CATA: Criticality Aware Task Acceleration for Multicore Processors

Emilio Castillo[*†], Miquel Moreto[*†], Marc Casas[*], Lluc Alvarez[*‡], Enrique Vallejo[‡], Kallia Chronaki[*], Rosa Badia[*§]
Jose Luis Bosque[‡], Ramon Beivide[‡], Eduard Ayguade[*†], Jesus Labarta[*†], Mateo Valero[*†]
[*]Barcelona Supercomputing Center, [†]Universidad Politecnica de Catalunya, [‡]Universidad de Cantabria,
[§]Artificial Intelligence Research Institute (IIIA) Spanish National Research Council (CSIC)
name.surname@bsc.es, name.surname@unican.es

***Abstract*—Managing criticality in task-based programming models opens a wide range of performance and power optimization opportunities in future manycore systems. Criticality aware task schedulers can benefit from these opportunities by scheduling tasks to the most appropriate cores. However, these schedulers may suffer from priority inversion and static binding problems that limit their expected improvements.**

Based on the observation that task criticality information can be exploited to drive hardware reconfigurations, we propose a Criticality Aware Task Acceleration (CATA) mechanism that dynamically adapts the computational power of a task depending on its criticality. As a result, CATA achieves significant improvements over a baseline static scheduler, reaching average improvements up to 18.4% in execution time and 30.1% in Energy-Delay Product (EDP) on a simulated 32-core system.

The cost of reconfiguring hardware by means of a software-only solution rises with the number of cores due to lock contention and reconfiguration overhead. Therefore, novel architectural support is proposed to eliminate these overheads on future manycore systems. This architectural support minimally extends hardware structures already present in current processors, which allows further improvements in performance with negligible overhead. As a consequence, average improvements of up to 20.4% in execution time and 34.0% in EDP are obtained, outperforming state-of-the-art acceleration proposals not aware of task criticality.

I. Introduction

In recent years, power limits and thermal dissipation constraints have extolled the importance of energy efficiency in microprocessor designs. For this reason, modern computer systems implement different hardware mechanisms that allow to reconfigure the computational capability of the system, aiming to maximize performance under affordable power budgets. For example, per-core power gating and Dynamic Voltage and Frequency Scaling (DVFS) are common reconfiguration techniques available on commodity hardware [1], [2]. In SMT processors, the number of SMT threads per core or the decode priority can be adjusted [3], [4], while in multicores, the prefetcher aggressiveness, the memory controller or the last-level cache space assigned to an application can be changed [5]–[7]. More recently, reconfigurable systems that support core fusion or that can transform traditional high performance out-of-order cores into highly-threaded in-order SMT cores when required, have been shown to achieve significant reductions in terms of energy consumption [8], [9]. However, the problem of optimally reconfiguring the hardware is not solved in general as all the aforementioned solutions rely on effective but ad-hoc mechanisms. Combining such solutions for a wide set of reconfiguration problems is complex [4], and introduces a significant burden on the programmer.

Recent advances in programming models recover the use of task-based data-flow programming models to simplify parallel programming in future manycores [10]–[15]. In these models the programmer splits the code in sequential pieces of work, called tasks, and specifies the data and control dependences between them. With this information the runtime system manages the parallel execution, scheduling tasks to cores and taking care of synchronization among them. These models not only ease programmability, but also can increase performance by avoiding global synchronization points. However, the issue of controlling reconfigurable hardware when using this simple data-flow model is still not properly solved in the literature.

This work advocates an integrated system in which the task-based runtime system controls hardware reconfiguration according to the criticality of the different tasks in execution. As such, the runtime can either schedule the most critical tasks to the fastest hardware components or reconfigure those components where the highly-critical tasks run. In this way, the programmer only has to provide simple and intuitive annotations and does not need to explicitly control the way the load is balanced, how the hardware is reconfigured, or whether a particular power budget is met. Such responsibilities are mainly left to the runtime system, which decouples the software and hardware layers and drives the design of specific hardware components to support such functions when required. To reconfigure the computation power of the system, we consider DVFS, as it is a common reconfiguration capability on commodity hardware. However, our criticality aware approach can target reconfiguration of any hardware component, as no DVFS specific assumptions are made.

The main contributions of the paper are:

- We compare two mechanisms for estimating task criticality with user-defined static annotations and with a dynamic solution operating at execution time. Both approaches are effective, but the simpler implementation of the user-defined static annotations provides slightly better performance and EDP results.
- We introduce *Criticality Aware Task Acceleration* (CATA), a runtime system level technique that recon-

figures the frequency of the cores while keeping the whole processor under a certain power budget. Average improvements reach 18.4% and 30.1% in execution time and EDP over a baseline scheduler on a 32-core system.
- For some applications, the DVFS reconfiguration penalties caused by inherent serialization issues can become a performance bottleneck. To overcome this problem, we introduce a hardware component denoted *Runtime Support Unit* (RSU), which relieves the runtime system of carrying out frequency reconfigurations and can be easily incorporated on top of existing solutions [16]–[18]. For sensitive applications, up to an additional 8.5% improvement in performance is obtained over CATA.

The integrated solution proposed in this paper, which goes from the source code to the hardware level passing through the runtime and the operating system, shows the need for a multi-layer approach to optimally exploit the heterogeneity and reconfiguration possibilities of future manycore systems.

This paper is organized as follows. Section II provides background on the work. Section III presents our approaches for task criticality-aware task acceleration. The experimental setup and evaluation results are detailed in Sections IV and V. Related work is discussed in Section VI and, finally, Section VII draws the main conclusions of this work.

II. Background

This section describes the characteristics of task-based programming models, the concept of criticality in them, the scheduling techniques to exploit such criticality in heterogeneous architectures and the problems they present.

A. Task-Based Programming Models

Task-based programming models such as OpenMP 4.0 [10] conceive the execution of a parallel program as a set of tasks with dependences among them. Typically, the programmer adds code annotations to split the serial code in tasks that can potentially run in parallel. Each annotation defines a different *task type*, while every execution of a given task type is denoted a *task instance*. Also, the programmer specifies what data is used by each task (called input dependences) and what data is produced (called output dependences). The runtime system is in charge of managing the execution of the tasks, releasing the programmer from the burden of explicitly synchronizing tasks and scheduling them to cores, thus easing programmability.

In order to manage task execution the runtime system builds a *task dependence graph* (TDG), a directed acyclic graph where nodes represent tasks and edges dependences among them. Similarly to how an out-of-order processor schedules instructions, the runtime system schedules a task on a core when all its input dependences are ready and, when the execution of the task finishes, its output dependences become ready for the next tasks. This execution model decouples the hardware from the application, allowing to apply many optimizations at runtime level in a generic and application-agnostic way [19], [20]. For instance, the information found in the task dependences can be exploited to perform data prefetching [21] or efficient data communication between tasks [22], [23]. In this context, this paper exploits for the first time the opportunities that task criticality information available in the runtime system offer to reconfigure current multicores.

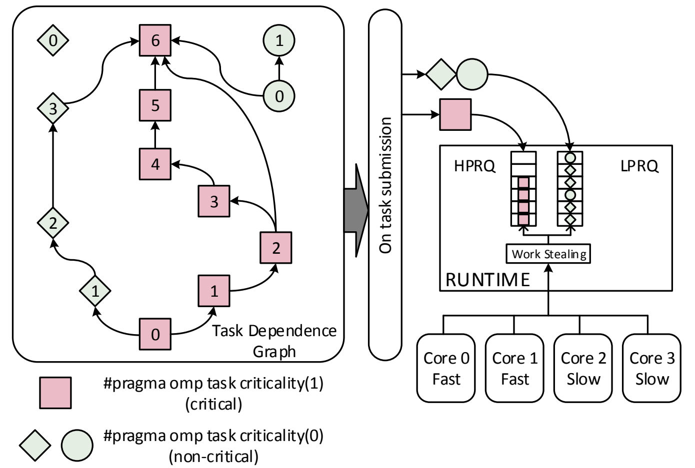

Fig. 1. Criticality assignment with bottom-level and static policies, and Criticality-Aware Task Scheduling.

B. Task Criticality in Task-Based Programming Models

Criticality represents the extent to which a particular task is in the critical path of a parallel application. In general, criticality is difficult to estimate as the execution flow of an application is dynamic and input-dependent. Two approaches can be used to estimate the criticality of a task.

One approach is to dynamically determine the criticality of the tasks during the execution of the application, exploring the TDG of tasks waiting for execution and assigning higher criticality to those tasks that belong to the longest dependency path [24]. Figure 1 shows a synthetic example of this method. In the TDG on the left, each node represents a task, each edge of the graph represents a dependence between two tasks, and the shape of the node represents its task type. The number inside each node is the *bottom-level* (BL) of the task, which is the length of the longest path in the dependency chains from this node to a leaf node. The criticality of a task is derived from its BL, where tasks with the highest BL and their descendants in the longest path are considered critical. Consequently, the square nodes in Figure 1 are considered critical.

The bottom-level approach does not require any input from the programmer and it can dynamically adapt to different phases of the application. However, this method has some limitations. First, exploring the TDG every time a task is created can become costly, specially in dense TDGs with short tasks. Second, the task execution time is not taken into account as only the length of the path to the leaf node is considered. Third, only a sub-graph of the TDG is considered to estimate criticality and some tasks marked as critical in such partial TDG may not be critical in the complete TDG.

Another approach is to statically assign criticality to the tasks, either using compiler analysis or allowing the programmer to annotate them. For this purpose, we have extended the task directive in OpenMP 4.0 to specify criticality, `#pragma`

`omp task criticality(c)`. The parameter c represents the criticality level assigned to the given task type. Critical tasks have higher values of c, while non-critical tasks have a value of $c = 0$. The bottom left part of Figure 1 shows how this directive is used to assign the criticality of the three different task types in the example, where square nodes are considered critical, while triangular and circular nodes are estimated as non-critical. In this example and for the sake of simplicity, tasks are assigned to the same criticality level with both approaches (static annotations and bottom-level), but this does not happen in general.

The main problem of static annotations is that estimating the criticality of a task can be complex and input dependent. However, by analyzing the execution of the application it is feasible to identify tasks that block the execution of other tasks, or tasks with long execution times that could benefit from running in fast processing elements.

The task criticality information obtained with any of these approaches can be exploited by the runtime system in multiple ways, specially in the context of heterogeneous systems.

C. Criticality-Aware Task Scheduling and Limitations

The task scheduler is a fundamental part of task-based runtime systems. Its goal is to assign tasks to cores, maximizing the utilization of the available computational resources and ensuring load balance. The typical scheduler of task-based runtime systems assigns tasks to available cores in a *first in, first out* (FIFO) manner without considering the criticality of the tasks. In this approach, tasks that become ready for execution are kept in a ready queue until there is an available core. The main limitation of the FIFO scheduler when running on heterogeneous systems is that tasks are assigned blindly to fast or slow cores, regardless of their criticality.

A *Criticality-Aware Task Scheduler* [24] (CATS) can solve the *blind assignment* problem of the FIFO scheduler. CATS is focused on heterogeneous architectures, ensuring that critical tasks are executed on fast cores and assigning non-critical tasks to slow cores. As shown in Figure 1, CATS splits the ready queue in two: a *high priority ready queue* (HPRQ), and a *low priority ready queue* (LPRQ). Tasks identified as critical are enqueued in the HPRQ and non-critical ones in the LPRQ. When a fast core is available it requests a task to the HPRQ, and the first ready task is scheduled on the core. If the HPRQ is empty, a task from the LPRQ can be scheduled on a fast core. If no tasks are ready, the core remains idle until some ready task is available. Similarly, slow cores look for tasks in the LPRQ. Task stealing from the HPRQ is accepted only if no fast cores are idling. Figure 1 illustrates the runtime system extensions and the scheduling decisions for the synthetic TDG on the left.

CATS solves the blind assignment problem of FIFO schedulers. However, even if it considers the criticality of the tasks, it may present misbehaviors in the scheduling decisions that lead to load imbalance in heterogeneous architectures:

- *Priority inversion*: when a critical task has to be scheduled and all the fast cores are in use by non-critical tasks, it is scheduled to a slow core.
- *Static binding* for the task duration: when a task finishes executing on a fast core, this core can be left idle even if other critical tasks are running on slow cores.

These problems happen because the computational capabilities of the cores are static and, once a task is scheduled to a core, it is not possible to re-distribute resources if the original circumstances change. In order to overcome these limitations, this paper proposes a runtime-driven *criticality-aware task acceleration* scheme, resulting in a responsive system that executes critical tasks on fast cores and re-distributes the computational capabilities of the architecture to overcome the priority inversion and static binding problems.

III. Criticality-Aware Task Acceleration Using DVFS Reconfigurations

This section proposes to exploit reconfiguration opportunities that task criticality information can provide to the runtime system to perform *Criticality-Aware Task Acceleration* (CATA) in task-based programming models. First, a pure software approach where the runtime system drives the reconfigurations according to the criticality of the running tasks is introduced. Then, hardware extensions required to support fast reconfigurations from the runtime system are described in detail.

DVFS is selected as a proof-of-concept for the reconfiguration mechanism, as it allows to accelerate different cores and it is already present in the majority of current architectures. Nevertheless, the proposed ideas and the runtime system extensions are generic enough to be applied or easily adapted to other reconfiguration techniques. We further assume that two frequency levels are allowed in the system, which can be efficiently implemented with dual-rail V_{dd} circuitry [25]. Extending the proposed ideas to more levels of acceleration is left as future work. In addition, Section VI discusses other reconfiguration approaches that could benefit from the ideas proposed in this paper.

A. Criticality-Aware Runtime-Driven DVFS Reconfiguration

The runtime system is extended with several structures to manage hardware reconfiguration according to the criticality of the tasks. Figure 2 shows these extensions. Similar to the CATS scheduler, the runtime system splits the ready queue in a HPRQ for the critical tasks and a LPRQ for the non-critical tasks. To manage the reconfigurations, the Reconfiguration Support Module (RSM) tracks the state of each core (*Accelerated* or *Non-Accelerated*), the criticality of the task that is being executed on each core (*Critical*, *Non-Critical*, or *No Task*), and the power budget. The power budget is represented as the maximum amount of cores that can simultaneously run at the fastest frequency, and is provided to the runtime system as a parameter.

When a core requests a new task to the scheduler it first tries to assign critical tasks from the HPRQ and, if no critical tasks are ready, a non-critical task from the LPRQ is selected. If there is enough power budget the core is set to the fastest power state, even for non-critical tasks. If there is no available

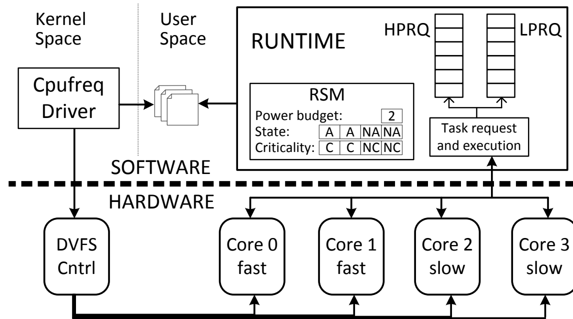

Fig. 2. Runtime system support for Criticality-Aware Task Acceleration (CATA) using DVFS reconfigurations. The runtime maintains status (Accelerated, Not Accelerated) and criticality (Critical, Non-Critical, No Task) information for each core in the Reconfiguration Support Module (RSM).

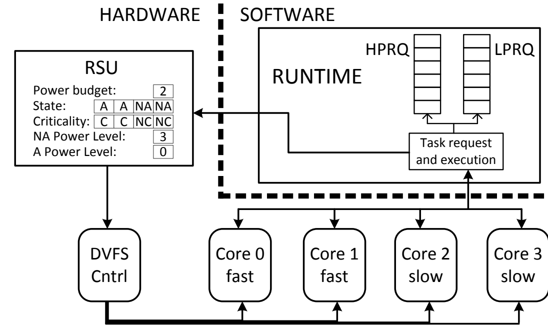

Fig. 3. Architectural and runtime system support for Criticality-Aware Task Acceleration (CATA) using DVFS reconfigurations. The RSU module implements the hardware reconfiguration functionality, and stores the same information as the RSM, plus the DVFS levels to use with Accelerated and Not Accelerated tasks.

power budget and the task is critical, the runtime system looks for an accelerated core executing a non-critical task, decreases its frequency, and accelerates the core of the new task. In the case that all fast cores are running critical tasks, the incoming task cannot be accelerated, so it is tagged as non-accelerated. Every time an accelerated task finishes, the runtime system decelerates the core and, if there is any non-accelerated critical task, one of them is accelerated.

To drive CPU frequency and voltage changes, the runtime system uses the standard interface provided by the `cpufreq` daemon of the Linux kernel. The `cpufreq` daemon governor is set to accept changes from user space. Figure 2 shows how the runtime system communicates with the `cpufreq` framework. Frequency and voltage changes are performed by writing the new power state in a configuration file mapped in the file system, having one file per core. The `cpufreq` daemon triggers an interrupt when it detects a write to one of these files, and the kernel executes the `cpufreq` driver. The driver writes the new power state in the DVFS controller, establishing the new voltage and frequency values for the core, and then the architecture starts the DVFS transition. Finally, the kernel updates all its internal data structures related to the clock frequency and returns the control to the runtime system.

Although this approach is able to solve the priority inversion and static binding issues by reconfiguring the computational capabilities assigned to the tasks, it raises a new issue for performance: *reconfiguration serialization*. Some steps of the software-driven reconfiguration operations inherently need to execute sequentially, since concurrent updates could transiently set the system in an illegal state that exceeds the power budget. Furthermore, invoking an interrupt and running the corresponding `cpufreq` driver in the kernel space can become a performance bottleneck. As a result, all the steps required to reconfigure the core frequency can last from tens of microseconds to over a millisecond in our experiments, becoming a potential point of contention for large core counts.

B. Architectural Support for DVFS Reconfiguration

With the trend towards highly parallel multicores the frequency of reconfigurations will significantly increase. This will be exacerbated by the increasing trend towards fine-grain task programming models with specific hardware support for task creation, data-dependencies detection and scheduling [26]–[28]. Consequently, software-driven reconfiguration operations be inefficient in future multicores. In such systems, hardware support for runtime-driven reconfigurations arises as a suitable solution to reduce contention in the reconfiguration process.

We propose a new hardware unit, the *Runtime Support Unit* (RSU), which implements the reconfiguration algorithm explained in the previous section. The RSU avoids continuous switches from user to kernel space, reducing the latency in reconfigurations and removing contention due to reconfiguration serialization. As illustrated in Figure 3, the RSU tracks the state of each core and the criticality of the running tasks to decide hardware reconfigurations and notify per-core frequency changes to the DVFS controller.

1) RSU Management

The RSU stores the criticality of the task running on each core (*Critical*, *Non-Critical*, or *No Task*), the status of each core (*Accelerated* or *Non-Accelerated*), the corresponding *Accelerated* and *Non-Accelerated Power Levels* to configure the DVFS controller, and the overall power budget for the system.

To manage the RSU, the ISA is augmented with initialization, reset and disabling instructions (`rsu_init`, `rsu_reset`, and `rsu_disable`, respectively), and control instructions to notify the beginning of the execution of tasks (`rsu_start_task(cpu,critic)`) and the completion of tasks (`rsu_end_task(cpu)`). Finally, another instruction is added to read the criticality of a task running in the RSU for virtualization purposes (`rsu_read_critic(cpu)`).

An alternative implementation could manage the RSU through a memory mapped schema. We have selected the ISA extension approach due to its simple implementation. As the RSU is only accessed twice per executed task, both solutions are expected to behave similarly.

2) RSU Operation

The RSU reconfigures the frequency and the voltage of each core upon two different events: task execution start and end. Whenever one of these two events occurs, the RSU inspects its current state to determine which cores have to be accelerated and which ones decelerated. This decision is taken with the same algorithm presented in Section III-A. When a task starts and there is available power budget the core is accelerated. If

the task is critical and there is no power budget available but a non-critical task is accelerated, the core of the non-critical task is decelerated and then the core of the new task is accelerated. If all the other tasks are critical the new task is executed at low frequency. When a task finishes the RSU decelerates its core and, if there is a critical task running on a non-accelerated core, it is accelerated.

3) RSU Virtualization

The operating system (OS) saves and restores the task criticality information at context switches. When a thread is preempted the OS reads the criticality value from the RSU and stores it in the associated kernel `thread_struct` data structure. The OS then sets a *No Task* value in the RSU to re-schedule the remaining tasks. When a thread is restored its task criticality value is written to the RSU. This design allows several concurrent independent applications to share the RSU.

4) Area and Power Overhead

The RSU requires a storage of 3 bits per core for the criticality and status fields, and $\log_2 num_cores$ bits for the power budget. In addition, two registers are required to configure the critical and non-critical power states of the DVFS controller. These registers require $\log_2 num_power_states$ bits and are set to the appropriate values at OS boot time. This results in a total storage cost of $3 \times num_cores + \log_2 num_cores + 2 \times \log_2 num_power_states$ bits. The overhead of the RSU has been evaluated using CACTI [29]. Results show that the RSU adds negligible overheads in area (less than 0.0001% in a 32-core processor) and in power (less than $50\mu W$).

5) Integration of RSU and TurboMode

The RSU can be seen as an extension to TurboMode implementations such as Intel's Turbo Boost [16], AMD Turbo Core [17], or dynamic TurboMode [18]. TurboMode allows active cores to run faster by using the power cap of the sleeping cores. A core is considered active as long as it is in the C_0 or C_1 ACPI power states: C_0 means that the core is actively executing instructions, while C_1 means that it has been briefly paused by executing the `halt` instruction in the OS scheduler. If a core remains in a C_1 state for a long period, the OS suggests to move the core to C_3 or a deeper power state, considering it inactive. Transitions between different power states are decided in a hardware microcontroller integrated in the processor die. Whenever a core is set to C_3 or deeper power state, some of the active cores in the C_0 state can increase their frequency as long as it does not exceed the overall power budget. Thus, the RSU registers could be added to the TurboMode microcontroller to accelerate parallel applications according to the task criticality with minimal hardware overhead.

IV. Experimental Setup

We employ Gem5 [30] to simulate an x86 full-system environment that includes application, runtime system and OS. Gem5 has been extended with a DVFS controller that allows to reconfigure the voltage and the frequency of each core [31]. Two frequency levels are allowed, 1 and 2 GHz, similar to an efficient dual-rail V_{dd} implementation [25]. In addition, the RSU described in Section III-B and a TurboMode model have also been implemented and integrated with the DVFS module. Power consumption is evaluated with McPAT [32] using a process technology of 22 nm and the default clock gating scheme of the tool. We perform simulations with 32 cores, using the detailed out-of-order CPU and detailed memory model of Gem5 configured as shown in Table I. Three heterogeneous configurations are considered: 25%, 50%, and 75% of fast cores, which corresponds to 8, 16, and 24 fast cores out of the total of 32 cores. In the experiments with CATS, the frequency of each core does not change during the execution, simulating a heterogeneous multicore with different performance ratios among cores. For CATA, the DVFS module varies core frequencies, reassigning the computational power across cores without exceeding the total power budget.

TABLE I
Processor Configuration.

Chip details	
Core count	32
Core type	Out-of-order single threaded
Core details	
DVFS configurations	Fast cores: 2 GHz, 1.0 V Slow cores: 1 GHz, 0.8 V $25\mu s$ reconfiguration latency
Fetch, issue, commit bandwidth	4 instr/cycle
Branch predictor	4K selector, 4K G-share, 4K bimodal 4-way BTB 4K entries, RAS 32 entries
Issue queue	Unified 64 entries
Reorder buffer	128 entries
Register file	256 INT, 256 FP
Functional units	4 INT ALU (1 cyc), 2 mult (3 cyc), 2 div (20 cyc) 2 FP ALU (2 cyc), 2 mult (4 cyc), 2 div (12 cyc) 2 Ld/St unit (1 cyc)
Instruction L1	32KB, 2-way, 64B/line (2 cycles hit)
Data L1	64KB, 2-way, 64B/line (2 cycles hit)
Instruction TLB	256 entries fully-associative (1 cycle hit)
Data TLB	256 entries fully-associative (1 cycle hit)
NoC and shared components	
L2	Unified shared NUCA, banked 2MB/core, 8-way 64B/line, 15/300 cycles hit/miss
Coherence protocol	MESI, distributed 4-way cache directory 64K entries
NoC	4×8 Mesh, link 1 cycle

The simulated system is a Gentoo Linux with a kernel 2.6.28-4. Nanos++ 0.7a [11] is used for the task runtime library, which supports OpenMP 4.0 constructs [10] and task criticality annotations. We have developed a driver to manage the DVFS controller within the `cpufreq` framework. Nanos++ requests frequency changes to the `cpufreq` framework by writing to a specific set of files, one per core. Any modification to these files triggers the `cpufreq` daemon to invoke the developed driver that sets the DVFS controller to the requested state. Note that this is the same mechanism found in real systems that support user space governors for DVFS.

To test the different proposals, we make use of a subset of six benchmarks from PARSECSs [33], a task-based implementation of the PARSEC suite [34] written in OpenMP 4.0. The benchmarks are compiled with Mercurium 1.99 source-to-source compiler [11], using gcc 4.6.4 as the backend compiler to generate the final binary. *Simlarge* input sets are used for

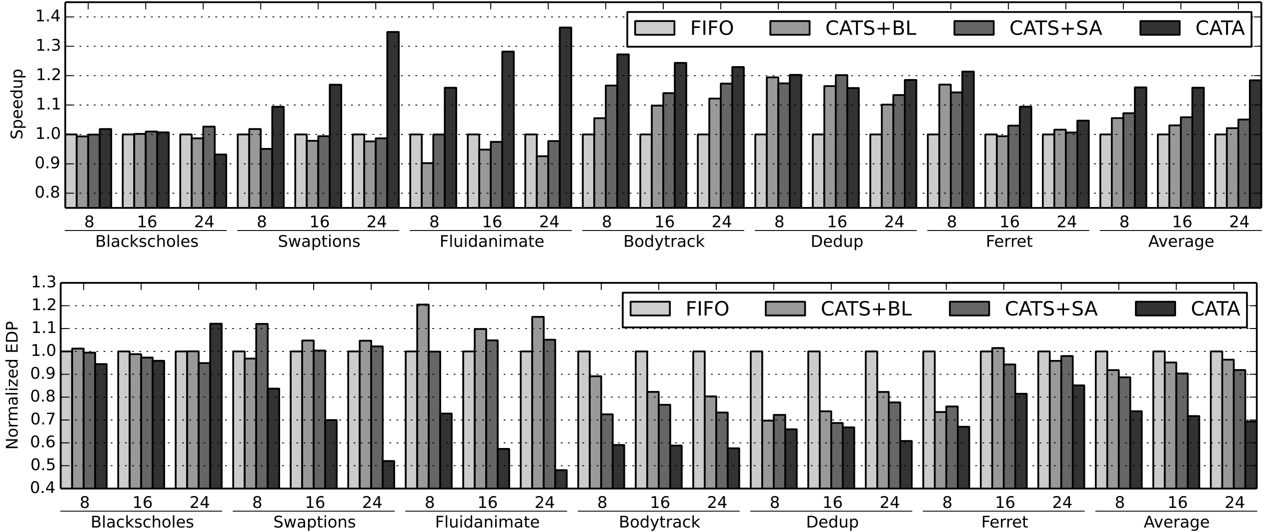

Fig. 4. Speedup and Energy-Delay Product (EDP) results with an increasing number of fast cores (8, 16, 24) on a 32-core processor. CATS+BL makes use of bottom-level and CATS+SA of static annotations methods to estimate task criticality. Results are normalized to the FIFO scheduler.

all the experiments and the whole parallel section of each benchmark is simulated.

The selected benchmarks are representative state-of-the-art parallel algorithms from different areas of computing and exploit different parallelization approaches: Blackscholes and Swaptions use fork-join parallelism, Fluidanimate is a 3D stencil, and Bodytrack, Dedup and Ferret use pipeline parallelism. For versions with static annotations, each task type is annotated with the desired criticality level. Fluidanimate has the maximum number of task types, eight, and on average, four criticality annotations were provided. To chose the proper criticality level, we make use of existing profiling tools to visualize the parallel execution of the application and identify its critical path. We complement this analysis with the criticality level identified by the bottom-level approach to decide the final criticality level of each task type.

V. Evaluation

A. Criticality-Aware Task Scheduling

Figure 4 shows the execution time speedup and the normalized Energy-Delay-Product (EDP) of the four different software-only implementations of the system: FIFO, two variants of CATS which employ bottom-level (CATS+BL) and static annotations (CATS+SA) as criticality estimation methods, and CATA, which is analyzed in the next section. All results are normalized to the FIFO scheduler.

Results show that CATS solves the *blind assignment* problem of FIFO, providing average speedups of up to 5.6% for CATS+BL and up to 7.2% for CATS+SA with 8 fast cores. Static annotations perform better in these applications than bottom-level, which was originally designed and evaluated for HPC applications [24]. This happens because the static annotations approach does not suffer the overhead of exploring the TDG of the application, in contrast to bottom-level.

However, not all benchmarks benefit from exploiting task criticality in CATS. Fork-join or stencil applications (Blackscholes, Swaptions and Fluidanimate) present tasks with very similar criticality levels. As a result, scheduling critical tasks to fast cores does not significantly impact performance. In fact, the overheads of the bottom-level approach can degrade performance, reaching up to a 9.8% slowdown in Fluidanimate, where each task can have up to nine parent tasks.

Applications with complex TDGs based on pipelines (Bodytrack, Dedup and Ferret) benefit more from CATS. These applications contain tasks with significantly different criticality levels. For example, in the case of Dedup and Ferret there are compute-intensive tasks followed by I/O-intensive tasks to write results that are in the critical path of the application. In these cases, a proper identification and scheduling of critical tasks yields important performance improvements, reaching up to 20,2% in Dedup. In the case of Bodytrack, task duration can change up to an order of magnitude among task types. Since CATS+BL identifies critical tasks based only on the length of the critical path in the TDG, it obtains smaller performance improvements than CATS+SA. In the case of Dedup and Ferret, both schedulers perform similarly, although the lower overhead of CATS+SA slightly favors performance in some cases.

Figure 4 also shows the normalized EDP of all the mechanisms. We observe that the improvements in execution time translate into energy savings. CATS+SA obtains average EDP reductions between 8.2% and 11.4%, while CATS+BL EDP reductions ranges between 3.7% and 8.2%. Fork-join or stencil applications do not obtain significant EDP reduction. It is noticeable the effect of CATS+BL overhead in the case of Fluidanimate with 8 fast cores, as EDP increases by 22.1% over the baseline. In contrast, significant EDP improvements occur in the benchmarks with complex TDGs, achieving EDP reductions up to 31.4% in Dedup with 16 fast cores.

B. Criticality-Aware Task Acceleration

CATA can dynamically reconfigure the DVFS settings of the cores based on the criticality of the tasks they execute, avoiding the *static binding* and *priority inversion* issues of

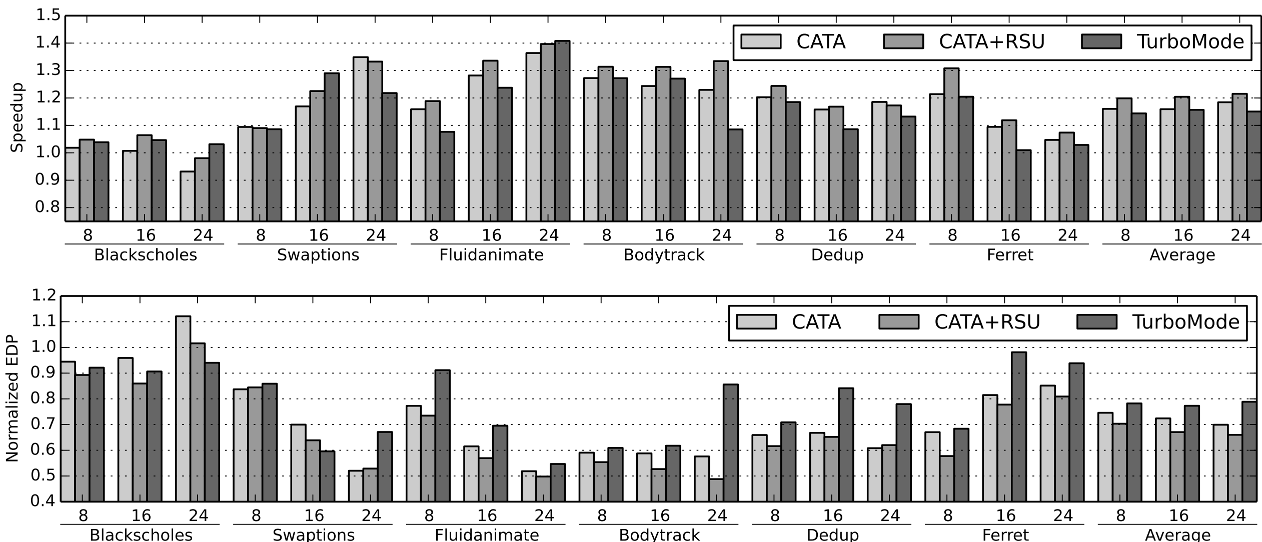

Fig. 5. Speedup and Energy-Delay Product (EDP) results with an increasing number of fast cores (8, 16, 24) on a 32-core processor. Results are normalized to the FIFO scheduler.

CATS, as discussed in Section II-C. Figure 4 shows the performance and EDP improvements that CATA achieves over FIFO. Based on the results in Section V-A, we evaluate CATA using static annotations for criticality estimation.

Results show that CATA achieves average speedups of 15.9% to 18.4% over FIFO, and from 8.2% to 12.7% better than CATS+SA. The main improvements of CATA are obtained in fork-join or stencil applications, in particular Swaptions and Fluidanimate. In these applications, when tasks finish their execution before a synchronization point, CATA reassigns the available power budget to the remaining executing tasks, reducing the load imbalance. In contrast, in Blackscholes the number of tasks is very large and the load imbalance is low. This causes CATA to provide minimal performance benefits and even to present slight slowdowns with 24 fast cores. The slowdown is due to the overhead of frequency reconfigurations. In the applications with pipeline parallelism the performance improvement over CATS is lower, but still CATA obtains noticeable speedups of up to 28% in Bodytrack with 8 fast cores. CATA average improvements in EDP are significant, ranging from 25.4% to 30.1%. These gains are larger than the improvements in execution time as CATA reduces the power consumption of idle cores while it avoids priority inversion and static binding problems. Benchmarks with a large amount of load imbalance such as Swaptions and Fluidanimate dramatically reduce EDP, halving the baseline with 24 fast cores. When a task finishes and there are no other tasks ready to execute, CATA decelerates the core reducing the average number of fast cores decreasing power consumption.

C. Architecturally Supported CATA

Despite the significant performance and power benefits, CATA can be further improved by reducing the overhead of reconfiguring the computational power of the cores. As described in Section III-B, frequency reconfigurations have to be serialized to avoid potentially harmful power states. In CATA this is done using locks and, as a result, it suffers from *reconfiguration serialization* overheads as the number of cores increases. This issue can become a bottleneck when one of the two following conditions hold: i) the amount of time spent performing reconfigurations is significant, or ii) the distribution of reconfigurations over time has a bursty behavior, which is the case in applications with synchronization barriers.

An analysis of the execution of the applications shows that the average reconfiguration latency of CATA ranges from 11 μs to 65 μs. However, maximum lock acquisition times in Blackscholes, Fluidanimate and Bodytrack reach several milliseconds (from 4.8 ms to 15 ms) due to lock contention. Additionally, although the average overhead of the reconfiguration time of the six applications ranges acceptable values of 0.03% to 3.49%, this overhead can be in the critical path and introduce load imbalance, increasing execution time significantly more than this average percentage as a result.

The RSU hardware component introduced in Section III-B speeds up reconfigurations and avoids taking locks as it centralizes all the reconfiguration operations. Figure 5 shows performance and EDP results using CATA, CATA+RSU and also TurboMode, which is discussed in the next section. Results are normalized to the FIFO scheduler to ease the comparison with Figure 4. On average, CATA+RSU further improves the performance of CATA, reaching an average 20.4% improvement over FIFO on a 32-core processor with 16 fast cores (and 3.9% faster than CATA). Performance improvements are most noticeable in applications that suffer lock contention (Blackscholes, Fluidanimate and Bodytrack), reaching an average speedup over CATA of 4.4% on the analyzed applications, significantly reducing the performance degradation shown by Blackscholes with 24 fast cores, and achieving 8.5% speedup over CATA in Bodytrack with 24 fast cores. CATA+RSU reaches a maximum speedup over FIFO of 40.2% in Fluidanimate with 24 fast cores. Regarding the other applications (Swaptions, Dedup and Ferret), the additional

improvements are on average small as lock contention is very low. Observed performance differences are mainly caused by changes in scheduling decisions induced by reconfigurations.

In EDP the average improvements range from 29.7% to 34.0% over FIFO and from 5.6% to 7.4% over CATA. The main reasons behind EDP reduction are the performance improvements and faster reconfigurations. Furthermore, in applications with a high lock contention the EDP reductions against CATA range from 4.0% to 9.4%. This proves the effectiveness of the proposed CATA+RSU and justifies the usefulness of such architectural support.

D. Comparison with Other Proposals

Finally, we compare CATA and CATA+RSU with an implementation of TurboMode [18]. For a fair comparison, our implementation of TurboMode considers the same two frequencies as in the previous experiments, with an overall power budget assigned in terms of maximum number of fast cores. TurboMode is not aware of task criticality, so the base FIFO scheduler is employed and all active cores (in state C_0) are assumed to be running critical tasks. Whenever an accelerated core executes the `halt` instruction triggered by the OS to transition from C_0 to C_1 state, the core notifies the TurboMode controller. The TurboMode controller lowers the frequency of the core, selects a random active core, and accelerates it. When the OS awakes a sleeping core, it notifies the TurboMode controller, and the core is accelerated only if there is enough power budget. Being able to quickly accelerate or decelerate at the C_1 state benefits applications with barriers or short idle loops, which do not need to wait for deeper sleeping states to yield their power budget to other cores.

Figure 5 shows the performance and EDP results of TurboMode. On average, TurboMode obtains slightly worse results than CATA, reaching between 14.4% and 15.7% performance improvements over FIFO. CATA+RSU outperforms TurboMode, with speedups between 4.0% and 5.3% and equivalent hardware cost. TurboMode presents competitive performance with CATA+RSU in fork-join and stencil applications (Blackscholes, Swaptions and Fluidanimate), but at the cost of higher energy consumption. This happens because CATA+RSU reconfigures the frequencies right at the moment that a task finishes its execution, while with TurboMode the reconfiguration must wait until the thread goes to sleep and triggers the `halt` instruction in the OS scheduler. In pipeline applications that overlap different types of tasks (Bodytrack, Dedup and Ferret), TurboMode performs worse than CATA+RSU with performance degradations up to 18.7% in Bodytrack with 24 fast cores. In the case of EDP results, pipeline applications obtain moderate improvements, with average results close to the ones obtained with CATS+SA.

TurboMode significantly improves performance over FIFO as it can solve the *static binding* issue. Since TurboMode is not aware of what is being executed in each core and its corresponding criticality, it may accelerate a non-critical task or runtime idle-loops. In contrast, CATA and CATA+RSU always know what to accelerate, effectively obtaining performance improvements. However, we have observed that TurboMode exhibits some characteristics that our proposals could benefit from. A thread executing a task can suddenly issue a `halt` instruction if the task requires any kernel service that suspends the core for a while; I/O operations, contention on locks that protects the page-fault and memory allocation routines are some examples that we have measured in Swaptions, Dedup and Ferret applications. CATA approaches are not aware of this situation causing the halted core to retain its accelerated state. On the contrary, TurboMode can drive that computing power to any other core that is doing useful work.

VI. Related Work

A. Task-Based Programming Models

All the task-based programming models that have emerged in the last years can benefit from the ideas in this paper. To attain this goal, these programming models require some basic support to identify task criticality. The static annotations approach can be easily adopted in all task-based programming models by incorporating compiler directives or language extensions that allow programmers to express the criticality of the tasks. In contrast, the bottom-level approach [24] is only suitable for task-based programming models that manage the execution of the tasks with a TDG, such as OpenMP 4.0 [10], OmpSs [11], Intel TBB [35], Codelets [12], StarPU [13], task extensions for Cilk [15], and Legion [14], but cannot be applied to task-based programming models that require the programmer to manage the execution of the tasks, either by organizing them manually (like in Sequoia [36]) or adding explicit synchronization between tasks (like in Charm++ [37], Habanero [38]). Provided that task criticality information is present in the runtime system of any task-based programming model, the extensions proposed in this paper can also be incorporated to accelerate critical tasks.

B. Criticality Estimation and Exploitation

The identification of critical code segments of parallel applications has been studied in fork-join programming models, where performance is often conditioned by the slowest thread. Meeting Points [39] instruments programs to monitor the progress of each thread and boosts the execution in delayed cores. However, it only works for balanced programs, in which all threads run the same loop count with similar amount of computation. DCB [40] extends the model to unbalanced loops and to threads with different code in pipeline parallel programs, using an epoch-based approach in the latter case. However, it requires a significant profiling and instrumentation of the application. BIS [41] and Criticality Stacks [42] determine criticality based on hardware counters and idle thread detection, and raise frequency accordingly. Additional proposals migrate threads to fast cores [43], [44] or perform task stealing [45]. These approaches are suitable for traditional fork-join programming models where all threads perform similar computation. However, they are not applicable to task-based programming models where tasks have different criticality according to the task dependence graph.

C. Dynamic Voltage and Frequency Scaling

DVFS exploits slack in the computation to lower voltage and frequency, and save energy. Per-core DVFS is introduced by Donald and Martonosi [1] in the context of thermal management. Fine-grained DVS employing on-chip buck converters for per-core voltage switching is first studied by Kim et al. [46]. A significant drawback of on-chip regulators is their low energy efficiency, especially for low voltages. Dual-rail V_{dd} systems used for near-threshold designs [47], [48], are more energy efficient but only for two possible frequencies. Due to its energy efficiency and low switching latency, we consider dual-rail V_{dd} support in our evaluation, although our proposals can be adapted to other DVFS support as well as other hardware reconfiguration capabilities.

DVFS is typically used to save energy in program phases where running at the highest frequency is not crucial for performance. These phases can be either determined at compile time [49] or tracking the evolution of hardware performance counters at execution time [50], [51]. In this paper we propose a runtime-driven approach that exploits task criticality to determine which program parts can be executed at low frequency.

D. Reconfiguration Techniques

Multiple techniques to exploit heterogeneous architectures and reconfiguration capabilities have been proposed, such as migrating critical sections to fast cores [43], *fusing* cores together when high performance is required for single-threaded code [9], reconfiguring the computational power of the cores [8], *folding* cores or switching them off using *power gating* [4], or using *application heartbeats* to assign cores and adapt the properties of the caches and the TLBs according to the specified real-time performance constraints [52]. Our proposal is independent from these mechanisms, which could be also applied. However, Vega et al. [4] show that multiple power-saving mechanisms (in their study, DVFS and power-gating) can interact in undesired ways whereas a coordinated solution is more efficient. A coordinated solution of CATA and other power-saving mechanisms is left for future work.

VII. Conclusions

Hardware mechanisms that allow reconfiguring the computational capabilities of the system are a common feature in current processors, as they are an effective way to maximize performance under the desired power budget. However, optimally deciding how to reconfigure the hardware is a challenging problem, because it highly depends on the behavior of the workloads and the parallelization strategy used in multi-threaded programs. In task-based programming models, where a runtime system controls the execution of parallel tasks, the criticality of the tasks can be exploited to drive hardware reconfiguration decisions in the runtime system.

This paper presents an integrated solution in which the runtime system of task-based programming models performs Criticality Aware Task Acceleration (CATA). In this approach the runtime system schedules tasks to cores and controls their DVFS settings, accelerating the cores that execute critical tasks and setting the cores that execute non-critical tasks to low-frequency power-efficient states. Since performing DVFS reconfigurations in software can cause performance overheads due to serialization issues, this paper also proposes a hardware component, the Runtime Support Unit (RSU), that relieves the runtime system of carrying out DVFS reconfigurations and can be seen as a minimal extension to existing TurboMode implementations [16]–[18]. With this hardware support, the runtime system informs the RSU of the criticality of the tasks when they are scheduled for execution on a core, and the RSU reconfigures the voltage and the frequency of the cores according to the criticality of the running tasks.

Results show that CATA outperforms existing scheduling approaches for heterogeneous architectures. CATA solves the blind assignment issue of FIFO schedulers that do not exploit task criticality, achieving improvements of up to 18.4% in execution time and 30.1% in EDP. CATA also solves the static binding and priority inversion problems of CATS, which results in speedups of up to 12.7% and improvements of up to 25% in EDP over CATS. When adding architectural support to reduce reconfiguration overhead, CATA+RSU obtains an additional improvement over CATA of 3.9% in execution time and 7.4% in EDP, while it outperforms state-of-the-art TurboMode as it does not take into account task criticality when deciding DVFS reconfigurations.

Acknowledgment

This work has been supported by the Spanish Government (grant SEV2015-0493, SEV-2011-00067 of the Severo Ochoa Program), by the Spanish Ministry of Science and Innovation (contracts TIN2015-65316, TIN2012-34557, TIN2013-46957-C2-2-P), by Generalitat de Catalunya (contracts 2014-SGR-1051 and 2014-SGR-1272), by the RoMoL ERC Advanced Grant (GA 321253) and the European HiPEAC Network of Excellence. The Mont-Blanc project receives funding from the EU's Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 610402 and from the EU's H2020 Framework Programme (H2020/2014-2020) under grant agreement n° 671697. M. Moretó has been partially supported by the Ministry of Economy and Competitiveness under Juan de la Cierva postdoctoral fellowship number JCI-2012-15047. M. Casas is supported by the Secretary for Universities and Research of the Ministry of Economy and Knowledge of the Government of Catalonia and the Cofund programme of the Marie Curie Actions of the 7th R&D Framework Programme of the European Union (Contract 2013 BP_B 00243). E. Castillo has been partially supported by the Spanish Ministry of Education, Culture and Sports under grant FPU2012/2254.

References

[1] J. Donald and M. Martonosi, "Techniques for multicore thermal management: Classification and new exploration," in *ISCA*, 2006, pp. 78–88.

[2] S. Kaxiras and M. Martonosi, "Computer architecture techniques for power-efficiency," *Synthesis Lectures on Computer Architecture*, vol. 3, no. 1, pp. 1–207, 2008.

[3] C. Boneti, F. J. Cazorla, R. Gioiosa, A. Buyuktosunoglu, C.-Y. Cher, and M. Valero, "Software-controlled priority characterization of POWER5 processor," in *ISCA*, 2008, pp. 415–426.

[4] A. Vega, A. Buyuktosunoglu, H. Hanson, P. Bose, and S. Ramani, "Crank it up or dial it down: coordinated multiprocessor frequency and folding control," in *MICRO*, 2013, pp. 210–221.

[5] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *MICRO*, 2006, pp. 423–432.

[6] V. Jiménez, R. Gioiosa, F. J. Cazorla, A. Buyuktosunoglu, P. Bose, and F. P. O'Connell, "Making data prefetch smarter: Adaptive prefetching on POWER7," in *PACT*, 2012, pp. 137–146.

[7] H. Cook, M. Moretó, S. Bird, K. Dao, D. A. Patterson, and K. Asanovic, "A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness," in *ISCA*, 2013, pp. 308–319.

[8] Khubaib, M. A. Suleman, M. Hashemi, C. Wilkerson, and Y. N. Patt, "Morphcore: An energy-efficient microarchitecture for high performance ILP and high throughput TLP," in *MICRO*, 2012, pp. 305–316.

[9] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez, "Core fusion: Accommodating software diversity in chip multiprocessors," in *ISCA*, 2007, pp. 186–197.

[10] "OpenMP architecture review board: Application program interface," 2013.

[11] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "OmpSs: a proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.

[12] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao, "Using a "Codelet" program execution model for exascale machines: Position paper," in *EXADAPT*, 2011, pp. 64–69.

[13] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: A unified platform for task scheduling on heterogeneous multicore architectures," in *Euro-Par*, 2009, pp. 863–874.

[14] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *SC*, 2012, pp. 66:1–66:11.

[15] H. Vandierendonck, G. Tzenakis, and D. S. Nikolopoulos, "A unified scheduler for recursive and task dataflow parallelism," in *PACT*, 2011, pp. 1–11.

[16] "Intel Turbo Boost Technology in Intel Core Microarchitecture (Nehalem) Based Processors," 2008.

[17] AMD, "The new AMD Opteron processor core technology," Tech. Rep., 2011.

[18] D. Lo and C. Kozyrakis, "Dynamic management of TurboMode in modern multi-core chips," in *HPCA*, 2014, pp. 603–613.

[19] M. Valero, M. Moretó, M. Casas, E. Ayguadé, and J. Labarta, "Runtime-aware architectures: A first approach," *International Journal on Supercomputing Frontiers and Innovations*, vol. 1, no. 1, pp. 29–44, Jun. 2014.

[20] M. Casas, M. Moretó, L. Alvarez, E. Castillo, D. Chasapis, T. Hayes, L. Jaulmes, O. Palomar, O. Unsal, A. Cristal, E. Ayguade, J. Labarta, and M. Valero, "Runtime-aware architectures," in *Euro-Par*, 2015, pp. 16–27.

[21] V. Papaefstathiou, M. G. Katevenis, D. S. Nikolopoulos, and D. Pnevmatikatos, "Prefetching and cache management using task lifetimes," in *ICS*, 2013, pp. 325–334.

[22] M. Manivannan, A. Negi, and P. Stenström, "Efficient forwarding of producer-consumer data in task-based programs," in *ICPP*, 2013, pp. 517–522.

[23] M. Manivannan and P. Stenstrom, "Runtime-guided cache coherence optimizations in multi-core architectures," in *IPDPS*, 2014, pp. 625–636.

[24] K. Chronaki, A. Rico, R. M. Badia, E. Ayguade, J. Labarta, and M. Valero, "Criticality-aware dynamic task scheduling on heterogeneous architectures," in *ICS*, 2015.

[25] T. Miller, R. Thomas, and R. Teodorescu, "Mitigating the effects of process variation in ultra-low voltage chip multiprocessors using dual supply voltages and half-speed units," *Computer Architecture Letters*, vol. 11, no. 2, pp. 45–48, July 2012.

[26] S. Kumar, C. J. Hughes, and A. D. Nguyen, "Carbon: architectural support for fine-grained parallelism on chip multiprocessors," in *ISCA*, 2007, pp. 162–173.

[27] Y. Etsion, F. Cabarcas, A. Rico, A. Ramírez, R. M. Badia, E. Ayguadé, J. Labarta, and M. Valero, "Task superscalar: An out-of-order task pipeline," in *MICRO*, 2010, pp. 89–100.

[28] D. Sanchez, R. M. Yoo, and C. Kozyrakis, "Flexible Architectural Support for Fine-Grain Scheduling," in *ASPLOS*, 2010, pp. 311–322.

[29] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP Laboratories*, 2009.

[30] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[31] V. Spiliopoulos, A. Bagdia, A. Hansson, P. Aldworth, and S. Kaxiras, "Introducing DVFS-management in a full-system simulator," in *MASCOTS*, 2013, pp. 535–545.

[32] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO*, 2009, pp. 469–480.

[33] D. Chasapis, M. Casas, M. Moreto, R. Vidal, E. Ayguade, J. Labarta, and M. Valero, "PARSECSs: Evaluating the impact of task parallelism in the PARSEC benchmark suite," *ACM TACO*, pp. 1:1–1:25, 2015.

[34] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *PACT*, 2008, pp. 72–81.

[35] J. Reinders, *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007.

[36] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: programming the memory hierarchy," in *SC*, 2006.

[37] L. V. Kalé and S. Krishnan, "CHARM++: A portable concurrent object oriented system based on C++," in *OOPSLA*, 1993, pp. 91–108.

[38] J. Shirako, J. M. Zhao, V. K. Nandivada, and V. N. Sarkar, "Chunking parallel loops in the presence of synchronization," in *ICS*, 2009, pp. 181–192.

[39] Q. Cai, J. González, R. Rakvic, G. Magklis, P. Chaparro, and A. González, "Meeting points: Using thread criticality to adapt multicore hardware to parallel regions," in *PACT*, 2008, pp. 240–249.

[40] H. K. Cho and S. Mahlke, "Embracing heterogeneity with dynamic core boosting," in *CF*, 2014, pp. 10:1–10:10.

[41] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt, "Bottleneck identification and scheduling in multithreaded applications," in *ASPLOS*, 2012, pp. 223–234.

[42] K. Du Bois, S. Eyerman, J. B. Sartor, and L. Eeckhout, "Criticality stacks: Identifying critical threads in parallel programs using synchronization behavior," in *ISCA*, 2013, pp. 511–522.

[43] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt, "Accelerating critical section execution with asymmetric multi-core architectures," in *ASPLOS*, 2009, pp. 253–264.

[44] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt, "Utility-based acceleration of multithreaded applications on asymmetric CMPs," in *ISCA*, 2013, pp. 154–165.

[45] A. Bhattacharjee and M. Martonosi, "Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors," in *ISCA*, 2009, pp. 290–301.

[46] W. Kim, M. S. Gupta, G.-Y. Wei, and D. Brooks, "System level analysis of fast, per-core DVFS using on-chip switching regulators," in *HPCA*, 2008, pp. 123–134.

[47] T. Miller, X. Pan, R. Thomas, N. Sedaghati, and R. Teodorescu, "Booster: Reactive core acceleration for mitigating the effects of process variation and application imbalance in low-voltage chips," in *HPCA*, 2012, pp. 1–12.

[48] R. G. Dreslinski, B. Giridhar, N. Pinckney, D. Blaauw, D. Sylvester, and T. Mudge, "Reevaluating fast dual-voltage power rail switching circuitry," in *WDDD*, 2012.

[49] C.-H. Hsu and U. Kremer, "The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction," in *PLDI*, 2003, pp. 38–48.

[50] G. Keramidas, V. Spiliopoulos, and S. Kaxiras, "Interval-based models for run-time DVFS orchestration in superscalar processors," in *Comput. Front.*, 2010, pp. 287–296.

[51] S. Eyerman and L. Eeckhout, "Fine-grained DVFS using on-chip regulators," *ACM TACO*, vol. 8, no. 1, pp. 1:1–1:24, Feb. 2011.

[52] H. Hoffmann, J. Eastep, M. D. Santambrogio, J. E. Miller, and A. Agarwal, "Application heartbeats: A generic interface for specifying program performance and goals in autonomous computing environments," in *ICAC*, 2010, pp. 79–88.