard real-time attributes — `period`, `deadline`, `phase`, and worst-case execution time (`wcet`) — and a non-empty `places` set specifying allowed cores. These properties are conveyed via the configuration file (see Subsection 2.3).

### 2.2 Execution Model

OpenMP-RT supports coarse-grained inter-task parallelism via `rttask`, and fine-grained intra-task parallelism using standard OpenMP constructs within `rttask` contexts.

Each real-time task runs on a dedicated periodic thread, supported by statically allocated thread pools per priority level (see Figure 1). Pool sizes are computed at compile time based on the maximum concurrency per priority level, derived from the configuration file.

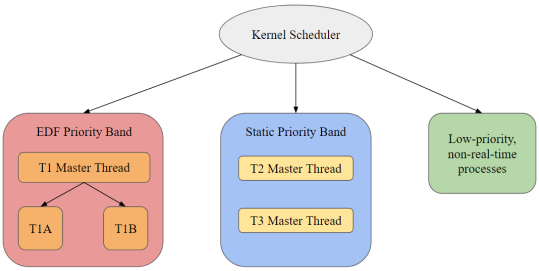


**Fig. 1.** Hybrid scheduling system, showing the ordering between the priority bands.

Each periodic `rttask` is restricted to a specified subset of cores. When it enters a parallel section, it attempts to obtain threads from the pool associated with its priority level, distributing work via the fork-join model.

However, due to system preemption policies and the state of other tasks, an `rttask` may not always obtain the requested number of threads, which is consistent with OpenMP's best-effort model (Section 1.3, OpenMP 5.1). For example, if another real-time task of the same priority is already using threads from the pool, contention may force sequential execution, as shown in Figure 2. To address this, developers can either conservatively account for sequential execution in worst-case execution time (WCET) analysis (which is undesirable), or isolate tasks of the same priority to disjoint core sets. Such core isolation is easily specified in the OpenMP-RT configuration file.

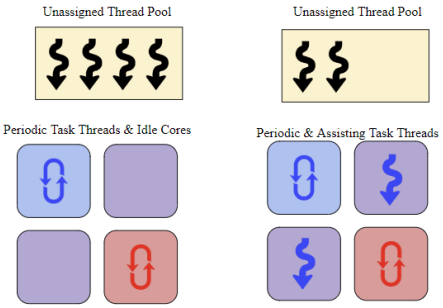


**Fig. 2.** Periodic real-time task threads, mapped to available threads from the thread subpool. Red is starved be cause Blue has claimed both shared cores.

By segmenting the allowed cores for the blue and red tasks, we are able to guarantee (fine-grained) parallel execution for both tasks regardless of which one parallelizes first, avoiding the issue seen in the second part of the figure.

### 2.3 Real-Time Specifications via a Configuration File

The configuration file specifies all `rttasks`, including core mappings and dependencies. Each task is identified by a unique name, referenced in corresponding `rttask` pragmas in the source code. Provided at compile time, the configuration file enables static analysis of the task set. This would not be feasible if task

properties were embedded directly in pragmas as they may be spread across multiple files. We assume a one-to-one mapping between cores and places, consistent with the OpenMP specification, and this mapping is enforced by checking the `OMP_PLACES` environment variable. Figures 3 and 4 show the syntax and clauses for task definitions.

```
rttask [clause[ [,] clause] ... ] new-line
```

**Fig. 3.** Task definition for rttasks in the configuration file.

```
name(identifier)
period(integer-expression)
deadline(integer-expression)
phase(integer-expression)
wcet(integer-expression)
depend(dependency-type : list)
priority(integer-expression)
threads(integer-expression)
place(list)
```

**Fig. 4.** Task descriptors & accepted data types for tasks in the configuration file.

The first two lines of the configuration file define the expected value of the `OMP_PLACES` environment variable (using the standard syntax defined in Section 6.5 of the OpenMP 5.1 spec), and the set of places (i.e., cores) where non-real-time tasks will be permitted to execute. While this is not a full description of the required hardware and its capabilities to realize real-time performance, this does allow the user to specify at least some system requirements, namely number & distribution of cores. We note that this sort of hardware reference in software design is commonplace in real-time development. Defining the expected `OMP_PLACES` value is also necessary for the core isolation discussed previously. Next, subsequent lines define a single `rttask` at a time, with the following parameters:

- `name` (required): The unique identifier used in the corresponding `rttask` pragma.
- `period` (required): The task's period (time between releases of the task), specified as a positive integer.
- `deadline`: The relative deadline (time after release by which task execution must be complete), also a positive integer. Defaults to be equivalent to the period if omitted.
- `phase`: The task's initial offset (non-negative integer). Defaults to zero, meaning the task starts immediately upon encountering the `rttask` construct.

- `wcet`: The worst-case execution time, given as a positive integer.
- `depend`: Specifies communication dependencies using a subset of the syntax from the standard `depend` clause.
- `priority`: The task's static priority (positive integer). Required for statically scheduled tasks; if omitted, the task is scheduled under EDF.
- `threads`: The number of threads requested by the task.
- `place` (required): The set of cores (places) to which the task is restricted. For tasks with intra-task parallelism, each thread is confined to this set.

In our implementation, time-related parameters (`period`, `deadline`, `wcet`, and `phase`) are expressed in microseconds. At a minimum, an EDF-scheduled `rttask` must define both a `period` and `wcet`, while a statically scheduled task must specify both `period` and `priority`. Thus, the presence or absence of the `priority` field determines whether the task is scheduled under EDF or static priority. In a system containing both EDF and static priority tasks, all tasks in the EDF band are scheduled at higher priority than static priority tasks.

### 2.4 The `rttask` Construct

The `rttask` pragma encapsulates periodic real-time task code and accepts a single `name` clause. All other attributes are defined in the configuration file, allowing the compiler to instantiate the task using this reference. Figure 5 shows the syntax.

```
#pragma omp rttask name(identifier) new-line
    structured-block
```

**Fig. 5.** Syntax for an `rttask` construct.

To avoid dynamic thread creation, all threads are initialized at startup. Execution of all `rttasks` is delayed until a defined absolute time after all threads have been created, ensuring that all `rttasks` will have the same time zero (and avoiding problems of favoritism caused by the time required to create all required threads). Each `rttask` is further delayed by its `phase` parameter, which may be zero in many cases.

The `rttask` pragma must be used outside other OpenMP pragmas, as it relies on dedicated real-time threads incompatible with OpenMP's standard parallel model. However, a subset of OpenMP constructs is supported within an `rttask`, enabling intra-task parallelism.

Specifically, OpenMP `task` and `parallel` pragmas retain their syntax but are adapted to OpenMP-RT's real-time constraints. Parallel regions reuse preallocated threads from the pool associated with the `rttask`'s priority level, which are returned after use.

While thread existence is guaranteed, availability is not as other `rttasks` with the same priority and overlapping core bindings may have already acquired

them. Availability can be ensured by assigning disjoint core sets to `rttasks` of the same priority, verifiable via static analysis of the configuration file [7]. Preallocated threads already have the correct priority and remain idle in the pool until needed.

Within parallel sections, OpenMP-RT supports `task` pragmas, which behave similarly to standard OpenMP tasks, including support for `taskwait` and critical sections. Each `rttask` defines an isolated execution context, so synchronization constructs are scoped accordingly. Thus, two threads may simultaneously execute critical sections if they belong to different `rttask` teams. This requires extending the OpenMP `critical` construct (Section 2.19.1) to scope contention groups to individual `rttask` contexts.

### 2.5 Inter-task Communication Framework

Communication between `rttask`s is facilitated through a lock-free framework designed to support data exchange both among real-time tasks and between real-time and non-real-time threads. Our work introduces and compares two such frameworks, each with distinct semantics: (1) A retry-based mechanism, implemented via the `rtread` and `rtwrite` pragmas, and (2) a double-buffered approach, using `rtreadbuffer` and `rtwritebuffer`. Both rely on lightweight atomic operations to ensure efficiency and correctness. Atomic operations themselves are thread-safe, making it possible to construct a thread-safe communication framework atop them, so long as the ordering of certain memory accesses is respected (enforceable via memory fences). Lock-free communication is essential when interfacing real-time and non-real-time threads. Allowing a non-real-time thread to block a resource required by a real-time task would compromise real-time guarantees. Notably, the operating system (in this case Linux) does not permit non-real-time tasks to inherit real-time privileges, rendering priority inheritance ineffective in this context. Nonetheless, bidirectional communication between real-time and non-real-time tasks is often necessary (e.g., for user interfaces in autonomous vehicle systems).

For diagnostic and evaluation purposes, a simpler lock-based alternative is also provided. This version employs the Priority Ceiling Emulation Protocol and is implemented via the `rtreadlock` and `rtwritelock` pragmas. It uses per-channel mutexes and ensures deadlock avoidance by enforcing a total ordering over shared data. Lock acquisition and release operations are automatically generated to follow this order, with unlocks occurring in reverse, thereby adhering to a sufficient condition for deadlock freedom [16]. Each `rttask` declares its shared memory dependencies in the configuration file. These dependencies specify both the identifier names and the task's access mode. An `in` dependency denotes read-only access (i.e., acting as a consumer), while an `out` dependency indicates write-only access (i.e., acting as a producer).

**Retry-based method** In the retry-based communication framework, each shared dependency variable is accompanied by two timestamp registers in addition to

the data itself. These timestamps are used to validate the integrity of the data during concurrent access, enabling a lock-free, single-producer, multi-consumer communication model.

```
#pragma omp rtread source(identifier) dest(identifier) size(integer-expression) flag(identifier) [num_tries(integer-expression)] new-line
#pragma omp rtwrite source(identifier) dest(identifier) size(integer-expression) new-line
```

**Fig. 6.** Syntax for the `rtread` and `rtwrite` pragmas.

The `rtread` pragma supports the following clauses:

- `source` (required): The shared memory variable to read from. Must be declared as an `in` dependency in the configuration file.
- `dest` (required): The local destination address for the copied data. Must be private to the executing thread.
- `size` (required): The number of bytes to copy from `source` to `dest`.
- `numtries` (optional): The maximum number of retry attempts if timestamps do not match. Defaults to 1 if unspecified.
- `flag` (required): A pointer to a flag variable that is set to 0 upon a successful read, or 1 if all retry attempts fail.

The `rtwrite` pragma includes:

- `source` (required): The local variable to write from. Must be private to the executing thread.
- `dest` (required): The shared memory variable to write to. Must be declared as an `out` dependency in the configuration file.
- `size` (required): The number of bytes to copy from `source` to `dest`.

The write operation proceeds as follows: The current timestamp is first written to the initial timestamp register. Then, the data is copied from the local source to the shared destination. Once the copy completes, the same timestamp is written to the second timestamp register. This ensures that during the write, the two timestamps are temporarily inconsistent, signaling that the data is in an invalid state. So long as the order of access (timestamp 1, data, timestamp 2) is adhered to (enforceable via memory barriers), this property holds.

During a read operation, the flag is initially set to 1. The first timestamp is read and stored, followed by copying the data from the shared source to the local destination. After the copy, the second timestamp is read and compared to the first. If they match, the read is considered valid, the flag is set to 0, and execution continues. If the timestamps differ, the read is invalidated, indicating that a concurrent write occurred. The operation may retry, up to the number of retries allowed by the `numtries` value.

So long as the ordering of the steps in each `rtwrite` or `rtread` is respected (e.g., a writer only equalizes the timestamps values *after* completing write to the shared data), these operations are thread-safe. This property is enforceable via a memory fence.

**Double-Buffer method** An alternative to the retry-based lock-free communication method is the double-buffer approach, implemented using the `rtreadbuffer` and `rtwritebuffer` pragmas. These pragmas share the same syntax as their retry-based counterparts, namely `rtread` and `rtwrite`, including the `source`, `dest`, and `size` clauses. However, `rtreadbuffer` omits the `numtries` and `flag` parameters, as it does not rely on retry logic.

The double-buffer mechanism operates by maintaining two memory buffers for each shared dependency. These buffers are managed by the producer task, as specified by the `depend(out:)` clause in the configuration file. A shared pointer field designates which buffer is currently readable. While one buffer is being read, the other is available for writing. Once a write operation completes, the producer atomically updates the shared pointer to reference the newly written buffer, effectively swapping the roles of the two buffers. This ensures that readers always access a consistent snapshot of the data, while writers can update the alternate buffer without interference. A memory fence is used to ensure the pointer update only occurs after writing data is complete.

Notably, read operations in this model are guaranteed to succeed without retries. Even if a write occurs concurrently with a read, the reader continues to access the previous buffer, which remains valid and unaffected by the ongoing write (which is occurring in a separate location in memory and is therefor irrelevant). This design eliminates the risk of read failures and ensures deterministic behavior in real-time systems.

## 3 Implementation

**Listing 1.1.** Pseudocode of lock-free communication.

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

We implemented a prototype of OpenMP-RT by extending the OpenMP language and runtime support within the LLVM compiler infrastructure. The

**Listing 1.2.** Sample RT-Task. Some variable declarations truncated for brevity.

```
#pragma omp rttask name(domain2)
{
    int f = 0;
    int** U;
    #pragma omp rtread source(border1) dest(U) size(4000) flag(f) num_tries(3)
    #pragma omp parallel for
    for(j=1;j<Coord2Dsize;j++){ //iterate over subdomain for calculation
        godunov_domaincol(j, U, Coord2D); //invoke subdomain calculation
    }
    #pragma omp rtwrite source(U) dest(border2) size(4000)
}
```

current implementation targets C programs running on Linux systems with the PREEMPT_RT patch, supporting both static priority and EDF scheduling.

The prototype serves as a proof-of-concept to demonstrate OpenMP-RT's feasibility and enable experimental evaluation via a benchmark suite. While support for other languages (e.g., C++) is possible, it is beyond the scope of this work.

Linux was selected for its widespread use, mature ecosystem, and increasing adoption in real-time systems, including industrial applications like Tesla's autonomous driving stack. With PREEMPT_RT, Linux provides the necessary OS-level support for OpenMP-RT, including EDF and static priority scheduling in high-priority bands isolated from standard threads. This dual support for real-time and non-real-time tasks allows OpenMP-RT to demonstrate hybrid scheduling, where both types of threads coexist and interact safely.

As described in Section 2, OpenMP-RT includes frameworks for safe data exchange between `rttask`s and across the real-time boundary. Listing 1.1 shows a pseudocode example of the retry-based communication mechanism triggered by `rtread` and `rtwrite` pragmas, represented as function calls.

## 4 Results and Example

Let us provide an example application created using OpenMP-RT, including a configuration file (see Listing 1.3) as well as one relevant `rttask` construct (see Listing 1.2, consolidated for space). Deadline misses in this sample application are compared against a version of the task set implemented without OpenMP-RT, using a combination of Linux PREEMPT-RT syscalls as well as ordinary OpenMP `task` pragmas.

This sample task set is based on a version of the HYDRO Benchmark, which incorporates OpenMP tasks [4] in a parallel computation of fluid dynamics equations. This workload was selected because it includes clearly-defined producer and consumer dependencies between tasks, and the computational intensity makes it easier to observe deadline misses. Our modified version of this benchmark uses real-time threads (in one version implemented as OpenMP-RT

**Listing 1.3.** Sample configuration file.

```
ompplaces   "{0:24}"
nonrtplaces "{4:24}"
task name(domain1) period(1000) wcet(700)
    depend(inout:border1) threads(5) place(0,1)
task name(domain2) period(1000) phase(100)
    depend(inout:border1) depend(inout:border2) priority(10) threads(5) place(2,3)
task name(domain3) period(1000) phase(200)
    depend(inout:border3) priority(20) threads(5) place(0,1,2,3)
```

**Table 1.** Utilization versus observed deadline misses.

| Total Utilization | Deadline Misses | | |
|---|---|---|---|
| | OMP-RT (retry) | OMP-RT (double-buffer) | PREEMPT-RT + OMP |
| 3.6 | 0 | 0 | 59,124 |
| 3.72 | 0 | 0 | 109,628 |
| 3.8 | 8 | 3 | 161,452 |
| 3.96 | 7622 | 8195 | 182,929 |

`rttask` pragmas, and, for comparison, in another using PREEMPT-RT Pthread syscalls), to handle each subdomain calculation. The OpenMP-RT version is compared using both Retry and Double-Buffer synchronization primitives. As the actual workload is static between experiments, we are instead able to control utilization by varying the periods and deadlines associated with subdomain calculations. Experimentation was conducted using an Intel i7-14700 KF processor with 24 cores. For our purposes, the benchmark was constrained to 4 cores. Slightly over 2 million task releases occur in the calculation, giving context to the number of observed deadline misses.

As can be seen in Table 1, OpenMP-RT dramatically reduced the number of deadline misses, eliminating them completely at utilization values below 3.8 (at 4 cores, this is equivalent to a per-core utilization of 0.95, or 95%). Note that deadline misses without OpenMP were mostly caused by the fine-grained parallelization inside subdomain calculations, where threads did not inherit priority from the initial launching thread before launch, thus resulting in potential priority inversions. This demonstrates the benefit of OpenMP-RT's real-time priority support.

Further experiments were conducted in order to compare the two options for synchronization in OpenMP-RT, namely the Retry and Double Buffer methods. The same task set as in the previous experiments was used.

Both techniques result in relatively similar response times, with Double-Buffer expressing superior response times on average only in the case of very high utilization. (This is a side effect of the retry-based method having a much higher likelihood of invalidating data during a read.) Notably, under very high utilization, there is a significant increase in reader response time but no corresponding increase for writers. This is caused by the `rtread` and `rtwrite` implementations

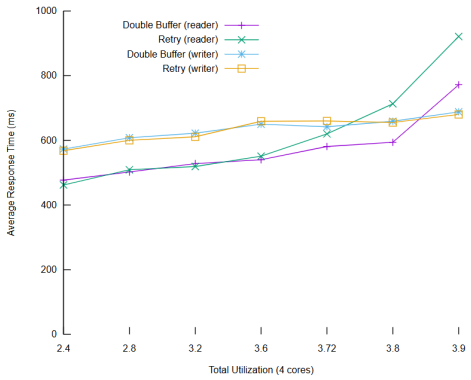


**Fig. 7.** Response time comparison between Double-Buffer and Retry based synchronization methods.

favoring write operations in the event of a preemption. The increased response time for readers is an artifact of increased frequency of retries (or, in the double buffer case, switching to a new buffer).

As one of the driving objectives for the development of OpenMP-RT is developer ease-of-use, we can compare the length (in lines of code) of an application implemented via OpenMP-RT versus the same application using existing PREEMPT-RT Pthread syscalls. While lines of code is not an exact metric for overall complexity, it is useful in a before/after comparison. As we already have two versions of a real-time application based on the HYDRO Benchmark with and without OpenMP-RT, we use it as an example for reduction in code complexity. Previous experiments highlighted that the non-OpenMP-RT application struggles to meet deadlines, but that is not relevant to this analysis. Reduction in utilization (accomplished through increasing periods and deadlines, with no change in workload) allows both versions of the test application to execute with no missed deadlines in any case.

**Table 2.** Reduction in lines of code for HYDRO benchmark derived application.

| | Pthread | OpenMP-RT | Percentage Decrease |
|---|---|---|---|
| Lines of Code | 1562 | 1297 | 17% |

Table 2 shows the difference in total lines of code between the implemented PREEMPT-RT Pthread and OpenMP-RT versions of the sample application. The reduction in code length is mainly due to the removal of all code involved in setting up real-time Pthreads (as this functionality is relegated to the OpenMP runtime). For completeness's sake the length of the OpenMP-RT configuration file has been added to the line total of the OpenMP-RT version. However the configuration file, included above, totals 8 lines, i.e., its effect on the length is inconsequential. We also note that the PREEMPT-RT Pthread version includes

*only* the minimum required to launch a real-time thread, along with the timer setups for inducing periodic behavior. It includes no error checking, which would be vital in a real application to prevent silent failure, and is included in the OpenMP-RT implementation.

## 5 Related Work

There has been significant interest in the use of multicore architectures in real-time systems, despite the inherent difficulties of real-time scheduling on such platforms [12][5][14]. Several studies have evaluated OpenMP's potential in this domain, particularly in the study of timing bounds [14][15]. These works highlight limitations in the predictability of OpenMP, mostly stemming from the best-effort execution model and lack of native support for real-time semantics [3]. In comparison, OpenMP-RT introduces new pragma constructs and semantics for expressing periodic real-time execution and synchronization primitives with calculable bounds.

Serrano et al. [11] propose extending the existing OpenMP `task` pragma with `event`, `deadline`, and `priority` clauses to support recurrent tasks and coarse-grained parallelism. This approach uses user-defined event expressions to define periodic tasks, in contrast to OpenMP-RT's use of dedicated `rttask` construct with explicitly-defined `period` (and other real-time task descriptors) information for periodic task creation and expressing hierarchical scheduling. Our implementation integrates with the Linux `PREEMPT_RT` kernel, supporting both static priority and EDF scheduling, making OpenMP-RT the first to provide hard real-time guarantees under Linux using OpenMP.

OpenMP-RT was first introduced in [9] with a focus on ease of real-time programming and synchronization primitives. In contrast, our current manuscript focuses on OpenMP compatibility, semantic constraints and discusses benefits and challenges in the context of considering OpenMP-RT for potential inclusion in the OpenMP standard.

## 6 Conclusion

We introduced OpenMP-RT, a framework built on top of OpenMP to support the development of parallel real-time applications. Our implementation, targeting the C language via the LLVM compiler infrastructure, leverages Linux's real-time scheduling capabilities to enable predictable execution and supports coarse-grained parallelism among periodic tasks, core isolation, hybrid scheduling strategies, along with multiple lock-free inter-task communication mechanisms. Future work will explore expanding the inter-task communication model, supporting additional scheduling policies beyond EDF and static priority, and developing tooling to assist with system design, such as automated optimization of core-to-task bindings, core migration and schedulability analysis.

## Acknowledgments

This work supported in part by NSF awards CISE-1747555, CISE-1813004, CISE-2316201 and a grant by CISCO.

## References

1. Alrawais, A.: Parallel programming models and paradigms: Openmp analysis. In: 2021 5th International Conference on Computing Methodologies and Communication (ICCMC). pp. 1022–1029 (2021). https://doi.org/10.1109/ICCMC51019.2021.9418401
2. Ayguade, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Teruel, X., Unnikrishnan, P., Zhang, G.: The design of openmp tasks. IEEE Transactions on Parallel and Distributed Systems **20**(3), 404–418 (2009). https://doi.org/10.1109/TPDS.2008.105
3. Chandra, R., Dagum, L., Kohr, D., Menon, R., Maydan, D., McDonald, J.: Parallel programming in OpenMP. Morgan kaufmann (2001)
4. Gaidamour, J., Lecas, D., Lavallée, P.F.: Introducing OpenMP Tasks into the HYDRO Benchmark (2021), https://arxiv.org/abs/2106.13465
5. Godabole, P., Bhole, G.: Timing analysis in multi-core real time systems. In: 2021 IEEE International Symposium on Smart Electronic Systems (iSES). pp. 38–43 (2021). https://doi.org/10.1109/iSES52644.2021.00021
6. Liu, C., Layland, J.: Scheduling algorithms for multiprogramming in a hard-real-time environment. Journal of the ACM **20**(1), 46–61 (Jan 1973)
7. Liu, J.: Real-Time Systems. Prentice Hall (2000)
8. Marongiu, A., Capotondi, A., Tagliavini, G., Benini, L.: Improving the programmability of sthorm-based heterogeneous systems with offload-enabled openmp. In: Proceedings of the First International Workshop on Many-Core Embedded Systems. p. 1–8. MES '13, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2489068.2489069, https://doi.org/10.1145/2489068.2489069
9. McDonald, B., Mueller, F.: Openmp-rt: Native pragma support for real-time tasks and synchronization with llvm under linux. In: ACM SIGPLAN Conference on Language, Compiler, and Tool Support for Embedded Systems. pp. 119–130 (Jun 2024)
10. Peng, J., Hu, C., Xi, J.: Msi a new parallel programming model. In: 2009 WRI World Congress on Software Engineering. vol. 1, pp. 56–60 (2009). https://doi.org/10.1109/WCSE.2009.114
11. Serrano, M.A., Royuela, S., Quiñones, E.: Towards an openmp specification for critical real-time systems. In: de Supinski, B.R., Valero-Lara, P., Martorell, X., Mateo Bellido, S., Labarta, J. (eds.) Evolving OpenMP for Evolving Architectures. pp. 143–159. Springer International Publishing, Cham (2018)
12. Sha, L., Caccamo, M., Mancuso, R., Kim, J.E., Yoon, M.K., Pellizzoni, R., Yun, H., Kegley, R., Perlman, D., Arundale, G., Bradford, R.: Real-time computing on multicore processors. Computer **49**, 69–77 (09 2016). https://doi.org/10.1109/MC.2016.271
13. Vaidehi, M., Nair, T.R.G.: Multicore applications in real time systems. In: Journal of Research & Industry. vol. 1, pp. 30–35 (2008)

14. Vargas, R., Quinones, E., Marongiu, A.: Openmp and timing predictability: A possible union? In: Design, Automation and Test in Europe. pp. 617–620 (2015). https://doi.org/10.7873/DATE.2015.0778
15. Wang, Y., Guan, N., Sun, J., Lv, M., He, Q., He, T., Yi, W.: Benchmarking openmp programs for real-time scheduling. In: 2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA). pp. 1–10 (2017). https://doi.org/10.1109/RTCSA.2017.8046322
16. Wang, Y., Liao, H., Nazeem, A., Reveliotis, S., Kelly, T., Mahlke, S., Lafortune, S.: Maximally permissive deadlock avoidance for multi-threaded computer programs (extended abstract). In: 2009 IEEE International Conference on Automation Science and Engineering. pp. 37–41 (2009). https://doi.org/10.1109/COASE.2009.5234118