Exploiting Elasticity via OS-runtime Cooperation to Improve CPU Utilization in Multicore Systems

Javier Rubio, Carlos Bilbao, Juan Carlos Saez, and Manuel Prieto-Matias
Facultad de Informática
Complutense University of Madrid
Madrid, Spain
Email: {jrubio05,cbilbao,jcsaezal,mpmatias}@ucm.es

***Abstract*—The chip multicore processor (CMP) architecture has become the predominant design choice for contemporary general-purpose systems across multiple sectors of commercial technology. Thanks to technological progress, CMP systems can now feature hundreds of cores. While multithreaded applications may potentially benefit from the increasing core counts, leveraging all available cores is not always feasible due to limited Thread-Level Parallelism (TLP), load imbalance among threads, and other scalability bottlenecks.**

Colocating multiple applications on the same node is becoming a popular practice to maximize processor utilization. In HPC, malleability –the ability to dynamically alter the number of active threads within the same application–, is also being exploited at the runtime-system level to better deal with scenarios exhibiting time-varying scalability. In the cloud, application colocation is leveraged along with different forms of coarse-grained elasticity to cater to the varying resource demands. This work introduces an operating system (OS) level elastic mechanism designed to efficiently leverage idle CPU periods in workloads consisting of unmodified applications, many of which do not rely on a runtime system to function. This mechanism constitutes a form of fine-grained vertical elasticity that leverages cooperation between the runtime system and the OS to maximize CPU utilization. To this end, it opportunistically increases the active thread count of malleable applications during idle periods. We implemented our proposed OS extensions in the Linux kernel, and augmented the GNU's OpenMP runtime to show a proof of concept of the required OS-runtime interaction. By using diverse multithreaded programs, we demonstrate the ability of the proposed OS support to substantially improve the system throughput.

***Index Terms*—Multicore processors, operating system, elasticity, Linux kernel, runtime system, OpenMP, malleability.**

1. Introduction

The chip multicore processor (CMP) architecture stands today as the *de facto* design choice for modern general-purpose systems. CMPs find a place in embedded devices, personal computers, high-performance servers, and virtually all sectors of commercial technology. Thanks to the striking advances in technology and processor design, CMP systems already reach hundreds of cores in server platforms. Likewise, a substantial increase in the core count is also taking place in the desktop market [1]. Take for instance the 14th generation of Intel Core i9 processors, with up to 24 cores.

Multithreaded applications constitute a straightforward way to benefit from increasing core counts. However, not every multithreaded program can effectively leverage all the system's cores, due to program phases with a limited degree of Thread-Level Parallelism (TLP) –e.g., explicit sequential sections– [2], the presence of load imbalance between worker threads [1], or other scalability bottlenecks associated with contention on shared memory-related resources between cores [3], [4].

In HPC environments, colocating multiple applications on the same node is becoming a popular practice towards improving processor utilization [4], [5]. Several colocation approaches have been explored in this context. One option is to statically partition the system's CPU resources, so that each co-running applications is assigned an exclusive set of cores throughout the execution [4]. However, the dynamic degree of parallelism that many HPC application exhibit over time makes this approach not only challenging but suboptimal [5]. Another alternative is to dynamically adjust the number of active threads of the co-running applications, so that they use an appropriate number of cores that caters to the needs of its current execution phase [6]. To this end, the application must support dynamic *malleability*, this is, the ability to utilize a varying number of threads/cores at runtime. In HPC, delivering malleability support to applications is simpler than in other scenarios, as HPC applications often make use of runtime systems; implementing malleability support at the runtime system level often requires little or no changes in the application code. In turn, cooperation between the multiple runtime systems of applications colocated in the same node makes it easier to implement policies that optimize resource utilization [6], [7].

In the cloud, maximizing resource utilization is paramount to increase revenue and reduce utility costs [8]. Increasing resource usage is effected via oversubscription [9] and elasticity [10]. Specifically, elasticity can be exploited horizontally (i.e., by increasing the number of instances of the application using multiple containers/VMs),

or vertically (i.e. by dynamically increasing the associated CPU and memory resources). Cloud elasticity mechanisms are based on the continuous monitoring of resource usage and/or the external requests that an application receives. Previous research has highlighted that, while this is suitable for client-server applications, it does not meet the needs of workloads that do not depend on external requests, as scientific applications [10]. Another limitation of current cloud dynamic elasticity mechanisms is that they are applied at coarse granularities; this often makes it difficult to effectively utilize cores that go idle just for a few seconds.

In this work, we propose an operating-system (OS) level elastic mechanism to efficiently harness CPU resources during idle periods on a multicore system. Specifically, we showcase the potential of automatically maximizing CPU utilization and improving system throughput when running a mix of unmodified applications, some of which feature malleability support in the runtime system. To do so, our proposal leverages explicit interaction between the OS and the runtime system, and reacts to OS-visible thread activations and deactivations (i.e., a thread blocks when waiting). Our proposal is meant to be used as a complementary method for vertical elasticity in the cloud, providing a means to increase CPU utilization and throughput that operates at a much shorter timescales than existing methods of vertical elasticity [8]. Moreover, it can be also used in general-purpose OSs to opportunistically improve CPU utilization in desktop environments.

To the best of our knowledge, this proposal is the first to leverage elastic OS-runtime scheduling for workloads spanning both unmodified malleable and non-malleable application types, all without the need for instrumentation or alterations to the applications' code. While previously-proposed techniques to improve CPU utilization also rely on application colocation and/or perform elastic scheduling decisions [4], [6], [7], they require that every application in the workload use a customized runtime system. Unlike our approach, these techniques do not leverage OS-runtime interaction, and are incompatible with legacy software, which may not rely on a runtime system to function.

Our work makes the following contributions:

1) We designed an OS-level mechanism that leverages idle cores left by non-malleable applications to opportunistically increase the active thread count of malleable applications. The proposed OS extension also reclaims the extra cores populated by malleable programs automatically, when non-malleable ones increase their amount of TLP.
2) We implemented the proposed OS-level support as a loadable kernel module, which can be loaded in unmodified Linux kernels (no kernel patches required).
3) As a proof of concept of malleability delivered by the runtime system, we extended the GNU's OpenMP runtime with malleability support for loop-based parallel programs. To benefit from this support, no changes in the application source code are required.
4) To assess the effectiveness of our proposal, we employ program mixes that include malleable and non-malleable applications. Our experimental evaluation shows that our approach brings substantial throughput gains (24.5% average improvement w.r.t. the default Linux scheduler). This is accomplished by accelerating malleable applications without causing a significant decline in the performance of non-malleable programs.

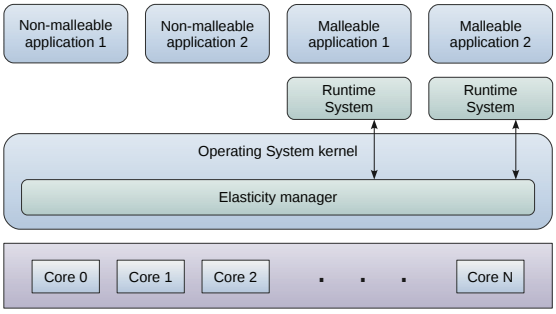


Figure 1: System overview

The remainder of this paper is organized as follows. Sec. 2 describes the design and implementation of our proposal. Sec. 3 covers the experimental evaluation. Sec. 4 discusses related work. Finally, Sec. 5 concludes the paper.

2. Design and implementation

Our proposal leverages cooperation between the runtime system and the OS kernel to maximize CPU utilization and improve system throughput via dynamic malleability.

Fig. 1 depicts the interaction between the various components of our framework and its intended use. The workload combines a mix of applications; each application may run directly on top of the operating system or inside a separate container. Some applications run unmodified programs, potentially legacy software, where threads not doing useful work block. These applications can be single- or multi-threaded, and they may or may not use a runtime system. In case a runtime is used, it does not interact with the OS or exploits malleability. For simplicity, we refer to these programs as *non-malleable* applications. Other programs leverage malleability via a cooperative runtime system that runs at user-space and interacts with our proposed OS-level elasticity manager. As a proof of concept to show the potential of this OS-runtime interaction, we augmented the GNU OpenMP runtime system with malleability support in *parallel for* constructs. This brings malleability to a wide range of well-known OpenMP benchmarks that allow us to quantify the benefits when using unmodified applications.

At a high level, our proposed system works as follows. The OS-level elasticity manager continuously monitors the CPU utilization of non-malleable applications. When at least one non-malleable application consistently leaves idle cores, the kernel reassigns them to malleable programs. To do so, it communicates the target active thread count to the runtime system through a shared memory region, where changes are notified by OS via a signal delivered to the associated process. The runtime system is ultimately responsible for enforcing the desired active thread count. In addition, the

elasticity manager exploits thread packing to confine malleable and non-malleable programs in different core groups, so as to minimize interference between applications.

We implemented our elasticity manager as an OS scheduler extension in the Linux kernel v5.16.20. Specifically, this extension is bundled as a plugin of the PMCSched framework [1], which can be dynamically loaded in unmodified kernels (no patch required) via a customizable kernel module. The malleability extensions for OpenMP were incorporated into the GNU OpenMP runtime system that comes with GCC 11.2, also referred to as *libgomp*.

In the remainder of this section, we first describe the way in which the kernel-level elasticity manager interacts with the user-level runtime system. We then present the inner workings of the elasticity manager. Lastly, we explain how the runtime system enforces the necessary active thread count in parallel for OpenMP constructs.

2.1. OS-Runtime interface

Communication between the malleability-enabled runtime system and the OS kernel takes place through a per-application shared memory region. In our implementation, this region holds a simple two-field data structure. The first field, called `is_malleable`, is a flag that the runtime system sets to 1 to inform the OS that the application supports malleability. In our OpenMP runtime implementation, this flag is set to 1 when the program enters the first parallel section, although the runtime has the potential to selectively disable it when the application enters phases that are not amenable to malleability, such as explicit sequential regions. The second field (`target_threads`) is exclusively modified by the elasticity manager and is used to inform the runtime system of the maximum allowed number of active threads for the application. Every time this integer field is altered, the elasticity manager sends the SIGUSR1 signal to the application process. We opted to deliver that signal as it is one reserved for user-defined purposes in the POSIX standard. The runtime system must install the associated signal handler and carry out the necessary implementation-specific actions upon signal reception to enforce that maximum target thread count.

We should highlight that allowing an application to communicate with the OS kernel via shared memory to exchange scheduling-relevant information is inspired by the `schedctl()` system call present in the Solaris operating system [11]. This call returns a pointer to a scheduling-related data structure, allowing efficient bidirectional communication between user and kernel spaces without the need for additional system calls. To create the per-application memory region shared with the Linux kernel, we leverage the built-in support provided by PMCSched [1]. This allows us to seamlessly create the region by relying on the existing set of Linux system calls. To this end, the main thread must open the */proc/pmc/schedctl* special file –exported by PMCSched– and then pass the returned file descriptor to the `mmap()` system call. Internally, PMCSched associates the custom data structure with the corresponding process descriptor (Linux `task struct`), thus making it available to any PMCSched plugin, including our elasticity manager, described in the next section.

2.2. Elasticity manager

The elasticity manager (EM) is an OS scheduler extension that distributes the system's cores among malleable and non-malleable applications in the workload. It works cooperatively with the Linux process scheduler by dynamically adjusting CPU affinities and by communicating with the runtime system of each malleable application. EM reacts to key scheduling events, such as when a new process/thread is created, when a thread terminates or blocks, and when it becomes runnable again. Additionally, EM's code is activated periodically in a system-wide fashion to assign cores to the various applications on the system. EM was implemented as a plugin of the PMCSched tool [1]; the interface of a PMCSched plugin consists of a set of callbacks allowing to capture the aforementioned scheduling events [12]. The periodic activation of EM is also triggered by PMCSched.

EM maintains three global linked lists: *active* applications (processes with at least one runnable thread), *idle* applications (non-malleable programs with no runnable threads at this point), and *malleable* applications whose target thread count and affinity masks needs to be updated. The first two lists may require updating when a thread enters the system, becomes runnable, blocks, or exits. The third list is populated by the periodic system-wide core allocation algorithm described next, which is activated every `em_period` ms. Notably, `em_period` is a configurable parameter of EM.

Between executions of the core allocation algorithm, EM continuously monitors changes in threads' states (i.e., blocked and runnable) to keep track of the amount of time that each application spends with different runnable thread counts. For this purpose, it maintains a per-application *activity vector* – an array whose i-th entry stores the amount of time (in nanoseconds) the application spends with i runnable threads. Activity vectors of non-malleable applications are taken into consideration by EM to decide how to distribute cores among processes; these vectors are reset by the core allocation algorithm so as to start a new monitoring interval, which lasts a full `em_period`. For efficiency reasons, the activity vector is accompanied by a bitmask, where each bit indicates the validity of a particular entry. Maintaining this bitmask allows us to efficiently determine the application's usual thread count, and makes it possible to reset the activity vector without zero-filling all its entries (the entire bitmask is cleared instead). We should also highlight that EM makes changes in a process's activity vector only when one of its threads becomes inactive (blocks or terminates) or runnable again; these events are captured by leveraging specific callbacks provided by PMCsched. When the system is idle, these callbacks are not called, and the periodic core allocation algorithm is not engaged; therefore, EM's code is not invoked during idle system periods.

Alg. 1 depicts the pseudo-code of EM's periodic core allocation algorithm, which comprises two steps. In Step 1,

Algorithm 1 Core allocation algorithm

```
1:  function periodic_core_allocation(void)
2:      nr_remaining_cores = num_online_cpus()
    ▷ lists for malleable and non-malleable applications
3:      m_list=[] ; n_list=[]
    ▷ STEP 1: Determine application types and number of cores used by
    non-malleable applications
4:      for each app in active_list do
5:          if app.shared_region and app.shared_region.is_malleable then
6:              m_list.append(app)
7:          else
8:              nr_remaining_cores -= get_typical_thread_count(app)
9:              n_list.append(app)
10:         reset_activity_vector(app)
    ▷ STEP 2: Assign remaining cores to malleable applications
11:     nr_remaining_apps = len(m_list)
12:     if len(idle_apps) > 0 then
13:         nr_remaining_cores−−
    ▷ Ensure there is at least one core for each malleable application
14:     if nr_remaining_cores < len(m_list) then
15:         nr_remaining_cores = len(m_list)
16:     non_mall_cores = num_online_cpus() - nr_remaining_cores
17:     for each app in m_list do
18:         nr_fair_cores = max(1, ⌊nr_remaining_cores / nr_remaining_apps⌋)
19:         nr_remaining_apps−−
20:         nr_remaining_cores -= nr_fair_cores
21:         if app.shared_region.target_threads != nr_fair_cores then
22:             app.shared_region.target_threads = nr_fair_cores
23:             update_affinity_mask(app, nr_fair_cores, non_mall_cores)
24:             apps_to_update.append(app)
25:     if len(apps_to_update) > 0 then
26:         wake_up_kernel_thread()
```

EM traverses the list of active applications and divides them into two classes: malleable and non-malleable. To determine the application's class, EM attempts to obtain a reference to the structure stored in the memory region shared with the runtime system. If present, this structure is retrieved from the task descriptor of the process's group leader (the task structure from the main thread). The application is classified as malleable only if the shared memory region exists, and the current value of the `is_malleable` attribute is one (recall that the runtime system may dynamically alter this flag). Otherwise, the application is considered to be non-malleable. The loop in Step 1 also calculates the number of cores utilized by each non-malleable application, by invoking the `typical_thread_count()` auxiliary function. This function traverses the application's activity vector from higher to smaller thread counts, disregarding invalid entries, and returns the highest thread count whose entry stores a number no smaller than the `min_threshold` parameter. This configurable parameter establishes the minimum amount of CPU time a process must run within the `em_period` to be considered sufficiently CPU intensive. Note also that EM reserves one core for all idle applications. This substantially reduces the number of context switches in the event a thread from these applications suddenly becomes runnable. This is a common scenario in compute-intensive programs like those we used for evaluation (see Sec. 3).

Step 2 of the algorithm takes care of evenly distributing the remaining cores[1] (variable `nr_remaining_cores`) among malleable applications. To do so, it traverses the local list of malleable applications, and for each one it assigns a target thread count that matches its fair share of `nr_remaining_cores`. If the desired target thread count is different from the current one (i.e., the value stored in the shared memory region), EM updates the `target_threads` attribute in the shared structure. This step of the algorithm also takes care of determining the new CPU affinity mask of each malleable application, so that their threads are confined in specific sets of cores, different from those used by non-malleable applications. Note that in doing so, EM leverages thread-packing [13], as the total number of threads of malleable applications is typically greater than the number of cores assigned to it (see Sec. 2.3). Affinity masks are assigned in such a way that the `nr_remaining_cores` CPUs (cores) with the highest IDs are assigned to malleable programs. Therefore, all CPUs with IDs smaller than `nr_remaining_cores` are devoted to running non-malleable programs. Clearly, this implementation does not consider scenarios where the user explicitly reserves specific cores to run critical threads (e.g. a resource manager), or the fact that threads of non-malleable applications may be pinned to specific cores. As future work we plan to augment EM with a more sophisticated thread-to-core assignment policy that factors in these constraints.

Lastly, the algorithm activates a kernel thread to signal processes whose maximum thread count need adjusting, and to impose the designated per-application affinity masks to all of their threads. We should highlight that a kernel thread is required in this scenario, because the functions of the Linux kernel API that deliver signals and enforce affinity masks –which may trigger thread migrations– are blocking calls. Therefore, they must be invoked from *process context*, which is not the context type where the periodic core allocation algorithm runs (*interrupt context*).

2.3. Malleability in the OpenMP runtime system

The OpenMP runtime system already allows to request the utilization of different number of worker threads in the various parallel regions of a program. While this constitutes a form of malleability, its coarse granularity makes it impossible to react in our target time frame (a matter of milliseconds) in the vast majority of loop-based OpenMP programs. To achieve a finer-granularity and react as soon as possible when a thread change is requested from the kernel, we chose to implement the malleability control within loop-scheduling methods. As as proof of concept, we modified the `dynamic` and `guided` loop-scheduling methods. Note that the `static` loop-scheduling technique is not amenable to malleability. Indeed, in the *libgomp* implementation each worker thread invokes a single runtime call at the beginning of the loop, to get its full share of it.

To adopt malleability within the GNU OpenMP runtime system, we made a number of modifications, some of which affect environment variables. Specifically, we introduced two new environment variables: `GOMP_MALLEABLE` and `GOMP_MAX_THREADS`. These variables allow the user to enable or disable malleability and to indicate the max-

1. Exploring uneven core distribution among applications is out of the scope of this work. However, previous research [6] suggests that this would bring further throughput improvements.

imum number of threads the application can run with, respectively. In our experiments, we systematically set `GOMP_MAX_THREADS` to the total number of cores in the system, which is standard practice when exploiting malleability [6]. Note also that when malleability is enabled, the purpose of the standard `OMP_NUM_THREADS` environment variable was altered to allow specifying the initial number of *active* threads the application uses. Likewise, the `omp_set_num_threads()` OpenMP API function was also modified to allow establishing the same property. Regarding global settings, our modified runtime system automatically sets the default wait policy in synchronization primitives to the `passive` mode, just like in previous work [6]. By allowing worker threads to sleep instead of busy-wait on synchronization primitives (e.g., barriers), we effectively avoid wasting CPU cycles. This is paramount in oversubscription scenarios, which become apparent in multi-program settings, where EM grants a number of cores to the application that is often smaller that its total thread count.

To engage malleability within parallel loops, we employ a global a bit array, including an entry for each thread, to indicate whether it is *active* or *inactive*. Within a loop, active threads act as normal OpenMP worker threads, while inactive ones remain blocked inside the runtime calls associated with removing loop-iterations from the pool in the `dynamic` or `guided` methods. To determine the current chunk when a thread removes iterations from the pool in the `guided` method, we use the current number of active threads in the calculation rather than the total thread count. Note also that to enforce synchronization, we employ standard POSIX mutexes and condition variables. Within the runtime system implementation, the application's master thread takes care of initializing all these global resources. It also requests the creation of the per-application memory region shared with the EM (kernel space), and installs the signal handler for the `SIGUSR1` signal, which EM delivers when a change of the active thread count is in order.

Upon receiving the `SIGUSR1` signal, the runtime system reads the new target thread count value from shared memory, modifies the bit array of active threads accordingly, and wakes up the necessary threads (those in the active state). When a newly active thread wakes up, it proceeds to remove iterations from the pool and executes them. Should the number of active threads decrease upon signal reception, the threads that were recently forced to become inactive will block upon invocation of any runtime call for removing loop-iterations from the shared pool. Lastly, we should highlight that in our implementation all threads (active and inactive) must synchronize with each other in the implicit barrier present at the end of most parallel loops. To this end, when an active thread detects that all iterations for this parallel loop have been completed, it will awake all blocked (inactive) threads, ensuring they can arrive at the barrier.

3. Experimental evaluation

In this section, we begin by describing our experimental setup and introducing the applications and methodology employed in our experiments (Sec. 3.1). The detailed discussion of the results can be found in Sec. 3.2.

3.1. Experimental setup and methodology

Our evaluation was performed on a 16-core Intel Xeon Gold 5218 "Cascade Lake" processor, where cores run at 2.3Ghz. All cores feature two private cache levels (64KB L1 + 1MB L2) and share a 22MB last-level (L3) cache.

To assess the effectiveness of our proposal, we experimented with both POSIX threads (aka *pthreads*) and OpenMP programs. Specifically, we considered multi-threaded applications from different benchmark suites, encompassing NAS Parallel Benchmarks [14], PARSEC [15] and Rodinia [16]. In addition, we included `RNASeq` [17], an OpenMP-based RNA sequencing application, and two pthreads-based programs: `BLAST`, a bioinformatics application, and `FFTW3D` [2], a scientific benchmark performing the fast Fourier transform. For the NAS and PARSEC benchmark suites, we employed the `C` and `native` input sets, respectively. Given the exceedingly short execution time of Rodinia benchmarks when using their default command-line settings, we opted to use the alternative input sets employed by previous research [1], [18]. The exact command-line parameters we used can be found in prior work [18].

All OpenMP programs (unmodified) were compiled with a customized GCC 11.2 compiler [1], which contains a simple patch that just alters the default loop-scheduling method from `static` to `runtime`. This allows the user to enforce the desired loop-scheduling method and other parameters (via environment variables) for all parallel loops not including the `schedule` clause; this is the case of the vast majority of the loops in the considered applications. We should also highlight that application loops where the programmer explicitly established the `dynamic`, `guided`, or `runtime` scheduling methods also make use of our malleabillity extensions, which are globally enabled when the `GOMP_MALLEABLE` environment variable is set to 1.

In building the workloads, we first proceeded to identify parallel applications that exhibit execution phases with limited thread-level parallelism (TLP). These applications often leave idle cores, allowing our elasticity manager to opportunistically increase the thread count of other (malleable) programs within the workload, thus maximizing the processor's utilization. To identify applications of this kind, we built a PMCSched plugin that efficiently monitors the amount of time an application spends with different number of runnable (i.e., non-blocked) threads. Fig. 2 shows the fraction over the total execution time that different programs spend with various runnable thread counts. In this experiment, we launched the programs with 8 threads, as this is the thread count used to run non-malleable programs in our workloads. The results reveal that some programs spend more than 40% of its execution time running with just a single runnable thread. This is the case of one OpenMP application (`pathfinder`) and three pthreads-based programs: `blackscholes`, `BLAST` and `FFTW3D`. The limited

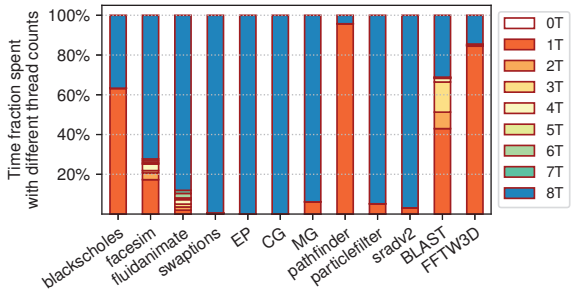


Figure 2: Time fraction spent with different runnable thread counts in various applications.

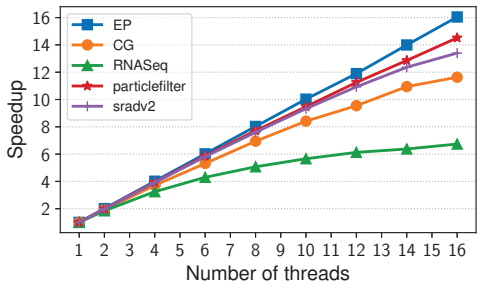


Figure 3: Scalability of several OpenMP programs in our experimental platform.

TLP in this context stems from the presence of long sequential phases (predominantly in initialization code), coarse-grained critical sections, or severe load imbalance among worker threads. By contrast, other applications, like `EP` or `particlefilter`, consistently run with the maximum thread count during most of their execution time.

To find OpenMP programs that potentially benefit from malleability in the presence of idle cores, we ran the various analyzed applications with different thread counts (from 1 to 16). Fig. 3 shows the speedup of a number of OpenMP programs that experience diverse performance improvements as the thread/core count increases.

In constructing the workloads, we selected 12 programs from the full set of benchmarks considered. Among these programs, four exhibit limited TLP (`blackscholes`, `BLAST`, `FFTW3D` and `pathfinder`), while the rest are OpenMP programs that experience significant benefit when increasing their thread counts. Table 1 presents the 24 random program mixes employed in our experiments. Each workload includes a non-malleable (N) program with limited TLP, and a malleable (M) OpenMP one. Both programs are launched with 8 threads each on our 16-core experimental system. Note also that OpenMP programs run with the best-performing loop-scheduling method among those supporting malleability (`guided` or `dynamic`).

We ran each workload in three different scenarios. In the first one, referred to as *Stock-Linux*, no malleability OS extensions are loaded, and applications use the unmodified version of the OpenMP runtime system. Under these circumstances both applications use a fixed thread count – 8 threads each. Moreover, the threads of the N-program are mapped to CPUs 0-7 via user-enforced affinities, while threads of the other application run on the remaining cores (CPUs 8-15). In the second scenario –denoted as *Oversubscription*– we run the M-program with 16 threads, whereas the N-program still uses 8 threads; affinities are only imposed to threads of the N-program, which are pinned to CPUs 0-7. This scenario does not leverage malleability, but exploits the fact that the N-program temporarily leaves idle cores, which can be used by the additional threads (one thread per core) used by the M-program. We considered this scenario to assess (1) how effectively the stock OS scheduler together with the unmodified runtime system deal with idle cores, and (2) what is the impact of leveraging oversubscription in the performance of the non-malleable program. In the last scenario, we enable our kernel-level elasticity manager (EM), and activate the malleabillity support in our modified OpenMP runtime system. No user-enforced CPU affinities are used in this case, and, as opposed to the remaining scenarios, the active thread count of the OpenMP (malleable) application is dynamically adjusted based on the number of idle cores left by the other multithreaded program. Notably, we conducted a sensitivity study (omitted due to space constraints) to determine a suitable choice of the `em_period` and `min_threshold` configurable parameters of EM in our platform. Based on the results, we opted to set these parameters to 100ms and 2ms, respectively.

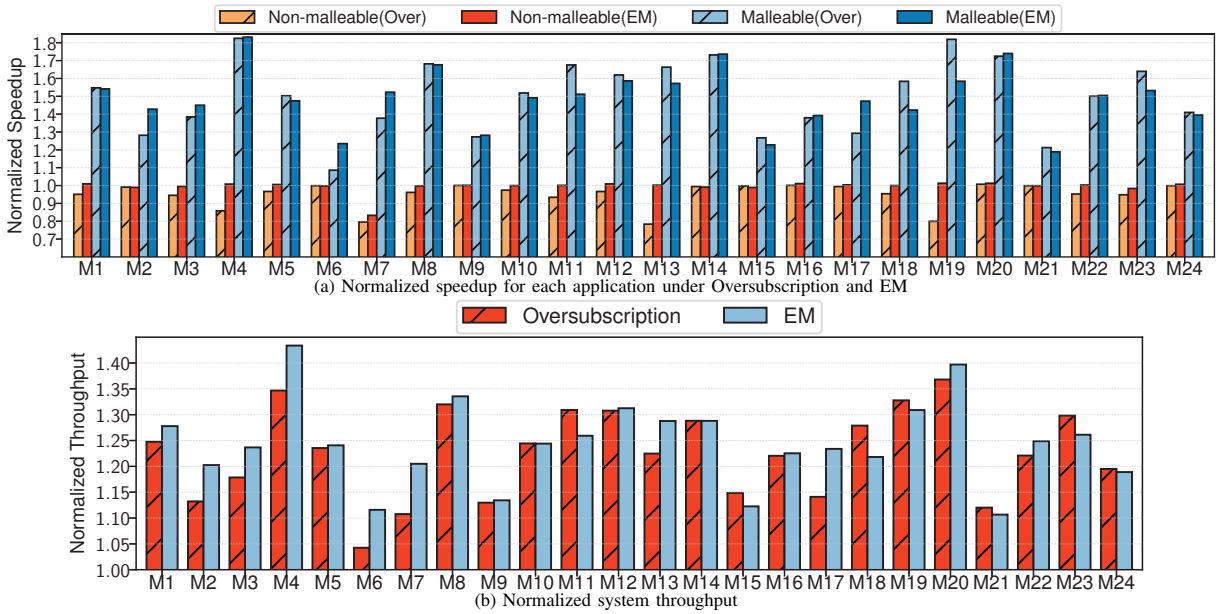
For each workload and scenario we measure the application performance (completion time) and system throughput (STP metric [19]). It is worth highlighting that running each program in the workload just once until completion, provides misleading performance results, especially when the N-program is shorter than the M-program. In that case, the M-program automatically experiences a performance boost under the elasticity manager, simply because threads of this program effectively populate all the platform's cores whe the N-program terminates. For accurate assessment of the performance gains provided by the elasticity manager when both applications run together the whole time, we employ a similar method than in prior work [20], so as to keep the system's load uniform throughout an experiment. To elaborate, we ensure that all applications in the mix commence simultaneously. When one of them finishes its execution, it is consistently restarted until the program with the longest runtime completes three iterations. Then, we measure system throughput based on the geometric mean of the completion times observed for each application.

3.2. Discussion of experimental results

Fig. 4a shows the relative performance that each application in the workload obtains under EM and Oversubscription vs. Stock Linux. Clearly, EM delivers substantial performance gains for M-programs (up to 83%, for workload M4). This is due to the effective utilization of idle cores left by N-programs, all with limited TLP. In most cases this is accomplished without significantly degrading the performance of the N-program. For the vast majority of the workloads, the performance of the non-malleable applications remains within a 1.7% range of that obtained with Stock-Linux. The substantial acceleration of M-programs across the board also leads to a considerable increase in the system throughput,

TABLE 1: Each table column M_i depicts the composition of the i-th workload used in our experiments. The (N) suffix in the application name (see row labels) indicate that the application is non-malleable. Conversely, malleable programs are listed with the (M) suffix. When a program is included in a workload, a black dot is displayed; otherwise, the associated cell is blank.

	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	M11	M12	M13	M14	M15	M16	M17	M18	M19	M20	M21	M22	M23	M24
blackscholes(N)	•				•				•				•				•				•			
BLAST(N)		•				•				•				•				•				•		
FFTW3D(N)			•				•				•				•				•				•	
pathfinder(N)				•				•				•				•				•				•
CG(M)						•										•								
EP(M)				•									•						•					
heartwall(M)					•					•													•	
leukocyte(M)			•									•						•						
particlefilter(M)	•										•									•		•		
RNASeq(M)								•							•						•			
sradv2(M)		•					•										•							
streamcluster(M)									•					•										•



(a) Normalized speedup for each application under Oversubscription and EM

(b) Normalized system throughput

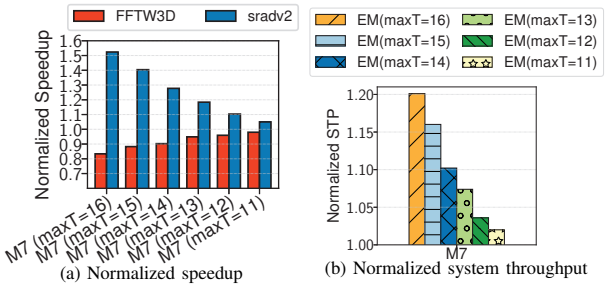
Figure 4: (a)Relative speedup for each application in the workload under Oversubscription and EM vs. Stock-Linux, and (b) normalized throughput for the various program mixes.

as Fig. 4b reveals. Compared to Stock-Linux, EM improves throughput by up to 43%, and by 24.5% on average.

Now we turn our attention to the results under Oversubscription (*Over*), and compare them with the EM counterparts. Running the malleable OpenMP application with 16 threads throughout the execution (what happens under *Over*), makes it possible in many workloads to achieve higher performance for this program w.r.t. EM. Despite the issues and overheads stemming from oversubscription [9], [21], the loop-scheduling methods used in our experiments (dynamic and guided) are key in improving the performance of the M-program. In this context, these methods automatically assign more work to threads running on a dedicated core than to threads time-sharing a core with other threads. This partially mitigates the imbalance arising from oversubscription. Unfortunately, the acceleration of the malleable OpenMP application under *Over* usually comes at the expense of degrading the performance of the N-program. Take for instance the results of M19 workload in Fig. 4a, where a substantial performance gain is accomplished for the M-program in exchange for a 21% performance degradation of the N-program. This stems from the fact that, under *Over*, threads of the N-program often have to time-share a core with threads from the other application; this causes uneven progress among threads[2], thus deteriorating the N-program's performance. All in all, we conclude that leveraging oversubscription may bring additional throughput improvements in some cases (e.g. see results for M8, M11, or M19 in Fig. 4b), but it also negatively (and consistently) impacts the performance of the non-malleable program. Conversely, EM guarantees lower overheads of non-malleable programs. After all, EM strives to avoid oversubscription when exploiting malleability dynamically.

The results provided by EM for the M7 and M23 workloads –both including the `FFTW3D` (non-malleable) application– are also of special attention. Specifically, as shown in Fig. 4a, the acceleration of the malleable partner

2. The N-programs we used do not exploit dynamic load balancing among threads; instead, threads are assigned a fixed amount of work upon creation. Hence, oversubscription gives rise to uneven thread progress.



(a) Normalized speedup (b) Normalized system throughput

Figure 5: Limiting the active thread count of the malleable application of workload M7 under EM.

comes at the expense of noticeable performance degradation of `FFTW3D`, being particularly noteworthy the 16.6% performance penalty in M7. After a thorough analysis of these scenarios, we determined that this degradation is caused by severe bandwidth contention on the platform. Essentially, `FFTW3D` is a highly bandwidth-intensive program, and when it shares the system with another bandwidth-intensive application, the system's bandwidth saturation point is reached. This situation is further aggravated by the effect of the elasticity manager. Granted, increasing the number of threads of a bandwidth-intensive program causes its bandwidth consumption to grow [22], thereby exacerbating the bandwidth contention problem. This especially deteriorates the performance of the non-malleable program, which does not enjoy a dynamic increase in its thread count.

We found that this issue can be addressed by limiting the number of threads assigned to the malleable application by our elasticity manager. As an illustrative example, Fig 5 shows the impact on application performance and system throughput that comes from imposing an upper limit to the number of threads assigned to the malleable application in workload M7, ranging from 16 threads (maxT=16) to 11 threads (maxT=11). Clearly, setting a lower limit on the number of threads utilized by the malleable program (`sradv2`) reduces the performance degradation of the non-malleable one (`FFTW3D`). However, this also comes at the expense of a consistent reduction of both `sradv2`'s performance and the system throughput. Moreover, throughput gains reach a peak value when no capping is applied (16 threads maximum, matching the total core count). Therefore, this experiment underscores that in specific cases a trade-off must be made to reap throughput benefits without deteriorating the performance of non-malleable programs. More importantly, memory bandwidth contention –in addition to the idle core count– should be taken into consideration when leveraging dynamic malleability to avoid performance degradation. Incorporating contention awareness into our elasticity manager constitutes a promising research avenue.

4. Related work

Elasticity is a key feature in the cloud, as it allows providers to offer scalable resources on demand while maximizing profits [8]. Notably, existing CPU elasticity mechanisms operate at higher time frames and coarser granularity than our node-level OS-oriented approach. For example, virtual machines or containers may be migrated to another node in response to a request to increase CPU resources [8], [9], which can take several seconds.

Huang et al. [9] propose an OS-level elasticity strategy that exploits various optimizations in the synchronization primitives. While this approach is also based on rapid scheduling of runnable threads during short idle periods, it focuses on the efficient handling of CPU oversubscription, rather than on the dynamic adaptation of thread counts in malleable programs. Moreover, unlike our proposal, Huang's work [9] does not propose any kind of OS-runtime cooperation method, or explicitly supports malleable applications. Other works not leveraging malleability either [1], [21], [23] acknowledge the importance of the coordination and information exchange between the OS and the application. For example, [21], [23] underscore the potential of making the OS scheduler aware of threads that are busy waiting in synchronization primitives by issuing notifications from the application level. Saez et al. [23] exploit this OS-runtime interaction in an actual OS scheduler implementation. Our proposal could be augmented so as to leverage information on busy waiting threads to improve effective CPU usage.

Malleability has been widely exploited in HPC in a more general form, this is, by performing dynamic resource reallocation across nodes, including adjustments in the number of worker processes/threads, and the necessary data redistribution [24]. By contrast, we looked at a more specific type of malleability that leverages active thread adjustments in a single node and is orchestrated by the OS. While our proposal focuses on describing the OS-runtime interaction and uses malleable OpenMP parallel for work-sharing constructs as a proof of concept, the OpenMP tasking model also lends itself (even more) to dynamic thread count adjustments. A wide range of techniques that leverage malleability, such as free-agent threads [25]–[27], have been proposed for OpenMP, most of which operate entirely at the runtime system level or in the cluster/job scheduler (e.g., Slurm) [5], [28]. These techniques could also benefit from OS-runtime cooperation and are exciting avenues for future research.

In elastic OpenMP [29], authors propose a mechanism to provide elasticity support for OpenMP applications that make the dynamic provisioning of cloud resources possible. This proposal supports the dynamic adjustment of resources (vCPUs) and a set of additional routines to enable the configuration of the elastic execution. Our OpenMP prototype has also adapted the inner workings of the parallel construct, but the problems we have addressed are orthogonal.

Other works also applied dynamic thread adjustment under colocation. Cho et al. introduced NuPoCo [6], a runtime-system level solution that strives to efficiently utilize both the CPU and memory controller when multiple parallel applications share the system. Our proposal conversely places its emphasis on the exploitation of idle cycles left by unmodified applications through elasticity support within the OS. More importantly, unlike our approach, NuPoCo requires that every application in the workload use a customized runtime system, thus being incompatible with legacy single-

threaded and multithreaded software.

5. Conclusions and Future Work

In this work, we have proposed an OS-level elasticity mechanism that strives to maximize CPU utilization and system throughput, when running workloads consisting of a mix of non-malleable programs and malleable applications (i.e., with the ability to vary their thread count). To accomplish its goals, our proposal leverages idle cores left by non-malleable applications (even for short periods of time), and exploits synergistic cooperation between the OS kernel and the runtime system for the dynamic adjustment of active threads in malleable applications.

To demonstrate the effectiveness of the proposed OS extension, we have implemented it in the Linux kernel. The associated OS-level support is bundled as a loadable kernel module that can be loaded in unmodified (unpatched) Linux kernels. We also augmented the GNU OpenMP runtime system to provide a proof-of-concept malleability support at the runtime system level that exploits the OS-runtime interaction required by our proposal. This malleability support can be utilized by a wide range of unmodified loop-based parallel applications. Our experiments reveal that our proposal delivers improved system throughput, while having minimal impact in the performance of non-malleable programs. This is accomplished thanks to the opportunistic utilization of idle cores to accelerate malleable applications.

As for future work, we plan on augmenting our proposal to explicitly deal with contention on shared resources among cores (e.g., shared last-level-cache). Providing the necessary support to deliver benefits to applications beyond compute-intensive applications and loop-based OpenMP programs is also an interesting avenue for future work.

Acknowledgments

This work has been supported by the Spanish MCIN under Grant PID2021-126576NB-I00, funded also by MCIN/AEI/10.13039/501100011033 and by "ERDF A way of making Europe".

References

[1] C. Bilbao, J. C. Saez, and M. Prieto-Matias, "Flexible system software scheduling for asymmetric multicore systems with PMCSched: A case for Intel Alder Lake," *Concurrency and Computation: Practice and Experience*, p. e7814, 2023.

[2] M. Annavaram *et al.*, "Mitigating Amdahl's Law through EPI Throttling," in *Proc. of ISCA '05*, 2005, pp. 298–309.

[3] M. A. Suleman, M. K. Qureshi, and Y. N. Patt, "Feedback-driven threading: Power-efficient and high-performance execution of multi-threaded workloads on cmps," in *Proc. of ASPLOS '08*, 2008, p. 277–286.

[4] F. V. Zacarias *et al.*, "Intelligent colocation of HPC workloads," *Journal of Parallel and Distributed Computing*, vol. 151, pp. 125–137, 2021.

[5] D. Álvarez, K. Sala, and V. Beltran, "nos-v: Co-executing hpc applications using system-wide task scheduling," https://arxiv.org/abs/2204.10768, 2022.

[6] Y. Cho, C. A. C. Guzman, and B. Egger, "Maximizing system utilization via parallelism management for co-located parallel applications," in *PACT*, ser. In Proc. of PACT '18, 2018.

[7] T. Harris, M. Maas, and V. J. Marathe, "Callisto: Co-scheduling parallel runtime systems," in *In Proc. of EuroSys '14*, 2014.

[8] Y. Al-Dhuraibi *et al.*, "Elasticity in cloud computing: State of the art and research challenges," *IEEE Transactions on Services Computing*, vol. 11, no. 2, pp. 430–447, 2018.

[9] H. Huang *et al.*, "Towards exploiting cpu elasticity via efficient thread oversubscription," ser. In Proc. of HPDC'21, 2021, p. 215–226.

[10] G. Galante and L. C. Erpen De Bona, "A programming-level approach for elasticizing parallel scientific applications," *Journal of Systems and Software*, vol. 110, pp. 239–252, 2015.

[11] J. Mauro and R. McDougall, *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture, Second Edition*. Prentice Hall, 2006.

[12] C. Bilbao and J. C. Saez, "PMCSched website," https://github.com/Zildj1an/pmcsched-website, GitHub, 2022, accessed: 2023-10-23.

[13] J. Park *et al.*, "Hypart: A hybrid technique for practical memory bandwidth partitioning on commodity servers," in *Proc. of PACT '18*, 2018, pp. 5:1–5:14.

[14] D. H. Bailey, E. Barszcz, and J. T. Barton et al., "The NAS parallel benchmarks—summary and preliminary results," in *Supercomputing '91*, 1991, pp. 158–165.

[15] C. Bienia *et al.*, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proc. of PACT'08*, 2008.

[16] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc of IISWC '09*, 2009, pp. 44–54.

[17] H. Chitsaz *et al.*, "A partition function algorithm for interacting nucleic acid strands," *Bioinformatics*, vol. 25, no. 12, pp. i365–i373, 2009.

[18] T. Suzuki, A. Nukada, and S. Matsuoka, "Efficient execution of multiple cuda applications using transparent suspend, resume and migration," in *Proc. of Euro-Par '15*, 2015.

[19] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE Micro*, vol. 28, pp. 42–53, 2008.

[20] D. Shelepov *et al.*, "HASS: a Scheduler for Heterogeneous Multicore Systems," *Oper. Syst. Review*, vol. 43, no. 2, pp. 66–75, 2009.

[21] J. H. M. Korndörfer *et al.*, "How do os and application schedulers interact? an investigation with multithreaded applications," in *Proc. of Euro-Par '23*, 2023, pp. 214–228.

[22] M. A. Suleman, M. K. Qureshi, and Y. N. Patt, "Feedback-Driven Threading: Power-Efficient and High-Performance Execution of Multi-Threaded Workloads on CMPs," in *Proc. of ASPLOS '08*, 2008, p. 277–286.

[23] J. C. Saez *et al.*, "Operating system support for mitigating software scalability bottlenecks on asymmetric multicore processors," in *Proc. of CF' 10*, 2010, pp. 31–40.

[24] J. I. Aliaga *et al.*, "A survey on malleability solutions for high-performance distributed computing," *Applied Sciences*, no. 10, 2022.

[25] J. Criado *et al.*, "Exploiting openmp malleability with free agent threads and dlb," in *ISC High Performance 2022 International Workshops*, 2022.

[26] ——, "Role-shifting threads: Increasing openmp malleability to address load imbalance at mpi and openmp," *The International Journal of High Performance Computing Applications (In press)*, 2023.

[27] OpenMP Architecture Review Board, "Openmp technical report 12: Version 6.0 preview 2," https://www.openmp.org/wp-content/uploads/openmp-TR12.pdf, 2023, Accessed: 2023-11-12.

[28] M. D'Amico *et al.*, "Drom: Enabling efficient and effortless malleability for resource managers," in *Proc. of ICPP Workshops '18*, 2018.

[29] G. Galante and R. da Rosa Righi, "Adaptive parallel applications: From shared memory architectures to fog computing (2002-2022)," *Cluster Computing*, vol. 25, no. 6, p. 4439–4461, 2022.