Scientific Programming 22 (2014) 223–237
DOI 10.3233/SPR-140389
IOS Press

TACO: A scheduling scheme for parallel applications on multicore architectures

Jan H. Schönherr [a,*], Ben Juurlink [b] and Jan Richling [a]
[a] *Communication and Operating Systems, Technische Universität Berlin, Berlin, Germany*
E-mails: {schnhrr, jan.richling}@tu-berlin.de
[b] *Embedded Systems Architecture, Technische Universität Berlin, Berlin, Germany*
E-mail: b.juurlink@tu-berlin.de

Abstract. While multicore architectures are used in the whole product range from server systems to handheld computers, the deployed software still undergoes the slow transition from sequential to parallel. This transition, however, is gaining more and more momentum due to the increased availability of more sophisticated parallel programming environments. Combined with the ever increasing complexity of multicore architectures, this results in a scheduling problem that is different from what it has been, because concurrently executing parallel programs and features such as non-uniform memory access, shared caches, or simultaneous multithreading have to be considered.

In this paper, we compare different ways of scheduling multiple parallel applications on multicore architectures. Due to emerging parallel programming environments, we primarily consider applications where the parallelism degree can be changed on the fly. We propose TACO, a topology-aware scheduling scheme that combines equipartitioning and coscheduling, which does not suffer from the drawbacks of the individual concepts. Additionally, TACO is conceptually compatible with contention-aware scheduling strategies. We find that topology-awareness increases performance for all evaluated workloads. The combination with coscheduling is more sensitive towards the executed workloads and NUMA effects. However, the gained versatility allows new use cases to be explored, which were not possible before.

Keywords: Coscheduling, equipartitioning, multicore, topology-aware

1. Introduction

Since the availability of multicore systems to the mass market, the development of parallel applications has changed. Gone are the days where developers themselves have to care about the creation and management of threads. Instead, a lot of expertise has gone into the creation of more advanced parallel programming environments, which relieve today's programmers from some of the more mundane tasks, such as *OpenMP* [11] or the *Intel Threading Building Blocks* [13]. These parallel programming environments boost the number of available parallel applications, as they open the field also to those developers that lack some of the expertise required otherwise. Additionally, the environments enforce sensible parallelizations to some degree, avoiding common beginner mistakes. Often, the resulting applications are moldable or malleable as defined by Feitelson and Rudolph [5]: the degree of parallelism of a moldable application can be specified at startup, making it portable to some extend, while a malleable application also allows reconfigurations at runtime. While parallel application development has evolved, schedulers of current operating systems have not – at least not with respect to the execution of parallel applications. Here, schedulers still consider every thread of a parallel application on its own; and parallel programming environments have to cope with that.

Research on scheduling of multiple parallel applications basically suggests two approaches based on partitioning: partitioning in space and partitioning in time. While the former is usually just called *partitioning*, the terms *coscheduling* [12] and *gang scheduling* [4] were coined for the latter. With partitioning, each application is assigned to a different set of processing elements within a parallel system; coscheduling uses coordinated context switches to switch at ap-

*Corresponding author: Jan H. Schönherr, Communication and Operating Systems, Technische Universität Berlin, Einsteinufer 17 EN6, 10587 Berlin, Germany. Tel.: +49 30 314 79833; Fax: +49 30 314 25156; E-mail: schnhrr@tu-berlin.de.

plication level instead of thread level. Both approaches assign computational resources at application level – a fact that applications can take advantage of at design time: low latency communication is possible, busy waiting and static load balancing can be used. An application can also exploit that it gets exclusive access to resources that are closely associated with the computational resources (e.g., shared cache in a multicore processor). In short, partitioning schemes allow to apply many optimizations techniques that are widespread in, e.g., the HPC area. It is important to note that software, which is optimized based on certain assumptions, might experience extreme slowdowns when those assumptions are not correct: busy waiting without simultaneous execution is probably the most disastrous combination, followed by static load balancing without threads making uniform progress, or algorithms optimized towards a certain cache size without exclusively assigned caches.

In the absence of other objectives, such as job importance or job scalability, it is normally desired to distribute the available CPU time between multiple parallel programs in a fair manner, giving each program an equal share. In the past, this has led to *equipartitioning* [16]: the available computational resources are split in space evenly between the running jobs. Whenever a new job arrives or a running job terminates, the mapping of jobs to processors is reorganized. This, however, requires parallel applications to be malleable in order to maintain a high efficiency. Such adaptations of the degree of parallelism within applications usually do not happen instantaneously. Instead, we see a temporary over- or undersubscription of the system during phases where a change request has already been issued but the application has not yet responded to it. Coscheduling, on the other hand, has no reconfiguration overhead making job arrivals and terminations very cheap. Instead, we get overhead from time-sharing and reduced efficiency due to normally sub-linear speedups.

In this paper, we adapt the concept of equipartitioning to contemporary multicore systems. The result is TACO, a topology-aware coscheduler: topology-awareness, i.e., respecting the organization of hardware components, retains advantages associated with partitioning, while the combination with coscheduling avoids drawbacks normally associated with traditional equipartitioning. In particular, TACO avoids frequent reconfigurations and achieves a fair distribution of CPU time, even in presence of applications that must or should be executed at a different degree of parallelism than indicated by traditional equipartitioning. Furthermore, we address the issue of resource contention on multicore architectures and outline a method to combine existing contention-aware scheduling strategies with TACO, essentially making them fit for parallel programs – or our approach contention-aware, depending on the point of view.

The remainder of this paper is structured as follows: In Section 2, we give a detailed description of our approach. Its significance in the context of resource contention is analyzed in Section 3. Afterwards, our approach is evaluated and compared to other established approaches in Section 4. Section 5 reviews other work in the area, and we conclude our paper in Section 6.

2. TACO: Topology-Aware COscheduling

In this section we describe our approach in detail. We start with the basic idea and develop several variants from that, which are tailored towards specific needs.

Both, partitioning and coscheduling, give a parallel application the illusion of being the only application in the system while – unlike batch processing – allowing multiple applications to make progress. With partitioning, applications see a system that is smaller than the real one; with coscheduling, applications see the whole system but not for the whole time. As outlined in Section 1, both techniques allow to apply a wide range of optimizations within applications. From the vantage point of the system, partitioning causes less overhead than coscheduling: there are less context switches and there is no need to achieve a simultaneous context switch across multiple CPUs which usually does not scale. Additionally, running a job with less processors usually increases its efficiency due to avoided parallel overhead. However, partitioning and especially equipartitioning can be problematic for applications when there are dependencies between partitions that can cause performance asymmetries or fluctuations within a partition. For instance, applying partitioning at the granularity of individual processor cores results in multiple jobs sharing the same processor or one job being involuntarily spread out over multiple NUMA nodes. This makes certain optimizations (e.g. optimizations towards cache utilization) futile. Another example are SMT siblings mapped to different partitions. Here, any static load balancing at application level is void due to unpredictable delays caused by contention for execution units. Doing solely coarse-

grained partitioning, e.g., at the level of NUMA nodes, allows more optimizations within a program, but results in a balancing problem for equipartitioning.

2.1. Basis

In order to create a scheduling scheme for modern multicore architectures, we combine the ideas of equipartitioning and coscheduling.

Modern parallel systems are not symmetrical in the sense that arbitrary pairs of processor cores always have the same behavior regarding resource sharing and communication overhead. Instead, the topology of a machine defines sets of cores that are closer than others. Therefore, we propose to create partitions obeying these borders in order to minimize influences between applications. More precisely, a partition is either equivalent to a unit defined by the hardware topology or a reasonable sized fraction of it. For example, we never create partitions that span one-and-a-half NUMA nodes. Instead, possible partitions are a NUMA node, a fraction of such a node (e.g., half a node), or a fraction of the next higher layer of the topology (e.g., half a system, which would be two nodes in case of a four node system). Hence, every partition has a homogeneous topology itself (given that the system topology is homogeneous). This gives applications an environment for which optimization is already established.

In order to achieve fairness, we only use identically shaped partitions for all applications at first. In order to deal with varying numbers of applications, we adapt the granularity of our partitions dynamically. Due to the topology-awareness and the identical shape requirement, there are only relatively few possible partition shapes. When there are more applications than partitions of a certain size – but not yet so much that it makes sense to use the next smaller partition size – we use coscheduling to schedule multiple applications in the same place.

Compared to traditional equipartitioning which requires to adjust partition sizes with each job arrival or termination, and thus resizing and (depending on the implementation) migrating applications quite often, this is not necessary with our approach. We only perform this readjustment when certain thresholds are crossed. To avoid frequent reconfigurations in case that the number of applications is around the threshold, a short-term hysteresis can be added. Consider, for example, the system given in Fig. 1. When the groups are currently at the processor level, we switch to the system level, as soon as at least one processor has nothing to do. However, to switch from system to processor level, it can be sensible to require more than two jobs. This logic is captured in Listing 1. Please note, that the balancing logic is separate from the partitioning logic.

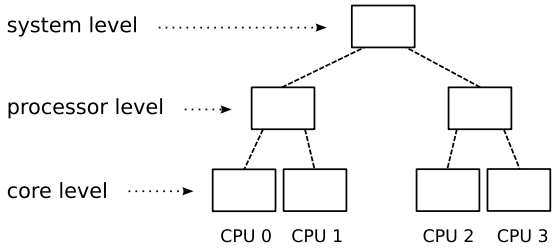

Fig. 1. Hierarchy levels of a dual-socket dual-core system.

```
int apps = 0;
int level = SYSTEM;

on_start(app a) {
    apps++;
    if (upper_threshold_reached(level, apps)) {
        level++;
        repartition_all_apps(level);
    } else {
        set_partition(a, level);
    }
}
on_terminate() {
    apps--;
    if (lower_threshold_reached(level, apps)) {
        level--;
        repartition_all_apps(level);
    }
}
```

Listing 1. Basic partitioning logic; balancing of partitions is handled separately.

It is also possible to replace the global decision of switching levels with local decisions: as soon as a node in the hierarchy has accumulated enough jobs, it switches to the next lower level; if a node has nothing to do, despite balancing, its siblings return their jobs to their parent. This spreads out reconfigurations, but results in more reconfigurations over time and creates imbalances in the CPU time distribution that cannot be addressed by rebalancing alone.

2.2. Reaching an equilibrium

Despite having only equally shaped partitions, our approach so far is still subject to load imbalances, when the number of coscheduled applications per partition differs. One way to solve this is to use a periodic rebalancing to even out this imbalance as sug-

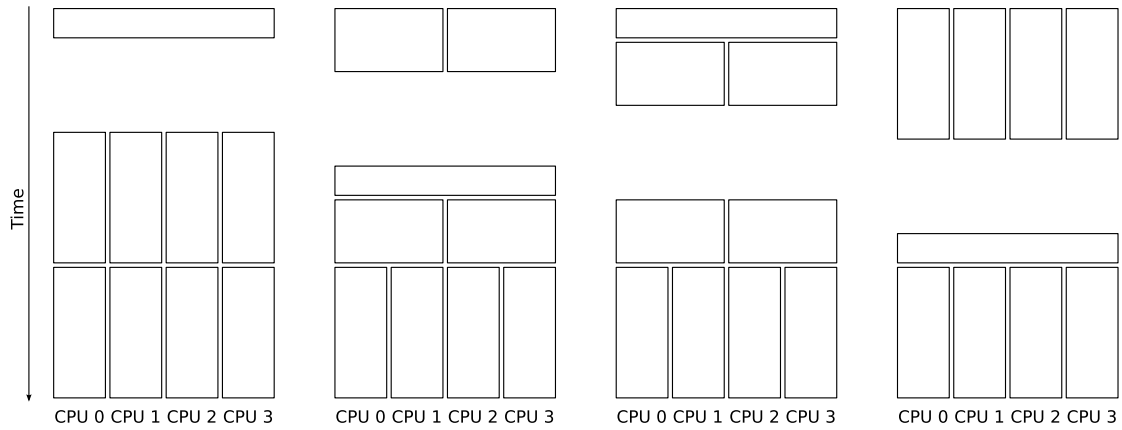

Fig. 2. Schedules generated by our approach for one to eight tasks on the example dual-socket dual-core system.

gested in [9]. We propose a different way to achieve uniform progress across all applications by again employing coscheduling and to coschedule multiple hierarchy levels. That is, we ensure that each partition in the current hierarchy level is equally loaded, and place odd elements in higher hierarchy levels. The scheduler alternates execution between different hierarchy levels, weighting each hierarchy level so that each job receives the same amount of CPU time. With scalable applications, this leads to a fair distribution of the available resources.

Figure 2 visualizes the resulting schedules for our example system from one to eight jobs. The figure shows no hysteresis, instead it is assumed that the state had time to settle after each reconfiguration. As this technique effectively reduces the number of coscheduled applications that execute in a partition, it also removes the buffer against reconfigurations, which our approach created in the first place. Hence, in order to avoid continuous reconfigurations, this kind of balancing is only feasible in more or less stable situations.

2.3. Incorporating reconfiguration delays

With traditional equipartitioning, every job arrival or termination leads to a short moment of over- or undersubscription, respectively, until the other jobs eventually adapt their degree of parallelism. Especially oversubscription can cause severe delays if applications depend on being coscheduled. With explicitly assigned partitions as realized by our basic approach, the consequences of oversubscription are kept per application, penalizing slow adapting applications.

However, the aforementioned coscheduling of hierarchy levels allows to handle reconfigurations of applications more gracefully. Instead of the operating system enforcing a new partition size immediately after issuing a change request to an application, applications have a grace period to react during which the old partition size is kept. When the application eventually adapts, the operating system scheduler can react immediately.

This helps reconfigurations in both directions. In case of a reconfiguration towards a larger partition, we avoid assigning processor cores to a partition that will not be used until the application actually reconfigures itself. In case of a reconfiguration towards a smaller partition, we avoid oversubscribing that partition and, with that, we neither create a balancing issue within the application nor do we violate the coscheduling property, which an application might rely on for efficient behavior. Again, by weighting hierarchy levels accordingly, we can realize a fair distribution of CPU time.

2.4. Being less restrictive

If we have additional information about individual applications, such as maximal or minimal parallelism degrees or knowledge on which assumptions about the system an application relies exactly, our approach can be easily fine-tuned. For instance, not every application really profits from being coscheduled, but still, avoiding oversubscription is beneficial. Thus, we can enable and disable coscheduling on a per application basis, allowing to execute threads of different non-coscheduled applications within the same partition in an uncoordinated way.

Due to the coscheduling of different hierarchical levels, it is no problem to accommodate applications with special needs regarding the parallelism degree. If

desired, they can be weighted appropriately, so that the amount of received CPU time is fair compared to other applications. Especially, the integration of moldable or evolving jobs is not problematic.

Furthermore, the knowledge on the characteristics of the individual applications can be used to optimize the assignment of applications to partitions. In the simplest case the application is characterized by its speedup behavior, so that applications with a near linear speedups get wider partitions (more cores) than applications with worse speedup behavior. However, as we show in Section 3.2, this usually ignores the impact of resource contention. Therefore, we propose to incorporate knowledge on the behavior of the application with respect to resource contention as discussed in Section 3.

3. Coscheduling and resource contention

Contrary to, e.g., nodes in a cluster, current multicore architectures share a multitude of resources between different processor cores. For instance, cores might share memory bandwidth, some caches, or even execution units. Hence, resource contention has become a problem. At the operating system level, research suggests to tackle this problem with a modified scheduler, which somehow takes care of dependencies between different processor cores and, thus, generates better schedules with hopefully reduced contention (see [17] for a survey). Those approaches that are able to handle more tasks than processor cores (e.g., [2,10]), require a coordination in time to make sure that certain tasks are executed simultaneously. This, essentially, is a form of coscheduling.

At first glance, these two reasons to employ coscheduling – (i) to improve and simplify parallel programs and (ii) to reduce resource contention – are in conflict with each other: because the former does not really consider resource consumption and the latter is typically unaware of dependencies between multiple tasks, they might decide on incompatible schedules. In an ideal world, the scheduling algorithm would be able to take care of both. It would determine whether it is beneficial for an application to be coscheduled or if it only needs a subset of the very strong guarantee of coscheduling. However, with scheduling being NP-hard, an efficient solution for the combined case without resorting to too much a priori knowledge is unlikely to be found.

In this section, we suggest a heuristic, which allows us to consider coscheduling of parallel applications (TACO's primary area of expertise) and the optimization of a schedule in order to reduce resource contention as mostly orthogonal problems, so that they can be solved independently. Additionally, we discuss resource contention issues that directly affect TACO – namely, we discuss the inadequateness of the speedup measure when faced with potential resource contention and we determine boundaries for the length of a time slice for coscheduling.

3.1. Combining coschedulers

A contention-aware scheduler has to find an efficient schedule out of a vast number of possible schedules. Coscheduling, when provided to parallel applications by the operating system as an instrument, helps by reducing the number of possible schedules due to the imposed constraints. Unfortunately, it is normally still a rather high number and nearly no contention-aware scheduler supports parallel programs.

Instead of developing a new contention-aware scheduling algorithm that can handle parallel programs, we suggest to impose an additional restriction on permitted schedules, which allows us to reuse existing algorithms with only minor modifications. This restriction makes it possible to split the scheduling decision into an outer scheduling decision – which parallel programs should be executed simultaneously – and an inner scheduling decision – which task should be executed on which processor core within a parallel program. For both, we can employ existing contention-aware algorithms.

Contention-aware scheduling algorithms usually gather resource statistics during runtime and base their decision on that. For the outer scheduling decision to work, this requires that statistics are not gathered on a per task or per CPU basis, but on a per application basis. This, on the other hand, requires those statistics to be somewhat reliable and not to fluctuate by design. Thus, we only allow parallel programs to execute in partitions that are identically shaped with respect to shared resources. This restriction also helps the inner scheduling decision, which has now a constant base to work on, where it can determine the optimal placement of tasks of a parallel program within such a partition. In order to fully decouple the inner and outer decision, the allowed partitions must also be aligned to the system topology, i.e., a partition can only be a full topological unit or a fraction thereof. Otherwise, the

outer decision would still depend on the inner mapping and vice versa. Consider for instance a NUMA system with two quad-core processors and a parallel application with two threads, one is memory bound, the other compute bound. We can create partitions of size two in two ways: (i) using two cores of one processor, or (ii) using one core of each processor. Only the former allows a contention-aware outer placement of two (or more) instances of the parallel application without knowledge of the inner mappings.

The most obvious case that our proposed heuristic can not handle is one memory intensive application together with multiple compute intensive applications. In that case, the most efficient schedule is probably to spread out the memory intensive application over all NUMA domains and fill the free cores within each domain with compute intensive applications. It is debatable, whether this is really a relevant case, as the need to spread out execution is reduced as soon as there is more than one memory intensive parallel program. Furthermore, several optimization techniques for parallel programs heavily rely on non-interference (e.g., cache size optimizations and static load balancing), which is impossible to achieve while such a spread out application is run. Therefore, we currently believe that supporting such a scenario is not worth the effort. However, further research in that direction should be done.

TACO already fulfills the necessary conditions that allow an easy integration of contention-awareness, and plan to combine these two areas of research in our future work.

3.2. Effective speedup

When working with parallel applications, their scalability is always an issue. Not every problem is suited to be run with all available processor cores in a machine. On the one hand, there is Amdahl's Law; on the other hand, there is overhead within an application to support more processors. In the end, we have a trade off between gained processing power and additional overhead to handle that power. With information about the scalability of parallel programs, scheduling algorithms can form more sophisticated scheduling decisions, assigning each program a convenient number of processors to, e.g., improve throughput or energy efficiency. The traditional measure for this is speedup: the ratio of sequential execution time to parallel execution time for a given number of processors. Based on known speedup functions, it is simple to decide on questions such as which parallel program should get an additional processor.

Speedup is usually measured under ideal conditions, i.e., no other programs are executed concurrently. Thus, resource contention is avoided – at least with respect to other applications and especially in the case of determining the sequential execution time. In the latter case, the program is executed on one processor of the machine in an isolated manner, i.e., no other load on other processors. This makes speedup and algorithms based on this measure unsuitable for current multicore architectures, because – usually – there is no isolated execution: if some processor cores are not assigned to a certain program, they get assigned to another one. Thus, the selection of partition sizes must become contention-aware.

To this effect, we propose to use *effective speedup* in algorithms, which is (conceptually) measured when all remaining cores of the machine are used by other programs. Obviously, this effective speedup is not simple to determine as resource contention heavily depends on the type of load on other cores, but it is possible to establish worst case boundaries by using "resource eaters" as load – programs that are designed for fully utilizing shared resources. The effective speedup that is actually realized in the face of resource contention is better than (or at least equal to) the speedup realized without interference of other applications – sometimes even up to super linear speedups. Of course, the parallel execution with all processor cores does not get faster – the cases with fewer processors just get slower.

As an example, consider a machine with p cores that has to execute p instances of the same parallel program. What is the ideal number of threads each program should create? One, p, or something in between? The traditional speedup is usually not quite linear and efficiency decreases with more processors, thus, one thread is normally a good answer. However, its baseline – sequential execution – is derived by executing the program with one thread on an otherwise idle system. The effective speedup uses a realistic baseline instead. In our example, we can even measure it by executing p copies with one thread each. With this, the answer to the previous question is no longer obvious: the more processor cores an application uses, the less resource contention it experiences.

Table 1 shows measurements for the above example taken with the benchmarks and evaluation system described in detail in Section 4.1. The machine in question has four processors with six cores each and we only used partition shapes and sizes suggested by our

Table 1
Traditional versus effective speedup of NAS benchmarks on our evaluation system for the described scenario

Benchmark with partition size	Traditional speedup					Effective speedup				
	2	3	6	12	24	2	3	6	12	24
`bt.A`	1.8	2.5	3.8	6.3	13.4	1.9	2.5	4.6	7.6	17.0
`cg.B`	1.8	2.5	2.9	5.5	9.2	2.6	**3.9**	7.8	14.6	24.6
`ft.B`	1.7	2.3	3.1	5.9	11.7	1.9	2.9	5.7	10.9	19.7
`is.C`	1.9	2.8	4.7	7.8	12.2	1.9	2.9	5.9	9.9	15.1
`lu.A`	1.8	2.4	3.7	6.2	12.9	2.9	8.8	**18.7**	32.8	64.0
`mg.B`	1.2	1.3	1.1	2.2	4.3	2.0	2.9	5.9	11.6	22.1
`sp.A`	1.5	1.7	1.8	3.3	7.0	1.9	2.9	5.5	9. 8	19.2
`ua.A`	1.6	1.9	2.7	5.8	15.9	1.9	2.8	6.9	15.1	**38.3**

approach. To measure the effective speedup, we started as many instances of a benchmark as necessary to fill the machine with the given partitions, e.g., eight instances with a partition size of three. If the best partition size is different from one, it is emphasized. The most extreme result is `lu.A` with an effective speedup of 18.7 for a partition size of six. That is, in our scenario with 24 instances, using coscheduling and a partition size of six would finish roughly three times faster than using only one thread per instance.

In a nutshell, the idea of effective speedup promotes the use of coscheduled, larger partitions in certain situations, even when partitioning in space would also work. With respect to TACO, effective speedup can be used for online tuning of the hysteresis guiding transitions between hierarchy levels in Section 2.1 and for the selection of odd elements for ideal load balancing in Section 2.2. While we do not realize this approach in our evaluation for this paper, we certainly plan to incorporate these ideas in our future work.

3.3. Time slice length

Coscheduling – when compared to partitioning in space – has additional overhead due to context switches necessary to realize time-sharing. The (reasonable) length of a time slice is closely related to this overhead: the more overhead is associated with a (collective) context switch, the longer the time slice should be to keep the overhead at a reasonable level. On the other hand, with longer time slices, interactive behavior suffers. So, what is a good time slice length for coscheduling on current multicore architectures, and is there a difference to sequential programs or parallel but non-coscheduled applications?

Beside the direct overhead caused by execution of the operating system context switch code, there is indirect overhead caused by cache misses that would not have occurred without a context switch. According to [8], the latter can be further split into cache misses that happened while other tasks were running and cache misses that happen although the original application is already running again (because of an upset LRU order). While that analysis was for uniprocessor systems, it can be applied to coscheduled workloads as well, assuming that coscheduling is used to avoid cache sharing between different applications and that tasks of an application use the cache collectively. If multiple tasks use the (shared) cache competitively at the same time, it gets more complicated: depending on individual memory access patterns and access frequencies, each task sees an effectively smaller cache.

Qualitatively, we can make the following observations: With large time slices, it is unlikely that there is reusable content in the cache. So, every time an application is scheduled, it has to refill the cache. With smaller time slices, the likelihood of cache hits after a context switch increases. But if many applications are scheduled in between, cache hits become again less likely. Very small time slices would show a behavior similar to partitioning in space if not for the overhead caused by the context switch code itself.

Interestingly, it is easy to estimate worst case delays that an application might experience based on cache size, cache/memory bandwidth and latency. In the worst case, the whole cache has to be refilled. This can be done either with independent or dependent accesses. Dependent accesses are latency limited and, hence, realize a lower bandwidth. However, this bandwidth is per core and all cores are able to work in parallel (given a somewhat sensible parallel program). Depending on the system, all cores together might or might not saturate the available memory bandwidth. Our evaluation system, for instance, takes roughly a

millisecond to overwrite its (shared) last level cache completely, regardless of which access method is used (and assuming that the memory on the local NUMA node is accessed). Of course, this delay is normally neither realized fully, nor realized directly after being scheduled – instead it is likely spread out over time.

Thus, in order to keep cache related overhead due to coscheduling (compared to partitioning in space, not uncoordinated time-sharing) to a reasonable level, say around one percent, we need time slices of about 50 ms to 100 ms on a machine behaving similar to our evaluation system.

Trying to guarantee that there will still be data in the cache when an application is rescheduled, would require time slices of only a fraction of a millisecond. But then, we would hurt performance because of repeated refillings of higher level caches. Additionally, the execution time of the context switch code – which is in the single-digit microsecond range – is no longer negligible.

4. Evaluation

In order to prove the applicability of TACO, we compare two variants of it to several standard approaches. We use randomly generated workloads stressing the malleability of tasks. Our criteria for the effectiveness of an approach are the realized response time of a task compared to its isolated execution, the overall makespan, and the number of reconfigurations. Our workload is described in detail in Section 4.1, followed by a description of all tested approaches in Section 4.2. Our evaluation closes with the presentation and discussion of the results in Sections 4.3 and 4.4.

4.1. Workload and evaluation system

The evaluated workloads are composed of several OpenMP applications taken from the NAS Parallel Benchmarks 3.3 described by Bailey et al. [1,6] and developed by Jin et al. [7]. We only selected short running benchmarks (around one to three minutes when executed sequentially) that are able to adapt the degree of parallelism at runtime, i.e., they repeatedly enter and exit parallel regions. Classifying these benchmarks according to their reconfiguration delay, we have fast adapting benchmarks (`bt.A`, `mg.B`, `sp.A` and `ua.A`) and slow adapting benchmarks (`cg.B`, `ft.B` and `is.C`). Benchmark `lu.A` is somewhat of a special case, as it is the only one that uses active waiting at application level. Table 2 gives more details about these benchmarks. All time related measurements in that table were obtained in absence of other interference. Thus, they are not valid when, e.g., memory bandwidth is shared with other applications, but they give a rough idea of the characteristics.

A workload consists of a selection of benchmarks with exponentially distributed inter-arrival times. That is, the jobs arrivals constitute a Poisson process. The benchmarks and their start times are randomly selected for each workload. Our evaluation infrastructure then allows to replay a certain workload over and over again. Thus, we can feed different scheduling approaches with identical workloads.

Our evaluation system is a quad AMD Opteron 8435, a NUMA system with four six-core 45 nm K10 processors (codename Istanbul) clocked at 2.6 GHz. It has 64 GB RAM (DDR2-533, 16 GB per NUMA domain) and runs Linux 3.8 with NUMA memory balancing enabled. The used version of the GNU Compiler Collection – and thus also of GNU OpenMP – is 4.7.2.

Table 2
NAS benchmarks used for evaluation and their characteristics

Benchmark	Description	Sequential exec. time (s)	Speedup on partitions of size 2, 3, 6, 12 and 24	Reconfigurations (parallel regions)	Avg. reconf. delay (when run sequentially) (ms)
`bt.A`	Block tridiagonal	94	1.8, 2.5, 3.8, 6.3, 13.4	1012	46
`cg.B`	Conjugate gradient	169	1.8, 2.5, 2.9, 5.5, 9.2	231	365
`ft.B`	Fast Fourier transform	82	1.7, 2.3, 3.1, 5.9, 11.7	112	365
`is.C`	Integer sort	52	1.9, 2.8, 4.7, 7.8, 12.2	16	1627
`lu.A`	Lower–upper symmetric Gauss–Seidel	75	1.8, 2.4, 3.7, 6.2, 12.9	518	73
`mg.B`	Multi grid	13	1.2, 1.3, 1.1, 2.2, 4.3	1281	5
`sp.A`	Scalar pentadiagonal	71	1.5, 1.7, 1.8, 3.3, 7.0	3616	10
`ua.A`	Unstructured adaptive	68	1.6, 1.9, 2.7, 5.8, 15.9	36,510	1

4.2. Considered approaches

We consider six different approaches. The first two, *Uncontrolled Execution* and *Load-adaptive Execution*, are readily available on today's systems as they do not need additional support from the operating system: all decisions are made locally by the applications themselves. They give us the off-the-shelf baseline. *Standard Equipartitioning* and *Batch Processing*, on the other hand, are established approaches that require additional support. They form the conceptual baseline. Finally, we have *Topology-aware Equipartitioning* (TACO without coscheduling) and TACO itself.

Uncontrolled Execution (*UE*, *UEp*). This is probably the variant that is most often used today. Each application just considers itself, and the operating system is not aware of parallel or malleable applications. Thus, every application does what it wants and is not hindered by the operating system. In case of OpenMP applications, each application usually spawns as many worker threads as there are CPUs.

GNU OpenMP allows the user to select from three different waiting policies: passive waiting, spin-blocking (the default), and active waiting. In our experiments, we used spin-blocking (UE) and passive waiting (UEp). With the former we get applications that assume exclusive system access, while the latter sacrifices single application performance for overall throughput.

Load-adaptive Execution (*LA*, *LAp*). Another standard approach. The operating system is still not aware of parallel applications, but at least applications now recognize the fact that they do not own the system. Instead, they regularly poll the system load and adapt their own degree of parallelism. GNU OpenMP supports this style of execution when `OMP_DYNAMIC` is set. However, adaptations only happen when a parallel region is entered. Thus, it heavily depends on the program itself how often these adaptations take place.

In addition to that, achieved efficiency and fairness also depends on the load adjustment implementation and whether it uses additional sources of information. For instance, the load itself does not carry information about the number of concurrently running applications. Further, system load is typically adjusted only in terms of seconds; thus, it is not possible to react appropriately fast to thread creations and destructions. The load adaptation of GNU OpenMP is rather primitive, sizing the next parallel region to fill the free capacity according to the 15 min load average. Here, we also tested spin-blocking (LA) and passive waiting (LAp).

Equipartitioning (*EQ*, *EQi*). While not supported by current operating systems, we applied the basic idea of equipartitioning without further consideration of machine topology or other factors. That is, we simply divide the available processor cores by the number of applications and do static assignments until the next reconfiguration occurs. Though, we avoid migrations if possible, i.e., we only add and remove processor cores to and from already assigned sets.

We realized this approach by explicitly managing the affinity of Linux CPU-sets and a modified GNU OpenMP version that queries the CPU-set size. We evaluated this approach with the default NUMA memory policy (EQ) and with the memory interleave policy (EQi).

Batch processing (*BP*). While not useful in the interactive scenarios we consider, batch processing gives another base line to compare our approach to. Arriving jobs are simply processed in a FIFO order, one after the other. As our test applications do not have ideal speedups, this style of execution does not necessarily result in the shortest possible makespan.

Topology-aware Equipartitioning (*TA*, *TAi*). This is a variant of our approach without coscheduling. Compared to the basic equipartitioning above, the partitions now respect the system topology, so that whole topological units or fractions thereof are used. Additionally, the possible partition sizes are reduced as we strive to give out only equally sized partitions. For our quad-socket, 24-core evaluation system, this results in partition sizes of 1, 2, 3, 6 (one socket), 12 (half a system) and 24 cores (whole system).

Just like the basic equipartitioning, this was realized with the help of Linux CPU-sets and a modified GNU OpenMP. Again, we ran this approach with the default NUMA memory policy (TA) and the memory interleave policy (TAi).

Topology-aware Coscheduling (*TACO*, *TACOi*). This is our approach as described in Section 2.1. We evaluated a basic version of TACO without extras to gauge the principle applicability. That is, we did not apply the ideas described in Sections 2.2–2.4. Also, we did not use a hysteresis, i.e., the thresholds for switching partition sizes up and down are identical and correspond to the number of available partitions on a particular level. For our evaluation system and due to an implementation restriction, we have partition sizes of 1, 3, 6, 12 and 24 cores. This means if there is one application, it gets scheduled system wide, two to three applications are coscheduled on 12-core partitions, four to seven ap-

plications are coscheduled on sockets, eight to 23 applications use half-socket partitions, and finally 24 or more applications are executed as single-threaded programs.

To realize this, we used a modified Linux 3.8 kernel with coscheduling support. The concept of that coscheduler and its Linux implementation are described by Schönherr et al. in [14]. Basically, it allows selected applications to be coscheduled while retaining the properties of the original scheduler. The Linux implementation gets information about groups of tasks to be coscheduled and their desired coscheduling granularity via the Linux `cgroup` interface. The actual placement and balancing is done with the normal Linux rules. This approach was also tested with default NUMA memory policy (TACO) and with the memory interleave policy (TACOi).

4.3. Results

For our evaluation, we analyzed different sets of workloads against the different scheduling approaches. The workload sets differ in their application mix and in their average number of jobs. Their properties are given in Table 3. Each experiment was repeated five times, to see how stable the results are. For each experiment we determined the makespan (i.e., the time the system is not idle while processing a workload), individual job slowdowns (i.e., response times normalized to isolated parallel execution times), and the number of reconfigurations. These values are summarized in Table 4 with the best approaches highlighted.

One exemplary workload of set A and one of set B is given in Fig. 3(a) and (b), respectively. They show the number of concurrently running applications over time. They are typical in that UE and LA generate unusually long makespans compared to the other approaches. This is due to the non-coscheduled oversubscription and applications making use of spin-blocking and, in case of `lu.A`, active waiting. LA is generally better than UE, as oversubscription subsides over time due to an increasing system load and a slowly reacting load adaptation. Switching the waiting policy of OpenMP to passive waiting, does not help significantly, as demonstrated by UEp and LAp, because of the active waiting at application level in `lu.A`. Only for workloads without any active waiting, such as those in our set C, UEp and LAp are almost competitive as

Table 3
Configuration of workload sets used for evaluation

Set	Workloads	Jobs per workload	Arrivals per minute	Application mix
A	8	40	6	all
B	8	40	9	all
C	8	40	9	all except `lu.A`
D	8	40	9	all except `ft.B` and `mg.B`

Table 4
Averaged results of different approaches per set

Set	Approach	Makespan (BP = 100%)	Average slowdown	Reconfigurations
A	BP	100%	5.7	0
	LA	134%	20.3	n/a
	LAp	120%	13.0	n/a
	EQ	87%	4.1	145
	EQi	**84%**	**3.8**	**132**
	TA	94%	4.3	76
	TAi	**85%**	**3.8**	**69**
	TACO	99%	5.5	70
	TACOi	**83%**	**3.7**	**59**
B	BP	100%	10.2	0
	LA	134%	22.8	n/a
	LAp	127%	19.0	n/a
	EQ	94%	8.4	170
	EQi	94%	8.2	167
	TA	**91%**	**7.3**	**103**
	TAi	95%	8.3	98
	TACO	100%	10.4	64
	TACOi	**91%**	**7.7**	**63**
C	BP	100%	10.8	0
	UE	198%	31.5	0
	UEp	113%	15.4	0
	LA	120%	18.4	n/a
	LAp	110%	14.4	n/a
	EQ	91%	8.5	173
	EQi	94%	9.0	172
	TA	**89%**	**7.7**	**106**
	TAi	93%	9.0	98
	TACO	98%	11.7	68
	TACOi	92%	9.0	75
D	BP	100%	11.9	0
	LA	132%	29.7	n/a
	LAp	127%	24.5	n/a
	EQ	92%	11.1	167
	EQi	102%	13.0	167
	TA	**86%**	**8.6**	**112**
	TAi	100%	12.7	106
	TACO	89%	10.3	51
	TACOi	91%	10.6	59

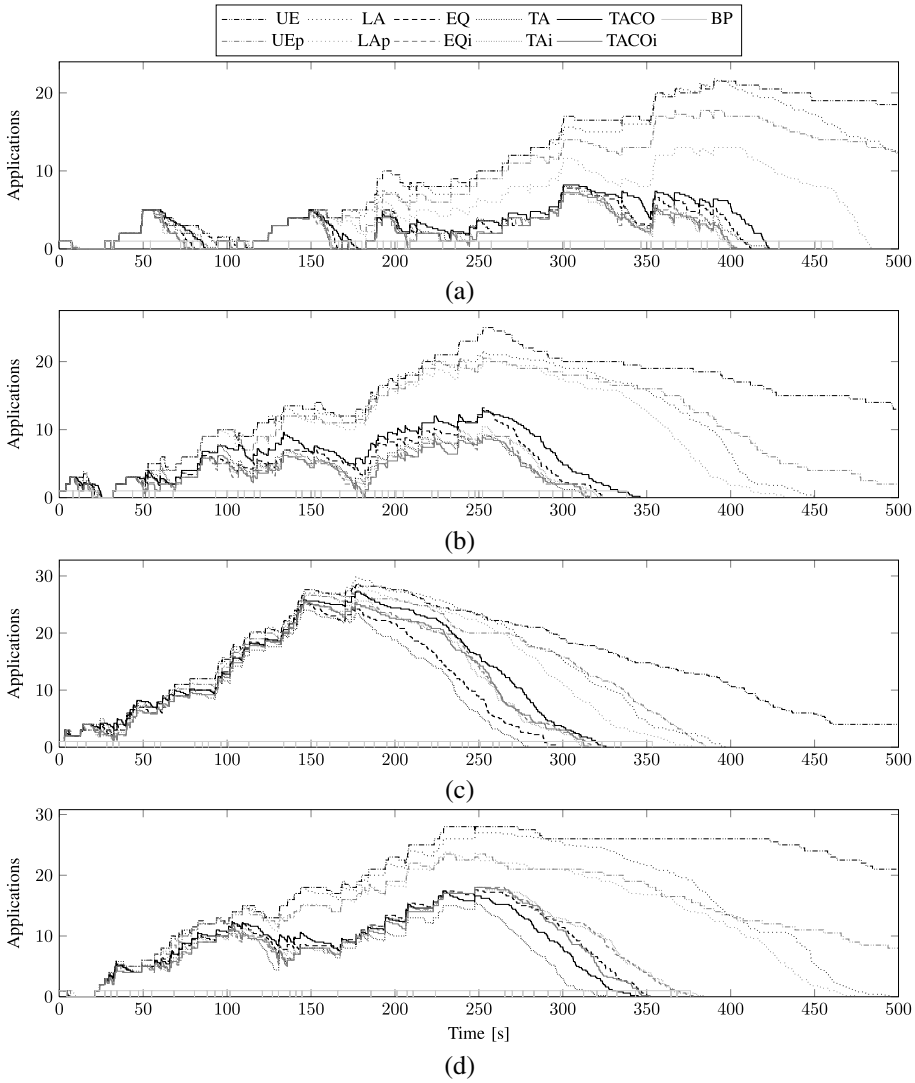

Fig. 3. Exemplary workloads. (a) Exemplary workload of set A. LA ends at 550, UEp at 2050, and UE at 2350 s. In this workload are five instances of `lu.A` with active waiting, which are responsible for the bad UE/UEp results. While the other approaches finish almost at the same time, the versions with memory interleaving realize more idle time during execution. (b) Exemplary workload of set B. UEp ends at 700, and UE at 1160 s. With only two instances of `lu.A`, UE/UEp finish earlier. (c) Exemplary workload of set C. UE ends at 570 s. This set has no active waiting at application level, but UE still suffers from OpenMP's spin-blocking. (d) Exemplary workload of set D. UEp ends at 2060 and UE at 2650 s. In this set, some workloads are particular bad for EQi and TAi. In other workloads, the performance of EQi and TAi is comparable to EQ or TACOi.

shown in Fig. 3(c). Because of these results, we refrained from exhaustive experiments with UE/UEp and concentrated on the partitioning approaches only.

Considering EQ and TA, both using partitioning and no coscheduling, we see that TA is better suited for workloads with a higher number of concurrent jobs than EQ. When there are not that many concurrent jobs, it is the other way around. This is because EQ ignores the topology of our system and applications likely end up spread across multiple NUMA domains. With many concurrent jobs, EQ produces many remote memory accesses, while TA keeps accesses mostly within one NUMA domain. With only a few concurrent jobs, reconfigurations with TA are more prone to cause job migrations across NUMA domains separating the job from its memory. EQ does not have this problem this pronounced. This is supported by the results of their counterparts EQi and TAi with memory interleaving enabled: here, memory is distributed across NUMA domains in the first place and it should not

matter where code is executed (except for multicore cache effects), making cross-NUMA migrations cheap at an overall increased cost for memory accesses. And indeed, EQi and TAi yield nearly identical results, outperforming their non-interleaving counterparts if (and only if) there are many cross-NUMA migrations, i.e., not many concurrent applications.

Comparing both topology-aware approaches with the default NUMA memory policy, we see that TACO was not able to outperform TA – not even once. Another general trend is that EQ is also better than TACO. Though, this really depends on the actual application mix within a workload as illustrated by our workload set D (see Fig. 3(d)), where benchmarks `mg.B` and `ft.B` were removed from the application mix and the result favors TACO over EQ. Here, two effects accumulate: Benchmark `mg.B` is severely memory-bound and does not profit from multiple cores of one socket. That TACO issues larger partitions on average than EQ or TA is also not helpful. Instead, `mg.B` profits from the likely spread out execution of EQ. When paired with some other application that is not that memory-bound, `mg.B` has more memory bandwidth available. Similar for `ft.B`, though it scales a bit better.

Besides TACO issuing larger partitions, there is another difference to the other partitioning schemes: our TACO implementation relies on the Linux balancing mechanism to distribute load while TA and EQ do it themselves. Thus, the load balancing for TA and EQ is done centrally and proactively with minimal migrations per reconfiguration, while the load balancing for TACO is distributed and reactive in nature. This seems to cause more migrations than necessary, resulting in more remote memory accesses. This theory is supported by the results of TACOi, its counterpart with NUMA memory interleaving enabled. TACOi also issues larger partitions, yet it is nearly always better than TACO – especially considering that memory interleaving causes a performance degradation for TAi and EQi in workloads with a higher arrival rate.

In fact, TACOi is the best of all evaluated approaches for set A, and a close second after TA for set B. In set C, benchmark `lu.A` is missing, which is very sensitive with respect to interferences on the L3 cache of our system and profits from having cache just for itself. Without this advantage, TACOi comes in third after TA and EQ. In set D, TA is first followed by TACO and TACOi.

4.4. Exploring the design space

In addition to the presented results, we also explored the design space of our topology-aware scheduling schemes. Foremost, for our topology-aware approaches being competitive on NUMA systems, a mechanism is necessary that keeps memory and tasks close together. If such a mechanism does not exist, topology-aware schemes often separate tasks from their memory and an approach like EQ or enforced NUMA memory interleaving are actually better, as there is probably at least some memory allocated at the NUMA node(s) where the tasks are executed. With Linux 3.8 a very simple NUMA memory balancing mechanism is available, which periodically enforces a migrate-on-next-touch policy to move memory to where it is needed. While not perfect, it helps not only TA and TACO, but also EQ and delivers consistently better performance in our experiments than doing nothing.

With TACO/TACOi we have the additional freedom to restrict allowed partition sizes without running into fragmentation or fairness issues. The results above use a mostly unrestricted set, where partition of sizes 1, 3, 6, 12 and 24 cores are allowed (due to an implementation restriction, supporting 2 and 3 at the same time is not possible). We also ran some experiments with more restrictions and only allowed sizes of 1, 6 and 24 cores – core, socket and system. However, with mostly `lu.A` profiting from this, it did not give good overall results. For a similar reason, we did not try the balancing scheme proposed in Section 2.2, as it requires application knowledge to be effective. Instead, we used periodic rebalancing for TACO and TACOi, as our coscheduling support is tightly integrated into the Linux scheduler which resolves load imbalances this way. While this works fine for TACOi, we had to modify the default load balancing settings for TACO, so that rebalancing applications across NUMA domains is kept at a very low frequency. Otherwise the resulting remote memory accesses and triggered page migrations can quickly kill the performance.

In Section 3.3 we discussed the optimal time slice length. Linux itself does not used fixed time slices, but adjusts them according to task weight, current load and the number of processor cores in the system. For our system, this translates to time slices from as short as 3 ms up to 24 ms. Paired with a regular 100 Hz timer, we should get 10 ms to 20 ms time slices for TACO and TACOi. Experiments with specific setups show, that some of the NAS benchmarks gain a few percent more performance with an increased average time slice length of 50 ms to 100 ms. However, this relies on specific partition sizes and specific coscheduled applications. For our workloads, it did not translate into a measurable advantage.

5. Related work

The idea of operating system enforced fairness between multiple parallel applications is not new. A pioneering work is [16], which introduces *Process Control*: a method to fairly distribute the available CPUs among running parallel applications. It includes a concept of malleability and also considers non-malleable applications by reducing the pool of available CPUs for malleable applications accordingly. CPUs are distributed in a round robin fashion, until either an application reaches its individual maximum or no more CPUs are left. The approach does not consider the system topology in any way, but for the targeted early shared memory systems this does not really matter.

On distributed memory systems, on the other hand, topology has always been important. In [9], two concepts for such systems are presented: *Equipartition* and *Folding*. Equipartition conceptually splits a regular, non-hierarchical system topology (e.g., a grid) into connected, almost equally sized partitions. Folding always splits the largest partition in two halves (with, e.g., hypercubes in mind). This has the benefit of avoiding parallel reconfigurations. The more unfair distribution of CPU time is countered with periodic rotations of applications. Folding is also recognized as a possibility to make rigid or moldable applications pseudo-malleable: due to the halving of partitions, non-malleable applications experience always a doubling of threads per processor, which works reasonably well as long as there is not much synchronization. Both approaches do not consider any form of coscheduling. However, as far as partition sizes are concerned, our approach is quite similar to Folding. For example, the idea of pseudo-malleable applications can be used with TACO without problems. Contrary to Folding, TACO achieves a fair CPU time distribution without periodic rotations.

Corbalan et al. suggest *Compress&Join* [3], a combination of coscheduling and partitioning, where job malleability is used to reduce fragmentation normally associated with coscheduling: based on an ideal number of processors for each application, their approach fits multiple applications into a coscheduled time slot, possibly sizing them down a bit with a bounded deviation from the ideal size. Fairness and system topology are not considered; and while exclusive resource usage due to coscheduling is mentioned, it is not considered when partitioning a time slice. Bhadauria and McKee [2], on the other hand, consider fairness and resource contention in their partitioning scheme. Similar to Corbalan et al., they also use partitioning within coscheduling. However, they use a sampling and feedback mechanism to intelligently select and size applications to be scheduled simultaneously, so that contention of system resources is hopefully minimized. A hierarchical system topology is not considered. Both approaches require large time slices (measured in seconds) and long running applications. Contrary to that, TACO works with short time slices (measured in milliseconds, similar to usual OS time slices) and does not disturb interactive behavior. TACO's nesting of time and space slicing only requires partition wide synchronization (instead of system wide synchronization) and enables variable length time slices. Additionally, it recognizes hierarchically arranged resources. We currently do not consider application speedups and do not arrange for certain applications to run simultaneously. However, our approach is flexible enough that these features can be easily added. In fact, we plan to integrate some of these ideas to exploit the potential of our approach and to make it more robust to a wide variety of workloads.

An example for a contention-aware scheduling algorithm that does not consider parallel applications is the work of Merkel et al. presented in [10]. Instead of explicitly selecting sets of tasks that minimize resource contention when executed simultaneously, they realize this feat as emergent behavior. They distribute tasks with similar resource demands evenly across the system. By grouping runqueues in pairs, sorting their tasks by their expected resource consumption – one ascending, one descending – and occasional synchronization between them, they increase the likelihood of executing tasks simultaneously that complement each other's resource demands. While this type of coscheduling is radically different from our variant, it can still be used together with TACO as outlined in Section 3.1, for instance to realize the outer scheduling decision.

Tam et al. [15] follow a different line of thought. They do not try to avoid contention – at least not directly – but they try to increase resource sharing. Specifically, they track cache misses which are satisfied by remote caches and form groups of tasks that exhibit similar sharing patterns. These groups, once identified, are placed on cores with a shared cache (unless it leads to a severe imbalance) hopefully reducing the number of cache misses. In case of more groups than number of independent caches, multiple groups are assigned to a cache, sharing it competitively without further management, such as coscheduling. On the one hand, such an approach seems unnecessary in the

context of our approach, as applications automatically receive shared caches. On the other hand, it allows to subdivide an application into multiple parts with distinct sharing patterns or to detect cross-application sharing, without having to specify it explicitly. Thus, approaches such as this are good candidates for the inner scheduling decision as described in Section 3.1.

6. Conclusion

In current operating systems, scheduling of applications is done from within the operating system scheduler that keeps all details, like system load or resource status, hidden from user-level applications, following the concept of separation of concerns. But in these days with emerging many-core systems and an increasing count of parallel applications, new challenges arise when it comes to scheduling targeting high and efficient CPU utilization in non-HPC environments, which might require a change in this policy. Despite a whole lot of research that has been published about efficient scheduling of parallel applications within the last decades, nothing of this is available in today's operating systems. Hence, parallel programs for end-user devices are on their own and must base their degree of parallelism on assumptions, such as the overall system load, and enforce their thread placement manually. We believe that this stems from the inflexibility of suggested approaches that cannot handle or incorporate legacy situations and thus force an all or nothing decision.

In this paper, we tackled the problem by introducing TACO, a topology-aware scheduling scheme that combines the approaches of partitioning and coscheduling. On the one hand, topology-aware partitions allow us to retain a high potential for application-level optimizations. On the other hand, we apply coscheduling to reduce the number of reconfigurations without sacrificing the advantages of partitioning and to reach a perfectly balanced distribution of computational power in every (stable) situation. The result is a flexible concept that can handle high application birth and death rates and that can easily incorporate applications with special requirements. We discussed the issue of resource contention on current multicore architectures and its impact on scheduling of parallel applications. We proposed a method to integrate contention-awareness into our approach. Furthermore, we showed that the traditional speedup measure is questionable in multicore systems under the presence of resource contention. Instead, we proposed to measure the effective speedup that takes resource contention into account.

We implemented our approach and executed a series of experiments with multiple parallel applications. Topology-awareness in itself turned out very beneficial when NUMA effects are minimized. The gain in communication speed outweighs everything else. The addition of coscheduling gives us our sought after conceptual flexibility, but makes our approach more sensitive towards the executed workload due to larger partitions. The integration of our TACO implementation with the Linux load balancer currently suffers from increased NUMA effects. Nevertheless, we see competitive results with memory interleaving enabled.

The continuation of our research presented here addresses on the one hand issues already mentioned: We want to incorporate application specific knowledge into our approach in order to apply it more selectively, such as enforcing coscheduled and topology-aware partitions only for applications that benefit from it, and doing a speedup- and ideally contention-aware mapping of differently sized partitions to applications. Also, we intend to make our approach aware of resource contention and influence application placement, so that unfortunate combinations are avoided. A second area of interest is the exploration of new use cases. The ability of our approach to utilize short time slices allows for dynamic adaptations during application execution to match the partition size to a varying degree of parallelism. This way, we can address short phases of lower parallelism and evolving applications.

In the end, we see TACO as a versatile, customizable, and eventually robust building block in upcoming operating system schedulers.

References

[1] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan and S. Weeratunga, The NAS parallel benchmarks, Technical Report RNR-94-007, NASA Ames Research Center, Moffett Field, CA, USA, March 1994.

[2] M. Bhadauria and S.A. McKee, An approach to resource-aware co-scheduling for CMPs, in: *Proceedings of the 24th ACM International Conference on Supercomputing (ICS'10)*, ACM, New York, NY, USA, 2010, pp. 189–199.

[3] J. Corbalan, X. Martorell and J. Labarta, Improving gang scheduling through job performance analysis and malleability, in: *Proceedings of the 15th International Conference on Supercomputing (ICS'01)*, ACM, New York, NY, USA, 2001, pp. 303–311.

[4] D.G. Feitelson and L. Rudolph, Distributed hierarchical control for parallel processing, *Computer* **23**(5) (1990), 65–77.

[5] D.G. Feitelson and L. Rudolph, Toward convergence in job schedulers for parallel supercomputers, in: *Proceedings of the IPPS'96 Workshop on Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science, Vol. 1162, Springer, Berlin/Heidelberg, Germany, April 1996, pp. 1–26.

[6] H. Feng, R.F. Van der Wijngaart, R. Biswas and C. Mavriplis, Unstructured adaptive (UA) NAS parallel benchmark, version 1.0, Technical Report NAS-04-006, NASA Ames Research Center, Moffett Field, CA, USA, July 2004.

[7] H. Jin, M. Frumkin and J. Yan, The OpenMP implementation of NAS parallel benchmarks and its performance, Technical Report NAS-99-011, NASA Ames Research Center, Moffett Field, CA, USA, October 1999.

[8] F. Liu and Y. Solihin, Understanding the behavior and implications of context switch misses, *ACM Transactions on Architecture and Code Optimization* **7**(4) (2010), 21:1–21:28.

[9] C. McCann and J. Zahorjan, Processor allocation policies for message-passing parallel computers, in: *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, ACM Press, New York, NY, USA, May 1994, pp. 19–32.

[10] A. Merkel, J. Stoess and F. Bellosa, Resource-conscious scheduling for energy efficiency on multicore processors, in: *Proceedings of the 5th European Conference on Computer Systems (EuroSys'10)*, ACM Press, New York, NY, USA, April 2010, pp. 153–166.

[11] OpenMP Architecture Review Board, OpenMP application program interface, version 3.1, July 2011.

[12] J. Ousterhout, Scheduling techniques for concurrent systems, in: *Proceedings of the 3rd International Conference on Distributed Computing Systems (ICDCS'82)*, IEEE Computer Society, Los Alamitos, CA, USA, October 1982, pp. 22–30.

[13] J. Reinders, *Intel Threading Building Blocks*, 1st edn, O'Reilly & Associates, Sebastopol, CA, USA, 2007.

[14] J.H. Schönherr, B. Lutz and J. Richling, Non-intrusive coscheduling for general purpose operating systems, in: *Proceedings of the International Conference on Multicore Software Engineering, Performance, and Tools (MSEPT'12)*, Lecture Notes in Computer Science, Vol. 7303, Springer, Berlin/Heidelberg, Germany, May 2012, pp. 66–77.

[15] D. Tam, R. Azimi and M. Stumm, Thread clustering: Sharing-aware scheduling on smp-cmp-smt multiprocessors, in: *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, EuroSys'07*, ACM, New York, NY, USA, 2007, pp. 47–58.

[16] A. Tucker and A. Gupta, Process control and scheduling issues for multiprogrammed shared-memory multiprocessors, in: *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP'89)*, ACM Press, New York, NY, USA, December 1989, pp. 159–166.

[17] S. Zhuravlev, J.C. Saez, S. Blagodurov, A. Fedorova and M. Prieto, Survey of scheduling techniques for addressing shared resources in multicore processors, *ACM Computing Surveys* **45**(1) (2012), 4:1–4:28.