PrioRAT: Criticality-Driven Prioritization Inside the On-Chip Memory Hierarchy

Vladimir Dimić[1,2(✉)][0000−0002−9176−7674], Miquel Moretó[1,2][0000−0002−9848−8758], Marc Casas[1,2][0000−0003−4564−2093], and Mateo Valero[1,2][0000−0003−2917−2482]

[1] Barcelona Supercomputing Center (BSC), Barcelona, Spain
{vladimir.dimic,miquel.moreto,marc.casas,mateo.valero}@bsc.es
[2] Universitat Politcnica de Catalunya (UPC), Barcelona, Spain

Abstract. The ever-increasing gap between the processor and main memory speeds requires careful utilization of the limited memory link. This is additionally emphasized for the case of memory-bound applications. Prioritization of memory requests in the memory controller is one of the approaches to improve performance of such codes. However, current designs do not consider high-level information about parallel applications. In this paper, we propose a holistic approach to this problem, where the runtime system-level knowledge is made available in hardware. Processor exploits this information to better prioritize memory requests, while introducing negligible hardware cost. Our design is based on the notion of critical path in the execution of a parallel code. The critical tasks are accelerated by prioritizing their memory requests within the on-chip memory hierarchy. As a result, we reduce the critical path and improve the overall performance up to 1.19× compared to the baseline systems.

1 Introduction

The growing gap between the processor and memory speeds, often referred to as the *Memory Wall* [39], has been one of the important factors driving the design of modern computing systems. In the last decades, DRAM chips have become more complex, introducing multiple levels of parallelism to allow for a higher bandwidth between the processor and the memory. On the processor side, several components, such as caches and prefetchers, serve to provide higher effective bandwidth and reduced latency between the cores and the memory subsystem. The interface between the processor and the main memory, the memory controller, is also an important point for optimization.

Early memory controller designs adopted simple request ordering algorithms from queuing theory, such as *first come-first served*, which was further improved to take into account the *row buffer* locality [32]. Multi-core processors introduce new challenges to memory request scheduling. It is necessary to take into account the priority of threads, avoid starvation and ensure forward-progress of all co-running threads [23, 25, 26, 27]. However, these proposals lack the awareness of the global impact of prioritization decisions on the execution of parallel applications. The

Dimic, V. [et al.]. PrioRAT: criticality-driven prioritization inside the on-chip memory hierarchy. A: International European Conference on Parallel and Distributed Computing. "Euro-Par 2021: Parallel Processing 27th International Conference on Parallel and Distributed Computing: Lisbon, Portugal, September 1–3, 2021: proceedings". Springer Nature, 2021, p. 599-615. ISBN 978-3-030-85665-6. The final authenticated version is available online at https://doi.org/10.1007/978-3-030-85665-6_37

relatively short-term knowledge available by observing hardware-level behavior is not always enough to achieve the best performance improvements.

Taking a look at the software level reveals that modern parallel computer systems are becoming more difficult to program due to their increasing complexity. Modern programming models aid a programmer in creating well-performing parallel applications by offering simple ways to describe parallel constructs. For example, task-based parallel programming models introduce the notion of a *task* as a sequential part of the application that can run simultaneously with other tasks [6, 16, 22]. A programmer specifies these tasks and their dependencies using pre-defined annotations. The correct execution and synchronization of the tasks are handled by the runtime system library. Thus, the runtime system contains by design the information about the parallel code and the underlying hardware, and therefore can serve as an interface between these two layers. The usefulness of runtime system-level information in the context of HPC applications and hardware has been extensively studied [1, 4, 5, 7, 10, 11, 19, 24, 38].

In this paper, we further explore the opportunities to exploit the high-level information about a parallel application inside the hardware. In particular, we focus on the notion of critical paths in the context of task-parallel codes. We define *critical tasks* as the tasks belonging to the critical path of the execution. Following the definition of the critical path, reducing the duration of these tasks reduces the execution time of the whole parallel code. Previous works have exploited task criticality to improve scheduling on heterogeneous systems [7] as well as for power management [5]. These proposals, however, do not target generic chip-multiprocessors as they depend on asymmetric processor design and voltage-frequency scaling, respectively. In addition, they focus only on runtime systems and core designs, without targeting the memory hierarchy.

To overcome these shortcomings, we design PrioRAT, a general solution for all modern chip-multiprocessors. We follow a holistic approach where the runtime system knowledge is used in hardware to drive the prioritization algorithm. Specifically, we exploit the notion of task criticality, motivated by the discussion in the previous paragraph. During the execution, the runtime system computes tasks' criticality and provides it to the underlying hardware. Then, on-chip hardware resources make use of this information to prioritize the memory requests coming from the critical tasks. As a result, the critical tasks have their memory requests served faster, which reduces their duration and, thus, improves the performance of the whole parallel application by reducing the length of the critical path.

This paper makes the following contributions:

- We extend scheduling algorithms inside the shared on-chip components to consider memory request criticality. Task criticality is estimated by the runtime system using existing methods and is forwarded to the hardware via the well-supported memory mapped registers, which does not require changes to the processor's ISA.
- We evaluate the performance of PrioRAT on a cycle-accurate microarchitectural simulator using a set of characteristic workloads from the high

performance computing (HPC) domain. PrioRAT outperforms the baseline system without prioritization by up to 1.19× in terms of execution time.

The remaining of this document is structured as follows. Section 2 provides the context for our work and introduces the intuition behind memory request prioritization. Section 3 explains our proposal in detail. Section 4 describes the experimental setup, while Section 5 presents the results of the evaluation. Finally, Section 6 describes related work and Section 7 concludes this paper.

2 Background and Motivation

Since the introduction of the first multi-core processor at the very beginning of this century, the importance producing well-performing parallel codes has been recognized. Many programming models have been designed to ease the development process of parallel applications [6, 16, 17, 22, 28]. Some of these programming models are based on task-based parallelization. A way to improve the performance of a task-parallel application is to identify the critical path in the execution and reduce its duration. There are many works devoted to identification [3, 12, 20] and acceleration of the critical path [5, 7, 8, 9, 14, 36].

On the hardware level, the increasing gap between processor and memory speeds warrants giving a special attention to improving the utilization of the main memory resources. Some previous works propose reordering DRAM commands to achieve higher throughput [29, 32, 37]. However, they do not target multi-core processors. Other solutions focus on ensuring fair share of memory resources among all the threads [25, 26, 27]. Another family of scheduling algorithms considers the criticality of each memory request [15]. These proposals try to improve the performance of each thread independently by accelerating requests that have most negative impact on the execution time.

However, such designs do not consider the high-level notion of the critical path at the application level. In certain parallel codes, it is possible to sacrifice the performance of non-critical tasks by giving priority to critical tasks in order to reduce the critical path and, thus, the overall execution time.

To illustrate the effects of critical task prioritization, we develop a synthetic application that performs a strided access to an array. The stride is a configurable parameter used to indirectly tune the pressure on the caches and main memory by controlling the reuse of the accessed cache lines. The application is split into tasks and each task accesses its portion of the input array. Tasks are split in two groups: (i) critical tasks, which are artificially serialized to simulate a critical path, and (ii) non-critical tasks, which can run in parallel with other tasks.

We configure the benchmark to run with 24 critical and 150 non-critical tasks, each accessing an array of 256 KB with a stride of 16. This is equivalent to one access per cache line and achieves the highest memory contention in our simulated environment. We simulate two executions of the benchmark on a cycle-accurate simulator modeling a 16-core processor with a three-level cache hierarchy and main memory[3]. The first run is performed on a baseline system

[3] Section 4 describes the experimental setup in detail.

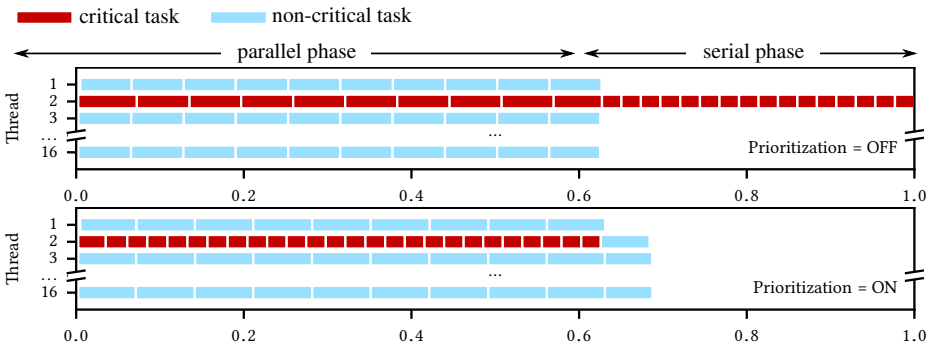

Fig. 1: Execution traces of the synthetic benchmark in the baseline configuration (top) and prioritized configuration (bottom).

without prioritization, while in the second execution, we prioritize the memory requests issued by the critical tasks. Figure 1 shows the traces of these two executions. Each trace displays the tasks and their duration during the execution (x-axis) and the thread where they execute (y-axis). Time is normalized to the execution on the baseline configuration and both traces have the same time scale. Critical and non-critical tasks are colored differently.

In the baseline configuration (see Figure 1, top), we observe the serialization of critical tasks and its negative impact on the total execution time – more than 30% of the execution time is spent running only serial code. We also note that during the parallel phase, the critical tasks execute slower compared to the last third of execution. This is a result of the high memory contention caused by the concurrent execution of many tasks in the parallel phase.

The bottom part of Figure 1 shows the trace when memory requests issued by critical tasks are given priority in the shared on-chip resources. The effects of the prioritization are clearly manifested through the reduced duration of the critical tasks. The non-critical tasks execute slower than in the baseline configuration, but the whole application finishes faster because the critical path is reduced.

This example clearly demonstrates the importance of having a high-level notion of the application within the hardware as it enables better decisions at hardware level. With these conclusions in mind, we develop PrioRAT, a solution that exploits the runtime system information about a task-based parallel code to guide prioritization of memory requests inside the on-chip memory hierarchy.

3 PrioRAT: Criticality-Driven Prioritization within the On-Chip Memory Hierarchy

PrioRAT is a holistic approach to prioritization in the on-chip memory hierarchy driven by the application-specific information available at the runtime system level. The runtime system library detects a critical path in an execution of a parallel application and marks the tasks belonging to the critical path as *critical*.

In hardware, we add a bit to each request to signal its criticality. The queues in the on-chip memory hierarchy are extended to prioritize critical requests (also called *prioritized requests*). Depending on the way the task criticality is exploited, we define two PrioRAT configurations:

- PR.static is a configuration where certain cores are configured to always issue prioritized memory requests. The runtime system is aware of the cores' configuration and schedules critical tasks onto prioritized cores.
- PR.dynamic is an approach where the runtime system controls whether each core issues prioritized requests depending on the criticality of a task scheduled on a given core. This approach requires an interface between the processor and the runtime system to enable switching of the configuration of the core.

In both configurations, the shared resources in the on-chip memory hierarchy, i.e., the last-level cache (LLC) and the memory controller, and the interconnect are extended to handle the prioritization of requests within their queues. In the following sections, we explain in detail the modifications to all relevant components necessary to implement PrioRAT.

3.1 Programming Model and Runtime System Support

We build PrioRAT on top of the existing parallel task-based programming model, OpenMP [28]. OpenMP is a directive-based programming model, where a programmer defines units of parallelism, such as tasks and loop iterations and defines dependencies between tasks. The runtime system library handles the scheduling of the defined tasks and loops on a multi-threaded machine. Internally, the runtime system tracks tasks using a Task Dependency Graph (TDG), which is constructed following the programmer-provided dependency information and implicit synchronization primitives. We use Nanos++ [2] as the runtime system that supports OpenMP-compatible task-parallel applications.

Inside Nanos++, we exploit an existing algorithm, called *bottom-level*, that analyzes the TDG and identifies a critical path of the execution by observing the distance of each task from the bottom of the TDG. This distance is updated on every insertion of a new task into the TDG by traversing the TDG from the bottom to the root. After the update, the root node stores the length of the longest path to the bottom level. The algorithm determines the critical path by following the longest path(s) along the way from the root to the bottom level. There may be several paths that satisfy this condition. In that case, the runtime system exposes to the user two configuration options: (i) choosing one random such path as the critical path, or (ii) marking all such paths as critical. The second option may result in more critical tasks compared to the first one.

The task criticality information is used by the runtime system to enable memory request prioritization inside the hardware. Depending on the PrioRAT configuration, the runtime system interacts differently with the processor, which is illustrated at the bottom of Figure 2.

PR.static is a static configuration where each core is set up to either always or never issue prioritized requests. The processor configuration is exposed to

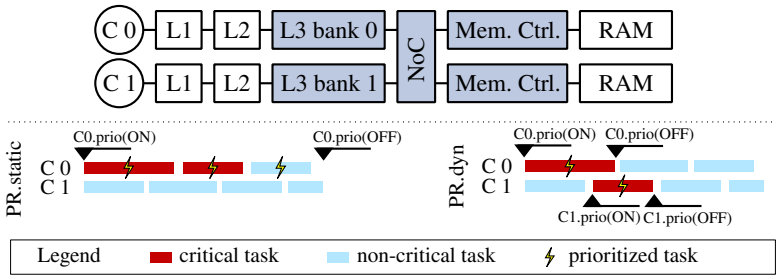

Fig. 2: Top: Overview of a dual-core system implementing PrioRAT. Colored components employ priority queues to prioritize memory requests. Bottom: Examples of executions of two PrioRAT configurations.

the runtime system via command line arguments or environmental variables. The scheduler within the runtime system assigns tasks to the cores taking into account task's criticality and whether a core issues prioritized requests. Critical tasks are preferentially scheduled on prioritized cores, while non-critical tasks can be scheduled on any core, depending on their availability.

In *PR.dynamic*, the task criticality information is provided to the processor cores by the runtime system using memory-mapped registers. Before each task begins execution, the runtime system signals to the core the criticality of that task. A core that executes a critical task issues high priority memory requests.

3.2 Hardware Extensions

This section introduces the micro-architectural extensions necessary to implement PrioRAT. The top of Figure 2 shows an overview of a system implementing PrioRAT, consisting of a dual-core processor connected to main memory. The hardware extensions for our proposal implement the following functionalities: (i) indicate to a core whether to prioritize memory requests of the currently-running task, (ii) flag memory requests as critical or non-critical and carry that information through the memory hierarchy, and (iii) prioritize requests in the memory hierarchy. We explain each feature in detail in the remaining of this section. As mentioned before, both PrioRAT configurations rely on exactly the same hardware extensions.

Awareness of Task Criticality in the Core. We add to each core a memory-mapped register, called *crit.reg*, which designates the priority of the memory requests issued by the currently-running task. In our implementation, the size of this register is one bit. In *PR.static*, each core is configured to always issue either high-priority or low-priority requests, i.e., the value of this register is constant for the duration of the application. It is the responsibility of the runtime system to carefully schedule tasks to utilize statically defined prioritization of the corresponding memory requests. In *PR.dynamic*, the runtime system updates

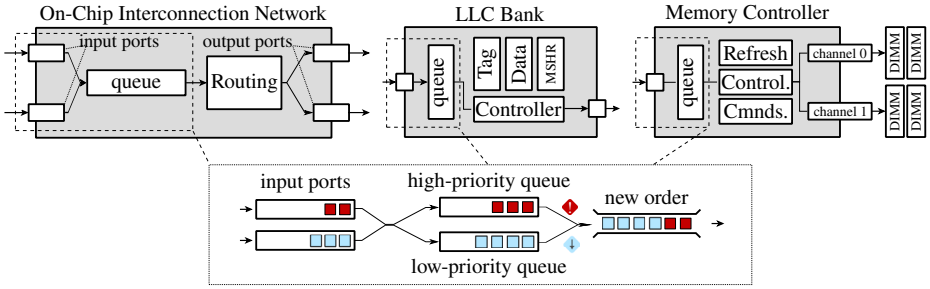

Fig. 3: Request prioritization inside shared on-chip components.

the value stored in *crit.reg* to hold the criticality of the task scheduled to run on the corresponding core. This is illustrated in Figure 2 using runtime system functions `Ci.prio(ON)` and `Ci.prio(OFF)`, which switch on and off the request prioritization for a given core `i`, respectively.

Assigning a Priority to Memory Requests. When issuing a request to the L1 cache, the core flags the request with criticality information stored in the *crit.reg*. This information is carried with the request along its way through the memory hierarchy. Since we model priority with two values, we require only one bit of information to be attached to each request. This adds a negligible overhead to the existing wires and logic in the on-chip memory hierarchy.

Memory Request Prioritization. To implement request prioritization within the shared on-chip resources, we extend the queues that hold incoming requests before they get processed. PrioRAT implements a double-queue design, where each queue holds either high or low priority requests, as shown in Figure 3. When selecting the next request for processing, the requests in the high-priority queue are preferred over the request from the low-priority queue. In the case of interconnection network and the LLC, the prioritization happens in the queue that holds incoming requests.

Memory controllers, on the other hand, already implement algorithms for request prioritization in order to exploit parallelism offered by DRAM chips. In order to take into account the request criticality in the memory controller, we extend the existing request scheduling policy, *First Ready-first come first served* (FR-FCFS). PrioRAT is independent of the baseline scheduling policy as it only adds another sorting criterion. This policy schedules first the request that hit in the already open rows in the DRAM chip. If no such request exists, the ordering is done in *first come - first served* manner. We augment this policy with the task criticality information. To preserve the performance benefits of exploiting row hits, we consider the criticality only after there is no request satisfying the row hit condition. For the requests of the same criticality we use the FCFS ordering.

To offer higher memory bandwidth to the cores, many modern processors make use of multiple memory controllers. The PrioRAT design does not require

any synchronization between memory controllers as the prioritization is done independently in each controller. Thus, PrioRAT can be directly applied to such processors without any modification.

3.3 Discussion

Support for Simultaneous Multi-Threading (SMT). Modern processors implement support for simultaneous multi-threading in order to better utilize the resources of superscalar out-of-order cores. In such systems, one physical core can execute two different threads. In the context of our work this means that two tasks of different criticality can share the same core. To ensure the correct prioritization of critical requests in an SMT processor, we would extend the input queue of the L1 cache in the same way as described for the LLC earlier in this section. The core would also be equipped with a *crit.reg* per thread.

Ensuring Fairness. PrioRAT's mechanism for memory request prioritization needs to ensure fair scheduling of tasks and applications. Within one application's tasks, it is the responsibility of the runtime system to ensure correct task and request prioritization. Well-known mechanisms for preventing starvation of non-prioritized requests can be employed in the shared resources' queues. On multi-programmed systems, it is necessary to ensure that one application does not starve other applications. For example, a "rogue" runtime system could make all tasks of a single application critical and therefore get an unfair share of hardware resources. To solve this issue, the operating system could, for example, enforce quotas for critical tasks or requests per application. Similar schemes are already proposed in previous works, which we briefly describe in Section 6. An alternative solution would be to integrate the algorithms for task prioritization within the operating system and, therefore, guarantee the their integrity.

Alternative Algorithms for Task Priority Estimation. PrioRAT hardware design is orthogonal to the algorithm to detect task criticality. We consider an existing algorithm to calculate task criticality based on the application's critical path, as described in Section 3.1. The critical path is calculated by considering the TDG's structure and the tasks' distance from the bottom of the TDG. An alternative TDG-based approach could also consider the task duration when calculating critical path. For example, it might be a better idea to prioritize a single long-lasting task compared to several chained short tasks. However, such approach requires a way to estimate task duration, which can be achieved via profiling, a heuristic based on code analysis or given as a hint by a programmer. Some proposals [5] employ static annotations of task priority introduced by a programmer. However, this requires good a knowledge of the code structure and potential previous profiling. While hand-tuned hints can provide better final criticality estimation, they are less practical than dynamic solutions which are able to adapt to different scenarios, input sets and platforms.

Table 1: Benchmark details.

Benchmark	Abbrev.	Input Parameters
Array scan	Scan	174 arrays of 256 KB; total array size 68.5 MB
Blackscholes	BS	16M options, 512K block size, 5 iterations
Conj. Grad.	CG	matrix qa8fm, 16 blocks, 97 iterations
Cholesky	Chol	16×16 blocks of 256×256 elem. (matrix 4096×4096 elem.)
Fluidanimate	Fluid	native: 5 frames, 500,000 particles
Heat-Jacobi	Heat	8192×8192 resolution, 2 heat sources, 10 iterations
Molec. Dyn.	MD	2000 atoms, periodic space, stretch phase change
miniAMR	AMR	64K max. blocks, 16×16×16 block, 2 objects, 40 variables
prk2-stencil	PRK2	8192 elem. per dimension, 1024 block size, 5 iterations
LU Decomp.	LU	12 blocks of 512×512 elem.
Specfem3D	Spfm3D	147,645 pts., 2160-elem. mesh, 125 GLL int. pts. per elem.
Sym. Mat. Inv.	SMI	8×8 blocks of 1024×1024 elem. (matrix 8192×8192 elem.)

Table 2: Parameters of the simulated system.

CPU	16 OoO superscalar cores, 192-entry ROB, 2.2 GHz, issue width 4 ins/cycle
Caches	64B line, non-inclusive, write-back, write-allocate
L1	private, 32 KB, 8-way set-associative, 4-cycle latency, split I/D, 16-entry MSHR
L2	private, 256 KB, 16-way set-associative, 13-cycle latency, 16-entry MSHR
L3	shared, 16 MB, 16-way set-associative, 68-cycle latency, 256-entry MSHR
Memory	120ns latency; 2 channels, each with the effective bandwidth of 4.3 GB/s
	128-entry buffer in the memory controller

4 Experimental Methodology

Benchmarks. To evaluate PrioRAT we use a set of benchmarks that covers common applications running on HPC systems implemented in a task-based programming model. The list of selected benchmarks with their corresponding input parameters is shown in Table 1. The synthetic benchmark described in Section 2, Scan, is also used in the evaluation.

Simulation Setup. We use TaskSim, a trace-driven cycle-accurate architectural simulator [30, 31]. TaskSim simulates in detail the execution of parallel applications with OpenMP pragma primitives [28] on parallel multi-core environments. The simulated system mimics an Intel-based processor and consists of 16 cores connected to main memory. We model a superscalar out-of-order cores with a detailed three-level cache hierarchy. Each core has two private cache levels, L1 and L2, while the L3 is shared. We use a main memory model with an effective bandwidth of 8.6 GB/s and latency of 120 ns. All relevant parameters of the simulated system are shown in Table 2.

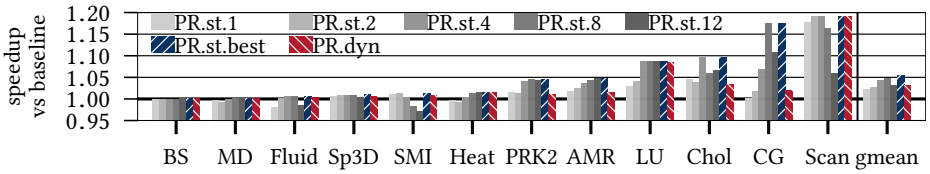

Fig. 4: Speedup of PR.static and PR.dynamic compared to the baseline.

5 Evaluation

5.1 Performance Evaluation

This section presents the overall performance of the two PrioRAT configurations for all evaluated benchmarks using the environment described in Section 4. Figure 4 shows the speedup of various PrioRAT configurations per benchmark compared to the execution on the baseline system. The following paragraphs explain the results for all evaluated configurations.

PR.static. First, we observe speedups achieved by a range of PR.static configurations. We consider configurations having 1, 2, 4, 8 and 12 prioritized cores in a 16-core system. These configurations are denoted as PR.st.X, where X is the number of prioritized cores. Some of the benchmarks, such as MD, Heat, AMR and LU, perform better when running on systems with more prioritized cores. In such configurations, there is a higher chance for all critical tasks to be prioritized. Moreover, sometimes it is beneficial to also accelerate some non-critical tasks, especially if that does not significantly penalize the performance of other non-prioritized tasks. On the other hand, some benchmarks, such as Fluid, SMI, Chol and Scan, achieve the best speedups when using less prioritized cores, i.e., 2, 4 or 8 cores. These codes have long critical paths and, therefore, do not benefit from prioritizing too many non-critical tasks. On average, the best performing PR.static configuration is PR.st.8, which performs 4.6% faster than the baseline. The best speedup of 1.19× is observed for Scan.

In addition, we define PR.st.best, which is the best-performing PR.static configuration per benchmark. On average, this configuration is 5.9% faster than the baseline. This shows that in scenarios where per-benchmark tuning is possible, application specific PR.static configuration can achieve better performance than a statically defined configuration.

PR.dynamic. Finally, we evaluate PR.dynamic, denoted PR.dyn in Figure 4. On average, this configuration outperforms the baseline by 3.0%. With 8 out of 12 benchmarks, this configuration achieves similar speedup as PR.st.best. In Chol, it achieves lower speedups due to scheduling artifacts that cause non-critical tasks to be scheduled for execution shortly before a critical task starts executing. This leads to a situation where all cores are busy executing non-critical tasks, while a critical task is waiting in a ready queue.

To better understand the performance gains, we observe the behavior at the microarchitectural level. The prioritization of critical requests reduces their round-

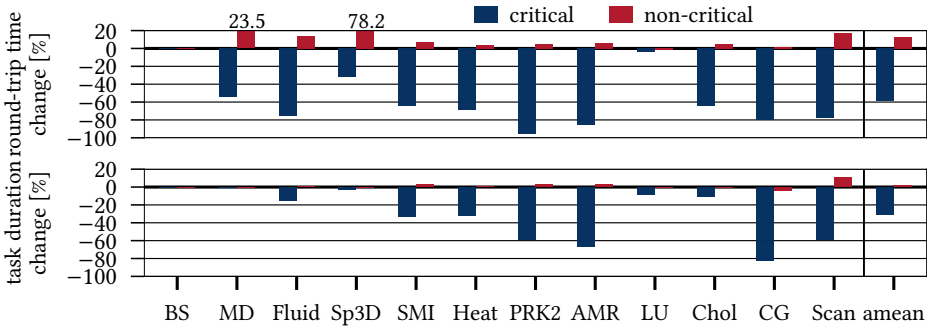

Fig. 5: Impact of request prioritization on request round-trip time (top) and task duration (bottom) for PR.dynamic

trip time from the core to the memory subsystem and back, at the expense of the increased round-trip time for non-critical requests. This effect is demonstrated on the upper plot in Figure 5, which shows the round-trip time only for requests that arrive to the main memory (i.e., the LLC misses).

Since critical memory requests observe reduced round-trip time, the critical tasks spend less time waiting for memory operations and therefore can execute faster. This finally results in a shorter execution time of the critical path, which may improve the overall performance of an application. These effects are shown at the bottom of Figure 5, which shows change in the duration of critical and non-critical tasks compared to the baseline configuration. We can observe that the critical path is significantly reduced in many applications.

However, this does not result in drastic performance improvement, as shown in Figure 4. The reason for such behavior is as follows. We use a critical path detection algorithm based on the structure of a TDG. It does not consider the duration of tasks, but rather the number of tasks on the path from the starting to the ending node. In addition, when the critical path is accelerated in PrioRAT, other non-critical tasks may become the new critical path (from the point of view of execution, not the TDG). Nevertheless, even with an infinite reduction of the critical path's length and a perfect scheduling algoritm, the achieved speedup is limited by the non-prioritized tasks. Only in the cases of LU and Chol, critical tasks stay in the critical path after the prioritization, which makes the reduction in critical path duration directly observable in overall performance gains.

5.2 Performance Sensitivity to Memory Latency

The evaluation presented so far is based on experiments using a memory latency of 120 ns, as explained in Section 4. Since memory latency varies across different systems, we explore how PrioRAT performs in systems with memory latencies ranging from 60 ns to 140 ns. Figure 6 shows the performance of the two PrioRAT configurations for the mentioned memory configurations. We do not show the results for the three benchmarks achieving lowest speedups, i.e., BS, MD and

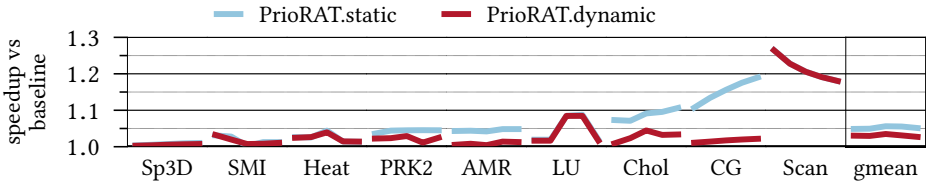

Fig. 6: Impact of memory latency (60ns, 80ns, 100ns, 120ns and 140ns) on PrioRAT's performance. Benchmarks with low speedups are omitted for clarity. Geometric mean includes the omitted benchmarks.

Fluid. The geometric mean is calculated taking into account all benchmarks, including the omitted ones. The results show that, for some benchmarks, the performance of PrioRAT varies across systems with different memory latencies. This is pronounced for LU, Chol, CG and Scan. Chol and CG perform better with PrioRAT in systems with higher memory latencies because prioritization in these systems makes more impact compared to the low-latency systems. The opposite behaviour is noticed for Scan. Due to a very high memory contention in this benchmark, most of the request round-trip time is spent waiting for a response from DRAM. Since prioritization only impacts the time requests spend in the queue waiting to be served, we conclude that prioritization has less effect in high-latency systems. We expect such behaviour to be rare in real codes. On average, we notice that PrioRAT performs equally across systems with different memory latencies.

5.3 Hardware Cost of PrioRAT Implementation

PrioRAT requires minor extensions in the core, the shared caches, the interconnect and the memory controllers, as explained in Section 3.2. The size of the *crit.reg* is one bit per core. In case of SMT support (see Section 3.3), PrioRAT requires one one-bit register per hardware thread. Inside the shared components, i.e., the caches, the interconnect and the memory controllers, we use double-queue to prioritize requests, which we design to be the same total size as the corresponding queue in the baseline system. Therefore, only added cost is a simple logic for scheduling requests into the two queues based on the request criticality. Finally, to enable passing criticality information with request, the structures that hold requests and corresponding communication lines are extended with one bit. All described extensions introduce negligible overhead in area compared to the baseline system.

6 Related Work

Acceleration of the Critical Path. In the context of fork-join programming models, the notion of a critical path applies to the slowest thread in a parallel

region. Accelerating such threads has been studied for heterogeneous chip multiprocessors [21, 35]. In the context of task-based programming models, critical path is often defined as the longest path from the starting until the final node in a task-dependency graph. The critical path detection has been extensively researched [3, 12, 20]. CATS [7] and CATA [5] are methods to accelerate critical paths targeting task-based programming models. CATS and CATA can only be implemented in heterogeneous systems or systems with support for dynamic voltage scaling, respectively. PrioRAT does not have these shortcomings and is applicable to any modern multi-core processor.

Scheduling in the Memory Controller. FCFS (First Come – First Served) is the simplest memory controller scheduling algorithm that considers only request arrival time. FR-FCFS (First Ready FCFS) [32] improves the FCFS algorithm by taking into account the locality of the row buffers inside DRAM chips to reduce costly *activate* and *precharge* actions. Many other previous works optimize the ordering of DRAM commands in order to improve memory bandwidth [29, 37].

Multi-core processors introduce new challenges, such as avoiding unfairness and starvation. FQM [27], STFM [26], PAR-BS [25] and ATLAS [23] are some of the many techniques that try to achieve fairness among threads by prioritizing requests inside memory controllers. Since these static schemes do not always achieve the best performance for a wide range of applications, researchers have proposed many adaptive scheduling algorithms. BLISS [34] prioritizes applications that are more sensitive to memory interference. Ipek et al. [18] propose an adaptive memory controller scheduling scheme based on reinforced learning. Hashemi et al. [15] identify that, so called, *dependent* cache misses are an important contributor to performance degradation when on-chip contention is present. This scenario occurs when an instruction causing a cache miss also depends on another instruction that results in a miss. Prioritizing such misses results in a reduced waiting time for the second miss.

However, the mentioned proposals focus only on the information visible to the core, which is generally observed on relatively short time intervals of several thousands of CPU cycles. Such fine-grained observations cannot capture the macro trends in the whole application as well as the impact of prioritization on the overall performance. Our scheme is driven by the runtime-provided knowledge on task criticality. Such approach significantly simplifies the hardware and enables better long-term decisions inside the on-chip resources.

Exploiting Criticality in Hardware. Subramaniam et al. [33] evaluate the impact of criticality information in the optimizations of on-chip components, such as the L1 data cache and the store queue. Ghose et al. [13] utilize the load criticality to augment the FR-FCFS scheduling policy in the memory controller. Both of these solutions observe criticality on the instruction level, contrary to our approach that uses task-level criticality. Moreover, they require precise predictors of criticality which comes with additional hardware cost in each processor core. PrioRAT utilizes runtime-provided information and does not introduce complex and expensive hardware predictors.

7 Conclusions

In this paper we present PrioRAT, a runtime-assisted approach for prioritization of memory requests in the processor. PrioRAT exploits the high-level information about the application, contrary to many state-of-the-art proposals. Our approach relies on the runtime system library to detect the critical path in a task-parallel application and forward task criticality information to the underlying hardware. The processor uses the knowledge of the task criticality to guide the prioritization of the memory requests inside the shared on-chip memory hierarchy.

We evaluate PrioRAT on a set of representative HPC codes. The extensive evaluation shows that PrioRAT outperforms the baseline system by up to 1.19× in terms of the execution time. Further analysis demonstrates a high impact of the prioritization on the request service time and task duration. Therefore, we demonstrate the importance of the availability of application-level knowledge inside the on-chip components.

Acknowledgements. This work has been partially supported by the Spanish Ministry of Science and Innovation (PID2019-107255GB-C21/AEI/10.13039/501100011033), by the Generalitat de Catalunya (contracts 2017-SGR-1414 and 2017-SGR-1328), by the European Unions Horizon 2020 research and innovation program under the Mont-Blanc 2020 project (grant agreement 779877) and by the RoMoL ERC Advanced Grant (GA 321253). V. Dimić has been partially supported by the Agency for Management of University and Research Grants (AGAUR) of the Government of Catalonia under Ajuts per a la contractació de personal investigador novell fellowship number 2017 FI_B 00855. M. Moretó and M. Casas have been partially supported by the Spanish Ministry of Economy, Industry and Competitiveness under Ramon y Cajal fellowship numbers RYC-2016-21104 and RYC-2017-23269, respectively.

References

1. Alvarez, L., Vilanova, L., Moreto, M., Casas, M., Gonzlez, M., et al.: Coherence protocol for transparent management of scratchpad memories in shared memory manycore architectures. In: ISCA'15. pp. 720–732 (2015). https://doi.org/10.1145/2749469.2750411
2. Barcelona Supercomputing Center: Nanos++ Runtime Library (May 2014), http://pm.bsc.es/nanox
3. Cai, Q., González, J., Rakvic, R., Magklis, G., Chaparro, P., González, A.: Meeting points: Using thread criticality to adapt multicore hardware to parallel regions. In: PACT'08. pp. 240–249 (2008). https://doi.org/10.1145/1454115.1454149
4. Casas, M., Moretó, M., Alvarez, L., Castillo, E., Chasapis, D., Hayes, T., et al.: Runtime-aware architectures. In: Euro-Par'15. pp. 16–27 (2015). https://doi.org/10.1007/978-3-662-48096-0_2
5. Castillo, E., Moreto, M., Casas, M., Alvarez, L., Vallejo, E., Chronaki, K., et al.: CATA: Criticality Aware Task Acceleration for Multicore Processors. In: IPDPS'16. pp. 413–422 (2016). https://doi.org/10.1109/IPDPS.2016.49

6. Chamberlain, B., Callahan, D., Zima, H.: Parallel Programmability and the Chapel Language. The International Journal of High Performance Computing Applications **21**(3), 291–312 (Aug 2007). https://doi.org/10.1177/1094342007078442
7. Chronaki, K., Rico, A., Badia, R.M., Ayguad, E., Labarta, J., Valero, M.: Criticality-Aware Dynamic Task Scheduling for Heterogeneous Architectures. In: ICS'15. pp. 329–338 (2015). https://doi.org/10.1145/2751205.2751235
8. Chun-Hsien Liu, Chia-Feng Li, Kuan-Chou Lai, Chao-Chin Wu: A Dynamic Critical Path Duplication Task Scheduling Algorithm for Distributed Heterogeneous Computing Systems. In: ICPADS'06. vol. 1, pp. 8 pp.– (2006). https://doi.org/10.1109/ICPADS.2006.37
9. Daoud, M., Kharma, N.: Efficient Compile-Time Task scheduling for Heterogeneous Distributed Computing Systems. In: ICPADS'06. vol. 1, pp. 11–22 (Jan 2006). https://doi.org/10.1109/ICPADS.2006.40
10. Dimić, V., Moretó, M., Casas, M., Ciesko, J., Valero, M.: Rich: Implementing reductions in the cache hierarchy. In: ICS'20. p. 13 pages (2020). https://doi.org/10.1145/3392717.3392736, https://doi.org/10.1145/3392717.3392736
11. Dimić, V., Moretó, M., Casas, M., Valero, M.: Runtime-assisted shared cache insertion policies based on re-reference intervals. In: Euro-Par'17. vol. 10417, pp. 247–259 (2017). https://doi.org/10.1007/978-3-319-64203-1_18
12. Du Bois, K., Eyerman, S., Sartor, J.B., Eeckhout, L.: Criticality Stacks: Identifying Critical Threads in Parallel Programs Using Synchronization Behavior. In: ISCA'13. pp. 511–522 (2013). https://doi.org/10.1145/2485922.2485966
13. Ghose, S., Lee, H., Martínez, J.F.: Improving Memory Scheduling via Processor-Side Load Criticality Information. In: ISCA'13. p. 8495 (2013). https://doi.org/10.1145/2485922.2485930
14. Hakem, M., Butelle, F.: Dynamic Critical Path Scheduling Parallel Programs onto Multiprocessors. In: IPDPS'05. pp. 7 pp.– (2005). https://doi.org/10.1109/IPDPS.2005.175
15. Hashemi, M., Khubaib, Ebrahimi, E., Mutlu, O., Patt, Y.N.: Accelerating dependent cache misses with an enhanced memory controller. In: ISCA'16. pp. 444–455 (2016). https://doi.org/10.1109/ISCA.2016.46
16. Intel Copropration: Intel® Cilk™ Plus Language Extension Specification (2013)
17. Intel Copropration: Intel® Thread Bulding Blocks (March 2020)
18. Ipek, E., Mutlu, O., Martnez, J.F., Caruana, R.: Self-Optimizing Memory Controllers: A Reinforcement Learning Approach. In: ISCA'08. pp. 39–50 (2008). https://doi.org/10.1109/ISCA.2008.21
19. Jaulmes, L., Casas, M., Moretó, M., Ayguadé, E., Labarta, J., Valero, M.: Exploiting asynchrony from exact forward recovery for due in iterative solvers. In: SC'15 (2015). https://doi.org/10.1145/2807591.2807599
20. Joao, J.A., Suleman, M.A., Mutlu, O., Patt, Y.N.: Bottleneck Identification and Scheduling in Multithreaded Applications. In: ASPLOS'12. pp. 223–234 (2012). https://doi.org/10.1145/2150976.2151001
21. Joao, J.A., Suleman, M.A., Mutlu, O., Patt, Y.N.: Utility-Based Acceleration of Multithreaded Applications on Asymmetric CMPs. In: ISCA'13. pp. 154–165 (2013). https://doi.org/10.1145/2485922.2485936
22. Kale, L.V., Krishnan, S.: CHARM++: A Portable Concurrent Object Oriented System Based on C++. In: OOPSLA'93. pp. 91–108 (1993). https://doi.org/10.1145/165854.165874
23. Kim, Y., Han, D., Mutlu, O., Harchol-Balter, M.: ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In: HPCA'10. pp. 1–12 (2010). https://doi.org/10.1109/HPCA.2010.5416658

24. Manivannan, M., Papaefstathiou, V., Pericas, M., Stenstrom, P.: RADAR: Runtime-Assisted Dead Region Management for Last-Level Caches. In: HPCA'2016. pp. 644–656 (2016). https://doi.org/10.1109/HPCA.2016.7446101
25. Mutlu, O., Moscibroda, T.: Parallelism-Aware Batch Scheduling: Enabling High-Performance and Fair Shared Memory Controllers. IEEE Micro **29**(1), 22–32 (2009). https://doi.org/10.1109/MM.2009.12
26. Mutlu, O., Moscibroda, T.: Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In: MICRO'07. pp. 146–160 (2007). https://doi.org/10.1109/MICRO.2007.40
27. Nesbit, K.J., et al.: Fair Queuing Memory Systems. In: MICRO'06. pp. 208–222 (2006). https://doi.org/10.1109/MICRO.2006.24
28. OpenMP Architecture Review Board: OpenMP Technical Report 4 Version 5.0 Preview 1 (November 2016)
29. Peiron, M., Valero, M., Ayguadé, E., Lang, T.: Vector Multiprocessors with Arbitrated Memory Access. In: ISCA'95. pp. 243–252 (1995). https://doi.org/10.1145/223982.224435
30. Rico, A., Cabarcas, F., Villavieja, C., Pavlovic, M., Vega, A., Etsion, Y., et al.: On the Simulation of Large-scale Architectures Using Multiple Application Abstraction Levels. ACM Trans. Archit. Code Optim. **8**(4), 36:1–36:20 (2012). https://doi.org/10.1145/2086696.2086715
31. Rico, A., Duran, A., Cabarcas, F., Etsion, Y., Ramirez, A., Valero, M.: Trace-driven simulation of multithreaded applications. In: ISPASS'11. pp. 87–96 (2011). https://doi.org/10.1109/ISPASS.2011.5762718
32. Rixner, S., Dally, W.J., Kapasi, U.J., Mattson, P., Owens, J.D.: Memory access scheduling. In: ISCA'00. pp. 128–138 (2000). https://doi.org/10.1145/339647.339668
33. Subramaniam, S., Bracy, A., Wang, H., Loh, G.H.: Criticality-Based Optimizations for Efficient Load Processing. In: HPCA'09. pp. 419–430 (2009). https://doi.org/10.1109/HPCA.2009.4798280
34. Subramanian, L., Lee, D., Seshadri, V., Rastogi, H., Mutlu, O.: The Blacklisting Memory Scheduler: Achieving high performance and fairness at low cost. In: ICCD'14. pp. 8–15 (2014). https://doi.org/10.1109/ICCD.2014.6974655
35. Suleman, M.A., Mutlu, O., Qureshi, M.K., Patt, Y.N.: Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures. In: ASPLOS'09. pp. 253–264 (2009). https://doi.org/10.1145/1508244.1508274
36. Topcuoglu, H., Hariri, S., Min-You Wu: Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. IEEE Transactions on Parallel and Distributed Systems **13**(3), 260–274 (2002). https://doi.org/10.1109/71.993206
37. Valero, M., Lang, T., Llabería, J.M., Peiron, M., Ayguadé, E., Navarra, J.J.: Increasing the Number of Strides for Conflict-Free Vector Access. In: ISCA'92. pp. 372–381 (1992). https://doi.org/10.1145/139669.140400
38. Valero, M., Moretó, M., Casas, M., Ayguade, E., Labarta, J.: Runtime-Aware Architectures: A First Approach. Supercomputing frontiers and innovations **1**(1) (2014). https://doi.org/10.14529/jsfi140102
39. Wulf, W.A., McKee, S.A.: Hitting the memory wall: implications of the obvious. ACM SIGARCH Computer Architecture News **23**(1), 20–24 (Mar 1995). https://doi.org/10.1145/216585.216588