Effective Runtime Management of Tasks and Priorities in GNU OpenMP Applications

Emiliano Silvestri, Alessandro Pellegrini, Pierangelo Di Sanzo, and Francesco Quaglia

Abstract—OpenMP has become a reference standard for the design of parallel applications. This standard is evolving quickly, thus offering new opportunities to the application programmers. However, OpenMP runtime environments are often not fully aligned with the actual requirements imposed by the evolution of such a standard. Among the main lacks, we find: (a) a limited capability to effectively cope with task priorities, and (b) the inadequacy in guaranteeing core properties while processing tasks such as the so-called *work conservativeness*—the ability of the OpenMP runtime environment to fully exploit the underlying multi-processor/multi-core machine through the avoidance of thread-blocking phases. In this article, we present the design of extensions to the GNU OpenMP (GOMP) implementation, integrated into gcc, which allow the effective management of tasks and their priorities. Our proposal is based on a user-space library—modularly combined with the one already offered by GOMP—and an external kernel-level Linux module—offering the opportunity to exploit raising hardware facilities for task/priority management. We also provide experimental results showing the effectiveness of our proposal, achieved by running either OpenMP common benchmarks or a new benchmark application (HASHTAG-TEXT) that we explicitly devised to stress the runtime environment in relation to the above-mentioned task/priority management aspects.

Index Terms—Multi-core computing, task parallelism, operating system support.

1 INTRODUCTION

In order to help programmers to structure their parallel applications, several paradigms have emerged. Among them, OpenMP [1] plays a central role and is widely considered a de-facto standard.

The OpenMP programming model has evolved over the years to support fine-grain and irregular parallelism, which is achieved by making the concept of *task* central [2]. In fact, programmers can use simple pragmas and an API supported by the runtime environment to schedule tasks, to define their dependency constraints and, as in recent developments of the OpenMP specification, to devise their priorities [3].

In this article we focus on the GNU OpenMP (GOMP) project, which has some limitations that are common to current implementations of OpenMP runtime environments. The main one stands in the impossibility to asynchronously preempt a running task in order to assign the computing power offered by some CPU-core to another task. In fact, task switching is allowed only if the running task spontaneously yields the CPU-core. This may happen either upon task completion, or when the running task reaches a Task Scheduling Point (TSP). This is used to explicitly include into the code a directive, like taskyield, which allows returning control to the runtime environment. One of the main reasons for the inclusion of TSPs is to let tasks safely access Thread Local Storage (TLS) in between two task scheduling points. However, when TLS is not used, the first drawback is that we may observe scenarios where a newly-scheduled higher-priority task is delayed by a lower-priority one that has already gained control of the CPU-core. This is somehow in contrast to the objective that higher priorities should be directly translated by the runtime environment into better timeliness and more prompt CPU-dispatching of task execution. This problem is exacerbated by the fact that a task in OpenMP simply corresponds to the execution of a function specified by the programmer, whose duration (namely, CPU demand) can be arbitrary. Also, the function might even interact with blocking services of the underlying Operating System (OS), so that its turnaround time (or the time to reach a TSP) can be further stretched. All these aspects can delay excessively the activation of some higher-priority task that is already standing.

Another drawback that we observe with GOMP is that joining a task with one it depends on (e.g. via the taskwait directive) makes heavy usage of blocking synchronization services like OS futexes. Hence, a thread T currently running task A blocks when A needs to join the execution of a child task B that has already been CPU-dispatched along another concurrent thread T'[1]. This scenario is somehow critical in terms of the capability of the GOMP runtime environment to exploit the underlying hardware resources. The latter aspect is also related to the default settings of

- *E. Silvestri is with Sapienza, University of Rome. E-mail: silvestri@diag.uniroma1.it*
- *A. Pellegrini is with Lockless S.r.l. E-mail: pellegrini@lockless.it*
- *P. Di Sanzo is with University of L'Aquila. E-mail: pierangelo.disanzo@univaq.it*
- *F. Quaglia is with University of Rome "Tor Vergata". E-mail: francesco.quaglia@uniroma2.it*

Manuscript received May, 2021.

1. The take-off of task A with no actual block of thread T can take place only under the scenario where task B is not currently taken in charge by, or has been bound to, another thread. Limited to this scenario, T can immediately—hence with no blocking phase—take task B, thus temporarily suspending task A. We recall that the default for OpenMP is to have tied tasks, which once taken by some thread cannot be migrated to another thread.

several OpenMP runtime environments, which limit the number of used threads to the number of available CPU-cores in the used machine—a choice that appears reasonable to avoid cross-thread competition for CPU resources. Therefore, having thread T blocked because of a dependency constraint with a child task bound to another thread leads in practice to renouncing to the computing power of a CPU-core. Furthermore, according to OpenMP Task Scheduling Constraints (TSCs), a ready tied task is admitted to be executed by whichever thread. However, a thread T that is processing tied ancestors of a generic tied task A cannot take any other tied task different from A until the thread that started processing A finishes its job. Such an issue is of high interest for the research community since it leads OpenMP applications to be challenging to be formally analyzed in terms of their capabilities to match deadlines in the presence of tied tasks [4]. Additionally, in GOMP thread T is simply blocked in this scenario, not allowing it to pick standing tasks, not even the untied ones.

Clearly, the presence of thread-blocking phases in situations where standing tasks could be ideally picked and processed—with no violation of TSCs—makes the run-time environment unable to guarantee the so-called *work-conservativeness* property. In fact, if we cannot guarantee that the computing power usage is preserved—since we renounce to some CPU-core at some point in time because of thread blocks—then proving that OpenMP applications can match specific deadlines becomes a hard job [5].

While a similar problem has already been addressed by lower-level technologies—as an example, OS-kernels do not renounce to using CPU-cores to dispatch threads at any point in time, or even provide mechanisms for enabling the spawning of additional threads when others are blocked [6]—it is still present in GOMP.

In this article, we present the design and the implementation of extensions to the GOMP runtime environment for Linux and x86-64 processors, which address all the above raised problems. In particular, our software modules, which are available as open-source[2], allow:

1) Prompt switch to any higher-priority task that is scheduled while a thread is processing a lower-priority one—promptness means having delays of at most the order of a few tens of microseconds on x86-64 chipsets; this provides better execution timeliness of higher-priority tasks;
2) The avoidance of thread blocking phases caused by dependencies across tasks bound to different threads—this allows us to better exploit the available CPU-cores, sliding towards work-conservativeness in the runtime environment.

The above two objectives are met in our design while still guaranteeing all the properties that are demanded from OpenMP runtime environments, such as the avoidance of moving tied tasks across threads and the avoidance of task dependency constraint violations. Further, we preserve the properties that programmers expect to be guaranteed when the coded tasks interact with common libraries—as an example, we preserve deadlock avoidance when multiple tasks tied to a same thread manage synchronization constructs such as spinlocks or mutexes/semaphores. Similarly, we still enable TLS to be correctly used by the application.

In this article, we focus on AMD x86-64 processors, although the design principles of our solution can be mapped to other hardware architectures. Also, our implementation is essentially a library that can be linked to the GOMP runtime environment and the application code, thus working with no need for particular intervention by the programmer or application code adaption. At the same time, the facilities that our proposal requires to nest into the Linux kernel are all supported by an external module only exploiting module-exposed programming facilities, thus not requiring any kernel recompilation. Such kernel side software is also available as open-source[3].

To assess our proposal, we run experiments with the Barcelona OpenMP Task Suite (BOTS) [7] and a new benchmark application we designed, called HASHTAG-TEXT. The latter has been devised to compensate for the lack of literature benchmarks in relation to our assessment objectives. Indeed, our search for OpenMP benchmarks found no solution explicitly including and exploiting task priorities, which (as hinted) have recently become of interest in the OpenMP specification. The source code of our new benchmark application is available as open source too[4], thus being released for usage to the community.

The remainder of this article is structured as follows. In Section 2, we discuss related work. The innovative OpenMP runtime system is presented in Section 3. Experimental data are provided in Section 4. Finally, conclusions are reported in Section 5.

2 Related Work

Our proposal introduces a new run-time support for controlling the program flow along threads in a parallel application with (potential) dependencies of the activities among threads. These environments have been thoroughly studied in the literature, and several solutions have been presented to achieve efficiency while carrying out the activities (the tasks) to be executed. A few proposals—such as Converse Threads [8] and Argobots [9]—are based on User-Level Thread (ULT) technology and on performing execution flow variations along threads at specific execution points—via an explicit synchronous invocation of an API that switches control between ULTs, each one encapsulating a task to be executed. The proposal in [9] integrates deferred-work concepts, such as tasklets, which have been historically used in OS-kernel technology to finalize—with an explicit synchronous call to a tasklet-processing routine—some previous request for task execution. Similarly, [8] integrates the notion of messages as a form of deferred work to be processed after that a poll operation by a thread identifies the presence of an incoming message. Differently, our solution is based on both asynchronous and synchronous switches of the execution flow used in combination, where the asynchronous switching capability is achieved thanks to the exploitation of rising hardware features in modern

2. https://github.com/HPDCS/ULMT-OpenMP-GCC

3. https://github.com/HPDCS/IBS-Support-ULMT
4. https://github.com/HPDCS/OpenMP-Task-Suite

CPUs. Further, we focus our design on GNU OpenMP, while the proposal in [8] supports an alternative programming model, such as Charm++.

Asynchronous switch of the execution flow is a classical target for OS technology—for both time-sharing and processing of signals/events (see, e.g. [10]). However, the granularity according to which a conventional OS induces control flow variations is unsuited for making a thread extremely reactive to the need for changing its execution flow. To make an example, the implementation of an asynchronous switch bringing a thread to run a higher-priority task via Posix signals would make the activation of the higher priority task delayed up to the end of the current tick assigned to the thread by the OS. In fact, the delivery of signals, with the associated control flow variation, mostly occurs when returning to user mode, e.g. after a timer interrupt (or a system call). In modern OS configurations, the tick can range from a few to several milliseconds, in order not to make the OS too "noisy" because of more frequent scheduler activations. In fact, these activations have an inner cost even in the scenario where the next-to-be-run thread coincides with the currently-running one. Classical Linux configurations are based on a tick ranging from 1 millisecond—as for the case of lower degree of parallelism in the hardware (2/4 CPU-cores)—to 4 milliseconds—as for the case of off-the-shelf machines with more CPU-cores, where CPU reassignment is less critical due to the larger amount of computing resources. A delay in the order of 1 to 4 milliseconds to switch the thread execution flow to a higher-priority OpenMP task is clearly non-adequate, especially when the higher-priority task is fine-grain, so that its waiting time for taking control under these settings can definitely overstep its running time[5]. Overall, the classical OS support for asynchronous (preemptive) execution-flow variations is extremely valid in scenarios where priorities are associated with threads to be processed in time-sharing mode, not with specific functions (the tasks) to be processed by each individual thread. This problem is directly tackled by our solution, which allows the asynchronous switch of the execution flow along a thread with very fine granularity—of the order of tens of microseconds—still with no intervention by the programmer.

Extremely reactive variation of the execution flow of threads has been dealt with in software-level speculative processing [11], [12]. In this context, fine-grain asynchronous execution flow variations enable detecting if the current speculation path of a thread is still consistent and can lead to early squashing it in the negative case. However, the proposed solutions rely on OS-kernel patching of the management of the hardware-timer interrupt logic with non-minimal intrusiveness. To better clarify this point, the Linux kernel has implementations of the timer-interrupt handler that vary across the various kernel releases. Our work provides an alternative way of exploiting modern CPUs for asynchronous execution flow variations, which requires minimal intervention at the OS-kernel level. In particular, our software architecture can be embedded within the Linux kernel by simply relying on loading an external module that does not patch the currently-running kernel logic. In our solution, instead of relying on hardware timers (and their kernel-specific management logic), we rely on common hardware profiling supports, which can be activated or deactivated by simply writing on specific (pseudo-)registers of the CPU. In more detail, our solution is founded on hardware facilities like the IBS (Instruction Based Sampling) support by AMD, or PEBS (Precise Event-Based Sampling) on the Intel counterpart.

This type of support has been exploited in the literature to build tools that allow understanding what are the actual dynamics of the hardware (cache misses, dominating executed-instruction types etc.) depending on the overlying running software. Among them, we can mention the well known Perf tool [13]. However, none of these tools has been exploited for directly driving the application execution flow. Instead, we use the support from the hardware profiler to proactively change the execution flow of the application threads in `GOMP`.

As for works specifically tailored to OpenMP runtime environments, the literature offers several proposals based on experimental and/or theoretical studies [4], [14]. One of the core objectives of these works is to determine whether OpenMP can be fruitfully exploited in the context of real-time applications and under what conditions, e.g. in terms of application design patterns. Our work is orthogonal to these studies since our objective is to include in the OpenMP runtime environment—particularly the `GOMP` implementation—innovative features for efficient processing of tasks. Clearly, such new features can trigger additional (formal) analysis in terms of achieving real-time capabilities in OpenMP applications. However, as noted in Section 1, the works in [4], [5] have already shown how real-time capabilities in OpenMP applications are strictly linked to the ability of the runtime environment to guarantee the work-conservativeness property. In our design, work-conservativeness is one of the core objectives, in combination with adequate support for reflecting task priorities into actual runtime dynamics.

The work in [15] presents an experimental study where different task scheduling policies, ranging from breadth-first [16] to work-first with work-stealing [17], are comparatively assessed. However, this study is still based on a runtime environment—which is called NANOS and is partially compliant with the OpenMP specification, although having influenced the definition of this specification along time—where task-switch operations occur synchronously with respect to the execution of threads, again via synchronous invocation of the ULT API. As an example, they occur when a task is terminated and another one needs to be picked by a thread, or when the creation of a new task is requested along a thread—in which case work-stealing can lead to processing the newly born task immediately along that thread. Contrarily, our runtime environment also includes the support for asynchronous (and preemptive) task switch operations. This is mandatory for the effective management of task priorities, which is an emerging aspect of the OpenMP specification.

5. We also note that high-resolution timers—used in Linux for deferred-work associated with very fine-grain delays—have their handlers processed along specific daemons, thus not enabling custom threads—those running the OpenMP application in our case—to directly receive periodic fine-grain interrupts enabling asynchronous control flow variations.

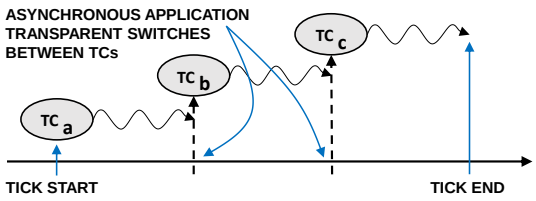

Fig. 1: A thread timeline with TC switches.

We also note that other environments for parallel (or distributed) computing, like those for the MPI (Message Passing Interface) specification [18], still lack native mechanisms for handling asynchronous switch of the control flow across different tasks along a same thread. Indeed, even though asynchronous messaging is supported, it is anyhow based on explicit calls to services that enable a thread to understand whether some specific communication has been asynchronously delivered to it—this may indicate that some different task needs to be processed by the thread. Instead, our solution is based on interrupting the thread execution flow, with no need for calls to services of the runtime environment (e.g. the check of an asynchronous receipt finalization) for handling task switch operations.

3 The Runtime Environment

3.1 Task Contexts

To reach the two core objectives listed in Section 1, namely fine-grain control of task priorities and better satisfaction of work-conservativeness, we introduced the notion of Task-Contexts (TCs). A TC is essentially a CPU context—made up of general-purpose registers, stack pointer, instruction pointer, etc.—which at any time instant can be either running on some CPU-core along a thread or can be saved into a `gomp_task_context` data structure.

TCs might resemble contexts managed by libraries implementing User-Level Threads (ULTs). However, we could not base our design on reusing ULT implementations because, as hinted, they cannot support asynchronous passage of control between a context TC_a and another context TC_b. As noted, ULTs allow switching between contexts—hence between the execution of two different functions along a thread—in a safe manner only under the assumption that all the switches are implemented via synchronous calls to a switch-supporting API (such as `setjmp/longjmp`).

Overall, TCs keep CPU-register values that might be otherwise clobbered when asynchronously transitioning the execution flow of the same thread to another context, without an explicit call observed at compile time.

In our implementation, the runtime system can create an arbitrary number of these TCs by reserving a different stack area for each of them. Each time the application uses the `GOMP_task` API to activate a task, the runtime environment intercepts it and links a fresh `gomp_task_context` data structure to the original task-management data structure.

In Figure 1, we show an example thread-execution timeline in our runtime environment. The thread execution flow can be asynchronously switched across three different task contexts, TC_a, TC_b, and TC_c, with no modification of the tick assigned by the underlying OS to the thread traversing these contexts, as well as no intervention by the programmer. Hence our solution is oblivious to how the OS decides to assign the CPU capacity to the hosted applications. At each switch, a higher priority task that has been injected in the system—or has become schedulable in the meanwhile since its task-dependency constraints have been satisfied—can promptly take control of the CPU-core. How to enable this kind of *preemptive* control flow variation, with a granularity of the order of a few tens of microseconds is discussed in the next section.

3.2 Implementation of the Preemption Support

The preemption support leading a thread to slide out of its control flow graph—as defined at application compile-time—must be based on some hardware support driving an asynchronous variation of the thread execution flow. As hinted, we do not exploit classical timer interrupts since moving to a custom management of them would make the solution intrusive at the OS-kernel level.

To reach our goal, we decided to exploit the rising hardware capabilities of modern CPUs, which have been not specifically devised for managing time-sharing and cross-context passage of control but rather for low-level monitoring of the hardware state. A few of these monitoring capabilities can generate interrupts, which are not exploited by the OS to do core work. In most cases, the usage of these interrupts is even not configured, depending on the OS version and on the users' needs.

In more detail, modern CPUs enable software to inspect their state at very fine grain. To achieve this objective, they generate programmable interrupts that asynchronously bring control to custom handlers. These handlers can do whatever action, and the typical action consists of accessing flushed shadow copies of CPU registers or non-programmable registers to inspect the state of the CPU upon their occurrence. The important aspect is that the interrupt handler runs on an interrupt stack where the original value of the instruction pointer—the one to be reinstalled upon the return of the handler—is readable and writable. Writing that value from inside the interrupt-handler allows generating a control flow variation along the thread, which can bring control to whatever instruction in the address space seen by the thread.

On x86-64 processors, this hardware support is implemented according to different flavors depending on the processor manufacturer. As hinted, on AMD processors, which we target in this work, we have the IBS (Instruction Based Sampling) support. Hence, we can program the CPU-core to send an interrupt on a given line after a number of machine instructions have been executed and committed by the pipeline, or after a given number of clock cycles. This feature can be exploited to build periodic interrupts hitting a given target thread (one running the `GOMP` application in our case) which stand aside from the ones associated with hardware timers. In our preemptive version of the `GOMP` environment, we exploit this hardware support in Linux in a lightweight and highly modular fashion. Rather than using the NMI (Non-Maskable-Interrupt) line for managing these interrupts, as it typically occurs in common OS drivers tailored to the exploitation of these hardware capabilities

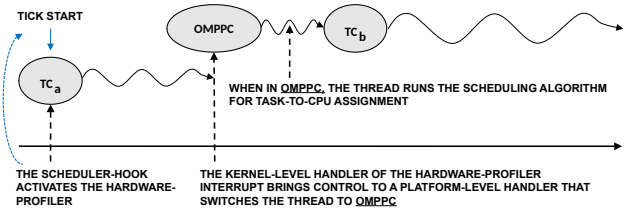

Fig. 2: A timeline with hardware-profiler interrupt and passage of control to OMPPC.

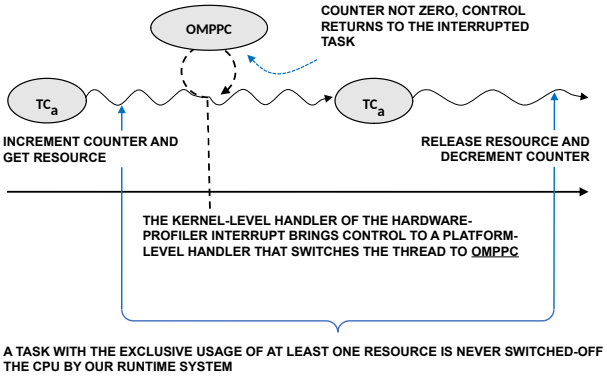

Fig. 3: A timeline with preserved execution of a resource holding task upon the hardware-profiler interrupt arrival.

exclusively for hardware profiling (see, e.g., [13]), we have selected a generic regular line that is not currently used by the Linux kernel. On that line, we install our kernel-level interrupt handler via an external loadable module[6].

To take advantage of the installed hardware-profiler interrupt handler when running `GOMP` threads, a few additional features are provided by the loadable Linux module. First, by relying on the Linux kernel `kprobe` infrastructure, we install a hook on the kernel `schedule()` function, which determines whether a thread running the `GOMP` application has been (re-)scheduled on CPU. In the positive case, the hardware profiler is activated to generate fine-grain interrupts along the time-quantum of that thread. To determine what threads of what processes need to undergo these fine-grain interrupts, the loadable module includes a special device file, whose `ioctl` commands allow to post the corresponding information (e.g. the `pid` of the process running the OpenMP application) to kernel-level data structures so as to make it accessible by the hook of the `schedule()` function.

In our architecture, the kernel-level handler of the hardware-profiler interrupt manipulates the CPU-context to be restored upon its return to bring control to a user-mode additional context, which we call OpenMP Platform Context (OMPPC). Passing through this context, our `GOMP` runtime environment allows the thread to execute the (asynchronously triggered) task-to-CPU reassignment algorithm aimed at the effective management of the priorities of standing tasks. The actual scheduling algorithm we embedded in our runtime environment and the data structures it relies on, will be presented in Section 3.3.

A timeline schematizing the evolution of a thread running the OpenMP application is shown in Figure 2. Initially, the thread might be running within the TC_a context. Earlier, upon its reschedule on CPU, the hook of the kernel scheduler activated the hardware profiler generating the fine-grain interrupts. As soon as one of these interrupts occurs, control bounces to OMPPC, and after running the task-to-CPU reassignment algorithm, the thread decides to switch to the context TC_b associated with another task, thus temporarily suspending the execution of the task with context TC_a.

Clearly, a few additional points need to be considered. First, the hardware-profiler interrupt handler cannot really change the current execution flow of the thread for bringing control to OMPPC if we were already running in OS-kernel mode. In fact, admitting the change would imply that a user-space portion of code could preempt the kernel level execution, a scenario that would break the overall kernel software design. To avoid this scenario, before any modification is put in place on the CPU-context to be restored upon returning from the hardware-profiler interrupt handler, a check is done whether the interrupted context was a kernel-level one. In the positive case, no execution flow variation is actuated. In other words, we are simply renouncing to exploit the received fine-grain interrupt—for possible asynchronous switch between `GOMP` tasks—in favor of coherent kernel-level operations. This also has the advantage of avoiding any risk of deadlocks when the thread has already entered kernel mode upon the arrival of the interrupt.

The same aspect of preemption avoidance needs to be considered for the execution of a task that has acquired exclusive usage of user-space resources, such as a spinlock. Otherwise, we might generate a scenario where the timing expectations for releasing the resource would be completely altered. Further, we might even incur deadlocks—as for the case of an access request by a task to a user-space resource held by another task that has been context-switched off the CPU. To avoid all these problems, we have followed an approach where any API that the programmer can use to acquire resources exclusively is transparently wrapped—at application link-time—by another API that before acquiring the resource puts the thread in a "non-preemptable state" (clearly, the thread can still be preempted by the operating system). In our solution, this wrapping is carried out fully transparently by the compile/link infrastructure, with no need for any intervention by the programmer of the `GOMP` application. In our implementation, this is not done by switching off the hardware-profiler interrupt that leads to the activation of the OMPPC, rather via a cheaper solution where we use a per-task atomic counter stored in the `gomp_task` structure. The counter is atomically incremented by one unit when we enter the wrapper of the target function invoked to acquire the resource and is atomically decremented when we leave the function releasing the resource. Therefore, the counter cannot be zero along any execution frame where some resource is locked in favor of a task. When switching to OMPPC due to a hardware-profiler interrupt delivery, the first action is to check the counter

6. Since we exploit a trap-handler trampoline that is already available in the kernel image, our solution is compliant with the Page-Table-Isolation service.

Algorithm 1 Case of Resource Acquisition

```
1: procedure WRAP_ORIG_RESOURE_GET_FUNCTION_NAME( ):
2:     ATOMIC_INCREMENT( )
3:     ret ← ORIG_RESOURCE_GET_FUNCTION_NAME( )
4:     if ret == FAILURE then ATOMIC_DECREMENT( )
5:     return ret
```

Algorithm 2 Case of Resource Release

```
1: procedure WRAP_ORIG_RESOURCE_RELEASE_FUNCTION_NAME( ):
2:     ret ← ORIG_RESOURCE_RELEASE_FUNCTION_NAME( )
3:     if ret == SUCCESS then ATOMIC_DECREMENT( )
4:     return ret
```

value, and if it is not zero, OMPPC simply switches back to the context of the task that was already running along the thread[7]. We also note that the counter value is incremented and decremented when entering or exiting OMPPC so that, if an interrupt was received but not yet fully processed (e.g. the thread was CPU-descheduled by the OS), then the receipt of other successive interrupts (e.g. after the thread has been CPU-rescheduled by the OS) leads to no action by the side of control flow management. Again, we note that this solution leads to no possibility of changing the execution flow of a thread when it has entered a software routine that needs to execute in a non-preemptable manner along the thread. This avoids any possibility of anomalies (like deadlocks) related to our mechanism. In Figure 3, we depict a timeline related to preemption avoidance given a previous acquisition of the exclusive usage of a resource. The structure of our wrappers for accessing and releasing user-space resources is shown in Algorithm 1 and Algorithm 2.

A similar approach, still based on wrapping, is used for non-reentrant user-space libraries (like, e.g., strtok), for which we cannot admit another task (along the same thread) to issue a call while a previous call is still being processed. For these libraries, we apply a small variation of the management of atomic counters where we simply increment the counter upon entering the wrapper of the application function exploiting these libraries, and we decrement the counter right before leaving the function, hence while executing the tail of the wrapper. In this way, the lifetime of the function execution always sees a counter with at least one unit, which avoids the actual preemption of the running task during the execution of that function.

This solution is also feasible for managing application functions that make use of TLS. As pointed out before, these cannot be interrupted asynchronously since we might give rise to interference on the usage of the same TLS instance if the thread starts running another task that needs to access this storage—this scenario would violate atomicity in the management of this storage. Our solution based on wrapping and atomic counters resolves this aspect.

7. This solution is similar in spirit to the one offered by the Linux kernel in order to manage preemptability of threads running in kernel mode. However, in that solution the check on the counter value is purely synchronous (e.g. it is carried out upon returning from some kernel level function along the thread to be possibly preempted, like for example the `printk` function). Rather, in our solution the check is carried out asynchronously when transiting into OMPPC.

As a final aspect, upon resuming the OMPPC context, the asynchronously interrupted thread does only a non-blocking attempt to acquire a `GOMP` internal mutex before it is allowed to execute the task-to-CPU reassignment, which is actually executed only if the mutex try-lock succeeds. Otherwise, the thread simply resumes the TC context of the task it was carrying on prior to being interrupted by the hardware profiler. This strategy makes interrupted threads not hang whenever another thread is already executing `GOMP` activities related to shared data structures. Hence, it provides minimal intrusiveness in terms of wait-access to the mutex-protected critical section by concurrent interrupted threads.

3.3 The Task Scheduling Architecture

The new task scheduling facilities that we added to the `GOMP` implementation are fully transparently activated via wrappers nested at compile/link time. In the rest of this section, we will illustrate the steps through which our innovative facilities have been built, thus posing attention to all the architectural updates/integrations that we paired with the original structure of `GOMP`. We again remark that our updates/integrations do not overwrite the original functionalities, rather they offer to the programmers (or end-users) the possibility to activate or not the functionalities we devised when running their applications. This can simply occur by configuring a `GOMP` environment variable—or invoking the newly offered routines of the run time environment as an alternative. Clearly, our re-engineered environment can exploit the loadable Linux module—and the associated user-space facilities—supporting task preemption, which we described in Section 3.2. This still occurs fully transparently to the native `GOMP` implementation and the overlying applications.

3.3.1 Basic control structures

When a #pragma omp parallel directive is encountered, which sets up the parallel region for the application execution, the actual executable code invokes the `gomp_team_star` function to setup the team of worker threads. We transparently injected within this function—via wrapping—a call to the `gomp_state_pool_init` function just before creating other threads. This function is part of a group of functionalities that manage activities related to a dynamic pool of `gomp_task_state` structures. These structures will be assigned to tasks (via a pointer in the `gomp_task` structure) upon their activation and will be released upon completion. They enclose both a pointer to the alternate stack memory area over which a task executes independently from the others and a `gomp_task_context` structure—which implements the TC—maintaining the content of the registers at the time of task preemption.

Before the worker threads reach the barrier preceding the work-sharing region, we check (again via a wrapper) if they can register themselves for exploiting the IBS hardware-profiler support offered by AMD processors. Registering will allow them to receive periodic interrupts that will be handled by our Linux kernel module supporting task preemption, as illustrated in Section 3.2.

When one of these interrupts is received, control is bounced to an "early logic" associated with the context

OMPPC, which implements the preamble of our preemptive task scheduling logic. In our design, this preamble verifies if the task currently executed by this thread may effectively be preempted in favor of another one[8], then saves the task's state at the time at which the thread was interrupted by storing the content of CPU-core registers into the corresponding `gomp_task_context` data structure. At this point, we can give control to the ULMT (User-Level-Micro-Thread) scheduler (to be described in Section 3.3.2).

The passage of control to the scheduler in our re-engineered environment is not only due to the occurrence of asynchronous events—namely, the hardware-profiler interrupts leading to the restoration of OMPPC and enabling the possibility to reevaluate the CPU assignment based on priorities. Rather, the environment also embeds a form of synchronous invocation of the task scheduler. More in detail, while executing a task, a thread may encounter different kinds of directives—executable and stand-alone directives to be more precise—corresponding to the invocation of the associated library functions. These function calls match the TSPs at which the runtime environment may or may not renew the scheduling decisions (namely, the assignment of tasks to threads). Nevertheless, the execution of some of these functions in `GOMP` can block the running thread—via futexes—because of the need to respect the dependency constraints among tasks imposed by the programmer whenever they are not found to be directly satisfied upon reaching the TSP. This is the scenario where task A executed along thread T_1 needs to wait for the finalization of a tied task B which has already been activated for processing along some other thread T_2. In this case, thread T_1 can no longer take care of passing control to B at the TSP to resolve the dependency—since this task has been already definitely tied to thread T_2—and in the original `GOMP` environment has no opportunity to rely on per-task contexts to pass control to another standing task. We refer to the #pragma omp taskwait directive, which finds its implementation in the `GOMP_taskwait` function, or the `gomp_task_maybe_wait_for_dependencies`. Overall, in `GOMP` these functions are implemented in a way preventing the environment to behave work-conservatively despite the possible presence of pending tasks whose execution along thread T_1 would not not violate TSCs. In our ULMT re-engineered environment we avoid this problem since we enable the passage of control to another task via a synchronous invocation to the ULMT scheduler upon reaching a TSP if the associated constraints are not currently satisfied. This is again achieved fully transparently since the synchronous call to the ULMT scheduler is embedded within a wrapper nested in the API functions `GOMP_taskwait` and `gomp_task_maybe_wait_for_dependencies` at linking time. In Figure 4, we depict the execution flows that can occur in the ULMT runtime environment, highlighting which of them cannot occur in the native `GOMP` implementation.

As a final note, `GOMP` also supports the #pragma omp critical directive that maps to the `GOMP_critical_start` function. This function allows tasks to rely on a global lock for executing critical sections. In ULMT, we have transparently wrapped this function to let it perform—beyond the management of the per-task atomic counters used for avoiding preemption based on asynchronous interrupts—a try-lock operation (rather than a blocking one), whose failure will lead the wrapper to invoke the ULMT task scheduler so as to switch off the CPU the task that needs to access the busy lock. Similarly, we wrapped those functions associated with the set of general-purpose locking routines that rely on the `omp_lock_t` structure for synchronization purposes. This again enables the execution to slide towards work-conservativeness by enabling a thread, which would otherwise be blocked while running a task that needs to access a busy critical section, to take care of processing other standing tasks.

8. As hinted in Section 3.2 we avoid preemption of tasks that acquired resources exclusively or are executing some non-reentrant function or some function exploiting thread-local-storage.

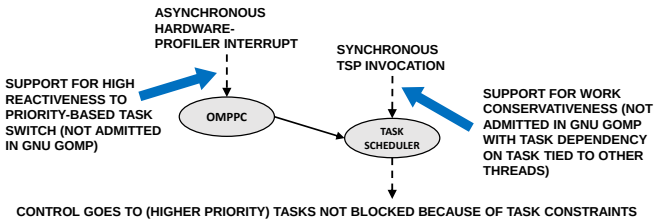

Fig. 4: Novel execution paths admitted in ULMT.

3.3.2 The ULMT scheduler

In this section, we present all the mechanisms at the base of the ULMT scheduler. As a first important note, while the original `GOMP` implementation only manages a single queue of standing tasks that are runnable, in ULMT we have introduced multiple queues because of the need to correctly satisfy OpenMP constraints in the more sophisticated task management environment entailing differentiated contexts for the different tasks. As noted, the last aspect is absent in `GOMP` since the arrival to a TSP never leads to switch to a different context—in fact, even under the scenario where the task reaching the TSP has unsatisfied constraints that lead the thread to run another task (e.g. a child task), this happens on the same stack, via synchronous invocation of the function associated with this task by the thread. Overall, the list of the task queues exploited in ULMT is the following one:

- GRQ (Global-Runnable-Queue), which is the original `GOMP` queue. It contains all the tasks that can be run by any thread. When a new tied/untied task is created, it is inserted into this queue. We kept this behavior operative in our version since it enables the task to be promptly taken by any thread accessing this global queue.
- GBQ (Global-Blocked-Queue), which is a new ULMT queue that keeps all the currently blocked untied tasks (so their contexts have been descheduled by the ULMT scheduler).
- LRQ (Local-Runnable-Queue), which is a new ULMT queue, with an instance per-thread, that keeps all the tied tasks that are currently runnable—possibly after a block phase—and must be finalized by the specific

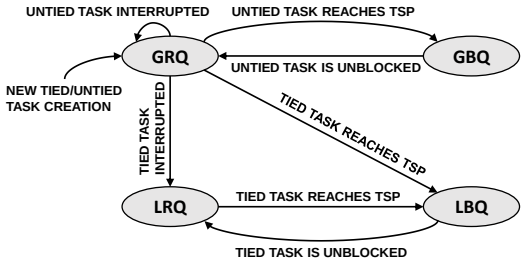

Fig. 5: Task-state diagram in ULMT.

thread given that they have been originally picked from the GRQ by that thread.

- LBQ (Local-Blocked-Queue), which is a new ULMT queue, with an instance per thread, that keeps all the currently blocked tied tasks, which must be finalized by the specific thread along which they have started their execution.

GRQ and LRQ are implemented as splay trees—this is the data structure originally exploited in `GOMP` for GRQ—so as to associate a different node to the sublist of tasks with a given priority. GBQ and LBQ have been implemented as simple doubly linked lists, allowing constant-time removal of an element once we have the pointer to the `struct gomp_task` representing the task to be removed from the queue. GBQ and LBQ keep tasks that are blocked because their scheduling constraints are not currently met and those that are blocked since they need to access some busy resource (e.g. a busy lock).

The moving of tasks among the queues, caused by the hardware-profiler interrupt (which is asynchronous with respect to the execution of the task) or by the invocation of a TSP (which is instead synchronous), is depicted in Figure 5. A task kept in GRQ can be moved to LRQ if it is tied, it has been picked by the thread—thus starting its execution—and is then interrupted because of the passage of control to some higher priority task. It will run again only along the thread that has picked it when it will become the highest priority task observed among GRQ and LRQ by that thread upon running the ULMT scheduler. The transit towards GBQ is only admitted for untied tasks that are CPU-dispatched along a thread and then trap into a TSP with non-satisfied constraints (or try to access some busy lock). On the other hand, tied tasks that are eventually CPU-dispatched either from GRQ or LRQ are always put into LBQ when a TSP with non-satisfied constraints is met (or the access to a busy lock is attempted). Overall, by the state diagram, we have that a tied task is never allowed to bounce back to GRQ or GBQ once picked by a given thread, thus always residing in the per-thread queues along its lifetime. This satisfies the OpenMP constraint that tied tasks cannot migrate to other threads once started up along a given thread.

Clearly, when a task is moved back to GRQ from GBQ—or to LRQ from LBQ—it is inserted at the priority level specified upon task creation. However, given that in the original implementation of `GOMP` no move back to a "runnable queue" from a "blocked queue" was in place—in fact, as hinted, "blocked queues" were even not present at all—a decision must be taken on the actual position to select for queuing the task into the corresponding priority level. As for this point, we decided to adopt different queuing policies of unblocked tasks into GRQ and LRQ. For LRQ, we adopted a classical tail insertion. On the contrary, for GRQ rather than inserting the task at the tail of the sublist corresponding to the correct priority level, we decided to take the opposite choice of queuing it at the head. In other words, within the same priority class of GRQ, a task being currently unblocked will be CPU-dispatched before other runnable ones already standing at the same priority level. The motivation for this choice is illustrated with the help of Figure 6. Specifically, the GRQ may contain both not yet CPU-dispatched tasks—in fact, by the task state-diagram, any task is inserted into GRQ upon its creation—and tasks that have already been dispatched at some point in the past and then were either: (A) interrupted in favor of some higher priority task by the hardware-profiler-based task preemption support; (B) blocked upon meeting non-satisfied TSCs or attempting the access to some busy lock along their execution—with consequent context switch to another task context—and then moved back to GRQ as soon as the constraints became satisfied or the needed lock becomes free.

Tasks that were not yet CPU-dispatched can be seen as "cold ones" (they are represented in blue in Figure 6), meaning that no resources (memory buffers, I/O channels, etc.) were yet committed for their execution. Instead, the ones that were already CPU-dispatched at some point in time can be seen as "hot ones" (they are represented in red in Figure 6) given the commitment of some resources for their partial execution. The insertion of unblocked tasks—the ones in point (B)—at the head of the per-priority level in GRQ when moving them back from GBQ can favor a more prompt execution of hot tasks, with the advantage of more timely release of the corresponding committed resources—this, in turn, can favor aspects such as locality. Clearly, all the tasks that stand on the per-thread LRQ are hot given that they where all CPU-dispatched at least one time in the past (see again the diagram in Figure 5), which is the reason for not applying the head insertion rule when moving tasks back from LBQ to LRQ. In the literature, some solutions discriminate classes of tasks (hot/cold) associated with a thread, like Cilk [19]. However, our solution also provides the integrated management of multiple queues, in combination with synchronous and asynchronous switch of the task contexts along thread execution.

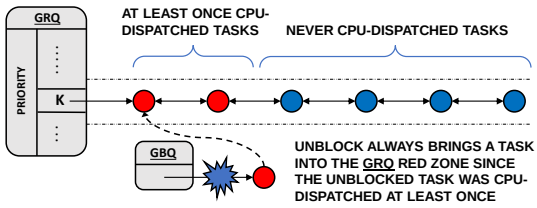

Fig. 6: Hot and cold task zones into GRQ.

An additional optimization we integrated within the ULMT task scheduler applies to GRQ and each individual LRQ associated with the different threads. This optimization is based on the idea of further favoring—within a same priority level—the tasks that more recently used the CPU.

This can additionally favor aspects such as locality, especially in scenarios of fine-grain tasks where the tasks that used a CPU-core after that others were context switched off the CPU may have only partially invalidated cached data that will be re-accessed by those same tasks when CPU-dispatched again. We note that this can be relevant even in scenarios where a task is untied and is moved across different threads along its lifetime, given the sharing of lower-level caches among the CPU-cores on top of which the threads may run—the same argument applies to top-level caches for, e.g. hyper-threaded CPUs.

This optimization is based on keeping for each task A a reference (a pointer) to the task B, which released the CPU in favor of A. Overall, each task keeps a reference to the "last running" task observed upon its CPU-dispatch operation. When the ULMT scheduler is invoked along a thread, the thread identifies the GRQ or LRQ sublist with the highest priority having runnable tasks and—instead of passing control to the task at the head of the sublist—it checks whether the last running task (B in our example) is on the same priority level and is now runnable. In the positive case, control goes to this task. Clearly, if such last running task B is currently blocked, we can exploit a reference it may keep to a second last running task C, thus favoring C in the schedule if it is runnable and stands at the highest priority level. This approach can iterate, but we can anyhow impose a bound on the iterations—falling back to CPU-dispatching the head task of the non-empty sublist at the highest priority level—to make the schedule operation executable in constant time. We also note that, since the "last running" referenced tasks have been CPU-dispatched at least one time—so they are hot tasks in our representation—then this optimization still prevents that a cold task kept by the GRQ is CPU-dispatched before some other hot task at the same priority level exists. Clearly, this optimization may result very useful in scenarios where a task A loses control of the CPU because of a hardware-profiler interrupt in favor of a higher priority task B, and then when B terminates—and the ULMT scheduler is again invoked, A stands at the highest non-empty priority level together with other runnable tasks. In this scenario, A is favored over the other tasks, thus hopefully finding again cached data to exploit along its execution.

4 Experimental Data

4.1 Platform

We assessed our ULMT proposal on top of two machines, whose configuration details are reported below:

Machine	CPU	Total cores	Clock frequency	Memory access model	Memory size
M1	2xAMD Opteron 6174	24	2.2 GHz	NUMA	32 GB
M2	1xAMD EPYC 7402P	24	2.8 GHz	UMA	64 GB

We assessed ULMT by comparing the values of performance indexes with the corresponding ones achieved by running the native version of GOMP. The comparison is carried out while scaling up the number of threads up to the maximum number of physical CPU-cores in the underlying machine. Also, we prevent the runtime from employing *task throttling* heuristics—which serialize the execution of tasks along threads when specific thresholds of standing tasks are met—since they have been proved to be harmful [20] for some application classes. However, we allow the applications to control the overhead for task creation and management by relying on the *cut-off* strategies provided by the same benchmarks. As a final note, on machine M1 we used kernel version 4.12, while on machine M2 we used kernel version 4.9. Also, gcc version 7.2 has been used as the compilation tool-chain.

4.2 Benchmarks

As benchmarks, we used various applications from BOTS (the Barcelona OpenMP Task Suite) [7]. This benchmark suite has been devised to bypass the limitations of previous benchmarks in terms of their capability to generate irregular and fine grain workloads with task dependencies. On the other hand, one limitation of BOTS, which has not yet been resolved by more recent proposals like KASTORS [21], is the lack of the definition of OpenMP applications specifically devised to stress the runtime system in the presence of task priorities. The motivation for this lack is related to the recent inclusion of priorities in the OpenMP specification—i.e. OpenMP 4.5. Given that the effective management of priorities is one of the central targets for our innovative ULMT proposal, we designed and implemented an innovative open-source benchmark application named HASHTAG-TEXT, entailing the possibility to configure different priority levels for OpenMP tasks. As pointed out in the introduction, this benchmark is already available for download[9]. Nevertheless, we still used applications from BOTS to assess the overhead produced by our re-engineered GOMP runtime, as well as the benefits that can be provided thanks to its orientation to work-conservativeness.

TABLE 1: Applications selected from BOTS

Name	Brief description	Domain	Nested tasks	Cut-Off	Worksharing	Taskwait
ALIGNMENT	Protein alignment with the Myers and Miller algorithm [22]	Dynamic programming	no	none	for	no
					single	
FLOORPLAN	Computation of the minimum area size including all cells	Optimization	yes	none	single	yes
				depth-based		
SPARSELU	LU matrix factorization over sparse matrices	Sparse linear algebra	no	none	for	no
			yes		single	yes
STRASSEN	Hierarchical decomposition for multiplication of large matrices [23]	Dense linear algebra	yes	none	single	yes
				depth-based		

In Table 1 we report the set of applications we selected from BOTS, together with a few of their most relevant characteristics (the reader can refer to the original specification of these applications for further details). With the exception of ALIGNMENT and SPARSELU in its configuration based on *for* work-sharing, the selected applications have *nested tasks*, which may lead to taskwait dependencies along the execution of the parent tasks. Hence, they can represent good test cases to determine whether our design can provide advantages through the avoidance of thread blocks upon a taskwait—this is where the orientation to work-conservativeness stands. On the other hand, all the selected benchmarks, except FLOORPLAN and SPARSELU

9. https://github.com/HPDCS/OpenMP-Task-Suite

in its configuration based on *single* work-sharing, are embarrassingly parallel (no thread ever blocks because of task dependency constraints) and have no particular scalability issues. Therefore, they are helpful especially for the determination of the overhead of our proposal in a scenario where work-conservativeness—as well as priorities—is not a real concern. Also, they cover a set of different important domains, ranging from dynamic programming to optimization, passing through both sparse and dense linear algebra applications[10].

As for our new HASHTAG-TEXT benchmark, it has been conceived to handle a large dataset of ⟨hashtag, text⟩ pairs to serve requests coming from users with different priority levels who wish to retrieve all texts associated with a given hashtag passed in input. This is the classical case of access to posts in scenarios like social networking. The benchmark uses a collection of *tweets* (captured during one day of listening on the Twitter's streaming channel) to populate the structures allocated at service startup. In more detail, the service relies on one or more hash tables to handle partitions of uniformly distributed data extracted from the same dataset, each one having a fixed but configurable number of buckets where colliding ⟨hashtag, text⟩ pairs are kept in a list. Since no data is ever replicated across different hash-table instances, relying on the functional parallelism offered by the OpenMP tasking model appears to be a good choice for accomplishing the application objectives. In fact, we can search for texts associated with a given hashtag in parallel across the different hash tables, also exploiting the advantages of reduced impact of scanning the list of conflicting items (since they are split into different hash tables, leading their size to be reduced compared to the case of a single hash table) and enabling fully parallel search on the same hashtag key. On the other hand, making a hash table maintain a subset of hashtags as its indexing keys (namely, partitioning the dataset on the basis of the hashtag) would not allow for embarrassingly parallel search in that same hash-table instance. The service is in charge of mapping requests arriving with a given priority to OpenMP tasks with corresponding priority levels. These tasks do not do any particular computation but spawn as many child 'text-search' tasks as hash tables before synchronizing with them through the taskwait directive. In our experiments, the application has been configured to generate requests with three different priority levels, namely 1, 2, and 3 (with higher levels having higher priority), according to a mixture of 60%, 30% and 10%, respectively. This mimics a scenario with a majority of regular users, a good percentage of "silver" users, and a minority of "gold" users[11]. Request arrivals to the HASHTAG-TEXTS service can be configured with average arrival rate λ, to which it corresponds an inter-arrival time period $1/\lambda$. This allows us to observe the outcomes of the service execution with different workload levels. We have set to 4 the number of hash tables keeping the data partitions, each one provided with 100 buckets of collision lists. We selected these values since we observed that they provide good load balancing even with the lowest workload level, while they do not impair parallelism with higher arrival rates of requests. Although the HASHTAG-TEXTS benchmark is designed to work correctly with both tied and untied tasks, we decreed the latter to best apply to the actual semantic of the application. Additionally, they stress the ULMT runtime more because of the expensive management required to handle this kind of task—e.g. caused by the need for concurrent accesses by multiple threads to the GBQ global task queue. This overhead would tend to reduce the actual performance gain we can achieve through our ULMT solution—since it represents a baseline execution cost to be anyhow spent—compared to what we may expect when porting this solution to runtime systems based on separated per-thread tasks queues and work-stealing across the different threads (like the Intel OpenMP environment [24]). Therefore, the selected configuration can help determining a sort of baseline expectation on the outcomes achievable through our proposal.

4.3 Results with BOTS

In this section, we focus on the experimental results obtained with BOTS applications. Given that they do not entail differentiated priority levels, there is no need to exploit the hardware-profiler periodic interrupts for promptly passing control to higher priority tasks. On the other hand, the management of different contexts performed by ULMT is still useful to block/unblock tasks depending on their dependency constraints. Hence, these experiments focus on assessing the trade-off between the overhead for separate contexts management—carried out according to the solutions proposed in Section 3—and the advantage thanks to more flexible management of tasks (and their time-shared execution) along threads. In Table 2, we report the execution times of the selected BOTS applications (averaged over 8 runs). We show the execution times with the original GOMP and with ULMT, and the relative speed-up values (i.e. the ratio between the two execution times).

As for ALIGNMENT, we observe almost negligible positive speed-up values by the ULMT runtime, with both machines M1 and M2. This is because the benchmark does not use nested tasks, thus it does not need to rely on the taskwait construct to accomplish synchronization between parent and child tasks. In this scenario managing separate task contexts to avoid thread-block caused by task dependency constraints, does not offer advantages. However, the overhead introduced by ULMT is essentially negligible, also motivated by the fact that a large fraction of the execution time is spent within code not included in the GOMP library. This phenomenon is linked to the relatively large granularity of ALIGNMENT tasks, which is about 14 milliseconds on M1 and 3.5 milliseconds on M2[12]. Overall, the relative cost for the creation of tasks (and their contexts) and the

10. Early data gathered for the other BOTS applications have show execution profiles well represented by the four benchmark applications we focus on in the analysis.

11. In the case where users would in their turn represent external services accessing the HASHTAG-TEXT service on behalf of their customers, the possible adoption of data caching at the level of the external services is not a concern for the validity of our benchmark configuration. In fact, in that case, thee load towards the HASHTAG-TEXTS service would simply represent the fraction of the volume of customers' requests not actually served through cached data.

12. We recall that M2 has a multi-core processor released in 2019, while M1 is based on multi-core processors released in 2010.

management of task-switch operations introduced in ULMT is irrelevant with this benchmark execution profile.

As for FLOORPLAN, the granularity of its tasks is much finer and varies depending on whether a cut-off strategy is employed or not—tasks have granularity of the order of nanoseconds when no cut-off is used, in contrast to the order of microseconds when manual cut-off or if clauses are employed. This leads to the scenario (opposite to ALIGNMENT) where a substantial percentage of time is spent executing code of the GOMP library rather than code of the overlying application. By the data, we note that tied and untied versions of this benchmark with no cut-off strategies produce results, in terms of execution time, which are two orders of magnitude worse than those achieved with cut-off. Also, execution times with no cut-off do not scale with the number of threads with both the baseline GOMP and the ULMT runtime. By looking at data, the ULMT runtime gives rise to speed-down compared to the baseline GOMP—caused by the additional task management operations—only under these pathological settings, essentially representative of an application-level "misconfiguration" that gives rise to thrashing phenomena while managing tasks. Instead, when running with cut-off (hence correct settings for this application profile) the ULMT runtime gives rice to positive speed-up (with about 30% of improvement). Since FLOORPLAN makes heavy usage of nested tasks, we observed with ULMT thousands of task-switches occurring upon the execution of code associated with the taskwait construct, which confirms the benefits achievable when forcing the task scheduler to behave work-conservatively whenever possible.

As for SPARSELU, the versions that use the for work-sharing construct immediately distribute all work across threads. They are characterized by an embarrassingly parallel execution scheme where no thread ever interferes with the work of the others, nor they need to synchronize on dependencies defined between parent and child tasks since no taskwait directive is provided in the code (see Table 1). Conversely, the versions that rely on the single work-sharing construct create all the first-level *explicit* tasks, which in their turn generate nested tasks to achieve a good work distribution and load balance. However, we observed by the experiments that, upon the execution of the taskwait construct, the threads have mostly found either locally spawned tasks available to be taken in charge or the dependency constraints already met, which are favorable cases to the baseline GOMP implementation. All these reasons explain the low speed-up values achieved by the ULMT runtime in a few configurations. In any case, no relevant penalty is noted for the additional work by ULMT related to managing separate task contexts.

As for STRASSEN, ULMT shows speed-down just for the configuration with no cut-off and untied tasks. The motivations are twofold. First, the absence of cut-off leads, as hinted, to more pressure on the handling (creation and scheduling) of tasks, which in ULMT is more costly because of the separation of task contexts. Second and more important, ULMT has an extra overhead paid whenever the execution of a nested task requires to suspend the currently executed untied one, which needs to be queued into GBQ, and then put back into GRQ when becoming ready again. In fact, these queues are fully shared among all the threads, thus imposing synchronized accesses for their manipulation[13]—the original version of GOMP only has the GRQ fully shared queue, while our version also has the fully shared GBQ, whose accesses require additional synchronization by the threads. However, with well-configured application settings, and proper usage of cut-off strategies, the overhead caused by ULMT pays off because of the achievement of better usage of CPU-cores thanks to its orientation to work conservativeness.

In any case, the ULMT overhead caused by an excessively frequent switch between task contexts under, e.g., misconfiguration of the application could be controlled by introducing an additional mechanism that supervises some performance index (e.g. the task throughput) and dynamically excludes the multi-context mechanism in scenarios where it cannot positively impact on the actual execution.

4.4 Results with Hashtag-Text

In this section, we focus on the results obtained with HASHTAG-TEXT (HT). As pointed out, this benchmark application is fully complementary to literature benchmarks since it makes use of task priorities. On the other hand, similarly to other benchmarks, it relies on parent/child task synchronization. With HT, ULMT can exploit the hardware-profiler periodic interrupt to promptly pass control to higher-priority standing tasks. It can also exploit context-switch between tasks to avoid thread block upon taskwait directives invoked by parent tasks. As for the periodic hardware-profiler interrupt (we recall that on the target platforms, which are based on AMD processors, we exploited IBS as the hardware profiler), we have selected three different periods, namely 100, 50 and 25 microseconds.

In the top plots of Figure 7 and Figure 8 we show the results obtained with no interarrival time between requests for the two different machines used in the experiments. meaning that we have a continuous injection of requests. This is the most-intensive workload scenario, in which the continuous creation of tasks also generates the heaviest contention for accessing data structures where tasks are placed before being executed. We report the speed-up achieved by UMLT vs. the baseline GOMP for both the application execution time and the request response time for the three different request/task priority levels. By the results, ULMT outperforms the baseline GOMP at all thread counts, with speed-up of the execution time that ranges from 2% to about 8% on machine M1, and from 3% to about 7% on machine M2. Noteworthy, the application takes benefits by the ULMT runtime thanks to its support for work-conservativeness, but it takes no benefits from the hardware-profiler support and the associated capability for asynchronous task switching. In the plots, IBS OFF represents the settings where IBS interrupts are simply switched off, while task context switches enabling the avoidance of block phases for threads are still active. This is somehow expected because

13. As pointed out before, full sharing task queues keeping untied tasks—compared to work stealing—provides an assessment with no specific queue management approach favorable to reduce the ULMT overhead. Hence, it represents a test scenario where ULMT is assessed to determine its baseline effects in an independent manner from other optimizations.

TABLE 2: Results with the applications from BOTS.

Version		8 Threads				16 Threads				24 Threads			
		Execution Time (seconds)		Speedup		Execution Time (seconds)		Speedup		Execution Time (seconds)		Speedup	
		M1	M2	M1	M2	M1	M2	M1	M2	M1	M2	M1	M2
Alignment													
for-untied	*Baseline*	18.936	4.502	1.002	1.001	9.752	2.330	1.002	1.005	6.627	1.602	1.004	1.016
	ULMT	18.898	4.499			9.730	2.319			6.598	1.576		
for-tied	*Baseline*	18.926	4.528	0.995	1.001	9.576	2.303	0.997	0.999	6.267	1.551	0.972	1.006
	ULMT	19.030	4.525			9.601	2.306			6.447	1.541		
single-untied	*Baseline*	18.980	4.496	0.998	0.999	9.769	2.306	1.002	1.002	6.669	1.568	1.008	1.006
	ULMT	19.022	4.501			9.746	2.301			6.616	1.559		
single-tied	*Baseline*	18.899	4.523	0.997	1.001	9.523	2.302	0.993	0.001	6.190	1.532	0.977	0.997
	ULMT	18.949	4.518			9.589	2.291			6.337	1.537		
Floorplan													
untied	*Baseline*	153.534	31.484	0.625	0.744	24.355	24.355	0.626	0.626	330.681	20.638	0.592	0.543
	ULMT	245.655	42.299			38.907	38.907			558.558	38.001		
untied-if	*Baseline*	5.956	1.223	1.263	1.298	3.145	0.561	1.194	1.047	2.020	0.373	1.056	0.820
	ULMT	4.716	0.942			2.633	0.536			1.912	0.455		
untied-manual	*Baseline*	3.359	0.514	1.271	1.204	1.781	0.240	1.267	1.013	1.079	0.160	1.194	1.013
	ULMT	2.642	0.427			1.406	0.237			904	0.158		
tied	*Baseline*	185.456	32.253	0.643	0.624	386.984	24.193	0.645	0.550	616.214	22.448	0.690	0.531
	ULMT	288.368	51.671			600.359	44.002			893.358	42.276		
SparseLU													
for-untied	*Baseline*	172.677	24.182	1.012	0.999	90.696	12.536	1.006	1.003	63.972	9.108	1.009	1.000
	ULMT	170.640	24.208			90.135	12.501			63.429	9.111		
for-tied	*Baseline*	172.752	24.143	1.012	1.000	90.352	12.494	1.008	0.997	63.785	9.033	1.010	0.994
	ULMT	170.714	24.143			89.631	12.528			63.177	9.085		
single-untied	*Baseline*	167.308	24.231	1.043	0.995	86.180	12.426	1.047	0.997	59.281	8.960	1.039	0.991
	ULMT	160.393	24.348			82.281	12.463			57.034	9.042		
single-tied	*Baseline*	167.064	24.719	1.042	0.995	86.195	12.644	1.048	1.002	59.216	9.048	1.047	1.002
	ULMT	160.308	24.848			82.210	12.614			56.576	9.031		
Strassen													
untied	*Baseline*	33.637	5.584	0.865	0.704	19.656	3.076	0.858	0.563	14.046	2.457	0.776	0.478
	ULMT	38.888	7.935			22.908	5.461			18.090	5.142		
untied-if	*Baseline*	35.585	5.771	1.029	1.037	22.285	3.551	1.046	1.005	17.819	2.947	1.073	0.999
	ULMT	34.587	5.567			21.312	3.532			16.602	2.950		
untied-manual	*Baseline*	35.661	5.787	1.033	1.043	22.220	3.537	1.054	1.008	18.246	2.942	1.089	1.009
	ULMT	34.514	5.546			21.078	3.510			16.752	2.915		
tied	*Baseline*	33.246	5.852	1.014	1.054	19.163	3.537	1.009	1.016	14.243	2.922	1.044	1.011
	ULMT	32.796	5.550			18.985	3.482			13.638	2.889		

of the corner case related to the absence of interarrival time between requests, which leads both UMLT and the baseline `GOMP` to mostly process the highest priority tasks along time, since the corresponding queues are unlikely observed as empty. This essentially annihilates the capability of ULMT to more promptly switch to the execution of standing higher priority tasks.

As soon as reasonable interarrival time of requests is set, the benefits provided by the asynchronous task-switching capabilities of ULMT become more evident. In the bottom plots of Figure 7 and Figure 8 we report data for the scenario with request interarrival time set to 10 microseconds. With this setting, the speed-up values of the application execution times provided by ULMT again come from the orientation to work-conservativeness, but the speed-up values achieved by ULMT on the response time of higher priority requests are considerably increased in various configurations, especially those with highest interrupt rate by the IBS hardware profiler. Globally, the achieved speed-up values of the response times range from 3% to 11%, from 5% to 38% and from 3% to 60% respectively for the tasks with priority 1, 2 and 3 with machine M1. At the same time, they range from 8% to 45%, from 5% to 27% and from 4% to 24% for these same priorities with machine M2. This is also an indication of the lightweight nature of our solution to support asynchronous task-switching at very fine granularity. Furthermore, ULMT enables improvements of the response time also with priority 1 requests, especially at larger core counts. This is motivated by the exploitation of work-conservativeness for these requests in ULMT in scenarios where we have several threads that get blocked with `GOMP` while processing these requests—because of unsatisfied task constraints. At the same time, given the non-minimal amount of available cores, higher priority requests are likely processed by threads running on top of another subset of the cores.

TABLE 3: Overhead with different IBS settings - machine M2 - 24 threads running HT.

IBS setting	100 microseconds	50 microseconds	25 microseconds
Measured overhead	2.12 %	2.46 %	4.81 %

To further assess the light weight of the support for the asynchronous switch, we report in Table 3 the overhead of the IBS interrupt management—expressed as the percentage of increase in the execution time when IBS is active compared to the case of `IBS OFF`. For these experiments we do not really exploit the IBS interrupt to change the execution flow of threads in favor of higher priority tasks. Rather, we simply let the handler of the IBS interrupt return, hence introducing the overhead of managing the interrupts with no real advantage. By the reported data we see that the overhead is below 5% when the IBS is configured to send the interrupt every 25 microseconds, while it is below 2.5% when the IBS is configured to send the interrupt every 50 or 100 microseconds.

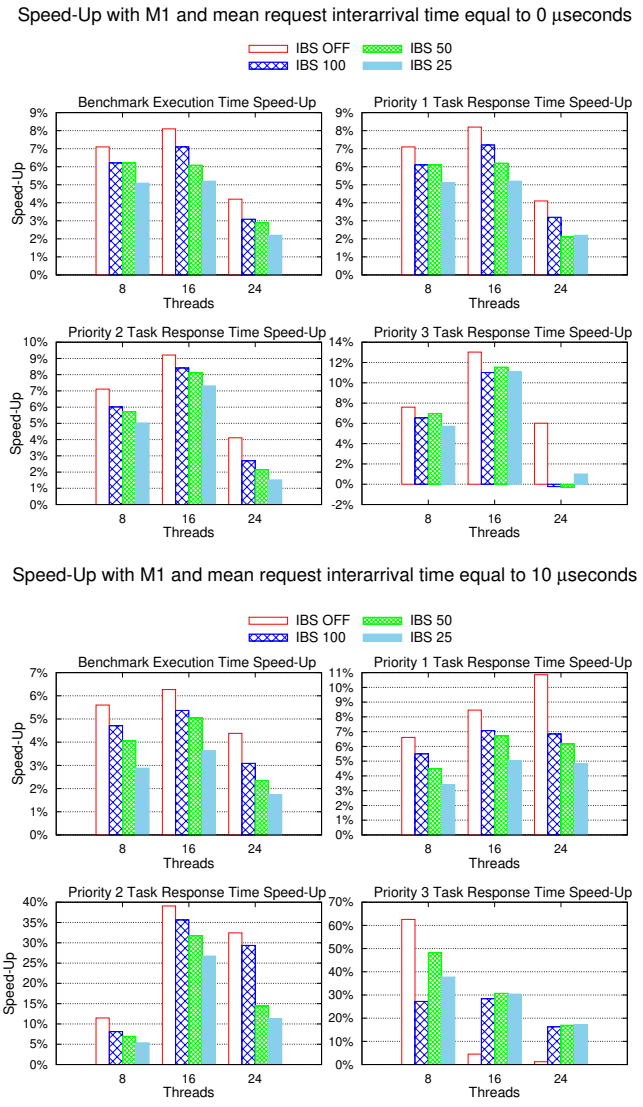

Fig. 7: Speed-up of benchmark execution time and task response time with priority 1-3 (machine M1)

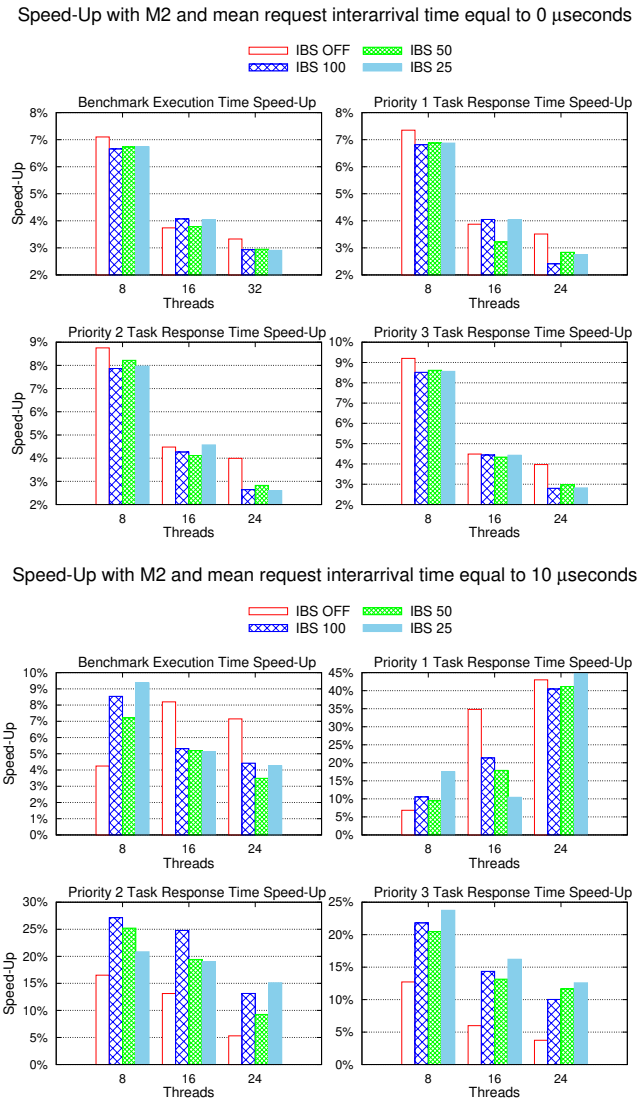

Fig. 8: Speed-up of benchmark execution time and task response time with priority 1-3 (machine M2).

5 Conclusions

The relevance of OpenMP for parallel programming is definitely growing, so that the corresponding runtime systems need to be reshuffled to match the objectives of its newer releases better. In this article, we have presented a reshuffle of the GNU OpenMP (GOMP) runtime environment, introducing two core innovative functionalities: 1) the support for fine-grain asynchronous reassignment of CPU-cores to higher priority tasks, and 2) the support for avoiding thread-block phases when unsatisfied parent/child task dependencies materialize (an aspect linked to the so called work-conservativeness property of OpenMP runtime systems). Since the introduction of task priorities in OpenMP is recent, none of the OpenMP benchmarks in the literature has revealed effective for testing our innovative runtime solutions. To bypass this problem, we have presented a fully new benchmark—called HASHTAG-TEXT—that has been released as open source. Our innovative GOMP runtime facilities have been released as open source as well, and we have also presented the results of a comprehensive experimental study demonstrating the effectiveness of our proposal.

References

[1] "OpenMP." [Online]. Available: https://www.openmp.org/
[2] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, "The design of openmp tasks," *IEEE Trans. Parallel Distributed Syst.*, vol. 20, no. 3, pp. 404–418, 2009.
[3] "OpenMP Application Programming Interface." [Online]. Available: https://www.openmp.org/specifications/
[4] M. A. Serrano, A. Melani, R. Vargas, A. Marongiu, M. Bertogna, and E. Quiñones, "Timing characterization of openmp4 tasking model," in *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, CASES 2015, Amsterdam, The Netherlands, October 4-9, 2015*. IEEE, 2015, pp. 157–166.
[5] J. Sun, N. Guan, Y. Wang, Q. He, and W. Yi, "Real-time scheduling and analysis of openmp task systems with tied tasks," in *2017 IEEE Real-Time Systems Symposium, RTSS 2017, Paris, France, December 5-8, 2017*. IEEE Computer Society, 2017, pp. 92–103.