Topology-aware Equipartitioning with Coscheduling on Multicore Systems

Jan H. Schönherr
schnhrr@tu-berlin.de
Communication and Operating Systems
Technische Universität Berlin

Ben Juurlink
b.juurlink@tu-berlin.de
Embedded Systems Architecture
Technische Universität Berlin

Jan Richling
jan.richling@tu-berlin.de
Communication and Operating Systems
Technische Universität Berlin

***Abstract*—Over the last decade, multicore architectures have become omnipresent. Today, they 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, which replace the sometimes crude results of ad-hoc parallelization. Combined with the ever increasing complexity of multicore architectures, this results in a scheduling problem that is different from what it has been, because 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. Due to emerging parallel programming environments, we only consider malleable applications, i. e., applications where the parallelism degree can be changed on the fly. We propose a topology-aware scheduling scheme that combines equipartitioning and coscheduling. It does not suffer from the drawbacks of the individual concepts and also allows to run applications at different degrees of parallelisms without compromising fairness. We find that topology-awareness increases performance for all evaluated workloads. The combination with coscheduling is more sensitive towards the executed workloads. However, the gained versatility allows new use cases to be explored, which were not possible before.

I. 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* [1] or the *Intel Threading Building Blocks* [2]. 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 to some degree sensible parallelizations, avoiding common beginner mistakes. Often, the resulting applications are moldable or malleable as defined by Feitelson and Rudolph [3]: 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.

978-1-4799-1010-6/13/$31.00 © 2013 IEEE

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* [4] and *gang scheduling* [5] 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 application 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 schemes 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* [6]: 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 usually do not happen instantaneously. 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. In order to keep properties normally associated with partitioning, our approach is topology-aware. Furthermore, we propose a combination with coscheduling to address drawbacks associated with traditional equipartitioning. In particular, our approach allows us to avoid frequent reconfigurations and to achieve 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.

The remainder of this paper is structured as follows: In Section II, we give a detailed description of our approach, which is then evaluated and compared to other established approaches in Section III. Section IV reviews other work in the area, and we conclude our paper in Section V.

II. Topology-aware Equipartitioning

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 I, 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.

A. Basis

In order to create an equipartitioning scheme for modern multicore architectures, we combine the idea of equipartitioning with coscheduling.

Modern parallel systems are not symmetrical in the sense that arbitrary pairs of processor cores would always have the same behavior regarding resource sharing and communication

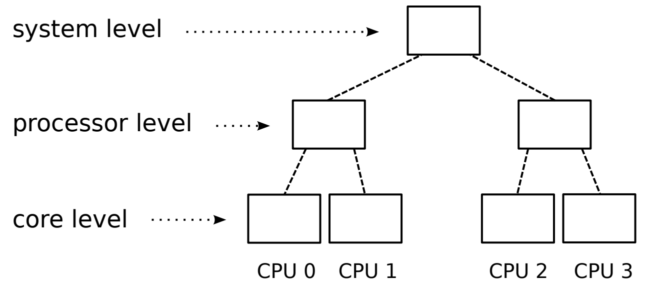

Fig. 1. Hierarchy levels of a dual-socket dual-core system.

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 Figure 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.

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

```
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.

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.

The frequency of reconfigurations is important, as every reconfiguration leads temporarily to over- or undersubscription within affected partitions. This is caused by the temporal granularity of the adaptation mechanism inside applications, i. e., there is a phase where a change request is already issued but the application has not yet responded to it. With our strategy the number of such reconfigurations is reduced compared to traditional approaches, therefore the impact of this problem is also reduced.

B. 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 suggested in [7]. 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 a 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. Under the assumption of similar speedup behavior, this leads to a fair distribution of the available resources. To avoid rapid reconfigurations, this kind of balancing is only feasible in more or less stable situations. 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.

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

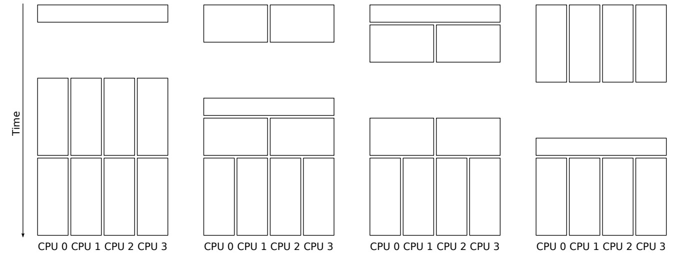

Fig. 2. Schedules generated by our approach for one to eight tasks on the example dual-socket dual-core system.

C. 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 speedup behavior of the individual applications can be used to optimize the assignment of applications to partitions, so that applications with a near linear speedups get wider partitions (more cores) than applications with worse speedup behavior.

III. Evaluation

In order to prove the applicability of our approach, 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 III-A, followed by a description of all tested approaches in Section III-B. Our evaluation closes with the presentation and discussion of the results in Section III-C.

A. 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. [8], [9] and developed by Jin et al. [10]. 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 I 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 GiB RAM (DDR2-533, 16 GiB 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.

B. 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 two variants of our *Topology-aware Equipartitioning*, with and without coscheduling.

a) **Uncontrolled Execution (UE):** 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 our case, OpenMP applications, that usually means that each application 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 and passive waiting. With the former we get applications that assume exclusive system access, while the latter sacrifices single application performance for overall throughput.

b) **Load-adaptive Execution (LA):** 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 minute load average. Here, we also tested spin-blocking and passive waiting.

c) **Equipartitioning (EQ):** 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.

d) **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.

e) **Topology-aware Equipartitioning (TA):** 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.

f) **Topology-aware Equipart. with Coscheduling (CO):** This is our approach as described in Section II-A. We evaluated the basic version without extras to gauge the principle applicability of our approach. That is, we did not apply the ideas described in Sections II-B and II-C. 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

TABLE I
NAS BENCHMARKS USED FOR EVALUATION AND THEIR CHARACTERISTICS.

Benchmark	Description	Sequential Exec. Time	Speedup on partitions of size 2, 3, 6, 12, and 24	Reconfigurations (parallel regions)	Avg. Reconf. Delay (when run sequentially)
bt.A	Block Tridiagonal	94 s	1.8, 2.5, 3.8, 6.3, 13.4	1012	46 ms
cg.B	Conjugate Gradient	169 s	1.8, 2.5, 2.9, 5.5, 9.2	231	365 ms
ft.B	Fast Fourier Transform	82 s	1.7, 2.3, 3.1, 5.9, 11.7	112	365 ms
is.C	Integer Sort	52 s	1.9, 2.8, 4.7, 7.8, 12.2	16	1627 ms
lu.A	Lower-Upper symmetric Gauss-Seidel	75 s	1.8, 2.4, 3.7, 6.2, 12.9	518	73 ms
mg.B	Multi Grid	13 s	1.2, 1.3, 1.1, 2.2, 4.3	1281	5 ms
sp.A	Scalar Pentadiagonal	71 s	1.5, 1.7, 1.8, 3.3, 7.0	3616	10 ms
ua.A	Unstructured Adaptive	68 s	1.6, 1.9, 2.7, 5.8, 15.9	36510	1 ms

TABLE II
CONFIGURATION OF WORKLOAD SETS USED FOR EVALUATION.

Set	Work-loads	Jobs per workload	Arrivals per minute	Application mix
A	5	40	6	all
B	8	40	9	all
C	4	40	9	all except lu.A
D	8	40	9	all except ft.B and mg.B

to seven applications 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 [11]. 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.

C. 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 II. 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 III.

One exemplary workload of each, set A and B, is given in Figures 3 and 4, 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 increasing system load and a slowly reacting load adaptation. Due to the active waiting in lu.A, switching the waiting policy of OpenMP to passive waiting, does not help significantly, as demonstrated by *UEp* and *LAp*, which are equivalent to UE and LA except for the passive waiting policy. Only for workloads without any active waiting, such as those in our set C, this changes as shown in Figure 5. Because of these results, we refrained from further experiments with UE and LA and concentrated on the three partitioning approaches only.

Another general trend in these workloads is that TA is better than EQ, which in turn is better than CO. Though, this really depends on the actual application mix within a workload. This is demonstrated on the one hand with the workload set C, where lu.A is missing (see Figure 5). This particular benchmark is very sensitive with respect to interferences on the L3 cache of our system, and it profits from having cache just for itself. EQ ignores the topology and applications likely end up spread across multiple sockets, a worst case scenario for lu.A. Thus, removing lu.A gives EQ a boost compared to TA and CO. Between TA and CO, lu.A favors CO, because this approach issues larger partitions than TA, though this is not visible in our presented workloads as the effect is countered by applications which prefer smaller partitions due to bad speedups. Another benchmark with similar characteristics, though not as extreme as lu.A, is cg.B.

If, on the other hand, we remove benchmarks mg.B and ft.B from the application mix as in set D, then CO is better suited for the resulting workloads than EQ (see Figure 6). Here, two effects accumulate: Benchmark mg.B is severely memory-bound and does not profit from multiple cores of one socket. That CO 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 the experiments discussed here, we also explored the design space of our topology-aware 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 is actually better, as there is probably at least some memory allocated at the NUMA

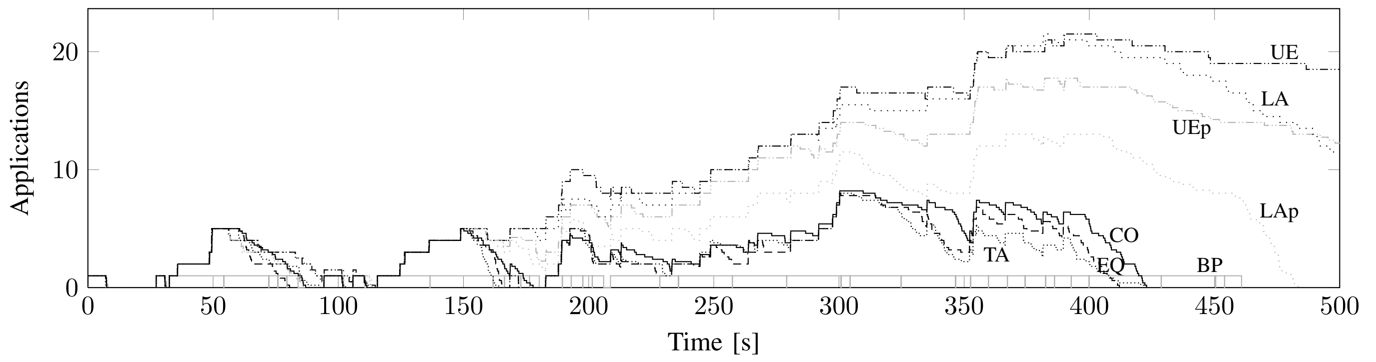

Fig. 3. Exemplary workload of set A. LA ends at 550, UEp at 2050, and UE at 2350 seconds.

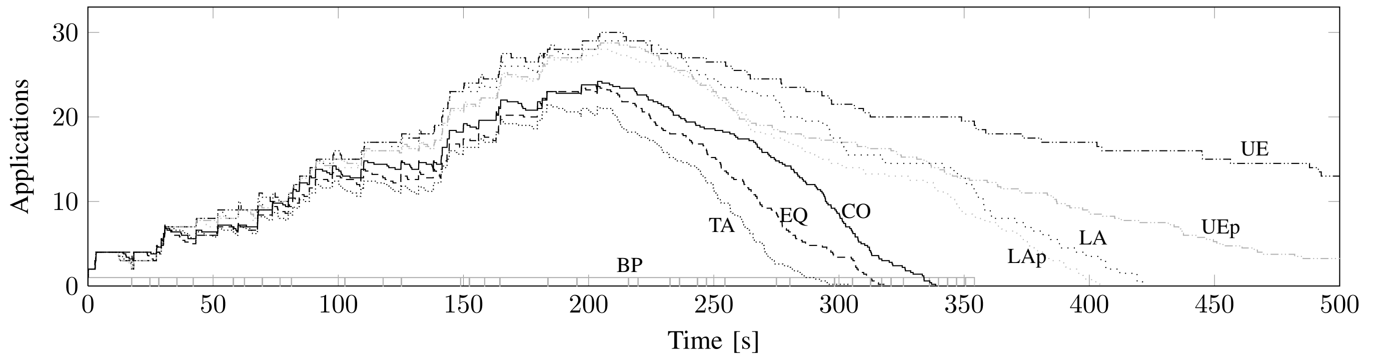

Fig. 4. Exemplary workload of set B. UEp ends at 1200, and UE at 1650 seconds.

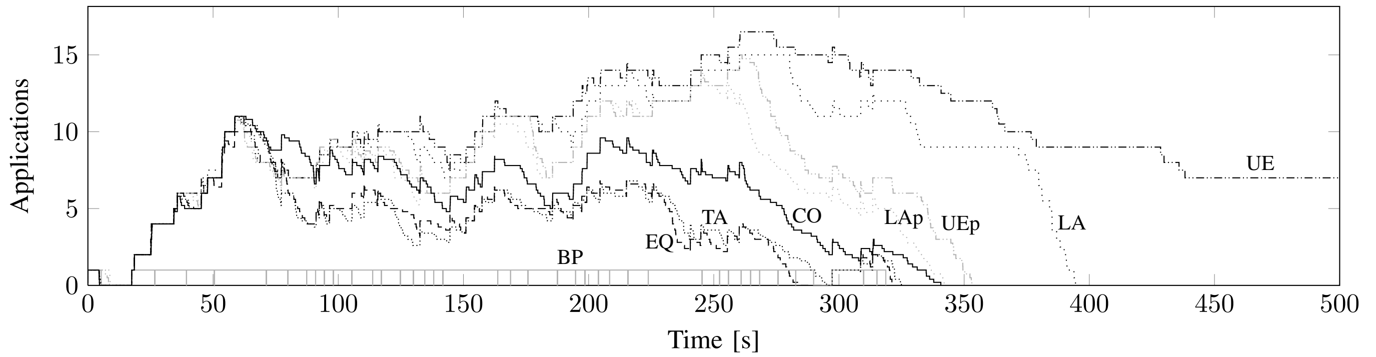

Fig. 5. Exemplary workload of set C. UE ends at 670 seconds.

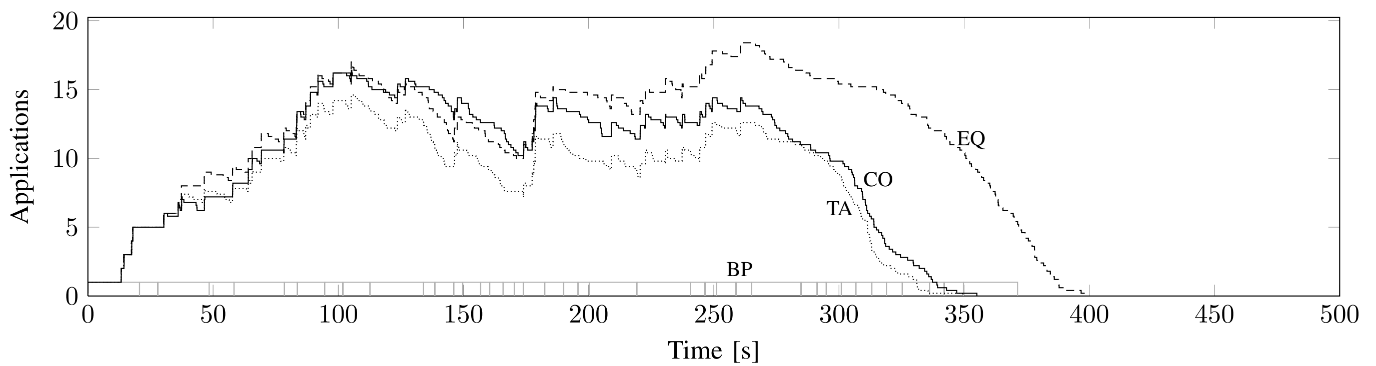

Fig. 6. Exemplary workload of set D.

TABLE III
AVERAGED RESULTS OF DIFFERENT APPROACHES PER SET.

Set	Approach	Makespan (BP=100%)	Average Slowdown	Reconfig-urations
A	BP	100%	6.7	0
	EQ	89%	4.8	155
	TA	95%	4.7	81
	CO	100%	6.2	74
B	BP	100%	10.2	0
	EQ	94%	8.4	170
	TA	91%	7.3	103
	CO	100%	10.4	64
C	BP	100%	10.7	0
	EQ	92%	8.9	178
	TA	90%	8.3	104
	CO	101%	13.1	88
D	BP	100%	11.9	0
	EQ	92%	11.1	167
	TA	86%	8.6	112
	CO	89%	10.3	51

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. This helps also EQ and delivers consistently better performance in our experiments than doing nothing or forcing an interleaved memory policy.

With CO 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 only `lu.A` and `cg.B` profiting from this, it did not give good overall results. For a similar reason, we did not try the balancing scheme proposed in Section II-B, as it requires application knowledge to be effective. Instead, we experimented with periodic rebalancing for CO, which works fine as long as 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. (Periodic rebalancing with an interleaved memory policy is fine, but that does not give performance either.)

Very brief experiments indicate, that we could get a few percent more performance out of CO with an increased average time-slice length. However, we intentionally use the operating system defaults, as our goal is a flexible, easily integrable scheme that does not cause inconveniences for applications out of its scope.

IV. RELATED WORK

The idea of operating system enforced fairness between multiple parallel applications is not new. A pioneering work is [6], 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 [7], 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 our approach without problems. Contrary to Folding, we achieve a fair CPU time distribution without periodic rotations.

Corbalan et al. suggest *Compress&Join* [12], 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 [13], 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, our approach works with short time slices (measured in milliseconds, similar to usual OS time slices) and does not disturb interactive behavior. Our nesting of time and space slicing only requires partition wide synchronization (instead of system wide synchronization) and enables variable length time slices. Additionally, we recognize 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.

V. 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 a new equipartitioning 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 also easily incorporate applications with special requirements.

We implemented our approach and executed a series of experiments with multiple parallel applications. The results were mixed. Topology-awareness in itself turned out very beneficial for all evaluated workloads. The gain in communication speed outweighs everything else. The addition of coscheduling, which gives us our sought after conceptual flexibility, makes our approach more sensitive towards the executed workload. However, the evaluated applications were not specifically selected to play the strengths of our approach: they do not scale particular well on multicore systems and they are not short enough (or their reconfiguration delays not long enough) so that the reduction in the number of reconfigurations becomes noticeable. Overall, this indicates that coscheduling (contrary to partitioning) should not be applied blindly.

The continuation of the research presented here basically addresses these remaining issues. First, there is the incorporation of application specific knowledge into our approach in order to apply it more selectively. This includes optimizations discussed in Section II-C: enforcing coscheduled and topology-aware partitions only for applications that benefit from it, and doing a speedup-aware mapping of differently sized partitions to applications. Another area of interest is that of resource contention, where application placement and partition size selection can be influenced, so that unfortunate combinations are avoided. Second, there 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 our approach as a versatile, customizable, and eventually robust building block in upcoming operating system schedulers.

References

[1] OpenMP Architecture Review Board, "OpenMP application program interface, version 3.1," Jul. 2011.

[2] J. Reinders, *Intel threading building blocks*, 1st ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2007.

[3] 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*, ser. Lecture Notes in Computer Science, vol. 1162. Berlin/Heidelberg, Germany: Springer, Apr. 1996, pp. 1–26.

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

[5] D. G. Feitelson and L. Rudolph, "Distributed hierarchical control for parallel processing," *Computer*, vol. 23, no. 5, pp. 65–77, May 1990.

[6] 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)*. New York, NY, USA: ACM Press, Dec. 1989, pp. 159–166.

[7] 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*. New York, NY, USA: ACM Press, May 1994, pp. 19–32.

[8] 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," NASA Ames Research Center, Moffett Field, CA, USA, Tech. Rep. RNR-94-007, Mar. 1994.

[9] H. Feng, R. F. V. der Wijngaart, R. Biswas, and C. Mavriplis, "Unstructured adaptive (UA) NAS parallel benchmark, version 1.0," NASA Ames Research Center, Moffett Field, CA, USA, Tech. Rep. NAS-04-006, Jul. 2004.

[10] H. Jin, M. Frumkin, and J. Yan, "The OpenMP implementation of NAS parallel benchmarks and its performance," NASA Ames Research Center, Moffett Field, CA, USA, Tech. Rep. NAS-99-011, Oct. 1999.

[11] 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)*, ser. Lecture Notes in Computer Science, vol. 7303. Berlin/Heidelberg, Germany: Springer, May 2012, pp. 66–77.

[12] 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)*. New York, NY, USA: ACM, 2001, pp. 303–311.

[13] 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)*. New York, NY, USA: ACM, 2010, pp. 189–199.