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

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

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

Abstract. In this paper we present a framework for automatic detection of the best binding between threads of a running parallel application and processor cores in an SMP 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 a lot of the 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 almost always find the best binding. 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.

Key words: 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. With multicore architectures becoming standard, clock frequencies have become more or less stable, but the number of cores on a die is increasing. 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.

Asymmetric properties 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 improve runtime, the nonexistence 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 her 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 interconnection and cache buffer hierarchies is evident. The ubiquity of multicore processors nowadays on standard computer hardware can easily lead to a situation where deterministic runtimes become a myth for programmers, forcing sequential programming on a 256 core processor for runtime predictability.

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 a first study, we used `autopin` to find optimal pinnings for applications 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.

2 Related Work

With the large amount of computer systems available today, there is no global strategy for performance optimization. One 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: Commercial software and library packages are available for certain classes of systems only. 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. A prominent example using this strategy is the Atlas

[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.

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

library [5]. Our automated search for best thread-to-core pinning takes a similar approach.

Initially used for internal correctness checks after production only, *Hardware Performance Counters* are integrated into processors on the market nowadays. The amount of different events that can be measured differs heavily among processors (e.g. see [6] or [7]). 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 was required to install a kernel patch (Perfctr), complicating the use significantly. However, HP has started to work on another kernel patch which is called perfmon2 [10]. This patch initially existed in the Linux Itanium architecture only. It provides support for latest Intel and AMD processors. Its user level parts (`libpfm`, `pfmon`) form the basis for `autopin`.

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:

Upon creation, each new thread is enumerated and pinned to one specific CPU core using the `sched_setaffinity()` system call. 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. If so, the tool will probe each of these sets for a certain time interval n which can be specified using the `-t` parameter. The probing is performed using the following algorithm:

1. Let the program "warm up" for $n/2$ seconds.
2. Read the current timestamp t_1 and value p_1 of the performance counter for each thread.
3. Run the program for n seconds.
4. Read the current timestamp t_2 and value p_2 of the performance counter for each thread.
5. Calculate the performance rate $r_i = (p_2 - p_1)/(t_2 - t_1)$ for each thread i 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 mapping in the list and return to 1.

The "warm up" time at the beginning of each probing cycle is needed for the actual rescheduling of the threads and to refill the cache.

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 n seconds the current performance rate is calculated and written to the console.

As non-optimal pinnings are used in the beginning, a slight overhead is imposed during this phase. However, in most cases this overhead can be neglected, especially when n is small compared to total application runtime.

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`.

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`. SPEC OMP 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[11].

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),

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 1 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 was detected with the authors' false sharing benchmark [12].

- 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 2). 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.
- A two socket machine, equipped with two AMD Opteron 2347. 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 1.9 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 3). 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

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
330.art	neural network simulation of adaptive resonance theory
320.equake	finite element simulation of earthquake modeling
324.apsi	weather prediction
326.gafort	genetic algorithm code
328.fma3d	finite-element crash simulation
332.ammp	computational chemistry

Table 2. Investigated CPU Pinnings for Different CPU Architectures

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

`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.

As described in chapter 3, `autopin` starts with a short warmup phase and then probes the different CPU pinnings read out of the environment variable `SCHEDULE` for a specified time period. For the measurements described in this paper, a time period of 30 seconds was chosen.

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.

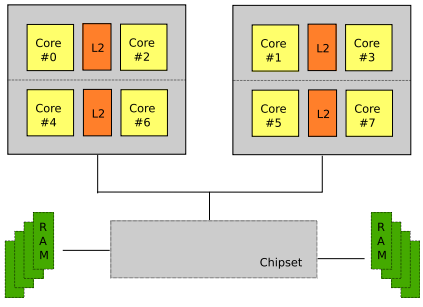

Fig. 1. Intel Clovertown System

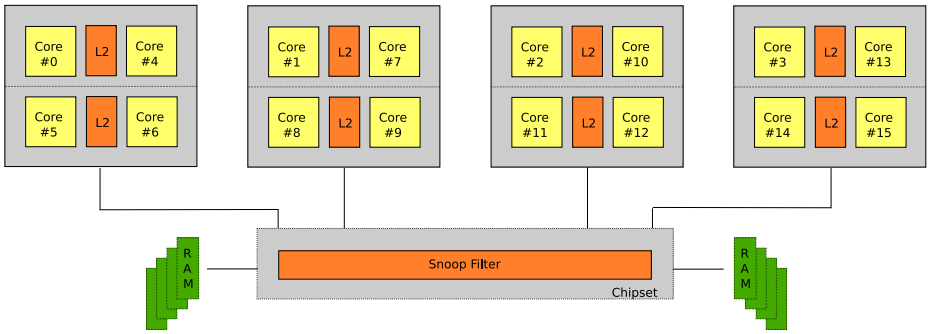

Fig. 2. Intel Caneland Platform

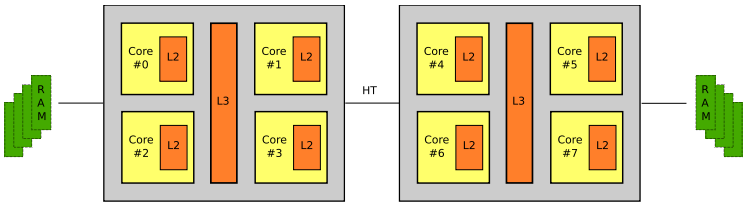

Fig. 3. AMD Barcelona System

– 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 on Caneland runs 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 (the first column shows the number of threads used, the second to fourth columns stand for the different thread-to-core pinnings). 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,

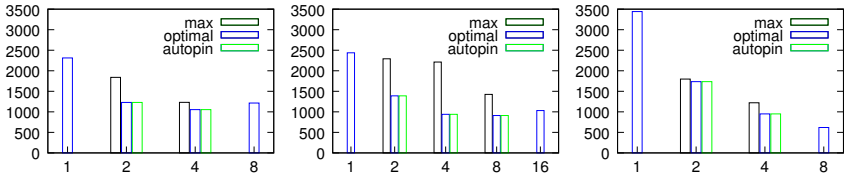

Fig. 4. 314.mgrid on Clovertown (left), Caneland (middle), Barcelona (right)

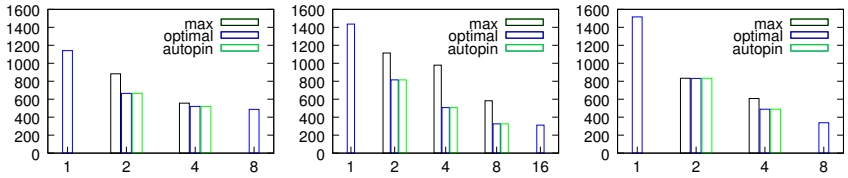

Fig. 5. 316.applu on Clovertown (left), Caneland (middle), Barcelona (right)

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, autpoin 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

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:

- On the Clovertown platform, our `autopin` tool found the optimal pinning for all benchmarks.
- On the Caneland platform it proposed a wrong pinning two times. However, the difference in total runtime of the optimal pinning and the pinning proposed by `autopin` was below 1% which is below metering precision.
- On the Barcelona platform, the pinning suggested by `autopin` seemed to be influenced by the order in which the pinnings were listed in the `SCHEDULE` variable: if the optimal pinning was listed first, it was found. If it was listed last it only was found if the runtime gap between the best and worst pinning was significant (over 10%). This is likely due to the fact that the Barcelona system is a NUMA architecture. Though the threads can easily be moved

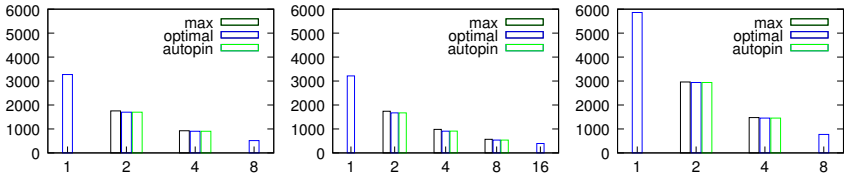

Fig. 6. 332.ammp on Clovertown (left), Caneland (middle), Barcelona (right)

to another core on another chip, the memory allocated by the thread remains on the origin node. All following memory accesses thus have to use the Hypertransport which has a higher latency and lower bandwidth than a direct memory access to local memory. However, now that we are aware of the problem, a solution to just obtain the optimal pinning is the following: Instead of rescheduling the running threads to other cores, the program has to be terminated and restarted with a different pinning. This way, each thread can access the memory locally and maximum bandwidth is therefore guaranteed.

- 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 exploited. 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 datasets, 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.

Figure 4 shows the total runtimes (in seconds) of the 314.mgrid micro 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 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 processors. 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 figure 5), 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 weak 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 the 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 figure 6: 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.

6 Conclusion and Outlook

In this paper, the `autopin` framework was presented. `autopin` allows to determine which thread pinning is best suited for a shared memory parallel program on a selected architecture. Performance analysis is done by means of hardware performance counters which can be freely specified by the user. It could be shown that `autopin` always proposed optimal pinning for the SPEC OMP benchmark on UMA architectures, aside from very few cases where the difference in runtime between optimal pinning and the pinning obtained by `autopin` was less than one percent. There is a drawback for `autopin` on the NUMA architectured Barcelona platform: When a thread is moved away from the chip where it has been created (and it's memory has been allocated on), all following memory accesses have to be performed over the slower Hypertransport which usually slows down program execution. Remarkably, the best and worst pinnings for some benchmark applications yielded a runtime difference of more than 100 per cent. Keeping these numbers in mind, it is obvious that CPU pinning is an important topic that will become even more crucial with future multicore processor architectures, which will have much more complicated on-chip interconnects with strongly varying access speeds.

Future versions of `autopin` can be improved in several ways. At the moment the user has to know the hardware infrastructure (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 analyze. 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 allow for specifying different execution phases, which even can be triggered by the application itself.

Finally, a future version of `autopin` will be able to migrate memory pages from one node to another: Whenever a thread is re-pinned to a core on another chip, the according memory pages are migrated too.

References

1. Graham, S.L., Kessler, P.B., McKusick, M.K.: gprof: a Call Graph Execution Profiler. In: SIGPLAN Symposium on Compiler Construction. (1982) 120–126
2. Intel: VTune Performance Analyzer. `http://www.intel.com/software/products/vtune`
3. Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S.: Cache-Oblivious Algorithms. In: FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science, Washington, DC, USA, IEEE Computer Society (1999) 285
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 '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing, Washington, DC, USA, IEEE Computer Society (2000) 42
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. 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: ISHPC '02: Proceedings of the 4th International Symposium on High Performance Computing, London, UK, Springer-Verlag (2002) 370–379
12. Weidendorfer, J., Ott, M., Klug, T., Trinitis, C.: Latencies of conflicting writes on contemporary multicore architectures. In Malyshkin, V.E., ed.: PaCT. Volume 4671 of Lecture Notes in Computer Science., Springer (2007) 318–327