Analyzing Resource Interdependencies in Multi-Core Architectures to Improve Scheduling Decisions

Anselm Busse
Technische Universität Berlin
Berlin, Germany
abusse@cs.tu-berlin.de

Jan H. Schönherr
Technische Universität Berlin
Berlin, Germany
schnhrr@cs.tu-berlin.de

Matthias Diener
Federal University of Rio Grande do Sul
Porto Alegre, Brazil
mdiener@inf.ufrgs.br

Gero Mühl
University of Rostock
Rostock, Germany
gero.muehl@uni-rostock.de

Jan Richling
Technische Universität Berlin
Berlin, Germany
richling@cs.tu-berlin.de

ABSTRACT

Since the advent of multi-core processors, different multi-core system and in particular processor architectures have emerged exhibiting individual advantages and disadvantages. One of the main distinguishing factors among these architectures is their varying degree and type of resource sharing among individual cores. On the one hand, resource sharing is necessary for the cores to communicate, while on the other hand resource sharing is often used for economic reasons. Depending on the degree and type of resource sharing, the impact on performance depends on the workload applied and can vary to a large extend.

In this paper, we investigate the impact of different kinds of resource interdependencies found in current processors on the performance of scheduling strategies using a set of benchmarks. Our results show that the architecture has a major impact on the performance of a process placement strategy. However, they also point out that simple strategies taking only a few basic architectural characteristics into account fall short. Thus, new holistic scheduling strategies are needed that take more characteristics into account.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management—*Scheduling, Threads, Multiprocessing/Multiprogramming/Multitasking*

General Terms

Experimentation, Performance

Keywords

Scheduling, process placement, multi-core

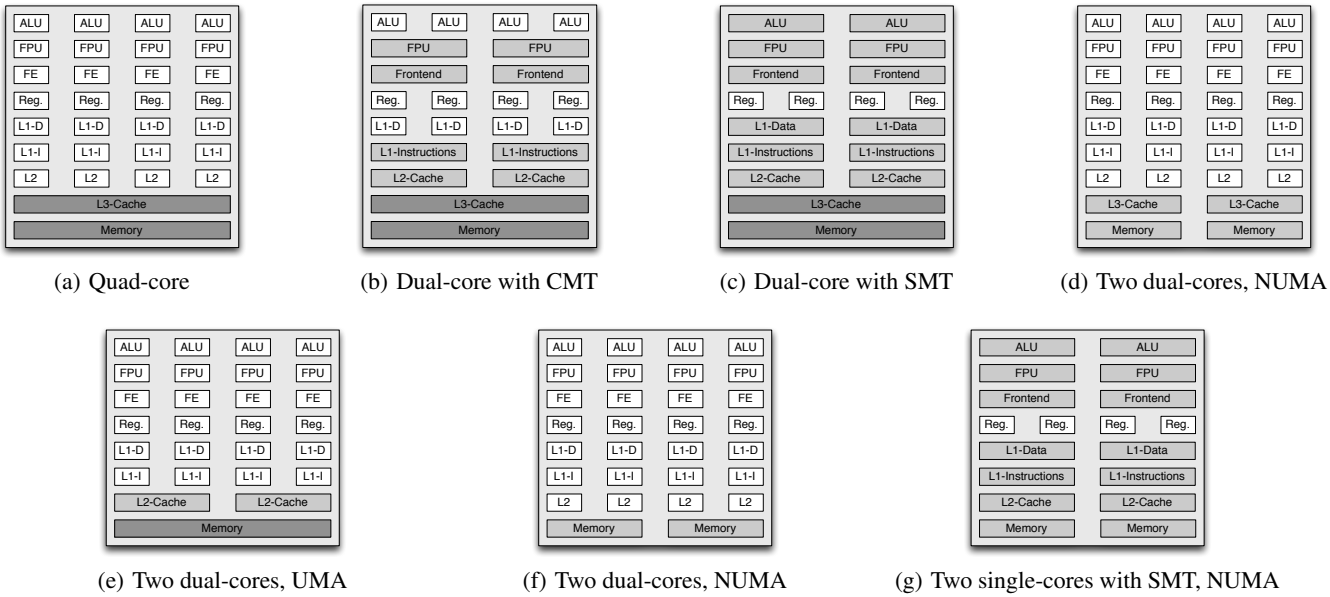
SAC'13 March 18-22, 2013, Coimbra, Portugal.

1. INTRODUCTION

One of the most important questions when deciding about a multi-core processor architecture is which resources should be shared among multiple cores and which resources should be available to each core exclusively. The need for shared resources arises from two facts: On the one hand, cores have to communicate with each other, so they must share at least a communication channel or some memory. On the other hand, there are economic reasons to share resources. For example, a floating point unit requires much more space on a die than an integer unit. Hence, it might be feasible to provide each core with an exclusive integer unit, but to share a floating point unit among several cores. However, such a decision usually has an impact on the performance of the system that depends on the applied workload. A workload with just integer operations is not affected by this design decision at all, whereas a pure floating point workload may suffer from a severe performance degradation. Another example is the memory hierarchy: For processes that have a high communication demand among each other, a fast shared cache might be the favorable solution, while processes that run in parallel, but have nothing in common might profit from an architecture with non-uniform memory access (NUMA) because of the resulting higher memory bandwidth.

Overall, such design decisions are complex and application-dependent. Therefore, the industry has come up with different multi-core system architectures, which led to a heterogeneity among current multi-core systems each with different strengths and weaknesses. Figure 1 shows a sample of existing system architectures that present themselves to the operating system as systems being able to execute four processes simultaneously. These consist either of four mostly independent cores, two pairs of simultaneous multithreading (SMT) siblings, or two pairs of cluster-based multithreading (CMT)[1] siblings. As a simplification, we will refer to a core or sibling that executes one dedicated process or thread as *processing element* (PE). Furthermore, we denote the effect that the performance of one PE is influenced by another PE as *interdependency*. For example, two SMT siblings exhibit such an interdependency because of shared processing resources (indicated by a darker shade in Fig. 1). The processing performance of one sibling

[1]Even though it is only a marketing term, we will use CMT as denotation for AMD's module design [1] to distinguish it from SMT in Intel CPUs.



(a) Quad-core (b) Dual-core with CMT (c) Dual-core with SMT (d) Two dual-cores, NUMA

(e) Two dual-cores, UMA (f) Two dual-cores, NUMA (g) Two single-cores with SMT, NUMA

Figure 1: System architectures with four processing elements and degree of sharing between PEs indicated by the shade of grey.

might decrease when a process is scheduled on the other sibling because of resource contention. However, interdependencies do not always have a bad influence on performance. For example, a shared cache might have a positive impact because of a faster communication between processes or threads.

In this paper, we investigate the effects of the different multi-core system architectures on different kinds of workloads. For brevity, we solely focus on systems with four PEs. While this reduces the number of possible architectures, these architectures are nevertheless a representative cross section, allowing us to transfer our findings to other systems with, e. g., more PEs or a different memory hierarchy. The goal of this investigation is to derive criteria for optimized scheduling decisions laying the foundation for a holistic scheduling approach.

The remainder of this paper is structured as follows: In the following section, we will present our vision of a holistic scheduler and a possible path towards it. Section 3 describes the performed experiments and presents the results. In Section 4, we discuss related work and, finally, we conclude and discuss potential directions for future work in Section 5.

2. VISION

In the context of an operating system, the goal of scheduling is to map the individual threads to the available PEs, while pursuing one or more optimization targets, such as response time, throughput, energy efficiency, fairness, or predictability. However, these goals and the strategies used to reach them often conflict with each other. Such conflicts are usually solved by integrating some kind of weight function.

It is our belief that an effective scheduling has to be holistic. That is, the scheduler must not only consider all properties of the workload and the hardware, but it must also actively control both in order to shape workload and hardware properties to better match each other with respect to the optimization goals. Such a holistic approach, as we envision it, is a complex challenge that has to be broken down into manageable units: Models for both, software and hardware, capture the relevant properties. Offline and online profiling and monitoring parametrize these models. Hardware and software interfaces give a certain degree of control over these parameters. And finally, the holistic scheduler continuously derives good system configurations with respect to the optimization targets from the model parameters and realizes them with the interfaces, realizing a feedback loop. Our related previous work mostly belongs to hardware interfaces: we analyzed the benefits of a direct interface to control the processor frequency [17]. We extended this to frequency-interdependent multi-core systems [18], thereby realizing that available process placement interfaces are too restrictive. Our coscheduling design [15] enriches the process placement interface and adds a needed feature non-intrusively. In an initial work on software interfaces, we allow to change the parallelism degree of applications on the fly [16].

In this paper, we address the hardware model within our holistic approach and analyze the influence of shared resources and other interdependencies between PEs on the optimization goal throughput. This is especially important, as our model should be able to accommodate future architectures, and it can be expected that – with increasing numbers of PEs – the amount of shared resources and the complexity of their interdependencies will further increase.

3. BENCHMARKS AND EVALUATION

To investigate effects of varying interdependencies, we need at least three processing elements: one reference PE and two auxiliary PEs which have different interdependencies with the reference PE. Of the two auxiliary PEs, we call the PE that shares more resources with the reference PE *close* and the other *distant*. (Note, that there is currently no architecture, where shared resources of one auxiliary PE are not a subset of the other; so this distinction can always be made.) We then measure the performance of specific workloads executed on a) the reference and the close PE, and b) the reference and the distant PE. We refer to former type of execution as *close scheduling* and to the latter as *distant scheduling*.

While we could make use of more than these three PEs and put load on more than two PEs at once, doing so would inevitably mix the effects of different interdependencies. As our goal is to isolate

Table 1: Configuration of the Evaluation Systems.

Name	Processor	Frequency	RAM	Sockets	Cores	PEs per Core	Diagram
italy	AMD Opteron 280	2.40 GHz	8 GB	2	2	1	Fig. 1(f)
zambezi	AMD FX-6100	3.30 GHz	8 GB	1	2	2 (CMT)	Fig. 1(b)
diamondville	Intel Atom 330	1.60 GHz	4 GB	1	2	2 (SMT)	Fig. 1(c)
yorkfield	Intel Core 2 Quad Q9400	2.66 GHz	8 GB	1	4	1	Fig. 1(e)
lynnfield	Intel Core i7 860	2.80 GHz	8 GB	1	2	2 (SMT)	Fig. 1(c)
sandybridge	Intel Core i7 2600	3.40 GHz	8 GB	1	2	2 (SMT)	Fig. 1(c)
gainestown	Intel Xeon X5570	2.93 GHz	48 GB	2	2	1	Fig. 1(d)
gainestown-smt	Intel Xeon X5570	2.93 GHz	48 GB	2	1	2 (SMT)	Fig. 1(g)

the individual effects, we explicitly refrain from doing this.

To evaluate performance differences between close and distant scheduling, we introduce the parameter Δ, which we define as:

$$\Delta = \begin{cases} -\left(\frac{t_{close}}{t_{distant}} - 1\right) & \text{if } t_{close} > t_{distant} \\ \left(\frac{t_{distant}}{t_{close}} - 1\right) & \text{if } t_{close} \leq t_{distant} \end{cases}$$

That is, a value of $\Delta > 0$ indicates that close scheduling should be favored for that specific case. A value of $\Delta = 0$ indicates indifference between the two scheduling types. Consequently, for $\Delta < 0$, distant scheduling should be preferred. Furthermore, the absolute value of Δ indicates the performance benefit for the favored scheduling decision. For example, a value of $\Delta = -1$ indicates that the benchmark took only half of the time to finish with distant scheduling than it took with close scheduling.

For our experiments, we gathered a variety of architectures. Their configurations with respect to resource sharing are shown in Table 1 and schematically depicted in Figure 1. The limitation to machines with four PEs is artificial: for our experiments, we need only three, and four PEs is the next best choice considering also available systems and our goal of focusing on interdependency differences (and not PE differences). Our selection covers many diverse ways of resource sharing which is present not only in current but also in past hardware. Investigating behavioral differences between architectures that have a similar high-level architecture, but differ in architectural details, does not only permit an extensive evaluation but also allows conclusions with respect to future architectures. In the following, the machines are referred to by the codename of their processor (with the exception of *gainestown* which we evaluate with and without SMT; to distinguish both configurations, the SMT configuration is called *gainestown-smt*).

With *italy* and *gainestown*, we have two different generations of NUMA systems without shared execution units. Our uniform memory access (UMA) systems *diamondville*, *lynnfield*, and *sandybridge* feature a shared last-level cache and SMT, while *zambezi* represents AMD's CMT approach. Furthermore, *yorkfield* is a front side bus (FSB) based system containing two dual-core packages, each with a large shared cache. Finally, *gainestown-smt* is an example of a NUMA system that also features SMT. This represents the combination of two alternatives considered only in isolation with the other machines. All of these machines present four PEs to the operating system, either inherently or forced via BIOS by disabling some cores per socket or by disabling SMT. Furthermore, any dynamic frequency adjustments are deactivated to increase predictability and to remove another degree of freedom. On the software side, all systems run Gentoo Linux with a Linux 3.3.4 kernel. We statically enforced the desired process placement with the `taskset` command or the `GOMP_CPU_AFFINITY` environment variable in case of an OpenMP program. This way the Linux scheduler had no influence on the running benchmarks.

At first, we analyze the interdependencies by using three different microbenchmarks, which are synchronization bound, memory bound and computation bound, respectively. These microbenchmarks serve as extreme cases in the performance space, as realistic (parallel) applications have elements of all three types. Finally, we run a series of high-level benchmarks to evaluate the insights we gathered through the previous microbenchmarks and check their validity. We will describe these benchmark in detail along with their results in the following subsections.

Each individual benchmark was run 30 times with close and distant scheduling. We then used the average value of the results for our evaluation. Furthermore, we also conducted all benchmarks with two quad-core machines with only symmetric interdependencies between the cores (Fig. 1(a)) to ensure we avoided systematic mistakes. As expected, the results for these machines showed no performance differences for selecting different pairs of cores.

3.1 Synchronization Microbenchmark

The goal of this benchmark is to isolate the issue of synchronization. The benchmark is a parallel program with two threads that is "synchronization bound": A shared variable is initialized with the thread ID of one of the two threads. Then, each thread reads the variable until it differs from the own ID. In this case, the own ID is written to the variable. Therefore, the mechanism of the synchronization is very similar to the semantic of a spinlock.

The results in Figure 2 show that on all systems close scheduling is the better option. The reason for this is the memory hierarchy: Both threads are accessing the same variable. Hence, the faster a modification by one thread can be recognized by the other (be-

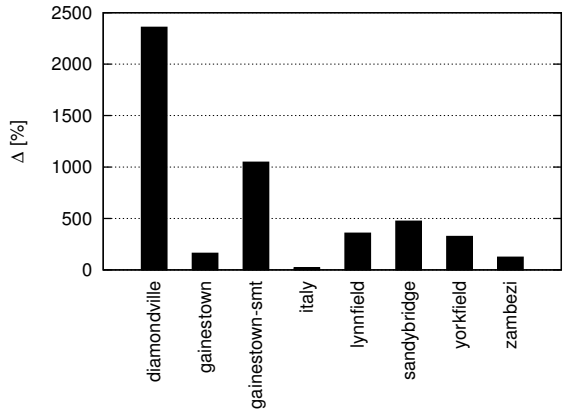


Figure 2: Results of the synchronization microbenchmark.

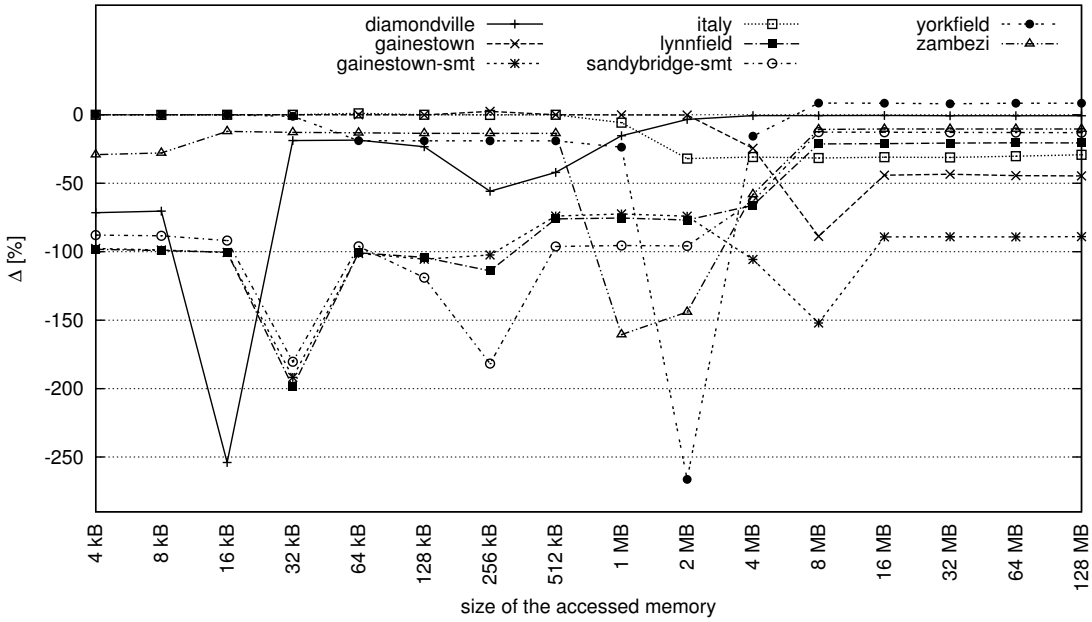


Figure 3: Results of the memory microbenchmark.

cause of a shared level in the memory hierarchy), the earlier the benchmark will be finished. An example for this are the results of the NUMA machines *gainestown-smt*, *gainestown* and *italy*. They show the differences in the memory hierarchy, where the advantage of close scheduling is much more pronounced in *gainestown-smt* with a shared L1-cache than in *gainestown* with a shared L3-cache than in *italy* without any shared cache.

On our SMT machines, the positive effect of a shared cache gets reduced due to the shared execution units. However, this counter-effect is in no case large enough to tip the balance in favor of distant scheduling in this purely synchronizing benchmark. The results of the SMT machines *diamondville*, *lynnfield* and *sandybridge*, although they have the same high level architecture (cf. Figure 1(c)), vary significantly. With respect to our goal of a holistic scheduler this implies that it is not sufficient to consider only the nature of shared resources, but also the actual implementation, i. e., advances between different microarchitectures.

The results of *diamondville* stand out against all others although this processor features a much simpler microarchitecture than the others. We suspect that the extreme superiority of close scheduling is caused by the in-order architecture leading to an additional optimal interlocked execution of both threads.

3.2 Memory Microbenchmark

The goal of this benchmark is to examine memory bound programs which consist of two threads. For this purpose, we used the `bw_mem` benchmark of `lmbench` [11] to measure the memory throughput of two threads running in parallel. We ran the benchmark for differently sized memory regions ranging from 4 kB to 128 MB per thread in order to evaluate the impact of different levels of shared resources. We expect to see that close scheduling will be slower because of the shared memory bandwidth and that it will show an additional drop in performance compared to distant scheduling as soon as it saturates a shared layer in the memory hierarchy and has to utilize the next lower one. At this point, distant scheduling can still use the higher, exclusive level of the memory hierarchy.

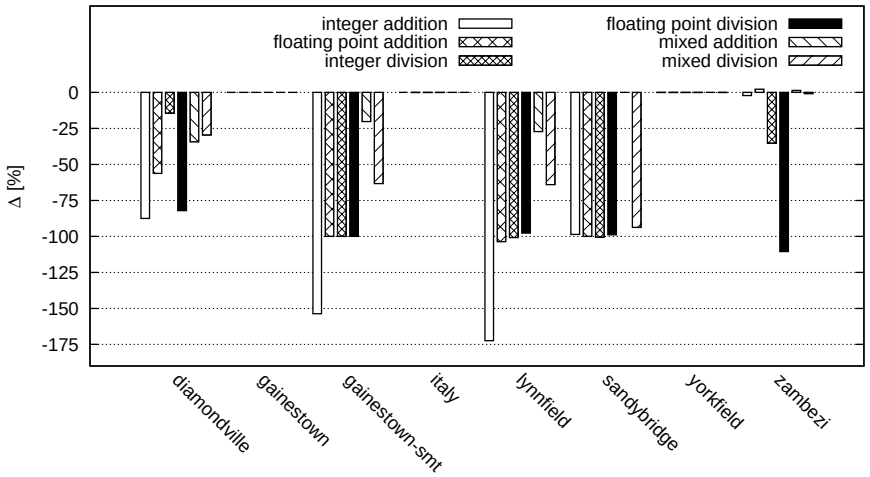
The results presented in Figure 3 show that on all test machines distant scheduling delivers better performance, except on *yorkfield* when using large memory regions. The latter is caused by the necessary arbitration between the two processor packages on the FSB. As expected, the results show a drop in performance for close scheduling compared to distant scheduling as soon as a shared level of memory is saturated (e. g., the L1-cache at 32 kB for *gainestown-smt*, *lynnfield* and *sandybridge*). The competing accesses explain the performance disadvantage of close scheduling.

The results imply for the holistic scheduler that it must have a precise knowledge about the penalty of saturating a level of the memory hierarchy.

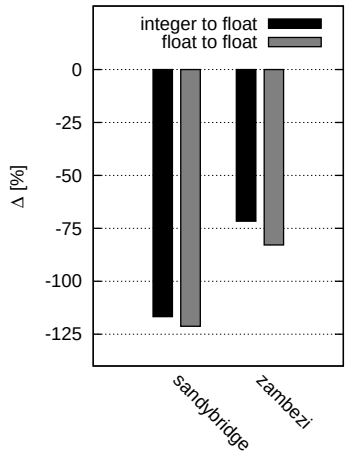
3.3 Computation Microbenchmarks

To investigate how different types of computations affect the performance of processes in the context of interdependencies, we devised two fully CPU bound benchmarks. The first benchmark saturates either the integer units or the floating-point units of the PEs by executing subsequent addition or division operations. The number of cache/memory accesses was minimized in order to avoid processing stalls. For all four combinations of close/distant scheduling and addition/division operations we measured: a pure integer workload, a pure floating-point workload, and a mixed workload consisting of one integer thread and one floating-point thread. The second microbenchmark mimics real software behavior, using code samples from Intel that demonstrate the applicability of AVX instructions in audio software [6]. The code samples consist of two copy functions: one that copies and transforms an array of 32 bit to 64 bit floating-point numbers and one that copies and transforms an array of 24 bit integer numbers to 64 bit floating-point numbers. We measured their behavior on our AVX-capable systems *sandybridge* and *zambezi*.

If already a single thread is capable of saturating all shared execution units of a PE, we expect to see a performance difference of 100% between close and distant scheduling. However, our experiments also include scenarios where a thread utilizes only a fraction of the execution units. This should give us an insight into the degree of performance loss caused by using these shared resources in two threads.



(a) Performance difference of integer and floating-point loads.



(b) Performance difference of AVX loads.

Figure 4: Results of the computation microbenchmarks.

Analyzing the results of the computation benchmarks in Figures 4(a) and 4(b) shows that performance differences are only significant for SMT and CMT systems as these benchmarks do neither synchronization nor many memory accesses, thereby not experiencing the effects demonstrated with the previous benchmarks. For these machines, distant scheduling is the preferred option, but the actual degree of the observed performance increase varies.

For *sandybridge*, distant scheduling is almost twice as fast for every benchmark with the exception of the mixed addition benchmark, where close scheduling is equally fast, i. e., both threads utilize distinct execution units and do not influence each other at all. Note, that on the other SMT systems, mixed addition still causes a performance degradation for close scheduling indicating a different bottleneck within the CPU. The results for *gainestown-smt* and *lynnfield* are almost similar compared to *sandybridge*, except for the integer addition, which slows down by a factor of around 2.5 for close scheduling. That is, even a sequential execution would be more efficient than close scheduling. We tracked this effect down to the *reorder buffer* of the CPUs, which becomes a bottleneck when results of previous calculations are reused too late (cf. [7]). We confirmed this suspicion by varying the number of instructions between the reuse of results.

The performance drop on *diamondville* is not as severe as on the other SMT machines. The explanation for this effect is the in-order execution of *diamondville*, while the other SMT machines have an out-of-order execution. The in-order execution slows down the benchmark in the distant scheduling case because not every execution unit inside the processor is fully saturated. The SMT feature reduces this effect and, thus, yields a better performance as we have initially suspected. The situation is more divergent with *zambezi*. While the mixed workloads and additions show no preference for either close or distant scheduling, we see a drastic performance difference for the division microbenchmarks as well as for the AVX benchmarks. The reason for this is that the floating point unit – which is also responsible for AVX instructions – is shared between two PEs in AMD's unique module design (cf. architecture diagram in Figure 1(b)) giving again distant scheduling a clear advantage.

Similar to the results of Section 3.1, the results of this benchmark show that it is not sufficient to simply consider the high level architecture. Instead, a holistic scheduler has to take machine specific properties into account.

3.4 NAS Parallel Benchmarks

In addition to the previously described microbenchmarks, we used several benchmarks from the OpenMP implementation of the NAS Parallel Benchmarks described by Bailey et al. [2, 4] and developed by Jin et al. [9]. The NAS Parallel Benchmarks measure the performance of parallel systems based on various common scientific calculations and computing problems. The benchmarks are available in increasing problem sizes: *S*, *W*, and *A* through *E*. Because of the very high execution times and memory requirements of the larger problem sizes, we only measured problem sizes up to *B* for FT and MG, and problem sizes up to *C* for the other used benchmarks. Table 2 contains a short overview of the benchmarks we used as well as their respective computational focus. We checked that the combined memory usage does not exceed the available main memory of our evaluation systems.

Table 2: Overview of the selected NAS benchmarks.

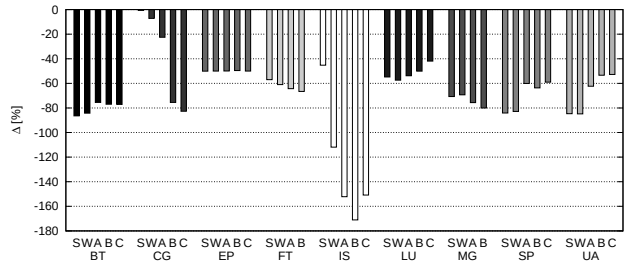
Name	Description	Focus
BT	Block Tridiagonal	Floating point performance
CG	Conjugate Gradient	Irregular communication
EP	Embarrassingly Parallel	Floating point performance
FT	Fast Fourier Transform	Long-distance communication
IS	Integer Sort	Integer performance
LU	Lower-Upper symmetric Gauss-Seidel	Regular communication
MG	Multi Grid	Regular communication
SP	Scalar Pentadiagonal	Floating point performance
UA	Unstructured Adaptive	Irregular communication

Contrary to the previous microbenchmarks, the NAS Parallel benchmarks solve complex problems. As such, they exhibit behavioral fragments of all our previous benchmarks: synchronization, computation, and memory accesses. Thus, we expect results that are a linear combination of the extreme cases determined by our microbenchmarks in Sections 3.1, 3.2, and 3.3.

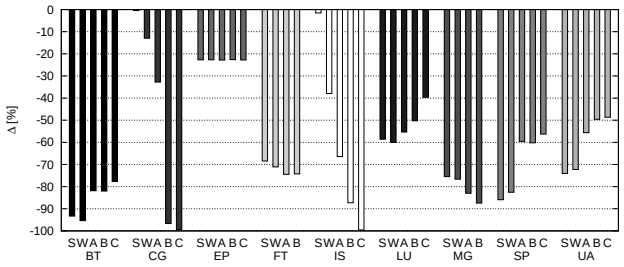
Figure 5 shows the results of the NAS Parallel Benchmarks for all evaluation systems. Please note, that every diagram has its own scaling on the y-axis. The first observation is that the EP benchmark, which consists almost only of floating point operations and no significant communication, only shows performance differences for each machine smaller than the respective differences in the floating point microbenchmark in Section 3.3. This confirms our assumption that the floating point results in Section 3.3 resemble a boundary for the performance differences for floating point based applications.

Considering our systems without shared execution units (*yorkfield*, *gainestown*, and *italy* in Figures 5(e), 5(f), and 5(h), respectively), we see that the positive effect of a faster inter-core communication with close scheduling (cf. Section 3.1) can – in most cases – not outweigh the benefits of distant scheduling: more cache and a higher overall memory bandwidth (cf. Section 3.2). The tendency here is that the larger the problem the better is distant scheduling.

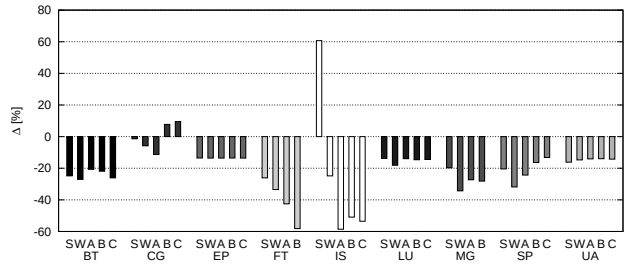
The SMT and CMT systems feature an even faster inter-core communication for close scheduling, which is now additionally countered by shared execution units. Overall, the effect of faster inter-core communication, which we have seen in Section 3.1, is in none of the even communication focused benchmarks prevail-



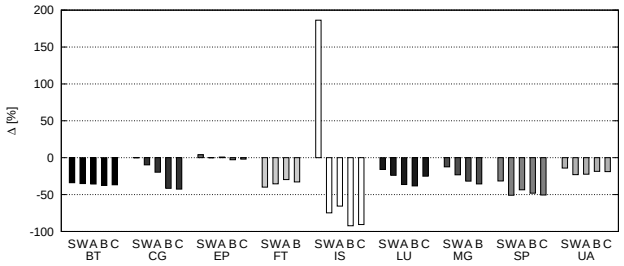
(a) Performance differences for *sandybridge*



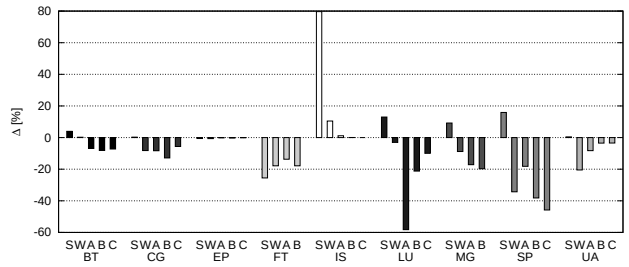
(b) Performance differences for *lynnfield*



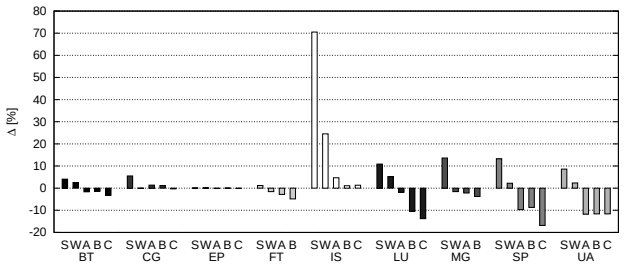
(c) Performance differences for *diamondville*



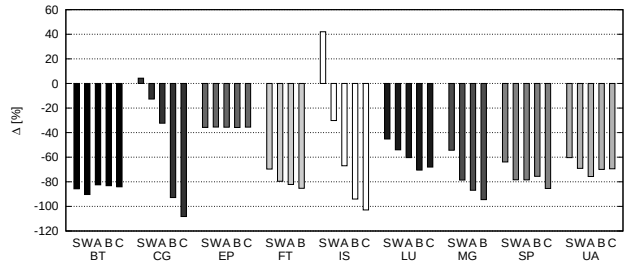
(d) Performance differences for *zambezi*



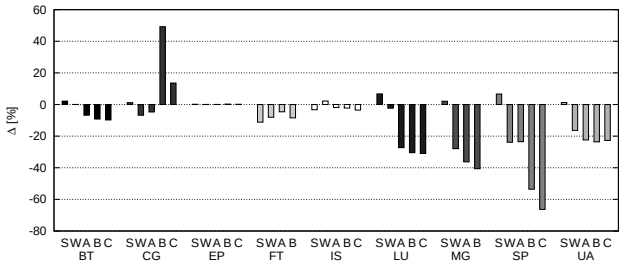
(e) Performance differences for *yorkfield*



(f) Performance differences for *gainestown*



(g) Performance differences for *gainestown-smt*



(h) Performance differences for *italy*

Figure 5: NAS benchmark results for all evaluation systems (note the different scaling of the y-axes).

ing over the loss in processing power, basically rendering close scheduling inferior even for those benchmarks. However, there is one exception: The integer sort benchmark with the smallest data set (IS.S) shows an advantage for close scheduling. A closer investigation using performance counters showed that the task is so small that the copying of program code to a second core takes a significant fraction of the whole processing time.

Furthermore, the results for the integer sort benchmark on *sandybridge* stand out, because they are the only ones where distant scheduling reaches a performance benefit of far more than 100%. These results are especially interesting when compared to the results of *lynnfield*, which has a very similar architecture. We were able to track down the difference to distinctions in the cache performance. As Figure 3 shows, the relative performance drop when saturating the L2-cache at 256 kB is much more severe on *sandybridge* than it is on *lynnfield*.

Overall, this leads to the conclusion that the effects found in Sections 3.2 and 3.3 are dominant in these benchmarks. Only if execution units are not shared and we do not hit one of the relative performance peaks for memory accesses, there is potential for close scheduling due to synchronization effects as demonstrated in Section 3.1.

4. RELATED WORK

In this paper, we studied interdependencies in multi-core systems. Jin and Hood et al. [10, 5] did similar studies and used an experimental setup like ours. However, they pursued the goal of identifying bottlenecks in HPC setups, so they evaluated MPI applications on fewer architectures and with less variations in the size of the data set compared to our work. Varying both aspects, architecture and data set size, gave us valuable insights that are relevant for a holistic scheduler. Still, there are other aspects that are also relevant, but that have been addressed in several publications already: First, the avoidance of contention of resources like execution units, cache or memory access. Second, the question of heterogeneity between PEs, i. e., different frequencies or instruction sets. And third, non-functional requirements.

The first aspect, contention avoidance, is mainly investigated by work on process schedulers: Zhuravlev et al. [21] and Laleel et al. [8] address contention within the cache hierarchy, Bulpin and Pratt [3] within SMT systems. Zhuravlev et al. and Laleel et al. use profiling of the cache behavior of processes for an improved scheduling decision. Zhuravlev et al. show the usability of their approach on a real system while Laleel et al. use a simulation. The scheduler of Bulpin and Pratt keeps records for process pairs that result in a system speedup when executed on SMT siblings, and takes this information into account during the scheduling decision leading to a better system performance. The avoidance of resource contentions is one major aspect that has to be handled by a holistic scheduler. However, as we have shown it is not sufficient for a holistic scheduler to only avoid specific resource contentions. It has to consider trade-offs between different kinds of resource contentions and other system properties that might lead to a better overall performance.

Shelepov et al. [19] and Saez et al. [14] focus on process scheduling on processors with heterogeneous PEs. They propose *architectural signatures* to classify particular processing elements. These signatures are then used to create a schedule that is better suited for specific processes. Srinivasan et al. [20] also deal with process scheduling in a heterogeneous system. They propose a hardware-based profiling mechanism to enable the operating system to choose the best PE for every task. However, they restricted their research to a CPU with a simple structure that, for example, only allows for a simple memory hierarchy. This model might, thus, not be rich enough to model current systems with sufficient accuracy, because subtle differences in architecture can already be significant as shown in Section 3. However, hardware-assisted profiling might be a required step towards a holistic scheduler, as it would considerably reduce the time needed to find an optimal scheduling solution. Those works give important insights on how to acquire process properties, which are an important foundation of our holistic scheduler.

Energy consumption and energy efficiency are examples for non-functional properties of the scheduling problem that can be considered in addition to performance. Merkel and Bellosa [12, 13] propose *task activity vectors* to capture the behavior of tasks for multiple dimensions. They use the gathered information not only to reduce contention by migrating and reordering tasks, but also to guide the CPUs' frequency selection reducing energy consumption without significant effects on performance.

5. CONCLUSIONS AND FUTURE WORK

The results of our evaluation in Section 3 demonstrate that many factors of a multi-core architecture can influence the performance of the processes executed. It became evident that taking just a few high-level characteristics of the architecture into account is not enough and that scheduling strategies that do so will not be able to exploit the full potential of modern multi-core architectures. Instead, just applying the optimizations valid for one generation onto the architectural similar next generation of hardware might actually invert the intended effect. Moreover, we confirmed that not only characteristics of the hardware architecture, but also those of the software that is being executed must be considered to gain optimal results. While the characteristics of the hardware can be determined before execution, this is not entirely possible for the software, e. g., because the behavior of software also depends on the actual input applied at runtime. Therefore, we suggest a holistic scheduling approach that takes more characteristics of the hardware into account than previous approaches and that also uses software characteristics gathered before the software is executed and while the software is executed. Giving the scheduler the additional ability to shape hardware and software opens even more optimization possibilities.

This paper clearly shows that the development of a holistic scheduler is a great challenge, especially when we consider that our examined loads in the experiments were very static and not very complex. A real world scenario will show much more dynamic process changes. The task of creating a scheduler will be even more complex when we take other requirements for an operating system scheduler into account. The most severe one is the fact that the scheduling decision should be reached with as little effort as possible, ideally in $\mathcal{O}(1)$.

Following this paper, we have to accomplish several tasks to create a holistic scheduler as already mentioned in Section 2. The first step will be to establish a model, which cannot only describe current computer architectures in a sufficient manner but is also able to describe upcoming yet unknown ones. We are planning to achieve this by a model that is as generic as possible. After that we will need facilities to profile applications with respect to their specific properties. Once again we will need to create a generic model for applications to do that. Furthermore, we will have to revisit the different possibilities to profile an application. As the related work presented in Section 4 shows, there is currently no prevailing method to profile applications. Finally, we will be able to deduct concrete scheduling algorithms that can be implemented and evaluated in a current operating system.

6. REFERENCES

[1] Advanced Micro Devices. *Software Optimization Guide for AMD Family 15h Processors*, Jan. 2012.

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

[3] J. R. Bulpin and I. A. Pratt. Hyper-threading aware process scheduling heuristics. In *Proceedings of the USENIX 2005 Annual Technical Conference (ATEC '05)*, pages 399–402, Berkeley, CA, USA, 2005. USENIX Association.

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

[5] R. Hood, H. Jin, P. Mehrotra, J. Chang, J. Djomehri, S. Gavali, D. Jespersen, K. Taylor, and R. Biswas. Performance impact of resource contention in multicore systems. In *Proceedings of the 24th IEEE International Parallel & Distributed Processing Symposium (IPDPS '10)*, Piscataway, NJ, USA, Apr. 2010. IEEE.

[6] Intel Corporation. Utilizing intel AVX with cakewalk SONAR* X1. White Paper, 2011.

[7] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Apr. 2012.

[8] A. Jaleel, H. H. Najaf-abadi, S. Subramaniam, S. C. Steely, and J. Emer. CRUISE: cache replacement and utility-aware scheduling. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '12)*, pages 249–260, New York, NY, USA, 2012. ACM.

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

[10] H. Jin, R. Hood, J. Chang, J. Djomehri, D. Jespersen, K. Taylor, R. Biswas, and P. Mehrotra. Characterizing application performance sensitivity to resource contention in multicore architectures. Technical Report NAS-09-002, NASA Ames Research Center, Moffett Field, CA, USA, Nov. 2009.

[11] L. McVoy and C. Staelin. lmbench: portable tools for performance analysis. In *Proceedings of the USENIX 1996 Annual Technical Conference (ATEC '96)*, pages 279–294, Berkeley, CA, USA, 1996. USENIX Association.

[12] 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)*, Berkeley, CA, USA, Dec. 2008. USENIX Association.

[13] A. Merkel and F. Bellosa. Task activity vectors: a new metric for temperature-aware scheduling. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European conference on Computer systems (EuroSys '08)*, pages 1–12, New York, NY, USA, Apr. 2008. ACM Press.

[14] J. C. Saez, D. Shelepov, A. Fedorova, and M. Prieto. Leveraging workload diversity through os scheduling to maximize performance on single-isa heterogeneous multicore systems. *Journal of Parallel and Distributed Computing*, 71(1):114–131, Jan. 2011.

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

[16] J. H. Schönherr, J. Richling, and H.-U. Heiss. Dynamic teams in OpenMP. In *Proceedings of the 22nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD '10)*, pages 231–237, Los Alamitos, CA, USA, Oct. 2010. IEEE Computer Society.

[17] J. H. Schönherr, J. Richling, M. Werner, and G. Mühl. Event-driven processor power management. In *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking*, pages 61–70, New York, NY, USA, Apr. 2010. ACM Press.

[18] J. H. Schönherr, J. Richling, M. Werner, and G. Mühl. A scheduling approach for efficient utilization of hardware-driven frequency scaling. In *Workshop Proceedings of the 23rd International Conference on Architecture of Computing Systems (ARCS '10)*, pages 367–376, Berlin, Germany, Feb. 2010. VDE Verlag.

[19] 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):66–75, Apr. 2009.

[20] S. Srinivasan, L. Zhao, R. Illikkal, and R. Iyer. Efficient interaction between OS and architecture in heterogeneous platforms. *SIGOPS Operating Systems Review*, 45(1):62–72, Feb. 2011.

[21] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the 15th International Conference on Architectural support for programming languages and operating systems (ASPLOS '10)*, pages 129–142, New York, NY, USA, Mar. 2010. ACM Press.