Resource-conscious Scheduling for Energy Efficiency on Multicore Processors

Andreas Merkel Jan Stoess Frank Bellosa
Karlsruhe Institute of Technology, System Architecture Group
{merkel,stoess,bellosa}@kit.edu

Abstract

In multicore systems, shared resources such as caches or the memory subsystem can lead to contention between applications running on different cores, entailing reduced performance and poor energy efficiency. The characteristics of individual applications, the assignment of applications to machines and execution contexts, and the selection of processor frequencies have a dramatic impact on resource contention, performance, and energy efficiency.

We employ the concept of task activity vectors for characterizing applications by resource utilization. Based on this characterization, we apply migration and co-scheduling policies that improve performance and energy efficiency by combining applications that use complementary resources, and use frequency scaling when scheduling cannot avoid contention owing to inauspicious workloads.

We integrate the policies into an operating system scheduler and into a virtualization system, allowing placement decisions to be made both within and across physical nodes, and reducing contention both for individual tasks and complete applications. Our evaluation based on the Linux operating system kernel and the KVM virtualization environment shows that resource-conscious scheduling reduces the energy delay product considerably.

Categories and Subject Descriptors D.4.1 [*Operating Systems*]: Process Management—Scheduling

General Terms Design, Management

Keywords activity vectors, CMP, energy-aware scheduling, frequency scaling, migration, resources, task characteristics, virtualization

EuroSys'10, April 13–16, 2010, Paris, France.

1. Introduction

Today's operating system schedulers treat cores of a chip multicore processor (CMP) largely like distinct physical processors. Yet, there are some interdependencies between cores that need be taken into account for optimal performance and energy efficiency.

The cores of a chip share resources such as caches and memory interfaces. This is likely to cause contention between the cores if activities with similar characteristics, for example several memory-bound programs, are running together [20]. Contention slows down the execution of the programs, and besides the performance penalty, also induces inefficient use of energy, since cores waiting for a resource to become available dissipate power without making progress. Aggressive clock gating or deep sleep states do not necessarily alleviate the energy inefficiency: a recent study shows, that, even with zero-power idle modes, race-to-halt like scheduling schemes are sub-optimal for all but the most CPU-bound applications [27].

A second cross-effect, also related to energy efficiency, stems from the fact that many chips only allow setting a single frequency and voltage for the entire chip, since allowing multiple frequencies and voltages introduces additional hardware complexity. The optimal frequency at which the processor can execute a program most efficiently in terms of runtime and energy depends on the program's characteristics, in particular on the frequency of memory accesses [12, 30]. Hence, if applications with different characteristics are running in parallel on a chip, not each one can be executed at its optimal frequency.

Since the scheduler is the component of an operating system responsible for deciding which applications run on the cores simultaneously, scheduling is crucial for performance and energy efficiency. In a cluster system consisting of multiple nodes, the same holds true for the assignment of applications or entire virtual machines (VMs) to the individual nodes, since this assignment determines the programs that are eligible to be co-scheduled on the processors of the node.

For a multicore chip that offers only chip-wide frequency scaling, the question arises whether it is advantageous to combine tasks with different characteristics in order to re-

duce resource contention, or rather to run tasks with similar characteristics together in order to be able to run all applications at their optimal frequency.

This paper extends our previous work [17], in which we have shown that addressing the problem of memory contention by co-scheduling can lead to a significant improvement of the energy delay product (EDP), which is defined as the product of runtime and expended energy [9]. In order to characterize tasks, we make use of the concept of *task activity vectors* [18], which represent the utilization of chip resources caused by tasks. Based on the information provided by activity vectors, we propose scheduling policies that avoid resource contention by co-scheduling tasks with different characteristics.

We make the following contributions:

- We show that combining tasks in order to reduce resource contention is more important than combining tasks that share a common optimal frequency.
- We propose a co-scheduling policy that avoids contention for bottleneck resources.
- We propose a migration policy that balances resource utilization across execution contexts.
- We propose a frequency heuristic that, if scheduling cannot avoid contention, reduces the impact of resource contention on energy efficiency.
- We show that using the concepts of virtualization and VM migration, our policies can be extended beyond the border of a single node, and can leverage the workload diversity of an entire group of nodes.

We integrate the policies into an operating system scheduler and into a virtualization system, allowing resource-conscious placement within and across nodes, and reducing contention both for individual tasks and complete applications. For the rest of the paper, we refer to a *task* to encompass both applications and VMs.

The focus of our co-scheduling policies lies on long-running, compute-intensive, and independent tasks: that is, we assume that tasks do little I/O and do not communicate with each other very often. Finally, our approach is currently limited to single-processor VMs.

We have extended the Linux 2.6.22 kernel scheduler to support our new strategies. We use the KVM virtualization system to extend scheduling support from individual tasks to complete software environments running in VM instances, and leverage VM migration to make placement decisions across different nodes. An evaluation of our policies using SPEC CPU 2006 benchmarks reveals that our policies manage to reduce EDP considerably.

The rest of this paper is structured as follows: Section 2 reviews related work. Section 3 presents our analysis of optimal multicore scheduling. Section 4 describes how we apply the concept of activity vectors to represent resources relevant for multicore scheduling. Sections 5 and 6, introduces our scheduling and migration policies as well as the frequency heuristic. Section 7 describes the implementation of our policies for Linux and KVM. Section 8 evaluates our proposed policies and Section 9 concludes.

2. Background and Related Work

Previous research has investigated the problem of selecting a frequency at which to run a task with given characteristics most efficiently in terms of energy consumption and performance [7, 12, 27, 30]. Memory-bound tasks can be executed at lower CPU frequencies without significant slowdown, since memory throughput and not CPU speed is the determining factor for their performance. In contrast, compute-bound tasks run more efficiently at higher frequencies, since lower frequencies prolong their runtime and cause them to consume power for a longer time, often negating the power savings gained by frequency scaling. However, all previous research was based on the assumption that a separate frequency can be chosen for each CPU.

Co-scheduling tasks based on memory bandwidth or other shared resources has been proposed for SMP [1, 31], SMT [16, 21, 26], and CMP systems [2, 6, 11, 25]. To our knowledge, no previous research has addressed resource-conscious co-scheduling and frequency selection in combination. Also, the constraint that multiple cores need to run at the same frequency has not been addressed in this context. Moreover, most research concentrates on finding optimal combinations of tasks, but does not discuss how a multiprocessor scheduler in a real system can succeed in combining tasks accordingly. In addition, research on co-scheduling for CMP systems has concentrated on the last-level cache as limiting resource. However, our experiments with the SPEC CPU 2006 benchmarks suggest that memory contention is becoming more important than cache contention.

A number of approaches have leveraged the dependency between optimal processor speed and memory intensity in the context of heterogeneous multicore processors, that is, chips whose cores support the same instruction set architecture, but operate with different speeds [8, 13, 24]. By assigning tasks to cores based on their memory intensity, runtime, energy, or EDP can be optimized. A related approach assigns a virtual machine monitor to a slower core, since its computations often overlap with I/O [14]. These approaches are orthogonal to ours, since they assume cores running at different speeds, while our approach is suitable for cores that have to share the same frequency/voltage domain.

Finally, there has been some research interest in using virtualization technology to improve energy efficiency, mostly, however, without specifically addressing resource contention a key factor in power consumption [22, 28, 29]. An exception forms the vGreen approach, which also proposes to link VM workload characteristics and scheduling to achieve better energy efficiency [5]. However, vGreen

focuses on cross-node placement only, while our approach also investigates VM scheduling within nodes; also, vGreen requires a central node to coordinate placement, while our approach uses a distributed balancing scheme between pairwise nodes.

3. Analysis

For investigating the effects of resource contention and frequency selection, we chose a 2.4 GHz Intel Core2 Quad Q6600. The Core2 Quad Q6600 is a multi-chip module that consists of two silicon dies in one package. Each die comprises two cores sharing 4 MiB of L2 cache. This allows observing the effects of cache as a shared resource, since it is possible to conduct experiments on cores sharing or not sharing L2 cache.

In our test system, the chip is connected to 8 GiB of DDR2 PC-6400 memory via a 266 MHz front side bus. The processor supports scaling the frequency down to 1.6 GHz. In this case, the core voltage is scaled from 1.24 V to 1.13 V. (For the rest of the paper, when we speak of frequency scaling, we imply that voltage scaling is also applied.)

We also performed experiments with an AMD Opteron 2354 quad-core chip. In contrast to the Core2, the Opteron does not access memory via a front-side bus, but possesses an integrated memory-controller on the chip. The cores of the Opteron possess private L1 and L2 caches and all share a common L3 cache, which makes it harder to analyze the importance of cache contention than for the Core2, where two cores share a common cache, respectively.

Our basic finding is the same for the Core2 and the Opteron: it does not pay off to co-schedule memory-bound tasks in order to be able to profit from lower chip frequencies. Therefore, and since the cache architecture of the Core2 allows a better analysis of cache contention, we will discuss our analysis of the Core2 in detail and only summarize the results with the Opteron at the end of this section.

As metric for our analysis, we choose the energy delay product (product of the energy expended by the processor for running a task multiplied by task runtime). The EDP emphasizes both performance and power consumption and reflects the goal of achieving energy efficiency without sacrificing too much performance. We deliberately consider only the energy consumption of the processor and not of other system components. This is a simplification, since we imply that the energy consumption of the other components is not influenced by scheduling, which means that the power consumption of all other components stays the same regardless which schedule is applied. For determining processor power consumption, we use a National Instruments SC-2345 board.

3.1 Resource contention

We evaluated resource contention between the cores using several microbenchmarks. The resources the cores are contending for are L2 cache (shared by two cores, respectively) and memory bandwidth (shared by all four cores).

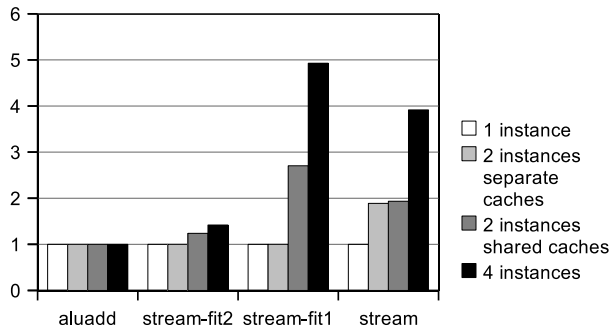

Figure 1. Normalized runtime of microbenchmarks running on the Core2 Quad

We selected microbenchmarks that differ in their use of the named resources: `aluadd` performs integer additions exclusively on the CPU registers. `stream` is a memory benchmark [15]. While originally, `stream` works on a data array considerably larger than the cache, we also created modified versions of the benchmarks that work on data arrays that fit into the L2 cache once (`stream-fit1`) or twice (`stream-fit2`).

Figure 1 shows the runtimes of the microbenchmarks when run alone, together with another instance of the same benchmark running on a core using a different L2 cache, together with an instance on a core using the same cache, and together with three instances on the other cores. All runtimes are normalized to the runtime of one instance running alone.

As expected, `aluadd`'s runtime is not influenced by other cores. The runtime of `stream-fit2` increases slightly when another instance uses the same cache because of conflict misses. The runtime of `stream-fit1` increases considerably when two instances share a cache because of conflict and, mainly, capacity misses. When four instances are running, memory contention causes a further increase in runtime. Finally, the original memory-bound `stream` suffers from memory contention already when two instances are running on different caches.

We did the same evaluation using the SPEC CPU 2006 benchmarks (Figure 2). Many SPEC benchmarks (those shown on the left half of the figure) behave like `aluadd` and `stream-fit2`, showing no or only little slowdown even when combined on the same cache. Of the benchmarks affected by resource contention (those on the right side of the figure), with the exception of `sphinx3` and `bzip2`, all show a notable increase in runtime already when running on cores with separate caches.

The results of the experiment with the SPEC benchmarks indicate that memory bandwidth is the critical resource for these benchmarks, and that the case where one task's working set fits into the cache but two tasks' working sets do not is rare. (Few benchmarks behave like `stream-fit1`, show-

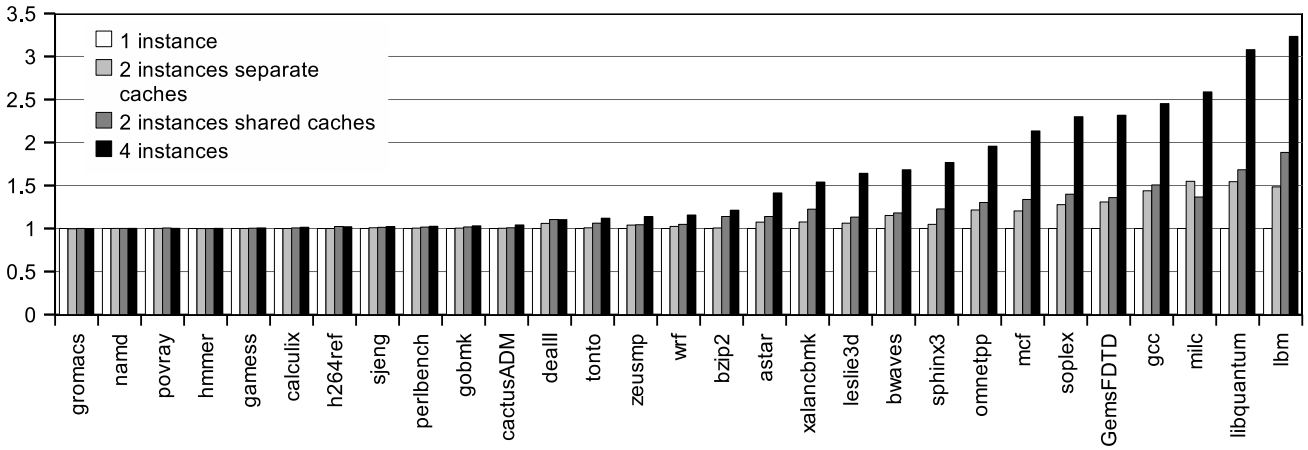

Figure 2. Normalized runtime of SPEC benchmarks on the Core2 Quad

instances	stream time	stream ener.	stream EDP	aluadd time	aluadd ener.	aluadd EDP	avg. EDP
4 aluadd	—	—	—	1.49	1.16	1.68	1.68
1 str. + 3 alu.	1.13	0.83	0.93	1.49	1.08	1.63	1.45
2 str. + 2 alu.	1.07	0.77	0.82	1.49	1.10	1.60	1.23
3 str. + 1 alu.	1.09	0.85	0.93	1.49	1.13	1.73	1.13
4 stream	1.04	0.80	0.83	—	—	—	0.83

Table 1. Relative runtime, energy, and EDP of microbenchmarks at 1.6 GHz compared to 2.4 GHz

ing little interference when running on different caches and heavy interference when running on the same cache.) Therefore, we will concentrate on memory bandwidth as constraining resource.

3.2 Frequency selection

As mentioned in Section 2, past research has indicated that memory-bound tasks are best executed at low frequencies, while compute-bound tasks are best executed at high frequencies. Based on this foundation, we want to explore (a) what frequency setting is optimal if multiple cores running applications with different characteristics have to share the same setting, and (b) whether it is beneficial to co-schedule similar applications in order to be able to run each one at its optimal setting.

For investigating the effects of frequency scaling, we ran different combinations of the `aluadd` and the `stream` benchmark on the cores. Table 1 shows the factor by which the EDP changes for each benchmark when dropping the frequency from 2.4 GHz to 1.6 GHz.

Since `aluadd` is compute-bound, its runtime increases when the frequency is reduced. This increase outweighs the decrease in power consumption, so the consumed energy and the EDP increase. For `stream`, the runtime hardly increases when the frequency is lowered, so here the EDP is dominated by the power consumption and thus decreases. However, when looking at the averaged EDP of all tasks, only a combination of four memory-bound tasks justifies frequency scaling.

The same holds true for the SPEC benchmarks, for which we obtained similar results. Figure 3 shows the runtime, expended energy, and EDP for four instances of each SPEC benchmark at the reduced frequency of 1.6 GHz, normalized to the respective values at 2.4 GHz. The order of the benchmarks is the same as in Figure 2. For the compute-bound benchmarks on the left, the same effect as for `aluadd` can be observed: runtime increases, negating the power savings and leading to an increased EDP. The memory-bound benchmarks on the right behave similarly to `stream`; their runtime only increases moderately, so frequency scaling yields energy savings and a reduced EDP.

As for the microbenchmarks, the reduction of EDP for memory-bound benchmarks is not nearly as big as the increase for the compute-bound benchmarks, so in order to profit from frequency scaling, only memory-bound tasks would have to be co-scheduled.

Hence, if we have more tasks available for execution than there are execution contexts, the question arises whether it is better to run memory-bound tasks together in order to be able to profit from frequency scaling, or to run compute-bound with memory-bound tasks in order to avoid resource contention. When we compare Figures 2 and 3, we see that the benchmarks that profit from DVFS are exactly the ones that suffer a tremendous slowdown when run in multiple instances because of contention. Contention causes a huge slowdown when all cores execute memory-bound tasks (see Figures 1 and 3), and thus the tasks consume power for a longer time, which outweighs the reduction in power consumption achievable by frequency scaling by far. This is not apparent in Figure 3, since the baseline already includes the penalty of co-scheduling four instances of a benchmark.

When looking at the results from both experiments, only the increase in runtime stemming from co-scheduling

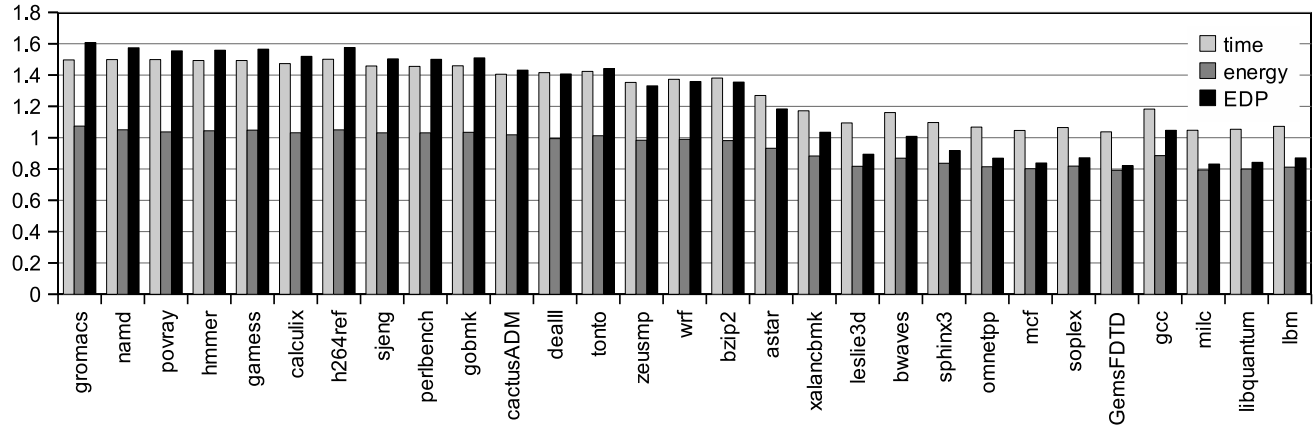

Figure 3. Relative runtime, energy and EDP of SPEC benchmarks at 1.6 GHz compared to 2.4 GHz

memory-bound tasks outweighs the reduction of EDP achievable by DVFS, even when the increase in expended energy caused by the longer runtime is not considered. So overall, more energy has to be spent when combining memory-bound tasks and using frequency scaling than when combining compute and memory bound tasks and running them at the highest frequency.

We illustrate this again with the results of an experiment with two SPEC benchmarks. We run four instances of `soplex`, a memory-bound benchmark and four instances of `hmmer`, a compute-bound benchmark. We compare the following three scheduling scenarios:

1. Run the four instances of `soplex` at their optimal frequency of 1.6 GHz, then run the four instances of `hmmer` at their optimal frequency of 2.4 GHz.
2. Run two instances of `soplex` with two instances of `hmmer` at 2.4 GHz and repeat.
3. Run two instances of `soplex` with two instances of `hmmer` at 1.6 GHz and repeat.

Table 2 shows runtime, CPU energy, and EDP for the scenarios. In scenario 1, resource contention slows down the four instances of the memory-bound `soplex` running in parallel, causing them to consume power for a longer time and resulting in the highest energy consumption of all three scenarios. Scenario 3 shows the lowest energy consumption. However, running the compute-bound `hmmer` at 1.6 GHz increases the total runtime substantially. As expected, scenario 2 shows the best EDP; here the benchmarks can be executed requiring only 80% the EDP of the other two scenarios.

3.3 Results for the AMD Opteron

We also conducted experiments with the microbenchmarks described above as well as with the SPEC benchmarks on a 2.2 GHz AMD Opteron 2354 quad-core. As on the Intel Core2, memory-bound benchmarks scheduled together suffer from substantial slowdown because of contention, although the slowdown is less severe than with the Core2. We attribute this to the integrated memory controller in the Opteron, as opposed to the front-side bus used in the Core2. Four instances of `stream` running together on the four cores of the Opteron are slowed down by a factor of 2.7 compared to a single instance (Core2: factor 3.9); the most memory intensive SPEC benchmark, `lbm`, suffers a slowdown of factor 2.5 (Core2: factor 3.2).

Our test chip supports frequency scaling to 2.0, 1.7, 1.4, and 1.1 GHz, with individual frequencies per core. Voltage is scaled accordingly, but all four cores are required to run at the same voltage. As with the Core2, the benchmarks that profit from frequency scaling are exactly the ones that are affected most by memory contention. Since the Opteron allows lower frequencies that the Core2, it offers more potential to conserve energy for memory-bound tasks. `lbm` profits most from frequency scaling; at the lowest processor frequency of 1.1 GHz, the benchmark can be executed with only 0.64 times the EDP compared to 2.2 GHz. (Core2: factor 0.87 at 1.6 GHz).

Although memory contention is not as severe on the Opteron as on the Core2 and although the Opteron can improve energy efficiency for memory-bound tasks more than the Core2, the loss of energy efficiency caused by co-scheduling memory-bound tasks still outweighs the benefits achievable with DVFS. Like with the Core2, only the increase in runtime that results from co-scheduling memory-bound tasks offsets the reduction of EDP that is achievable by frequency scaling, as can be seen at the example of `lbm`. The same is the case for the other memory-bound benchmarks, which suffer from less slowdown when co-scheduled, but for which frequency scaling also yields smaller savings.

4. Representing Resource Utilization

According to the results of our analysis, the scheduling policies we present in the following strive to combine tasks with

	time [s]		energy [kJ]		EDP [MJs]		
Scenario	hmmer	soplex	hmmer	soplex	hmmer	soplex	average
1: hmmer @ 2.4 GHz, soplex @ 1.6 GHz	923	1310	17.0	13.7	15.6	18.0	16.8
2: hmmer + soplex @ 2.4 GHz	952	837	15.7	13.8	15.0	11.6	13.3
3: hmmer + soplex @ 1.6 GHz	1420	911	17.0	10.9	24.1	9.9	17.1

Table 2. Runtime, energy, and EDP of the benchmark instances for different scheduling scenarios

different characteristics, and only engage frequency scaling if nothing but memory-bound tasks are available.

For representing resource utilization of a task, we use the concept of *task activity vectors*. In previous work, we have introduced task activity vectors as a means for task characterization in the context of temperature-aware scheduling [18]. An *activity vector* is part of a task's runtime context; it describes to what degree a running tasks utilizes various processor-related resources. The dimension of this vector is equal to the number of resources we want to consider. Each component of the vector denotes the degree of utilization of a corresponding resource. The components are normalized to the maximum utilization the respective resource can exhibit. Thus, the values of the vector's components range between 0 (no utilization) and 1 (maximum utilization).

Task activity vectors make the CPU resources a task uses part of the task's runtime context, so the operating system has detailed information about the characteristics of each task. Determining a task's activity vector requires determining the utilization of each resource. Information about the utilization could be provided directly by the hardware, for example via special registers. Unfortunately, this is not the case in today's processors. Therefore, we use performance monitoring counters to determine utilization, which were originally introduced for profiling and performance analysis. The counters count performance critical events such as bus transactions or cache requests and are therefore suitable to infer resource utilization.

For the Core2 processor, we decided to include three resources in the activity vector: memory bus, L2 cache, and "the rest of the core". While memory bus and L2 cache are the resources for which there is contention, the resource "rest of the core" stands for all resources which are not shared between cores, such as, for instance, L1 cache or integer and floating point units.

The activity vector of a task is not constant, but can change over time, as the task passes through different phases, for instance, runs different algorithms successively. Therefore, the operating system has to recalculate the activity vector of a task continuously. On every timer tick and on every task switch, we determine the utilization of the named resources by reading performance monitoring counters and update the activity vector of the currently running task. Note that, as a result, we assign events caused by asynchronous activity such as interrupt handling to the currently running task, although the activity may have been triggered by I/O requests of other tasks. Since we focus on tasks that do little I/O, we can neglect the potential error this introduces; the question of how asynchronous activity can be accounted for is a topic for future work.

Changing the processor frequency has an impact on task activity vectors. However, frequency changes affect different components in different ways: For example, memory bus utilization decreases with lower chip frequencies, since at a lower frequency, a core can issue fewer memory requests per time. On the other hand, the utilization of other chip resources increases, since at a lower frequency, the resources are able to process less instructions per time and thus the chip exhibits fewer stall cycles while waiting for memory. Especially when the frequency changes often, it is likely that there are tasks whose activity vectors were sampled at different frequencies. In order to be able to compare those vectors, it is necessary to predict what a task's activity vector would look like at another frequency.

For being able to compare activity vectors that were sampled at different frequencies, we supply a *translation vector* for each chip frequency. The components of the translation vector denote by which factor the components of an activity vector are expected to change when the task is running at the respective frequency, compared to the maximum frequency. We calibrate the translation vectors by running a set of representative benchmarks at each processor frequency while simultaneously monitoring resource utilization.

We can estimate the impact of a frequency change on an activity vector by doing a component-wise multiplication of the activity vector with the corresponding translation vector. We apply this translation when determining the activity vectors of tasks executed at a frequency lower than the maximum frequency. Hence, the activity vector of a task always denotes the (estimated) utilization the task would cause if run at the maximum frequency.

5. Resource-conscious Scheduling

Our analysis found that avoiding resource contention is of paramount importance for achieving an optimal EDP; we therefore strive to add resource contention awareness and avoidance to timeslice-based multiprocessor scheduling policies as they are found in today's general purpose operating systems. Since the degree of contention depends on the

combination of tasks running simultaneously on the execution contexts, we need to control the combination of tasks that run at a time.

This leads to the concept of gang scheduling, first proposed by Ousterhout [23]. While gang scheduling was proposed to co-schedule threads of the same multithreaded application in order to facilitate communication, our goal is to combine tasks using mutually exclusive resources if possible. As mentioned in the introduction, in order to limit the complexity of our research, we only consider independent single-threaded tasks, that is, we assume that there is no communication between tasks. If there is communication, co-scheduling based on communication patterns and co-scheduling based on resource utilization can have conflictive goals. This is a topic for future work.

A suitable distribution of tasks to processors and of applications to machines is a prerequisite for being able to co-schedule tasks, since co-scheduling depends on tasks with different characteristics being available. In the following, we will introduce a migration policy that makes sure that tasks with various resource utilization characteristics are available on each processor for co-scheduling. Based thereon, we present a co-scheduling policy that combines tasks with complementary characteristics. Our policies are applicable for systems that organize their tasks in CPU-local runqueues and perform round-robin-like scheduling on each CPU.

Since our policies are based on activity vectors, which represent resource utilization in an architecture-independent way, our policies are highly portable. In order to apply them to a new CPU architecture, only the underlying activity vectors (that is, the choice of vector components and the mechanism for determining them) need be adapted.

We apply our policies both to single-threaded applications and to entire VMs running on a single node; by means of VM migration, we further extend the migration policy to VMs running on multiple different nodes. Note again that we refer to a *task* to encompass both applications and VMs.

5.1 Vector balancing

We propose vector balancing as a policy that reduces resource contention by means of task migrations guided by activity vector information. A simple solution for distributing tasks to cores would be to collect tasks with similar characteristics on one core, for example, to run all memory-bound tasks on core 0 and all compute-bound tasks on core 1 of a dual-core processor. This way, even without a co-scheduling policy, there would never be two memory-bound task running simultaneously on the chip. However, it is not always optimal to collect similar tasks on one core. For example, several cache-intensive tasks on one core can overwrite each other's working sets.

Also, it is not always possible to divide tasks into sets that use mutually different resources; for instance, assume a situation with two tasks of medium memory intensity, a memory-bound task and a compute-bound task.

The goal of our balancing policy is to have tasks with different characteristics available on each core, so that a co-scheduling policies has a higher chance of finding a suitable task on each core. In other words, we want to have a high variance among unit utilization on each CPU. Therefore, we define variance to be our measure for balancing, or more specifically, the sum over the variance of all vector components, which we define formally in the following way:

Definition. *Let x_i be a random variable describing the i–th component of the activity vector of a task picked at random from a particular core's runqueue, and $V(x_i)$ the statistical variance of the random variable x_i. Then our measure for balancing is the sum of all components' variances,* varsum*:*

$$\text{varsum} \quad := \quad \sum_{i=1}^{n} V(x_i) \tag{1}$$

We express the Variance by using the expected values of x_i and x_i^2:

$$\text{varsum} \quad = \quad \sum_{i=1}^{n} E\left(x_i^2\right) - \left(E\left(x_i\right)\right)^2 \tag{2}$$

Since x_i is discrete (there is only a limited number of tasks in each runqueue), we can express $E(x_i)$ as $E(x_i) = \frac{1}{m}\sum_{j=1}^{m} a_{ji}$, where m is the number of tasks in the queue, and a_{ji} is the *i*–th component of the *j*–th task's activity vector. The same applies to the squared values.

Thus, we can calculate *varsum* if we keep track of the sum of the activity vectors of tasks in a runqueue and of the sum of the vectors, squared per component:

$$\text{varsum} \quad = \quad \sum_{i=1}^{n} \left(\frac{1}{m} \sum_{j=1}^{m} a_{ji}^2 - \left(\frac{1}{m} \sum_{j=1}^{m} a_{ji} \right)^2 \right) \tag{3}$$

Since our goal is to have a high variance in each runqueue, we strive to increase the *varsum* metric by task migrations. We migrate a task from one CPU to another, if after the migration the minimum of the *varsum*s of the two CPUs is bigger than before.

We integrate our balancing policy into the load balancing policy of the operating system. Since vector balancing is based on a scalar measure, we can easily adapt existing policies that are based on runqueue length to take the *varsum* metric into account. Thus, we can take advantage of existing strategies that honor hierarchical structures, for instance in non-uniform memory access systems, and provide scalability for systems with large numbers of processors. For instance, Linux uses a hierarchical load balancing algorithm that resolves load imbalances at the lowest level of the system's topology possible.

Whenever the scheduler checks for load imbalances, we make an additional check for opportunities to improve the *varsum* metric by migrations. In addition, we modify the algorithm for resolving load imbalances in a way that the induced task migrations do not decrease the *varsum* metric.

5.2 Cross-node migration

Taking tasks on multiple (physical) nodes into account for resource-conscious scheduling potentially increases workload diversity, and thus improves energy efficiency. Although migration of individual applications across nodes is a viable approach, it is also known to be cumbersome, mostly because of the problem of residual dependencies [19]. Virtualization technology leads a way out of this problem, since it encapsulates most of the dependencies in VM containers relatively easy to migrate [3].

We therefore leverage VM migration to allow resource-conscious scheduling across different computing nodes. Our cross-node migration algorithm is derived from the node-local version and implements a VM migration policy based on the *varsum* metric introduced in the previous section. In the cross-node case, however, we consider the resource utilization of *all* CPUs of a node when computing *varsum* rather than of individual CPUs only. Also, we *exchange* VMs right away rather than waiting for a complete balancing period for a migration decision to even out again. That is, the algorithm swaps two VMs on different hosts if, after the migration, the minimum of the *varsum* of the two nodes is bigger than before.

Cross-node migration is a heavy-weight operation, which leaves a potentially large cache and memory working set behind on the source node, induces additional power costs during the migration operation itself, and leads to substantially higher resource demands on the target node. Therefore is indicated to use larger time intervals between potential migration decisions (in the order of minutes). Note that, although our present algorithm is effectively capable to save energy (Section 8.4), it does not trade-off the mentioned costs at present; rather it uses fixed timing intervals and only optimizes for the *varsum* metric. A more elaborate algorithm remains subject of future work.

5.3 Sorted co-scheduling

Our co-scheduling policy concentrates on one resource that is assumed to be mainly responsible for contention. This requires knowledge about which resource is most important for avoiding contention, for example, that memory bandwidth is more important than cache for the SPEC benchmarks on the Intel Core2 processor.

Our policy is similar to previous approaches that have co-scheduled tasks with complementary resource demands [11, 31]. In contrast to related approaches, we make use of sorting and arrange the runqueues of the individual cores in a defined fashion in order to minimize the need for synchronization between scheduler instances running on the individual cores. In addition, our policy is not fixed to a certain resource by design. In the following, we will use memory bandwidth as an example for the most critical resource. However, our policy can consider any component of the activity vector and avoid contention for the respective resource. Since not only the architecture, but also the workload determines which resource is the most critical one, even a dynamic policy which changes the vector component considered according to the workload would be beneficial.

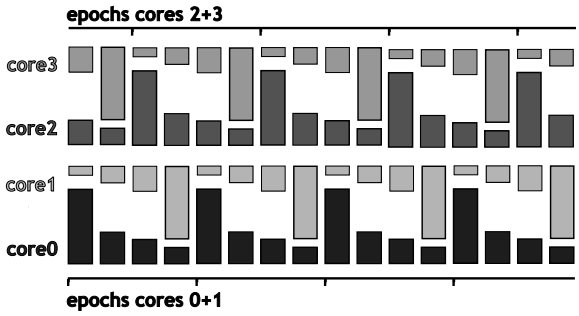

Figure 4. Sorted scheduling. Bars correspond to memory intensity.

The idea behind sorted co-scheduling is to group the cores into pairs of two and to execute tasks with complementary resource demands on each of them. For this purpose, we keep all runqueues sorted. In previous work [18], we have shown that it is possible to sort a runqueue lazily with low overhead, and have made use of sorting to arrange the tasks in a runqueue in a way that reduces hotspots on the chip. In this work, in order to avoid resource contention, we use the utilization of a single resource, in our case memory bandwidth, to sort the tasks in each processor's runqueue. For combining tasks with complementary memory bandwidth demands, we sort the tasks descendingly in runqueues of cores with even processor numbers and ascendingly for odd processor numbers.

To synchronize the scheduling decisions on a chip, we divide time into epochs. We choose an epoch to correspond to the timeslice length multiplied by m, where m is the maximum number of tasks in any of the chip's cores' runqueues. Thus, during an epoch, each core with exactly m tasks in its runqueue can process the runqueue exactly once. To achieve the same property for queues with fewer tasks, we execute each task in a runqueue with n tasks not for one timeslice, but for $\frac{m}{n}$ timeslices. The tasks in the runqueues are sorted in different directions, resulting in combinations of memory-bound and compute-bound tasks running at a time. Since only the starts of the epochs need be synchronized, the overhead for synchronizing scheduling decisions is small.

If there are more than two cores on a chip, to avoid running tasks with high memory intensity at the same time, we shift the beginning of the epochs for each additional pair of cores. For instance, on a quad core, we start a new epoch on cores 2 and 3 whenever cores 0 and 1 are in the middle of their epoch. This way, situations when cores 0 and 1 possibly both execute tasks with relatively low memory demands in the middle of their epoch can be used to run the most memory-bound task of cores 2 and 3 (see Figure 4).

Sorting the runqueues of the cores can be in conflict with scheduling schemes used for I/O-bound tasks. For instance, I/O-bound tasks typically receive a higher priority than compute-bound tasks and are scheduled preferably. However, our policy is focused at tasks that do no or only little I/O. I/O-bound tasks frequently block upon an I/O request before their timeslice expires and unblock again when the I/O request completes. Therefore, with I/O-dominated workloads, the contents of the runqueues constantly change and a sorting approach does not make sense. A scheduling policy for I/O-dominated workloads is a topic for future work.

6. Frequency Heuristic

In Section 3, we have pointed out that the primary lever to achieve energy efficiency is combining tasks in a way that avoids resource contention. Hence, we use frequency scaling only as a fallback engaged in situations when there are too many memory-bound tasks and our scheduling policies cannot avoid contention. In this section, we present a simple heuristic that detects such situations and reduces the common frequency and voltage of the cores of a chip in order to improve energy efficiency despite the workload being unsuitable for co-scheduling.

Sorted co-scheduling, being designed to avoid contention, also facilitates frequency selection, since the scheduling policy controls the combination of tasks running at a time. Hence, scheduling decisions do not occur randomly and independently across cores, and we can choose a frequency fitting the characteristics of the currently running tasks. Also, with sorted scheduling, we can expect the characteristics of tasks executed to change monotonically for each runqueue, which reduces the number of frequency changes.

Heuristics found in the literature are mostly based on the metrics of memory access frequency and on-chip or cache activity [4, 12, 30]. Snowdon et al. [27] has shown that the runtime of and the power consumed by a particular task at a certain frequency can be predicted using several architecture-specific performance counter metrics, which in turn can be used to infer the optimal frequency for running a task. Since our focus is not on finding a sophisticated heuristic, but rather on the interactions of resource contention and frequency scaling, we use a rather simple heuristic based solely on memory intensity.

The experiments with the microbenchmarks presented in Section 3 indicate that, on average, the EDP of tasks with memory utilization of 0% scales by a factor of 1.67 when lowering the frequency, whereas the EDP of tasks with memory utilization of 100% scales by factor of 0.88 (Table 1). Based on these corner cases, we would estimate the scaling of EDP of a task with memory utilization x using linear interpolation:

$$\text{EDP factor} = x * 0.88 + (1-x) * 1.67 = 1.67 - 0.79x \quad (4)$$

Note that using linear interpolation assumes a very simple processor. It does not account for features that modern processors possess for hiding memory latencies, such as out-of-order execution with outstanding loads or write buffers. As a result, the degree of slowdown that results from limited memory bandwidth depends on the instruction-level parallelism the task exhibits. Our benchmark used for calibration, the `stream` benchmark, performs loops over arrays, and the individual iterations are independent from each other, resulting in high instruction-level parallelism. Real-world applications can show less instruction-level parallelism, resulting in a lower EDP factor. (They benefit from lower frequencies already at lower memory utilizations than interpolated from the microbenchmark.)

Considering this effect despite our very simple model requires fine-tuning the model parameters. Engaging frequency scaling already for tasks with lower memory utilization can be accomplished either by reducing the constant or the proportional part of Equation 4. We performed experiments with several memory-intensive SPEC benchmarks and found modifying the constant part while only slightly changing the proportional part to be a viable solution. For our experiments presented in the next sections, we use the following estimation of the EDP factor:

$$\text{EDP factor} = 1.6 - 0.8x \quad (5)$$

Whenever a task switch has occurred on a core, we check whether a task switch is likely to occur on one of the sibling cores in the near future. We can infer this information from the current point in time within the epoch and the number of tasks in each other core's runqueue. If no switch is likely to occur on any other core of the chip, we check how the EDP of each task currently running on the chip would scale according to the EDP factor, and select a frequency of 1.6 GHz if the average of all EDP factors is smaller than one, and a frequency of 2.4 GHz otherwise.

Although our model is very simplistic and needs to be fine-tuned to the actual workload, it allows us to demonstrate how co-scheduling and frequency selection interact. However, since any selection policy that chooses an optimal frequency for a given workload can be applied on top of our co-scheduling policy, for deployment in a production system, a more sophisticated policy like those found in the literature could be used.

7. Implementation

We implemented support for task activity vectors, our proposed scheduling policies, and our frequency heuristic for a Linux 2.6.22 kernel. We based the implementation of VM scheduling on the KVM virtualization environment.

7.1 Activity vectors

For implementing activity vectors in Linux, we extend the `task_struct` data structure, which holds a task's runtime context, by an array for storing the vector's components.

We also export the activity vector of each task via the `/proc`–file-system to serve as input for the user-level programs that control virtual machine migration between nodes (see Section 7.3).

While our scheduling policies and the concept of activity vectors are platform independent, determining unit utilization and hence calculating the vector components depends on the chosen platform. In the following, we describe briefly how we determine the utilization of the resources represented by activity vectors for our evaluation platform, the Intel Core2 architecture. We believe, however, that activity vectors can be implemented on any platform that offers event monitoring counters capable of capturing the utilization of shared resources, which is the case with the performance monitoring counters available on many modern processors. For instance, in previous work, we have implemented activity vectors for the Intel NetBurst architecture [18].

Memory bus For determining bus utilization, we count the number of bus transactions initiated by a core during a timer tick and divide it by the theoretical maximum number of transactions the hardware supports during this period of time. The theoretical maximum is determined by the bandwidth of the memory bus and by the speed of the memory itself. Our test system has a front-side bus supporting 1066MT/s (MT/s: millions of transactions per second) and DDR2 PC-6400 RAM supporting 800MT/s, so the transfer rate is limited by the ram and is 800MT/s at most.

L2 Cache For the Core2 architecture, there is a large number of events available for counting requests to the L2 cache, including loads, stores, invalidations, and prefetch requests. Since there are not enough counters to count each event separately, we had to use the event `L2_RQSTS`, which accumulates all types of requests. However, it is hard to give a theoretical maximum rate for this event, since different kinds of requests have different maximum rates (for example, a prefetch from memory to L2 takes longer than a transfer from L2 to L1). To estimate an upper limit, we ran the `stream` memory benchmark, but with a reduced working set fitting completely into the L2 cache. We measured the number of L2 references during the run, which amounted to one reference every four cycles, and chose this value as the maximum.

Rest of the Core Rather than counting events for the various core units (which is impossible in practice because of a limited number of performance counters in the Core2 architecture), we use the number of retired instructions as a proxy for core activity. Doing so neglects various aspects, for instance, that different instructions keep the core busy for different amounts of time, and that some instructions do not retire because of misprediction. To obtain utilization, we divide the number of retired instructions by the total number of processor cycles.

Since a core can execute micro-operations in parallel, the number of retired instructions per cycle can be greater than one, meaning that the count of retired instructions is greater than the cycle count. We define core activity to be at 100% if the count of retired instructions is equal to or greater than the cycle count.

Our method is only a very rough estimation for the utilization of the resources of a core that are not shared with other cores, but proved sufficient for our purposes. Typically, memory or cache-bound tasks, which do not utilize other resources heavily, show a low number of retired instructions per cycle, while tasks with low memory and cache utilization that are bound to other resources show a high number of retired instructions per cycle.

7.2 Vector balancing and co-scheduling

For implementing vector balancing within single nodes, we modify Linux's load balancing algorithm to consider our measure of variance as defined in Section 5.1 next to load. Linux pursuits a hierarchical approach that strives to resolve load balances at the lowest level of the system's topology possible. Therefore, at each level of the topology starting with the lowest, the load balancer determines the CPU with the shortest and the longest runqueue, and, if needed, chooses a suitable task to migrate according to various criteria, for instance, that the task is not currently running, and is expected to have little data in the processor's cache.

We extend this algorithm in two ways: Firstly, we add to the mentioned criteria the requirement that the *varsum* metric of the runqueues the task is migrated between does not increase. Secondly, we initiate balancing operations between the runqueue with the highest ant the lowest *varsum* on each level of the topology, provided that the difference between these *varsum* values is greater than a pre-defined threshold.

Introducing coordinated co-scheduling into the Linux scheduler requires only few modifications. We replace the logic that checks whether a task's timeslice has expired. For implementing sorted scheduling, we switch tasks whenever one *n*th of an epoch has passed, where *n* is the number of tasks in the runqueue.

7.3 Virtual machine scheduling

To evaluate the benefits of virtualization in combination with resource-conscious scheduling, we have implemented node-local and cross-node VM scheduling based on the KVM 82 virtualization system [10]. In the node-local case, KVM spawns normal Linux tasks to host guest VMs, and therefore allows us to directly use our Linux scheduler for both tasks and VMs.

KVM also supports several types of cross-node migration (e.g., offline and live, and based on TCP sockets or files to transfer VM state). Our cross-node scheduler implementation consists of a script that runs at user-level on each participating node and leverages the existing KVM migration functionality. The script periodically (in our experiments ev-

ery 15 seconds) monitors the activity vectors of its own VMs and exports them to all other nodes by means of a shared NFS folder. At larger periods (in our experiments 10 minutes), the script balances VMs across nodes according to the policy described in Section 5.2.

Since the live migration proved to be unstable when being used with our extended kernel, we had to resort to an offline version; that is, to relocate a VM, the source host first stops the VM and then saves its memory and CPU state into a file in a shared folder. The target host afterwards loads the state files and then resumes the VM. Offline migration obviously slows down the VM transfer and breaks seamless user experience. Since our evaluation was based on benchmarks requiring no user interaction, those problems do not affect the validity of our results.

7.4 Frequency selection

Under Linux, the processor frequency is controlled by a governor framework, which allows selecting a policy for frequency scaling from user space. The framework is not suitable for initiating frequency switches from the scheduler; in particular, the functions provided by the device drivers controlling the chip frequency are not intended to be called from the scheduler, but only from the governors running in task context.

We therefore deactivated the framework and the driver and implemented our own prototype method for selecting a suitable frequency. To set the chip frequency according to the decisions of our frequency heuristic, we directly program the model specific registers of the Core 2. Ultimately, it would be beneficial to adapt the device drivers to support frequency changes initiated by the scheduler.

8. Evaluation

We evaluated our implementation on the Intel Core2 Quad described in Section 3, using the SPEC CPU 2006 benchmarks. For evaluating cross-node scheduling, we used a second machine with the same configuration; both machines were connected via Gigabit Ethernet.

In a preliminary experiment, we evaluated the overhead of maintaining activity vectors (without using them for scheduling). Determining activity vectors requires reading a small number of performance monitoring counters and few arithmetic operations. We measured the cost for reading a performance monitoring counter to be 54 cycles on our processor. Since activity vectors are only updated every timer interrupt and every task switch, we could not notice any increase in runtime for the SPEC benchmarks caused by enabling activity vectors compared to a setup without activity vectors enabled.

8.1 Sorted co-scheduling

To evaluate sorted co-scheduling, we ran four compute-bound benchmarks (`gamess`, `gromacs`, `hmmer`, `namd`) together with four memory-bound benchmarks (`lbm`, `libquantum`, `mcf`, `soplex`). Figure 5 shows runtime and EDP of the benchmarks when sorting is applied, relative to the respective values achieved using standard Linux scheduling.

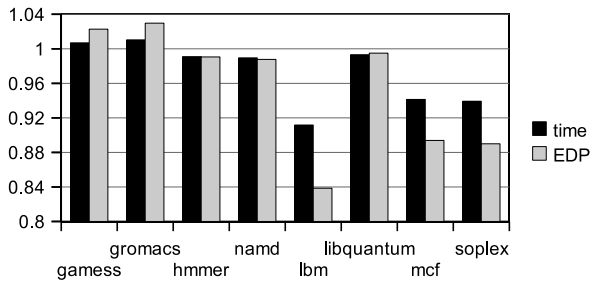

Figure 5. Runtime and EDP of SPEC benchmarks with sorted co-scheduling relative to standard Linux scheduling. Note that for readability, in this figure an in the following ones, the scale does not start at zero.

While the compute-bound benchmarks' runtimes hardly change (they are affected a little, since the compute-bound benchmarks show some memory references, too), the runtimes of all compute-bound benchmarks decrease. The reason why `libquantum`'s runtime is not reduced as much as the other memory-bound benchmarks' runtime is that even with runqueue sorting, two memory-bound tasks have to share the memory bus at a time, and depending on the applications' memory access patterns, bandwidth distribution can be unfair [20].

Since the power consumption is almost the same for sorted scheduling and standard Linux scheduling (frequency scaling is not beneficial, because at any time, two compute-bound tasks are running), EDP is determined solely by the runtime, and varies with runtime, but quadratically.

8.2 Frequency heuristic

Our proposed frequency heuristic engages frequency scaling as a fallback to conserve energy in situations when co-scheduling cannot avoid resource contention. For the experiments with the SPEC scenario presented in the preceding section, our frequency heuristic never invoked frequency scaling, because we selected scenarios containing enough compute-bound tasks for co-scheduling to be effective.

For evaluating the frequency heuristic, we choose four different SPEC scenarios. Scenario 1 contains only compute-bound benchmarks. Scenario 2 contains four compute-bound and four memory-bound benchmarks. For these two scenarios, frequency scaling should not be invoked; they are intended for revealing the overhead of our heuristic. Scenario 3 contains one compute-bound and seven memory-bound benchmarks, and scenario 4 contains only memory-bound benchmarks. Scenarios 3 and 4 represent the two configurations in which frequency scaling is beneficial. Table 3 shows the individual SPEC benchmarks used for the scenarios.

Scenario	**Benchmarks** (c = compute-bound, m = memory-bound)
1	`gamess` (c), `gobmk` (c), `gromacs` (c), `hmmer` (c), `namd` (c), `povray` (c), `sjeng` (c), `tonto` (c)
2	`gamess` (c), `gromacs` (c), `hmmer` (c), `namd` (c), `lbm` (m), `libquantum` (m), `mcf` (m), `soplex` (m)
3	`hmmer` (c), `GemsFDTD` (m), `lbm` (m), `libquantum` (m), `mcf` (m), `milc` (m), `omnetpp` (m), `soplex` (m)
4	`GemsFDTD` (m), `lbm` (m), `libquantum` (m), `mcf` (m), `milc` (m), `omnetpp` (m), `soplex` (m), `sphinx3` (m)

Table 3. SPEC scenarios used for evaluating the frequency heuristic

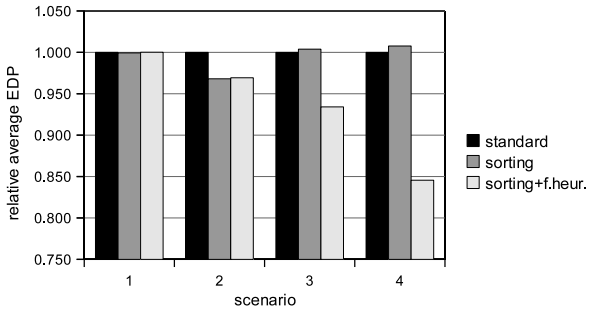

Figure 6. Effect of frequency heuristic for different SPEC scenarios

We compare three configurations: Standard Linux scheduling, sorted co-scheduling without the frequency heuristic, and sorted scheduling plus the frequency heuristic. Figure 6 depicts the average EDP for the SPEC benchmarks of our four scenarios, normalized to the EDP achieved by standard Linux scheduling.

For scenarios 1 and 2, the runtime is almost the same for sorted scheduling and sorted scheduling combined with the frequency heuristic, the only difference being that for scenario 2, sorted scheduling yields a better EDP than standard Linux scheduling by reducing contention. The frequency heuristic yields no benefits for these scenarios, since frequency scaling would lead to increased energy consumption by prolonging the runtime of the compute-bound benchmarks. On the other hand, the heuristic causes no measurable overhead for checking whether frequency scaling should be engaged.

For scenario 3, the heuristic engages frequency scaling whenever the one compute-bound benchmark is not running, which is every other timeslice, yielding an improvement of EDP over standard Linux scheduling and sorted co-scheduling (which is not beneficial, since contention cannot be avoided for the given workload). For scenario 4, runqueue sorting is not beneficial for the same reason. The benefit of the frequency heuristic is even bigger for this scenario, since without any compute-bound tasks, the heuristic can lower the frequency all the time.

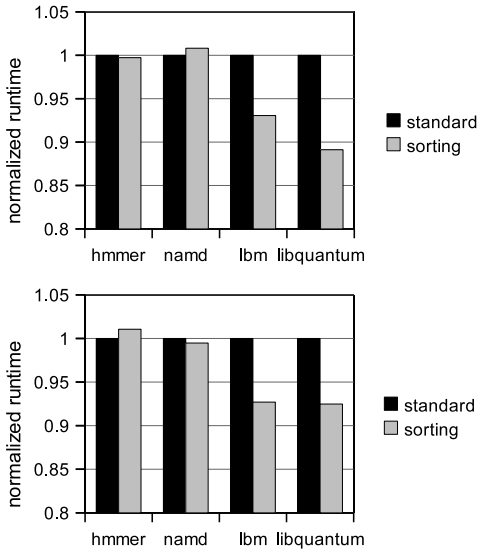

Figure 7. Runqueue sorting applied to benchmarks executed natively (top) and within VM instances (bottom)

8.3 Node-local virtual machine scheduling

We evaluated the effects of our scheduling strategies when applied to VM instances running on a single node. Owing to the limited amount of memory in our test machine, we ran the experiments with only two out of four cores enabled, effectively simulating a dual core.

In a first experiment, we started four VMs, two executing one memory-bound benchmarks each (`lbm` and `libquantum`), and the other two executing one compute-bound benchmark each (`hmmer` and `namd`). We measured the runtime of the SPEC benchmarks within the VMs, once using the standard Linux scheduler in the host, and once using sorted scheduling. For comparison, we did the same experiments without using virtualization, running the benchmarks natively on the host.

Figure 7 on the next page depicts the normalized execution times of the benchmarks for both scenarios. Both for natively executed memory-bound benchmarks and for memory-bound benchmarks executed within a VM, sorted co-scheduling manages to reduce the runtime. The actual reduction of runtime differs between native and virtualized execution. For `lbm`, the reduction is nearly the same both for the virtualized and the native scenario. `libquantum`, on the other hand, profits more from runqueue sorting in the native scenario. We attribute this difference to a slightly different behavior of the benchmarks when running within a VM, caused, for instance, by the need to maintain shadow page tables.

Overall, the results demonstrate that our resource-conscious scheduling strategies are as viable for VMs as they are for normal applications. Virtualization thereby provides the additional benefit of transparency and compatibility, and

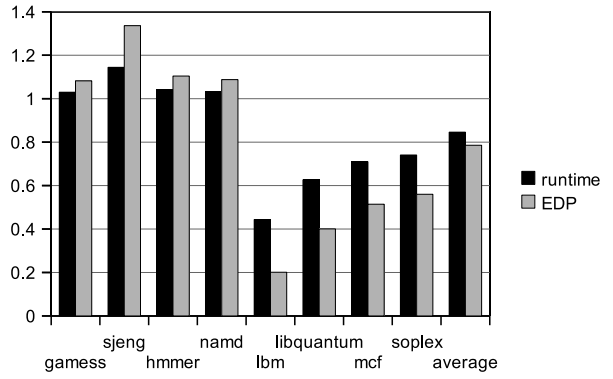

Figure 8. Runtime and EDP of benchmarks with cross-node migration relative to worst case static setup

enables resource-conscious scheduling to be applied in the host-layer and to improve energy-efficiency of virtualized workloads independent of the actual applications or operating system running within the VM.

8.4 Cross-node virtual machine migration

For evaluating cross-node migration, we ran a workload consisting of eight VMs on two physical nodes. On one node, we started four VMs running compute-bound benchmarks (`gamess`, `sjeng`, `hmmer`, and `namd`), on the other one four VMs executing memory-bound benchmarks (`lbm`, `libquantum`, `mcf`, and `soplex`).

For one test run, we disabled cross-node migration; then we repeated the test with our migration policy enabled. Without migration, the machine running the memory-bound benchmarks suffers from severe memory contention, while there is spare memory bandwidth on the other machine. Our policy mitigates the contention by migrating two of the memory-bound benchmarks and two of the compute-bound benchmarks, creating heterogeneous mixes on each machine.

Figure 8 depicts the relative runtimes and EDPs of the benchmarks when cross-node migration is enabled, relative to the scenario without migration. For all compute-bound benchmarks, the runtime and the EDP are reduced dramatically. The respective values of the compute-bound benchmarks are only moderately increased. Only `sjeng` suffers a bigger slowdown, which also leads to increased EDP. When cross-node migration is enabled, `sjeng` has to run on a machine with memory-bound benchmarks, which seems to affect `sjeng` more than the other compute-bound benchmarks. Still, on average, there is a reduction of runtime by 15% and of EDP by 21%.

Since the initial distribution above constitutes a worst-case scenario—memory-bound VMs on one node and compute-bound VMs on the other—we reran the test with a different starting point of three memory-bound and one compute-bound benchmark on one node (`lbm`, `libqantum`, `mcf`, and `gamess`), versus three compute-bound and one memory-bound benchmark on the other one (`sjeng`, `hmmer`, `namd`, and `soplex`). The results are accordingly in that migration still allows to save energy, this time, however, on a smaller scale of 4% reduction in both runtime and EDP on average.

9. Conclusion

In this paper, we have analyzed scheduling for avoiding resource contention and for optimal frequency selection. We have found that the two are oppositional goals, and that scheduling to avoid resource contention is crucial both in terms of performance and energy efficiency. Frequency scaling can lead to further savings, but combining tasks that run best at a certain frequency does not pay off if it leads to resource contention.

Based on the concept of activity vectors for representing resource utilization, we have designed scheduling policies that reduce contention significantly by co-scheduling tasks with complementary demands. By scheduling and migrating entire virtual machines, we have extended our policies to take advantage of workload diversity not only within a single node, but across several nodes. Our evaluations show that our policies are able to reduce EDP for scenarios where there is contention that can be reduced by migration and co-scheduling, or, if that is not possible, mitigated by frequency scaling.

At present, the focus of our strategies are long-running, independent, and compute-intensive tasks, whether they run as normal applications or are contained in a VM; also, the evaluation was based on applications with fairly homogeneous resource consumption patterns. The main area of future work is to tackle and to evaluate more complex application scenarios, for instance, to investigate algorithms for short-lived or heavily I/O bound tasks, or to evaluate highly dynamic resource patterns as they may be found in end-user or server workloads such as an office application or a virtualized web server.

References

[1] C. Antonopoulos, D. Nikolopoulos, and T. Papatheodorou. Scheduling algorithms with bus bandwidth considerations for SMPs. In *International Conference on Parallel Processing*, Oct. 2003.

[2] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, 2005.

[3] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation*, 2005.

[4] G. Dhiman and T. S. Rosing. Dynamic voltage frequency scaling for multi-tasking systems using online learning. In

Proceedings of the 2007 International Symposium on Low-Power Electronics and Design (ISLPED'07), 2007.

[5] G. Dhiman, G. Marchetti, and T. S. Rosing. vGreen: A system for energy efficient computing in virtualized environments. In *Proceedings of the 2009 International Symposium on Low-Power Electronics and Design (ISLPED'09)*, 2009.

[6] A. Fedorova, M. Seltzer, and M. D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, 2007.

[7] V. W. Freeh, F. Pan, D. K. Lowenthal, N. Kappiah, R. Springer, B. L. Rountree, and M. E. Femal. Analyzing the energy-time tradeoff in high-performance computing applications. *IEEE Transactions on Parallel and Distributed Systems*, 18(6), 2007.

[8] S. Ghiasi, T. Keller, and F. Rawson. Scheduling for heterogeneous processors in server systems. In *Proceedings of the Second Conference on Computing frontiers (CF'05)*, 2005.

[9] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid-State Circuits*, 31(9), 1996.

[10] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the Linux Virtual Machine Monitor. In *Proceedings of the Ottawa Linux Symposium 2007*, 2007.

[11] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using os observations to improve performance in multicore systems. *IEEE Micro*, 28, 2008.

[12] R. Kotla, A. Devgan, S. Ghiasi, T. Keller, and F. Rawson. Characterizing the impact of different memory-intensity levels. In *Proceedings of the Seventh IEEE International Workshop on Workload Characterization (WWC-7)*, 2004.

[13] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the* 36*th Annual IEEE/ACM International Symposium on Microarchitecture (MIRCO'03)*, 2003.

[14] V. Kumar and A. Fedorova. Towards better performance per watt in virtual environments on asymmetric single-isa multi-core systems. *SIGOPS Operating Systems Review*, 43(3), 2009.

[15] J. D. McCalpin. Sustainable memory bandwidth in current high performance computers, Oct. 1995.

[16] R. L. McGregor, C. D. Antonopoulos, and D. S. Nikolopoulos. Scheduling algorithms for effective thread pairing on hybrid multiprocessors. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers*, 2005.

[17] A. Merkel and F. Bellosa. Memory-aware scheduling for energy efficiency on multicore processors. In *Proceedings of the Workshop on Power Aware Computing and Systems (HotPower'08)*, 2008.

[18] A. Merkel and F. Bellosa. Task activity vectors: A new metric for temperature-aware scheduling. In *Third ACM SIGOPS EuroSys Conference*, Mar. 2008.

[19] D. S. Milojíčić, F. Douglis, Y. Paindaveine, R. Wheeler, and Z. Zhou. Process migration. *ACM Computing Surveys*, 32(3), 2000.

[20] T. Moscibroda and O. Mutlu. Memory performance attacks: denial of memory service in multi-core systems. In *SS'07: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, 2007.

[21] J. Nakajima and V. Pallipadi. Enhancements for hyper-threading technology in the operating system: seeking the optimal scheduling. In *WIESS'02: Proceedings of the 2nd conference on Industrial Experiences with Systems Software*, 2002.

[22] R. Nathuji and K. Schwan. VirtualPower: coordinated power management in virtualized enterprise systems. In *Proceedings of the Twenty-First ACM SIGOPS symposium on Operating System principles*, Oct. 2007.

[23] J. K. Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems*, 1982.

[24] D. Shelepov, J. C. Saez Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, and V. Kumar. Hass: a scheduler for heterogeneous multicore systems. *SIGOPS Operating Systems Review*, 43(2), 2009.

[25] S. Siddha, V. Pallipadi, and A. Mallick. Process scheduling challenges in the era of multi-core processors. *Intel Technology Journal*, 11(4), 2007.

[26] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous mutlithreading processor. *SIGPLAN Not.*, 35 (11), 2000.

[27] D. C. Snowdon, E. Le Sueur, S. M. Petters, and G. Heiser. Koala: A platform for OS-level power management. In *Fourth ACM SIGOPS EuroSys Conference*, Nuremberg, Germany, Apr. 2009.

[28] J. Stoess, C. Lang, and F. Bellosa. Energy management for hypervisor-based virtual machines. In *Proceedings of the USENIX 2007 Annual Technical Conference*, 2007.

[29] A. Verma, P. Ahuja, and A. Neogi. pMapper: Power and migration cost aware application placement in virtualized systems. In *Proceedings of the ACM/IFIP/USENIX 9th International Middleware Conference*, 2008.

[30] A. Weissel and F. Bellosa. Process cruise control: Event-driven clock scaling for dynamic power management. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'02)*, Oct. 2002.

[31] X. Zhang, S. Dwarkadas, G. Folkmanis, and K. Shen. Processor hardware counter statistics as a first-class system resource. In *HOTOS'07: Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, 2007.