`autopin` – Automated Optimization of Thread-to-Core Pinning on Multicore Systems

Tobias Klug, Michael Ott, Josef Weidendorfer, and Carsten Trinitis

Technische Universität München
Lehrstuhl für Rechnertechnik und Rechnerorganisation / Parallelrechnerarchitektur (LRR/TUM)
Boltzmannstraße 3, 85748 Garching bei München
{klug,ottmi,weidendo,trinitic}@in.tum.de

Abstract. In this paper we present a framework for automatic detection and application of the best binding between threads of a running parallel application and processor cores in a shared memory system, by making use of hardware performance counters. This is especially important within the scope of multicore architectures with shared cache levels. We demonstrate that many applications from the SPEC OMP benchmark show quite sensitive runtime behavior depending on the thread/core binding used. In our tests, the proposed framework is able to find the best binding in nearly all cases. The proposed framework is intended to supplement job scheduling systems for better automatic exploitation of systems with multicore processors, as well as making programmers aware of this issue by providing measurement logs.

Keywords: Multicore, CMP, automatic performance optimization, hardware performance counters, CPU binding, thread placement.

1 Introduction

During recent years, a clear paradigm shift from increasing clock rates towards multicore chip-architectures (CMP) has taken place. Considering chip manufacturers' long-term objective of integrating 128 and more cores onto one die, there are several open issues with respect to programmability and scalability that have to be examined. In the past, a serial program could benefit from a new processor model simply because processors' clock frequencies were increased from current models to successors. Consequently, even standard applications ran faster without any need to modify a single line of source code. With energy efficiency as a new optimization goal, clock frequencies have to stay more or less stable, and additional performance gains are only obtainable by parallelism on the core level. In order to take advantage of existing and future multicore processor architectures, it is essential to develop parallel applications and to adapt existing serial applications accordingly. Otherwise, all but one core remain idle, and no performance gain can be achieved at all. Parallel programming is leaving the high performance computing (HPC) niche and establishing itself as a mainstream programming technique.

P. Stenström (Ed.): Transactions on HiPEAC III, LNCS 6590, pp. 219–235, 2011.
© Springer-Verlag Berlin Heidelberg 2011

Asymmetric particularities of the memory subsystem are a big obstacle for runtime performance on shared memory machines, as they need to be taken care of explicitly. Non Uniform Memory Access (NUMA) architectures are a familiar example. A new type of asymmetric property comes with shared caches in multicore processors: The access history of one or multiple nearside cores can significantly influence the speed of memory accesses. While overlapping working sets in threads running on cores sharing a cache can reduce runtime, the non-existence of any overlapping usually degrades performance by cutting available cache space into half. Without sophisticated tools and detailed analysis, the programmer can only roughly assess the reason for acceleration or slowdown in the parallel code, let alone come up with optimization strategies for badly running code. This problem is expected to increase with the number of cores available on one chip[1], as in this case the need for complex on-chip interconnect and cache buffer hierarchies is evident.

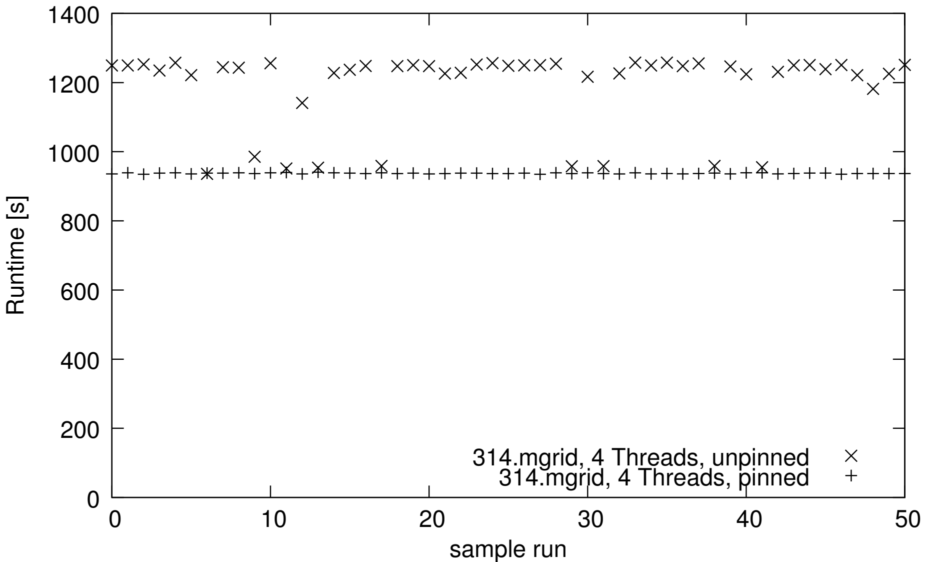

Fig. 1. Comparison of unpinned runs vs. runs under control of autopin. Four threads on the Caneland platform.

To overcome the issue with non-uniform memory subsystems, including the shared cache problem, we propose an automatic approach in this paper: While the application is running, the `autopin` tool checks a given set of fixed thread-to-core bindings (called *pinnings*) in order to find the pinning with optimal performance. In this study, we used `autopin` to find optimal pinnings for applications

[1] In the remainder of this work, the term chip will refer to a single physical processor chip which may consist of multiple processor cores plus cache. Hence, the term core will refer to a single x86 based physical processor unit.

in the SPEC OMP benchmark[2] on various multicore systems. We check pinnings where all cores are active as well as pinnings with a smaller number of threads than cores available on a given system. This is due to the fact that one core on a multicore processor can already fully exploit the available connection to main memory, thus slowing down any work on other cores on the same chip. In this case, it might be recommended to not use these cores for the parallel application. In addition, there exist applications that run with thread counts which do not match available core counts: Examples are parallel tree traversals or load balancing schemes generating/killing threads on the fly. The proposed framework is intended to supplement job scheduling systems for better automatic exploitation of systems with multicore processors, as well as making programmers aware of the given issue by providing measurement logs.

To illustrate this with an example, figure 1 shows runtimes for the SPEC OMP's 314.mgrid benchmark with four threads for 50 sample runs. The 314.mgrid benchmark will be explained in further detail in section 4. Without any control of the pinning, the operating system's scheduler decides on which cores the threads will run. As can be seen in figure 1, runtimes can vary significantly and are hard to predict. However, with the use of the `autopin` tool, optimal thread pinning ensures optimal performance as well as equally distributed runtimes over all sample runs.

2 Related Work

With the large amount of different computer systems available today, regarding available resources, from internode connection and memory subsystem parameters (e.g. cache sizes) to CPU features like superscalarity and vector units, it has always been difficult to come up with an algorithm implementation that optimally exploits these resources. A common approach is to use performance analysis tools such as GProf [1] or Intel VTune [2], and to adapt the code to a specific system. However, this approach is not always feasible: When software is run by a user on a site other than the development site, there usually will be an executable binary for a class of systems (as e.g. for commercial software). Often, the user can not even check if the application runs at optimal performance on his (expensive) system. To still allow for good exploitation, different approaches exist: Foremost, the best code optimization approaches are architecture independent, e.g. using algorithms with lower complexity. For caches, *cache oblivious algorithms* [3] use recursive splitting of data structures for blocking optimization, independent on cache size. Another approach is to check for hardware features at runtime (as in math libraries from vendors [4]) or at install time with an automated search for best parameters and according recompilation. This also includes a feedback compilation step as supported by most compilers (e.g. Intel Compiler Suite, PGI Compilers, GNU Compiler Collection), which can even adapt to a user's typical input data. A well known example for this strategy,

[2] `http://www.spec.org/OMP`

searching for best parameters for cache optimization, is the Atlas library [5]. Our automated search for best thread-to-core pinning takes a similar approach.

Documented hardware performance counters are built into every processor on the market today. Similar counters have always been present in processors to allow for internal correctness checks after production. The good news is that since quite some time these performance counters have been documented and can be used by various tools. Unfortunately, the amount of different events that can be measured varies from processor type to processor type (for example, see the manuals for Intel [6] or AMD [7] processors). This means that there is no standard (yet) that determines which performance counters have to be present in a processor. Recently, Intel has added a simple counter interface and a few specific events to its x86 architecture (for processors based on the Core microarchitecture). Typically, there are 2 or 4 counters available for a huge number of event types related to the processor pipeline, the cache subsystem, and the bus interface, thus allowing to check the utilization of resources. However, the semantics of events can be difficult to interpret, and often, detailed documentation is rare. Hardware Performance Counters are either used to read exact counts, or to derive statistical measurements. The most common commercial tool is VTune [2]. A library for multiple platforms and operating systems to read counters is PAPI [8]. For Linux, there is a statistical measurement tool called OProfile [9], available as part of the standard kernel. However, to get read access to counters, it is required to install a kernel patch (Perfctr), complicating the use significantly. Additionally, HP has started to work on another kernel patch called `perfmon2` [10]. This patch initially existed for the Linux Itanium architecture only but now also provides support for latest Intel and AMD processors. Its user level parts (`libpfm`, `pfmon`) form the basis for `autopin`.

So far, the de facto standard shared memory API OpenMP [11] was mostly used on large shared memory architectures. With the new memory and cache hierarchies being introduced by multicore architectures, thread pinning becomes increasingly important for OpenMP programs with regard to scalability issues. [12] and [13] discuss operating system and compiler dependent calls to control pinning as well as page allocation on ccNUMA, CMP (chip multiprocessing), and CMT/SMT (chip multithreading/simultaneous multithreading) architectures running Linux or SOLARIS. Carrying out several OpenMP benchmarks, the authors conclude that affinity is especially important for OpenMP performance on ccNUMA machines, with OpenMP nesting still being difficult on those architectures. From the operating systems' point of view, SOLARIS and SunStudio provide better tools to deal with the problem, however, Linux is catching up. The authors also observe performance benefits through shared caches in multicore architectures.

In [14], the author argues that using multicore platforms effectively will be a key challenge for programmers in the future. The article discusses the challenges posed by multicore technology, reviews recent work on programming languages potentially interesting for multicore platforms, and gives an overview on on-going activities to extend compiler technology with regard to multicore programming, which also affects thread pinning.

3 The autopin Tool

As a proof-of-concept implementation of our framework for automated CPU pinning, we extended the `pfmon` utility from the perfmon2 package (see [10]) with the required functionality.

The cores to be used and the order in which the cores are assigned to the threads is specified by the user via an environment variable called `SCHEDULE`. Each position of this string-variable defines a mapping of a thread ID to a CPU core ID. For example, `SCHEDULE=2367` would result in the first thread being pinned to core #2, the second thread to core #3, and so on. The user may pass several, comma-separated sets of scheduling mappings via this environment variable.

Upon creation, each new thread is enumerated and pinned to one specific CPU core using the `sched_setaffinity()` system call according to the first entry of the `SCHEDULE` variable. Pinning of the additional management thread created by Intel's OpenMP implementation is omitted. Hence, this thread is scheduled by the operating system.

If the user provided more than one scheduling mapping, the tool will probe each of these mappings for a certain time interval t. Probing is performed using the following algorithm:

1. Let the program initialize for i seconds.
2. Read the current timestamp ts_1 and value pc_1 of the performance counter for each thread.
3. Run the program for t seconds.
4. Read the current timestamp ts_2 and value pc_2 of the performance counter for each thread.
5. Calculate the performance rate $r_j = (pc_2 - pc_1)/(ts_2 - ts_1)$ for each thread j and the average performance rate r_{avg} over all threads.
6. If further mappings are left for probing, re-pin the threads according to the next pinning in the list, let the program "warm up" for w seconds, and return to 2.

The initialization step in 1. is required to avoid measuring potential sequential phases in the initial stage of the program [15].

The "warm up" time after each re-pinning is needed for the actual rescheduling of the threads and to refill the cache.

All parameters t, i, and w can be specified in the command line. Otherwise, the following default values (obtained by previous experiments) will be used: $t = 30$, $w = t/2$, and $i = w$.

The specific average performance rate r_{avg} of each scheduling mapping is written to the console. After all mappings have been probed, `autopin` displays the mapping which achieved the highest performance rate and re-pins the threads accordingly. The program then continues execution with this optimal pinning which will not be changed until the program terminates. Additionally, every t seconds the current performance rate is calculated and written to the console.

As non-optimal pinnings might be used in the beginning, a slight overhead is imposed during this phase. However, in most cases this overhead can be neglected, especially if t and w are small compared to total application runtime, see section 5 for detailed analysis.

The performance counter event which is used for the calculation of the performance rate can be specified by the user with the `-e` parameter. A list of events which are supported by `libpfm` for the used architecture can be retrieved by calling `autopin -L`.

As outlined in [16], thread migration on NUMA systems poses additional challenges: As accesses to non-local memory can decrease performance, not only the thread itself has to be migrated, but also its referenced memory pages. On operating systems which support next touch memory policy, pages are migrated automatically after thread repinning. The current stable Linux kernel only allows for manual page migration. However, there is a patch for the development branch which provides the required functionality [17]. `autopin` triggers automatic page migration as provided by this kernel patch.

4 Experimental Setup

This chapter describes the experimental setup that has been chosen in order to assess the performance of the `autopin` framework. First, the deployed benchmark suite SPEC OMP is described. After this, the hardware platforms that were used to perform the benchmark applications under control of `autopin` are specified. The last section deals with different CPU pinnings that were selected to be evaluated by `autopin` during the benchmark run.

4.1 Benchmark

SPEC OMP was used as a benchmark basis for `autopin`. It is an OpenMP benchmark suite for measuring performance of shared memory parallel systems consisting of eleven applications (see table 1), most of which are taken from the scientific area [18].

There are two different levels of workload for SPEC OMP: Medium and Large. All benchmark runs were executed with medium size, as the maximum number of cores used was 16, whereas runs with workload size large are intended to be used for large scale systems of 128 and more cores. In SPEC OMP all benchmark applications are provided in form of source code and have to be compiled with an appropriate compiler. For all hardware platforms described below, Intel Compiler Suite 9.1 was utilized.

4.2 Hardware Environment

Our testbed consists of several machines:

– One node with two Intel Clovertown processors. The Clovertown processor consists of four cores, while two cores have a shared Level 2 cache (4 MB),

Table 1. SPEC OMP benchmark applications

application name	description
310.wupwise	quantum chromodynamics
312.swim	shallow water modeling
314.mgrid	multi-grid solver in 3D potential field
316.applu	parabolic/elliptic partial differential equations
318.galgel	fluid dynamics analysis of oscillatory instability
320.equake	finite element simulation of earthquake modeling
324.apsi	weather prediction
326.gafort	genetic algorithm code
328.fma3d	finite-element crash simulation
330.art	neural network simulation of adaptive resonance theory
332.ammp	computational chemistry

respectively. Our system has 16 MB of cache in total, runs at a clock rate of 2.66 GHz and has 8 GB RAM, DDR2 667 MHz. The frontside bus has a clock rate of 1333 MHz.

Figure 2 demonstrates a schematical diagram of this machine, which will be referred to as Clovertown. The core numbers in the figure are corresponding to the logical processor id assigned by the Linux kernel. The drawing also illustrates which cores are sharing a cache (for instance core #0 and core #2). Whether two cores share a cache or not can be detected with the authors' false sharing benchmark [19].

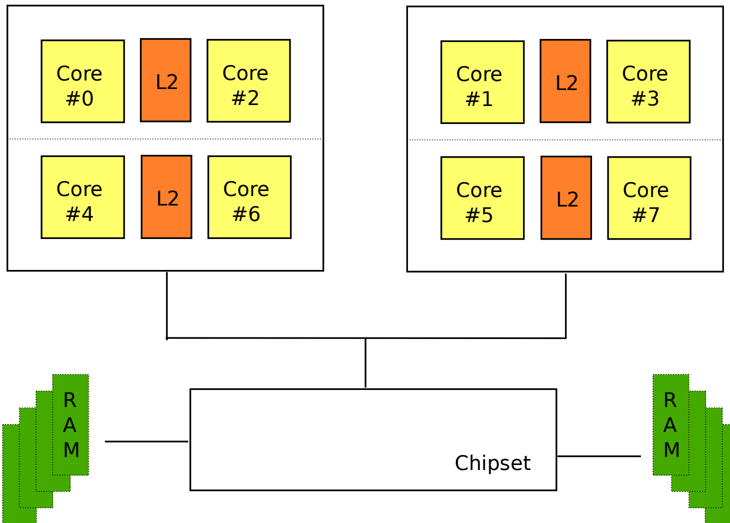

Fig. 2. Intel Clovertown System

– A system with four Intel Tigerton processors. The Tigerton processor consists of four cores, while two cores have a shared Level 2 cache (4 MB), respectively. There are four independent frontside buses (1066 MHz), so each CPU has a dedicated FSB. Each FSB is connected to the Chipset (Clarksboro) which has a 64 MB snoop filter. The memory controller can manage four fully buffered DIMM channels (see figure 3). Our system has 32 MB of cache in total, runs at a clock rate of 2.93 GHz and has 16 GB RAM (DDR2 667 MHz). This machine will be referred to as Caneland.

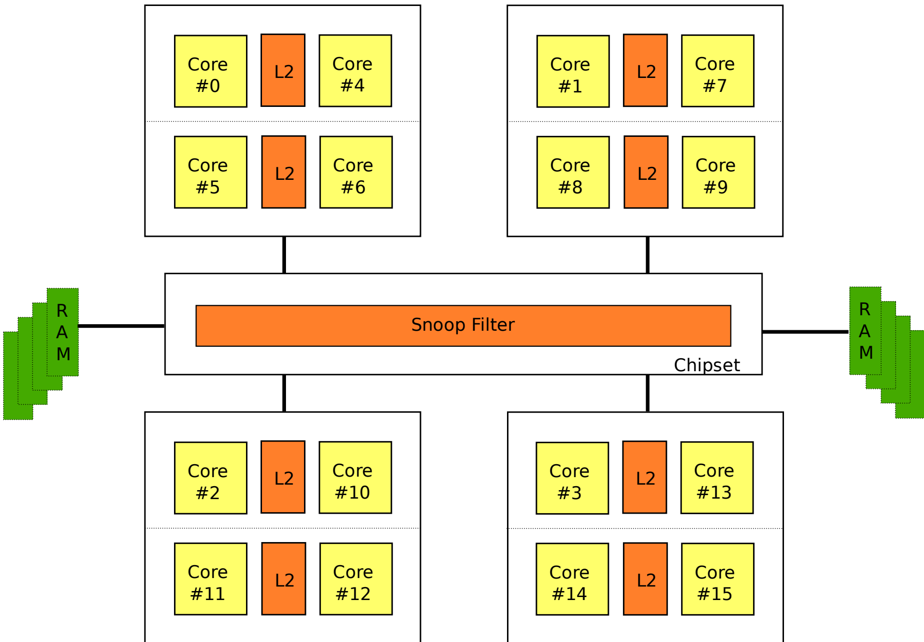

Fig. 3. Intel Caneland Platform

– A two socket machine, equipped with two AMD Opteron 2352. Each CPU has four cores, each of which has a L2 cache size of 512 KB. All cores on a chip are sharing a 2 MB L3 Cache. The four cores are running at a clock rate of 2.1 GHz. The system has 16 GB main memory, DDR2 667 MHz. In contrast to the two hardware platforms described above, this system represents a NUMA-Architecture. Each CPU has an integrated memory controller and can access local memory faster than remote memory. Access to remote memory takes place via HyperTransport (see figure 4). This machine will be referred to as Barcelona in the following sections.

4.3 Thread-to-Core Pinning

All benchmark applications were started with `autopin` monitoring the hardware counters `INSTRUCTIONS_RETIRED` on Intel architecture and accordingly

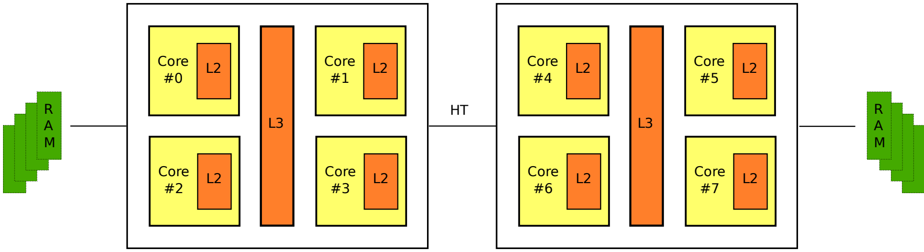

Fig. 4. AMD Barcelona System

RETIRED_INSTRUCTIONS on AMD architecture. As the deltas of the performance counters are divided by the measurement time interval, the measured metric represents the MIPS rate. For floating point intensive programs it might also be interesting to count the retired floating point instructions and calculate the FLOPS rate.

Table 2. Investigated CPU Pinnings for Different CPU Architectures. The first column shows the number of threads used, the second to fourth columns stand for the different thread-to-core pinnings.

#Threads	Caneland	Clovertown	Barcelona
1	1	4	1
2	1,2 1,7 1,8	2,6 4,5 4,6	4,5 2,6 4,6
4	1,7,8,9 1,8,2,11 5,8,11,14 8,9,11,12	4,5,6,7 2,3,6,7 1,3,5,7	2,6,3,7 4,6,5,7 1,3,5,7
8	1,7,8,9,2,10,11,12 4,6,7,9,10,12,13,15 5,6,8,9,11,12,14,15	0,1,2,3,4,5,6,7	0,1,2,3,4,5,6,7

We did not probe all possible pinnings, as most of them are redundant due to symmetries of the architectures:

– For the 1-thread runs we chose a core which is located on a different chip than core #0 as this one often is used for operating systems tasks and thus could disturb the benchmark.
– For runs with 2 threads we chose configurations on two different chips, on one chip with the 2 cores sharing the L2 cache (Intel only), and on one chip with both cores not sharing the cache.
– The measurements with 4 threads were carried out on 1 chip with all cores utilized, on 2 chips once with 2 cores not sharing the L2 cache and – where

applicable – once with 2 cores sharing the L2 cache. On the Caneland platform we additionally made a run on 4 chips, using one core per chip.
- On Clovertown and Barcelona 8 threads were pinned to the core IDs in the same order as they were forked (e.g. the 1st thread on core #0, the 2nd on core #1, and so on). On the Caneland platform we probed configurations exploiting all 4 cores on 2 chips, and 4 chips utilizing 2 cores each – once with shared cache once without.
- The 16-core runs on Caneland were conducted analogously to the 8-core runs on the Clovertown platform.

The detailed list of probed CPU pinnings can be found in table 2. In order to find the best and worst pinning, we made additional runs with `autopin` being called with one `SCHEDULE`-parameter only, so the CPU pinning stayed unchanged from start to finish. Such runs were performed for every pinning listed in table 2. So for example, on the Caneland platform for two threads there are the following CPU pinnings to investigate: (1,2), (1,7) and (1,8). Accordingly `autopin` was called with `SCHEDULE=12,17,18`. Additionally, autopin was called three times one after another with parameter `SCHEDULE=12` for the first run, `SCHEDULE=17` for the second run and `SCHEDULE=18` for the last run. This way it is possible to double-check if `autopin` really found the perfect CPU pinning.

5 Results

5.1 Verification of the `autopin` Approach

As described in chapter 4, we used the SPEC OMP benchmark suite to evaluate the effectiveness of our approach. As this suite consists of 11 individual benchmark applications, presenting the runtimes for all benchmarks, architectures, and configurations (# of cores used, pinning to cores) would go beyond the scope of this paper. Therefore, we only discuss three of the benchmark applications in detail: 314.mgrid, 316.applu, and 332.ammp. For the remaining benchmarks we will only sum up our observations shortly.

In comparison with the measurements presented in [16], the slightly modified algorithm (extended by the initialization phase), in combination with page migration on NUMA architectures as described in section 3, is able to find optimal pinnings for almost all benchmarks on all three platforms: In nearly all cases a pinning with a total runtime not exceeding the perfect pinning's runtime by one per cent was found. Only on the Caneland platform, in two cases (312.swim and 332.ammp) a pinning with a total runtime of less than 5% above the perfect pinning's runtime was found. The experiments have been carried out using `autopin`'s default parameters ($t = 30$, $w = t/2$, and $i = w$).

On all platforms, different CPU pinnings had only little effect on the total runtime of the benchmarks if only one core or all available cores were utilized. Note that this does not mean that one can neglect CPU pinning in these cases. Pinning is still important to prevent threads from moving from one core to another.

On the Clovertown platform, CPU pinning is most important for configurations with 2 cores. For 8 benchmarks, the difference in total runtime between the optimal and the worst configurations was over 20% (over 50% for 314 and 316). For the remaining three (324, 328, 332) it is in the range of 3-10%. For configurations with 4 utilized cores, pinning improved the total runtime between 1 and 7% and in one case (314) by 17%.

The Caneland platform is very sensitive to CPU pinning. Pinnings on two cores showed runtime differences in the range of 25-65% for 8 benchmarks out of 11. 324, 328, and 332 were in the range of 4-15%. The gap between the optimal and the worst pinning even increases for setups with 4 cores: only for 3 benchmarks (324, 328, 332) the difference was below 50%, 312 and 314 even showed differences over 100%. For 8 cores the runtime differences were widely distributed between 7 and 78%. Furthermore, for all benchmarks besides the usual suspects 324, 328, and 332, the best 4-core pinning showed better runtimes than the worst 8-core pinning. Utilizing all 16 cores improves runtime only slightly for most benchmarks. In fact, for 314 and 320, the optimal 4-core pinnings achieve better runtimes.

In general, the Barcelona platform seems to be more tolerant on wrong CPU pinnings. At least on 2-thread runs: runtime differences for the best and worst pinning were between 0.1 and 6.5%, except for 312 where the gap was 32%. On 4-thread configurations the pinning has a higher impact, though not as high as on the Caneland platform: for most benchmarks the runtimes differed between 1.5 and 28%, with 312 making an exception again by showing a gap of 58%.

For all data sets, the 2-core configurations which pinned the threads to cores on different chips showed the best runtimes. With 4 threads, it is best to pin them on cores which don't share a common cache on Intel Platforms. This is simply due to the fact, that with two cores sharing a common L2 cache, one core can utilize the whole 4MB L2 cache for one thread if the other one is idle. The same is true for 8-core configurations on Caneland. On the Barcelona it is best to distribute the threads equally to both chips. Being a NUMA architecture, this gives the highest aggregated memory bandwidth to all threads. Furthermore, as the L3 cache is shared between all cores on one chip, the available cache per thread is higher, if half of the cores are idling.

Figures 5, 7 and 9 show the total runtimes (in seconds) of the 314.mgrid benchmark on the Clovertown, Caneland and Barcelona platform utilizing 1, 2, 4, 8, and 16 (Caneland only) cores. For 1 core and 8 cores (16 on Caneland) we only show the runtime for one CPU pinning as different pinnings had only little effect on the total runtime in these cases. For the other core counts we show runtimes of the worst ("max") and the best ("optimal") pinning, as well as for the configuration `autopin` has proposed ("autopin") - which in all cases is identical to the optimal pinning. Note that on the Intel systems, utilizing more than 4 cores does not improve runtimes any further - even with perfect pinning. If the wrong pinning is chosen, the runtime can be worse than the runtime with perfect pinning on half the number of cores. This effect significantly influences performance on the Caneland platform: The worst 2- and 4-core setups are less

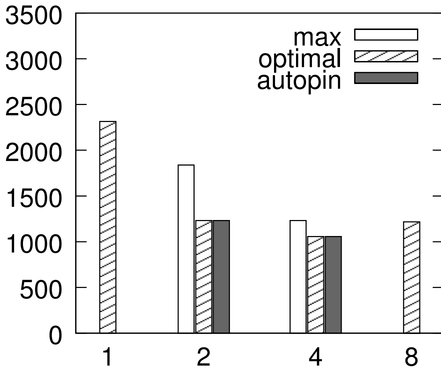

Fig. 5. 314.mgrid on Clovertown

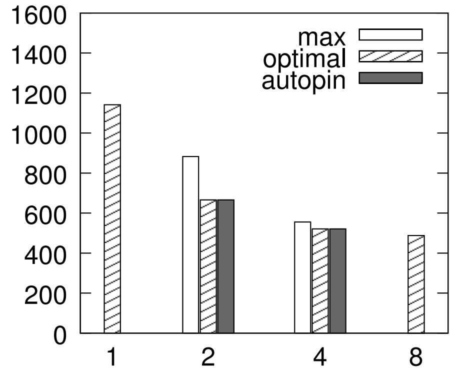

Fig. 6. 316.applu on Clovertown

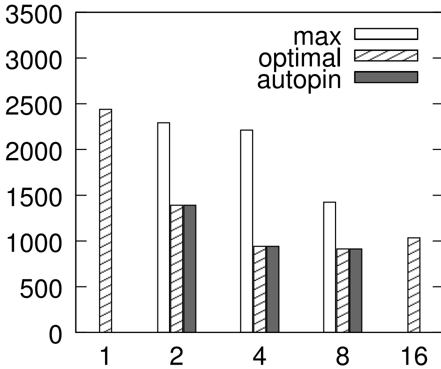

Fig. 7. 314.mgrid on Caneland

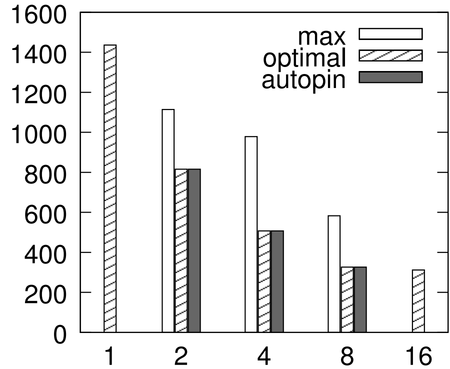

Fig. 8. 316.applu on Caneland

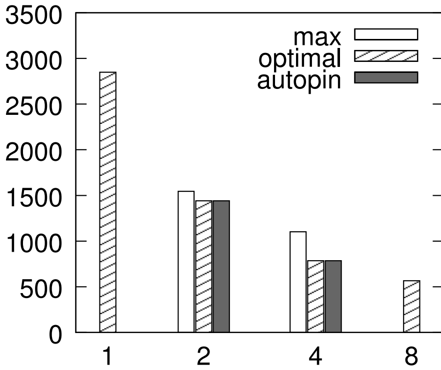

Fig. 9. 314.mgrid on Barcelona

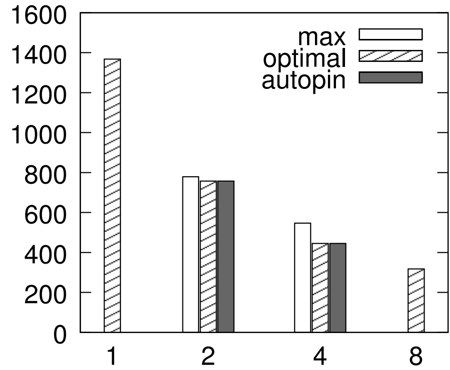

Fig. 10. 316.applu on Barcelona

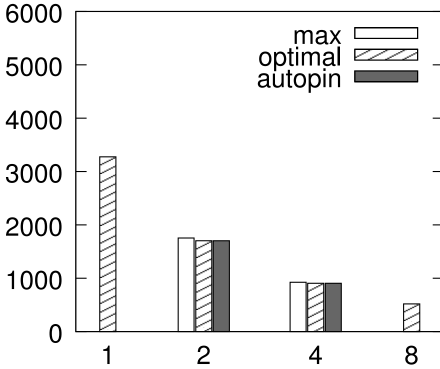

Fig. 11. 332.ammp on Clovertown

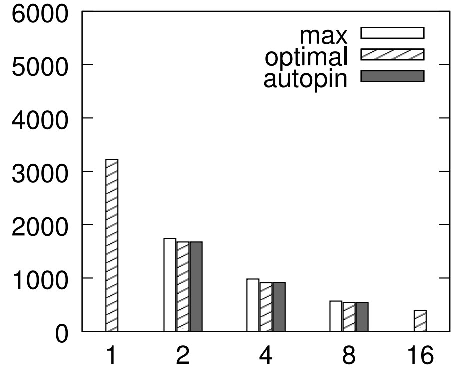

Fig. 12. 332.ammp on Caneland

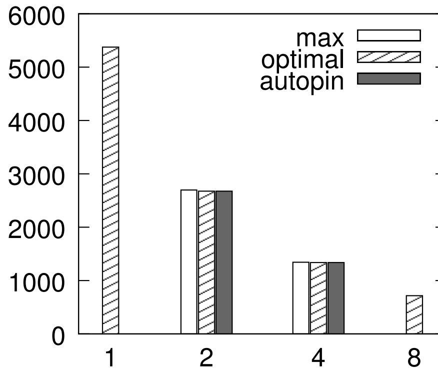

Fig. 13. 332.ammp on Barcelona

than 9% faster than the single-core setup. On Barcelona, wrong pinning does not show problems for 2 threads: the runtimes for both cases are within metering precision. For 4 cores the difference is approximately 20%. Furthermore, the scaling behavior on Barcelona is better than on Intel platforms: while the latter one can not benefit from more than 4 cores, the AMD system scales fine up to 8 cores. This leads to the fact that the total runtime for 8 Opteron cores is shorter than the runtime for 16 Tigerton cores. Given the fact that the single core runtime on the Opteron was 40% higher than on the Intel processors, this is remarkable.

Similar effects can be observed on the 316.applu benchmark (see figures 6, 8 and 10), especially on Caneland: Doubling the number of utilized CPU cores can slow down the computation if the wrong pinning is used. While this effect is weaker for the Clovertown, it still shows poor scaling performance. Again, using more than 4 cores does not improve performance at all. The Barcelona only shows runtime differences for the 4 core setup (44%). For optimal pinning, runtimes and scaling behavior is very similar to the Intel processors.

The 332.ammp benchmark draws a whole different picture as one can see on figures 11-13: Pinning of threads has almost no impact on the runtime and even on the Intel platforms we can see almost linear speedups up to 16 cores. We assume that this benchmark can run almost totally in cache and is therefore not limited by the memory bandwidth which is shared with the other cores.

5.2 Overhead Examination

As shown above, `autopin` was able to reliably detect optimal pinnings for nearly all benchmarks under consideration using the default parameters. However, to obtain maximum benefit for the user, the overhead imposed by `autopin` should be kept at a minimum level. This overhead is caused by the fact that during the different phases (initialization, warmup, and measurement) the application also runs with "slow" pinnings.

Hence, in order to find an optimal tradeoff between minimum overhead and reliable detection of optimal pinning, further experiments were carried out for different values of i, w, and t on the Barcelona and Clovertown platforms. These experiments showed that even for $i = 30s$, $w = 3s$, and $t = 10s$, optimal pinning is found for all benchmarks on the Clovertown platform. On the Barcelona

Table 3. `autopin` overhead on Clovertown: The first column shows the benchmark under consideration, the second column the number of threads, the third column the benchmark's total runtime with the fixed optimal pinning, the fourth column the total runtime under `autopin` probing several pinnings, the sixth column shows the difference between column three and four in per cent, the seventh column the total runtime with the slowest fixed pinning, and the last column shows the difference between column four and seven in per cent.

Benchmark	#Threads	Best Pinning [s]	autopin [s]	diff [%]	Worst Pinning [s]	diff [%]
310	2	655.12	659.27	0.63	785.95	19.22
310	4	484.53	486.94	0.5	656.02	34.72
312	2	932.41	949.13	1.79	1426.97	50.35
312	4	910.28	916.16	0.65	1388.78	51.59
314	2	1227.04	1231.28	0.35	1840.65	49.49
314	4	1058.51	1069.58	1.05	1779.32	66.36
316	2	656.73	667.29	1.61	882.99	32.32
316	4	522.95	528.57	1.07	787.69	49.02
320	2	287.24	295.01	2.71	379.63	28.68
320	4	253.64	258.25	1.82	350.08	35.56
324	2	572.85	574.82	0.34	604.66	5.19
324	4	313.63	314.83	0.38	345.05	9.6
328	2	1085.02	1090.55	0.51	1213.67	11.29
328	4	668.01	670.81	0.42	800.9	19.39
330	2	350.34	358.13	2.22	463.95	29.55
330	4	281.75	286.16	1.57	361.56	26.35
332	2	1755.84	1769.65	0.79	1795.86	1.48
332	4	941.98	941.57	-0.04	971.8	3.21

Table 4. `autopin` overhead on Barcelona. See caption of table 3 for an explanation of the table's columns.

Benchmark	#Threads	Best Pinning [s]	autopin [s]	diff [%]	Worst Pinning [s]	diff [%]
310	2	913.14	924.27	1.22	941.49	1.86
310	4	482.99	494.04	2.29	543.41	9.99
312	2	650.17	696.41	7.11	903.09	29.68
312	4	464.64	495.98	6.75	776.7	56.6
314	2	1440.37	1448.52	0.57	1545.21	6.68
314	4	784.52	793.36	1.13	1102.33	38.94
316	2	757.97	713.11	-5.92	779.39	9.29
316	4	445.45	420.22	-5.66	546.74	30.11
320	2	318.86	322.72	1.21	334.81	3.75
320	4	200.23	204.09	1.93	235.12	15.2
324	2	768.28	775.6	0.95	850.34	9.64
324	4	373.72	376.84	0.83	428.73	13.77
328	2	1064.43	1075.35	1.03	1125.79	4.69
328	4	571.38	599.47	4.92	655.09	9.28
330	2	567.76	569.73	0.35	573.98	0.75
330	4	297.51	299.4	0.64	306.17	2.26
332	2	2673.66	2697.62	0.9	2698.7	0.04
332	4	1334.3	1349.42	1.13	1343.9	-0.41

platform a slightly higher value of $w = 10s$ was required, which is necessary for page migration to take place.

Tables 3 and 4 show the total runtimes of the SPEC OMP benchmarks for the fixed optimal pinning, under `autopin` probing several pinnings, and for the fixed slowest pinning. Column five shows the relative runtime overhead in per cent imposed by `autopin`. On the Clovertown platform this overhead turns out to be below 3% in all cases. Running the application without `autopin` may cause runtimes up to 66% higher than those yielded by `autopin` in case the operating system's scheduler happens to choose the worst pinning as depicted in the last two columns. Due to the additional cost of memory page migration, the overhead on the Barcelona platform is slightly higher for memory intensive applications (up to 7.5%). Nevertheless, compared to the worst pinning, significant runtime improvements can be achieved. Interestingly, for the 316.applu benchmark the runtimes under `autopin` are even lower than the best fixed pinning. This might be due to the fact that this application prefers different pinnings in different program execution phases. See section 6 for a further discussion of this observation.

6 Conclusion and Outlook

In this paper we pinpointed the importance of correct CPU pinnings that account for both application characteristics as well as hardware properties. It is obvious that this topic will become even more crucial with future multicore processor

architectures, which will have much more complicated on-chip interconnects with strongly varying access speeds. Remarkably, the best and worst pinnings for some applications yielded a runtime difference of more than 100 per cent.

Additionally, we presented the `autopin` framework, which allows to automatically determine the thread pinning best suited for a shared memory parallel program on a selected architecture. This is achieved by evaluating the performance of different pinnings by means of hardware performance counters. It has been shown that `autopin` reliably proposes optimal pinnings for the SPEC OMP benchmark on UMA as well as NUMA architectures.

Future versions of `autopin` can be improved in several ways. At the moment the user needs profound knowledge on the hardware infrastructure (i.e. how many cores are available on how many sockets, how many cores are on a chip, which cores do share caches, etc.) in order to choose a reasonable set of schedule mappings. To make the tool easier to use for people with no background in computer architecture, a mechanism could be implemented that automatically detects the hardware infrastructure and selects appropriate schedule mappings to be analyzed. A promising idea that goes one step further is to integrate parts of `autopin` into the scheduler of the Linux kernel.

In its current version, `autopin` starts with one pinning and switches to the next pinning after a specified time frame and so on. When no more pinnings to be tested are left, `autopin` re-pins to the best mapping found so far and uses this pinning until the program terminates. This behavior could be inappropriate for programs that have strongly varying execution phases. For example, a parallel program with four active threads might have a first phase in which it is memory bound. Within this phase, distributing threads over four different chips makes much more sense than putting all threads together onto one chip. Consider the next phase to be dominated by very fine grain communication with all relevant data being held in caches. This time the situation is vice versa, and pinning all threads onto one chip with four cores sharing a L3 cache would be most efficient. Taking these considerations into account, the idea is to adapt `autopin` to continuously monitor the application and restart the repinning process if the application's performance drops under a certain threshold.

Acknowledgements

The work presented in this paper has been carried out in the context of the Munich Multicore Initiative MMI[3]. The authors would like to thank Sun Microsystems and Intel Corporation, who kindly provided hardware platforms for our experiments.

References

1. Graham, S.L., Kessler, P.B., McKusick, M.K.: gprof: a Call Graph Execution Profiler. In: SIGPLAN Symposium on Compiler Construction, pp. 120–126 (1982)
2. Intel: VTune Performance Analyzer, `http://www.intel.com/software/products/vtune`

[3] `http://mmi.in.tum.de`

3. Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S.: Cache-Oblivious Algorithms. In: FOCS 1999: Proceedings of the 40th Annual Symposium on Foundations of Computer Science, p. 285. IEEE Computer Society Press, Washington, DC (1999)
4. Intel: Math Kernel Library, `http://developer.intel.com/software/products/mkl`
5. Whaley, R.C., Dongarra, J.J.: Automatically Tuned Linear Algebra Software. Technical report (1997)
6. Intel Corporation: Intel 64 and IA-32 Architectures: Software Developer's Manual, Denver, CO, USA (2007)
7. Advanced Micro Devices: AMD64 Architecture Programmer's Manual. Number 24593 (2007)
8. Browne, S., Dongarra, J., Garner, N., London, K., Mucci, P.: A scalable cross-platform infrastructure for application performance tuning using hardware counters. In: Supercomputing 2000: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing, Washington, DC, USA, p. 42. IEEE Computer Society, Los Alamitos (2000)
9. Levon, J.: OProfile manual, `http://oprofile.sourceforge.net/doc/`
10. Eranian, S.: The perfmon2 Interface Specification. Technical Report HPL-2004-200R1, Hewlett-Packard Laboratory (February 2005)
11. OpenMP.org: The OpenMP API specification for parallel programming, `http://www.openmp.org/`
12. Chapman, B., an Mey, D.: The Future of OpenMP in the Multi-Core Era. In: ParCo 2007: Proceedings of the International Conference on Parallel Computing: Architectures, Algorithms and Applications, pp. 571–572. IOS Press, Amsterdam (2008)
13. an Mey, D., Terboven, C.: Affinity Matters!, `http://www.compunity.org/events/pastevents/parco07/AffinityMatters_DaM.pdf`
14. Chapman, B.: The Multicore Programming Challenge. In: Xu, M., Zhan, Y.-W., Cao, J., Liu, Y. (eds.) APPT 2007. LNCS, vol. 4847, p. 3. Springer, Heidelberg (2007)
15. Fürlinger, K., Moore, S.: Continuous runtime profiling of openmp applications. In: Proceedings of the 2007 Conference on Parallel Computing (PARCO 2007), pp. 677–686 (September 2007)
16. Ott, M., Klug, T., Weidendorfer, J., Trinitis, C.: autopin - Automated Optimization of Thread-to-Core Pinning on Multicore Systems. In: Proceedings of 1st Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG) (January 2008), `http://www.lrr.in.tum.de/~ottmi/docs/multiprog08.pdf`
17. Schermerhorn, L.T.: Automatic Page Migration for Linux - A Matter of Hygiene (January 2007); Talk at linux.conf.au 2007
18. Saito, H., Gaertner, G., Jones, W.B., Eigenmann, R., Iwashita, H., Lieberman, R., van Waveren, G.M., Whitney, B.: Large system performance of spec omp2001 benchmarks. In: Zima, H.P., Joe, K., Sato, M., Seo, Y., Shimasaki, M. (eds.) ISHPC 2002. LNCS, vol. 2327, pp. 370–379. Springer, Heidelberg (2002)
19. Weidendorfer, J., Ott, M., Klug, T., Trinitis, C.: Latencies of conflicting writes on contemporary multicore architectures. In: Malyshkin, V.E. (ed.) PaCT 2007. LNCS, vol. 4671, pp. 318–327. Springer, Heidelberg (2007)