Memory-aware Scheduling for Energy Efficiency on Multicore Processors

Andreas Merkel Frank Bellosa
University of Karlsruhe

Abstract

Memory bandwidth is a scarce resource in multicore systems. Scheduling has a dramatic impact on the delay introduced by memory contention, but also on the effectiveness of frequency scaling at saving energy. This paper investigates the cross-effects between tasks running on a multicore system, considering memory contention and the technical constraint of chip-wide frequency and voltage settings. We make the following contributions: 1) We identify the memory characteristics of tasks and sort core-specific runqueues to allow a co-scheduling of tasks with minimal energy delay product. 2) According to the memory characteristics of the workload, we set the frequency for individual chips so that the delay is only marginal. Our evaluation with a Linux implementation running on an Intel quad-core shows that memory-aware scheduling can reduce EDP considerably.

1 Introduction

Today's operating system schedulers treat cores of a chip multiprocessor (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. For example, many chips only allow setting frequency and voltage for the entire chip.

The optimal frequency at which the processor can execute a task most efficiently in terms of runtime and energy depends on the task's characteristics [5, 11], in particular on the frequency of memory accesses. For a multicore chip that offers global frequency scaling, the question arises whether it is advantageous to run tasks with similar characteristics together in order to run the chip at the corresponding optimal frequency.

On the other hand, the cores of a chip share some resources such as caches and memory interfaces. This is likely to cause contention between the cores if tasks with similar characteristics, for example several memory-bound tasks, are running together [9].

In this paper, we analyze what is the optimal way to co-schedule tasks on a multicore processor considering the criteria of frequency selection and contention. We find that in order to optimize the product of runtime and expended energy (energy delay product, EDP), the main goal must be to avoid contention by combining tasks with different characteristics. Only if nothing but memory-bound tasks are available, it is beneficial to apply frequency scaling.

Based on this analysis, we propose a scheduling policy that sorts the tasks in each core's runqueue by their memory intensity in order to co-schedule memory-bound with compute-bound tasks. Additionally, we apply a heuristic that lowers the frequency when only memory-bound tasks are available. An evaluation with a Linux implementation of sorted scheduling using SPEC CPU 2006 benchmarks reveals that our policies manage to reduce EDP for many scenarios.

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 our scheduling policy and frequency heuristic. Section 5 evaluates our proposed policy and Section 6 concludes.

2 Background and Related Work

Previous research [5, 11] has investigated the problem of selecting a frequency at which to run a task most efficiently. 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. To our knowledge, this is the first work that considers the constraint that multiple cores need to run at the same frequency in this context.

The memory bus has been identified as a bottleneck for symmetric multiprocessor (SMP) systems. For real-time SMP systems, throttling memory-bound tasks in order to guarantee bandwidth reservations has been proposed [2]. Combining tasks based on memory bandwidth demands has been proposed for SMP systems [12, 1] and simultaneously multithreaded (SMT) systems [7]. To our knowledge, no previous research has addressed memory-based co-scheduling and frequency selection in combination. Also, most research concentrates on finding optimal combinations of tasks for co-scheduling, but omits the discussion about how a real multiprocessor scheduler can succeed in combining tasks accordingly.

Up to now, research on co-scheduling for CMP systems has concentrated on the L2 cache as limiting resource [10, 4, 3]. However, our experiments with the recent SPEC CPU 2006 benchmarks suggest that memory contention is becoming more important than cache contention.

3 Analysis

For investigating the effects of contention and frequency selection, we chose a 2.4GHz Intel Core2 Quad Q6600. The chip houses four cores, two of which share 4MB of L2 cache, respectively. In our test system, the chip is connected via a 266MHz front side bus to 8GB of DDR2 PC-6400 memory. The processor supports scaling the frequency down to 1.6GHz. In this case, the core voltage is scaled from 1.20V to 1.12V. (For the rest of the paper, when we speak of frequency scaling, we will imply that voltage scaling is also applied.) This results in a reduction of typical processor power consumption from about 70W to about 45W.

For judging energy efficiency, we sampled the power consumption of the processor using a National Instruments Labview board. We use the EDP of the processor as measure since in many systems the processor is the component requiring most energy and cooling effort, so it is desirable to reduce its power consumption. On the other hand, it is undesirable to sacrifice too much performance. EDP considers both factors.

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).

We selected microbenchmarks that differ in their use of the named resources. `aluadd` performs integer additions exclusively on the CPU's registers. `stream` is a memory benchmark [6].We also use modified versions of the benchmark having working sets that fit into the L2 cache once (`stream-fit1`) or twice (`stream-fit2`).

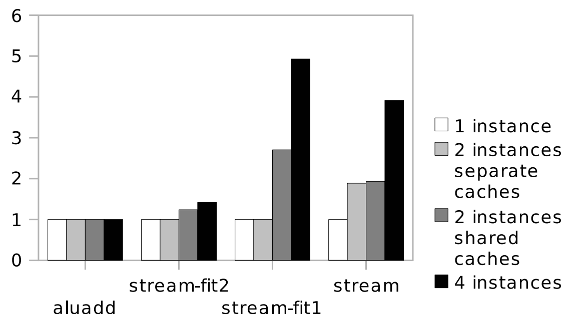

Figure 1: Normalized runtime of microbenchmarks

Figure 1 shows the runtimes for the microbenchmarks running alone, for two instances running on cores not sharing L2 cache, two instances on cores sharing L2 cache, and for four instances. All runtimes are normalized to the runtime of one instance running alone.

As expected, `aluadd`'s runtime is not influenced by other cores. `stream-fit2`'s runtime increases slightly when another instance uses the same cache because of conflict misses. `stream-fit1`'s runtime 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` benchmark suffers from memory contention already when two instances are running on different caches.

We did the same evaluation using the SPEC CPU 2006 benchmarks. It revealed that most SPEC benchmarks either behave like `aluadd` and `stream-fit2`, showing no or only little slowdown even when combined on the same cache, or like `stream`, showing heavy slowdown even when running on separate caches. This indicates that the case that one task's working set fits into the cache but two tasks' working sets do not is rare. Therefore, we will concentrate on memory bandwidth as constraining resource.

3.2 Frequency Selection

For evaluating 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 runtime, expended energy, and EDP changes for each benchmark when dropping the frequency from 2.4GHz to 1.6GHz.

Since `aluadd` is compute-bound, its runtime increases when the frequency is dropped. This increase

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	1.1	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: Effects of frequency scaling

outweighs the decrease in power consumption, so the 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 decreases. However, when looking at the averaged EDP of all tasks, only a combination of four memory-bound tasks justifies frequency scaling. We obtained the same results with the SPEC benchmarks.

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.

The experiments in Section 3.1 indicate a huge performance penalty (increase in runtime of up to a factor of four) for memory-bound tasks running simultaneously (see Figure 1), which leads to an increase in energy consumption by nearly the same factor. This outweighs the reduction in energy consumption achievable by frequency scaling (factor of 0.8, see Table 1) by far. These effects become even more prominent in EDP, which in addition to energy also considers the slowdown introduced by contention and frequency scaling.

We want to illustrate this with the results of an experiment with two SPEC benchmarks. We ran four instances of `soplex`, the memory-bound benchmark we found to profit most from frequency scaling, together with four instances of `hmmer`, a completely compute-bound benchmark. We compare the following three scheduling scenarios:

1. Co-schedule the four instances of `soplex` at their optimal frequency of 1.6GHz, then co-schedule the four instances of `hmmer` at their optimal frequency of 2.4GHz.
2. Always co-schedule two instances of `soplex` with two instances of `hmmer` at 2.4GHz
3. Always co-schedule two instances of `soplex` with two instances of `hmmer` at 1.6GHz

As expected, scenario 2 showed the best EDP; here the benchmarks can be executed requiring only $\frac{3}{4}$ the EDP of the other two scenarios.

Motivated by the results of this analysis, our scheduling policies presented in the next section strive to combine tasks with different characteristics, and only engage frequency scaling if nothing but compute-bound tasks are available.

4 Memory-aware Scheduling

4.1 Runqueue Sorting

We propose a policy for timeslice-based multitasking, multiprocessor scheduling. Our goal is to combine memory-bound with compute-bound tasks to reduce memory contention. Thus, our policy is a special form of gang scheduling.

Whenever the CPU executes a task for one timeslice, we use the processor's performance monitoring counters to determine the memory intensity of the task by counting the number of memory transactions. We use this characterization to sort the tasks in each processor's runqueue by their memory intensity. In previous work [8], we have shown that it is possible to sort a runqueue lazily with low overhead. We sort the tasks descendingly in runqueues of cores with even processor numbers and ascendingly for odd processor numbers to be able to co-schedule tasks of different memory intensities.

To achieve this co-scheduling, we ensure that the cores process their runqueues synchronously. Therefore, we introduce the concept of *epochs*. During an epoch, each runqueue shall be processed exactly once. We achieve this by executing each task in a runqueue for $\frac{1}{n}$ epoch, where n is the number of tasks in the runqueue. As the epoch length, we choose m standard timeslices, where m corresponds to the number of tasks in the longest runqueue. Thus, the tasks in the longest queue get executed for exactly one standard timeslice (100ms in Linux), while tasks in shorter queues get longer timeslices, resulting in all runqueues taking the same time to be processed. Since the tasks in the runqueues are sorted in different directions, this results in combinations of memory-bound and compute-bound tasks running at a time without having too much synchronization overhead.

If there are more than two cores on a chip, we shift the beginning of the epochs for each additional pair of cores. This way, situations when the first two cores both execute tasks with relatively low memory demands in the middle of their epoch can be used to run a memory-bound task on one of the remaining cores (see Figure 2).

To make sure that tasks of different memory intensity levels are available on each core, we employ a balancing mechanism that migrates tasks if needed.

4.2 Frequency Selection

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 if contention cannot be avoided.

On modern processors like the Core2, switching the frequency introduces delays in the order of microsec-

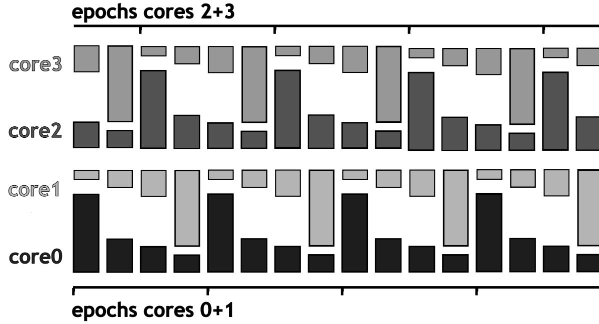

Figure 2: Sorted scheduling. Bars stand for tasks of different memory intensity.

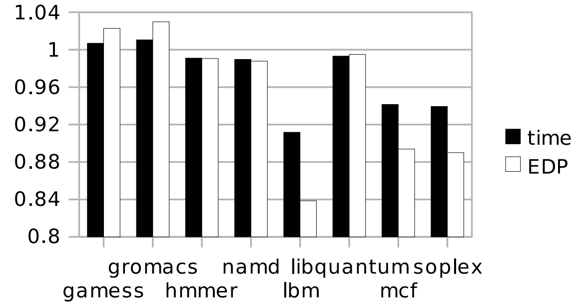

Figure 3: Effects of runqueue sorting: runtime and EDP relative to standard Linux scheduling

onds, which is several orders of magnitude smaller than the granularity of scheduling. This allows to select a suitable frequency on every task switch without introducing noticeable overhead.

In addition, sorted scheduling, being designed primarily to avoid contention, also facilitates frequency selection, since it controls the combination of tasks running at a time. Hence, scheduling decisions do not occur randomly and independently on the cores, which reduces the number of frequency changes.

The experiments with the microbenchmarks presented in the preceding subsection indicate that, on average, the EDP of tasks with memory bus utilization of 0% scales by a factor of 1.67 when lowering the frequency, whereas the EDP of tasks with memory bus utilization of 100% scales by a factor of 0.88 (Table 1). Based on this, we estimate the scaling of EDP of a task with memory bus utilization x with the following equation:

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

In practice, the situation is more complex. Since modern processors can have several memory requests outstanding and still continue executing instructions not dependent on the results of the memory references, the degree of slowdown caused by 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 more from lower frequencies.) In practice, we achieved good results with the following EDP calculation:

$$\text{EDP factor} = 1.4 - 0.8x$$

Whenever a scheduling decision has been made, we check whether the average EDP factor of the tasks currently selected for execution on the cores is smaller than one. If this is the case, we engage frequency scaling, else we disable it.

For chips that support more than two frequency levels, the EDP factor can be calculated analogously for each available frequency, and the frequency with the lowest EDP factor chosen.

5 Evaluation

We implemented sorted scheduling as well as our frequency heuristic for a Linux 2.6.22 kernel and evaluated it on the Intel Core2 Quad described in Section 3, using the SPEC CPU 2006 benchmarks.

In the first test, we evaluated runqueue sorting. We ran four compute-bound benchmarks (`gamess`, `gromacs`, `hmmer`, `namd`) together with four memory-bound benchmarks (`lbm`, `libquantum`, `mcf`, `soplex`) at a fixed frequency of 2.4GHz. Figure 3 shows the runtime and the EDP of the benchmarks when sorting is applied, relative to the runtime and EDP using standard Linux scheduling.

While the compute-bound benchmarks' runtime hardly changes (they are affected a little, since they show some memory references, too), the runtimes of all memory-bound benchmarks decreases. `libquantum`'s runtime is not reduced as much as the other memory-bound benchmarks' runtimes. The reason for libquantum not profiting as much is that even with runqueue sorting, two memory-bound tasks have to share the memory bus at a time, and bandwidth distribution can be unfair [9].

Since the power consumption is almost the same for sorted scheduling and standard Linux scheduling, EDP is determined solely by the runtime, and reduced with runtime, but quadratically. Enabling our frequency heuristic in addition to runqueue sorting yields almost the same results for this scenario (not shown), since at every point in time, two compute-bound tasks are running, so frequency scaling is never engaged.

To evaluate the effects of our frequency heuristic in isolation, we executed four instances of the compute-bound `hmmer` benchmark in parallel, followed by four instances of the memory-bound `lbm` benchmark running in parallel. We compare our adaptive policy with fixed frequency settings of 2.6GHz and 1.6GHz.

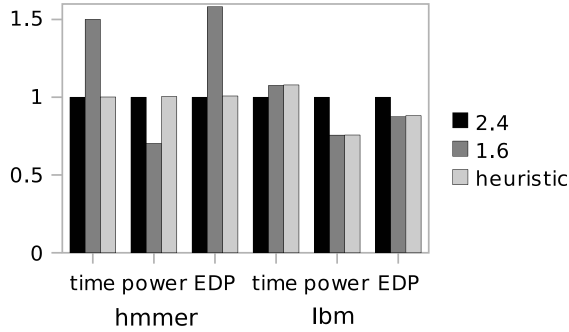

Figure 4: Effects of frequency heuristic

Figure 4 depicts runtime, power, and EDP normalized to the values at 2.4GHz. `hmmer` shows a better EDP at the high frequency, the reason being a substantial increase in runtime at 1.6GHz. `lbm`, on the other hand, shows a better EDP at the low frequency, since its runtime only increases slightly, so the decrease in power dominates the EDP. Our adaptive policy succeeds at selecting the best frequency for each situation, achieving the the same EDP as fixed 2.4GHz for `hmmer` and nearly the same EDP as fixed 1.6GHz for `lbm`.

Last, we evaluated the effects of our frequency heuristic in a dynamic scenario. We ran a workload consisting of one purely compute-bound benchmark (`hmmer`), and seven benchmarks that are memory-bound to varying degrees (`GemsFDTD`, `lbm`, `libquantum`, `mcf`, `milc`, `omnetpp`, `soplex`) and activated sorted scheduling as well as our frequency heuristic.

Here, sorted scheduling in itself leads only to marginal EDP improvements, since with only one compute-bound task, an over-saturation of the memory bus cannot be prevented.

The frequency heuristic, however, is able to reduce EDP by a factor of 0.918 compared to a static setting of 2.4GHz, averaged over all eight benchmarks by lowering the frequency whenever all cores run memory-bound tasks, and by using the high frequency as soon as one of the cores runs the compute-bound task. A fixed setting of 1.6GHz only reduces EDP by a factor of 0.930, since it does not consider the compute-bound task.

6 Conclusion

In this paper, we have analyzed scheduling for avoiding resource contention and for optimal frequency selection. We 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 only lead to savings for situations in which contention cannot be prevented by scheduling, and combining tasks that run best at a certain frequency does not pay off if it leads to resource contention.

We designed a scheduling policy that avoids memory contention by sorting the processors' runqueues by memory intensity, and a frequency heuristic based on memory intensity. Our evaluations show that our policies are able to reduce EDP for scenarios where there is contention that can be reduced by co-scheduling or, if that is not possible, mitigated by frequency scaling.

In future systems, the memory bandwidth will increase, but so will the number of cores, so memory contention can be expected to remain a problem. Our co-scheduling solution scales with larger number of cores, since only pairs of cores need be synchronized.

References

[1] ANTONOPOULOS, C., NIKOLOPOULOS, D., AND PAPATHEODOROU, T. Scheduling algorithms with bus bandwidth considerations for smps. *International Conference on Parallel Processing* (2003).

[2] BELLOSA, F. Process cruise control: Throttling memory access in a soft real-time environement. Tech. Rep. TR-I4-97-2, University of Erlangen, Department of Computer Science, 1997.

[3] CHANDRA, D., GUO, F., KIM, S., AND SOLIHIN, Y. 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).

[4] FEDOROVA, A., SELTZER, M., AND SMITH, M. D. 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).

[5] KOTLA, R., DEVGAN, A., GHIASI, S., KELLER, T., AND RAWSON, F. Characterizing the impact of different memory-intensity levels. In *Proceedings of the Seventh IEEE International Workshop on Workload Characterization (WWC-7)* (2004).

[6] MCCALPIN, J. D. Sustainable memory bandwidth in current high performance computers, 1995.

[7] MCGREGOR, R. L., ANTONOPOULOS, C. D., AND NIKOLOPOULOS, D. S. Scheduling algorithms for effective thread pairing on hybrid multiprocessors. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium* (2005).

[8] MERKEL, A., AND BELLOSA, F. Task activity vectors: A new metric for temperature-aware scheduling. In *Third ACM SIGOPS EuroSys Conference* (2008).

[9] MOSCIBRODA, T., AND MUTLU, O. 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).

[10] SIDDHA, S., PALLIPADI, V., AND MALLICK, A. Process scheduling challenges in the era of multi-core processors. *Intel Technology Journal 11*, 4 (2007).

[11] WEISSEL, A., AND BELLOSA, F. 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* (2002).

[12] ZHANG, X., DWARKADAS, S., FOLKMANIS, G., AND SHEN, K. 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).