A Topology-Aware Performance Monitoring Tool for Shared Resource Management in Multicore Systems

Nicolas Denoyelle(✉), Brice Goglin, and Emmanuel Jeannot

Inria Bordeaux - Sud-Ouest – LaBRI, Université de Bordeaux, Talence, France
{Nicolas.Denoyelle,Brice.Goglin,Emmanuel.Jeannot}@inria.fr

Abstract. Nowadays, performance optimization involves careful data and task placement to deal with parallel application needs with respect to the underlying hardware topology. Monitoring the application behavior provides useful information that still needs to be matched with the actual placement, for instance to understand whether bottlenecks are caused by the sequential code itself or by shared resources in parallel programs.

We propose an insightful monitoring tool based on two cornerstones of hardware performance counters monitoring and hardware locality modeling, respectively named PAPI and hwloc. It enables a dynamic visual analysis of parallel applications' phases at runtime, revealing their possibly variable and heterogeneous behaviors and needs. A purpose designed application shows that the topology-aware visual representation of hardware counters can help figuring out shared resource bottlenecks and ease the task placement decision process in runtime systems.

1 Introduction

The memory wall makes data locality increasingly important on the road to exascale. Data and computing tasks have to be colocated to better exploit the performance of parallel platforms. Many research projects focus on locality-aware data and/or task placement, for parallel programing models ranging from MPI and OpenMP to graphs of tasks. However finding out which placement is the best remains a difficult exercise that depends on the topology and characteristics of the hardware and on the application needs. Indeed, the hardware is increasingly complex, and software affinities can be of different kinds. For instance memory-bound tasks may prefer being scattered all across the machine, while, on the contrary, communication and synchronization may want to keep them close. Runtime systems require help identifying these needs and bottlenecks before they can place tasks accordingly.

Performance monitoring is a very active software area that offers many tools to gather information about the execution of tasks, the bottlenecks, *etc.* We introduce, in this paper, a new way to analyze performance by crossing the roads of performance monitoring and topology-aware placement. We propose an

S. Hunold et al. (Eds.): Euro-Par 2015 Workshops, LNCS 9523, pp. 710–721, 2015.
DOI: 10.1007/978-3-319-27308-2_57

extension of the Hardware Locality software (hwloc [2]) that enhances its graphical representation of the topology with performance monitoring information. This new tool enables optimization of the placement of parallel tasks based on visual monitoring.

The remaining of the paper is organized as follows. Section 2 presents the context of our work before the state of the art is described in Sect. 3. Section 4 details the goals, features and implementation of the proposed tool while an in-depth use case is studied in Sect. 5.

2 Context

The domain of parallel computing has undergone a shift in the way applications are executed with the advent of multicore systems. Nowadays, systems with more than 10 cores and a deep memory hierarchy (multiple levels of caches) are common place. Moreover, deeper data paths (flash, non-volatile memory, RAM, caches) and larger systems (the next Intel Knight Landing processor will provide more than 60 cores with 4 threads each) are expected in a near future. Such architectures feature many characteristics that make the execution highly sensitive to the way the computations are mapped. Indeed, threads exhibit different kinds of affinities as they do not use the same amount of memory or exchange the same amount of data.

Several works [6,7,10] showed that the way the affinity is managed has an important impact on the application performance: memory accesses depend on the mapping of the threads and the data location. Therefore, it is crucial to understand how this mapping impacts the performance. Moreover, it is the case for every steps of the application: the data allocation, the I/O (network, storage, etc.) or the computation. This is even more intricate when computation is composed of phases where the affinity between tasks changes. For instance a computation phase may be compute intensive, *e.g.* heavily use floating-point units, before another phase is memory intensively. To cope with this phase heterogeneity, some systems feature co-scheduling where compute-intensive applications are mixed with memory-bound ones expecting that the different application bottlenecks (here memory/cache vs. compute units/cores) will not interfere with each other.

In any case, whether the system executes one application or co-schedules several, it is very important to be able to efficiently map the processes and the threads on the different cores. To understand the impact of such mapping and analyze the performance of the running application(s), in the light of the mapping, applications and system developers need tools to monitor the application behavior. Such monitoring tool need not only to be able to display the performance of the application but also need to link the performance with the topology where the application is running. We believe that this last part is very important as the performance can only be understood if it is matched with the memory hierarchy and the used cores.

3 State of the Art

Performance analysis can be performed at different levels of granularity, from individual instructions up to the entire program. Analyzing the performance at the instruction level with tools such as MAQAO [1], PIN [9] or Intel VTune Amplifier is a part of the development and optimization process. We rather focus on optimizing through better placement during the execution. Moreover we target parallel applications with a regular execution pattern (for instance applications programmed with MPI, PGAS and/or threads) without dynamic scheduling of many tasks that would require a finer grain. Our coarse grain approach targets the entire application or different phases during its execution by observing its behavior from a higher level as the Unix `top` tool would.

Coarse-grain performance analysis still requires dynamic monitoring over time. It may involve real time tools such as `numatop` or tiptop [12], or offline temporal analysis with one of the existing tracing tools such as VampirTrace [8]. We base our work on these existing approaches while focussing on topology-aware performance study both in real-time or for post-mortem analysis.

Multiple metrics may be used to diagnose topology-related performance issues, including memory link contention, cache conflicts, or computing unit shared accesses. Performance counters are the main solution for analyzing the behavior of codes and numerous tools are available such as PAPI [3] or the Linux `perf` utility. We focus on intra-node performance in this paper while other metrics exist for entire cluster-wide, such as congestion in network switches or links, which may be studied with tools such as SCALASCA [4] or Paraver [11].

Analyzing performance based on the topology of the architecture is not a very active research topic yet. Indeed, discovering all the computing and memory resources in computing platform has only been recently mastered with tools such as hwloc [2]. Former approaches were often less portable or do not expose as many details about cache sharing *etc.* MemAxes [5] offers fine-grained memory performance analysis with a graphical radial hierarchy display. However, it only focuses on static post-mortem analysis of memory accesses while our approach is dynamic and works for all performance metrics and more kinds of resource sharing. LIKWID [13] is a set of performance analysis tools that use advanced knowledge of the hardware topology. This knowledge is used for task placement while we also propose to combine it with performance monitoring for better analysis. LIKWID is actually complementary to our work, it will soon use hwloc for better topology discovery, while we may use LIKWID performance monitoring abilities when they will be exported as a C programming interface.

4 Topology-Aware Performance Monitoring

Here, we describe the proposed extension of `lstopo` utility : design, usage and implementation.

4.1 Objectives and Features

`lstopo` is a utility from hwloc which function is to display machines topologies. The main goal of the new `lstopo` extension is to provide a fast and simple way to analyze the architecture behaviour during a program execution. Such a hardware based outlook can be used just to get a quick glance on the machine state as well as for more complex studies such as task placement optimization. It fits especially well multi-phase applications such as code coupling or complex structure traversal.

The original `lstopo` displays the hierarchy of computing resources (processors, NUMA nodes, cores, threads, caches, *etc.*) as nested boxes (see Fig. 1(a)). Inner boxes represents smaller resources (*e.g.* cores within processors or even hyperthreads).

The extension aims to stay as simple as the original tool. From the user perspective, the new graphical output keeps the existing boxes organization intact but changes their inner text and colors to report performance information. Box names are replaced with performance monitored values, whose variation (with respect to the maxima reached across the execution) is also represented by a horizontal line and the background color. At a given rate, the bar moves vertically and the color changes according to the monitored value (see Fig. 1(b)): from green to red (with different intermediate shades of yellow and orange). This implementation keeps the original technologies used in hwloc: The Cairo library is still used to draw the graphical output but it now periodically refreshes the display to update counters.

As explained in the next section, a few lines of input configuration enable the displaying of live, derived performance counters.

Because you cannot always get a graphical display on HPC platforms, performance counters may also be recorded with low overhead for later offline display. Indeed, just like hwloc lets you manipulate topologies of remote machines, the new `lstopo` may also load performance counters for more convenient displaying on another host. Moreover, performance counters may also be exported in a trace file in PAJÉ format[1] for later post-mortem analysis with any paje trace analyzer.

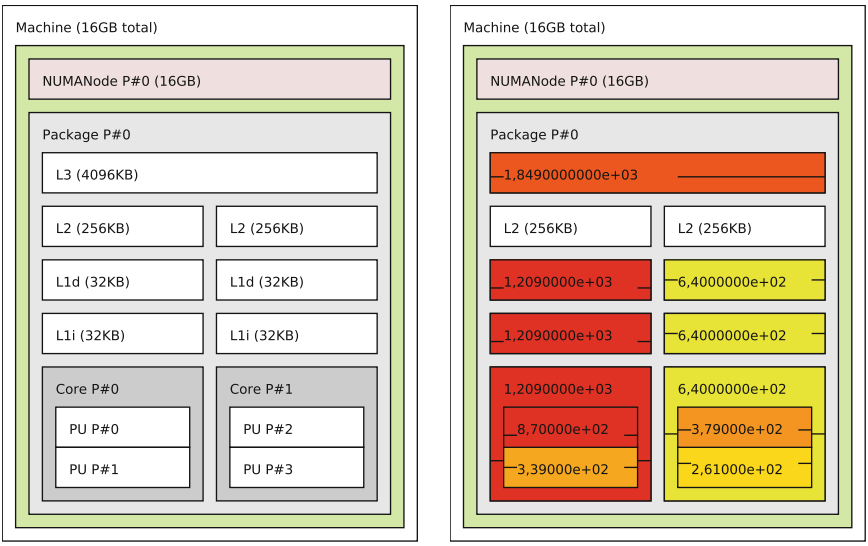
Several options can be set such as sampling rate, replay speed, accumulating values or not. Moreover, we provide two monitoring modes: *per node* where performance counters are displayed for all the process running on the NUMA node and *per process* where performance counters are displayed for a given process.

Finally, for those whom want to record specific part of an application, a library interface is provided to record parts of a whole application and set markers to delimit phases into the trace.

4.2 Usage and Configuration

Figure 1(a) shows the original `lstopo` output on a dual-core Intel i7-4600U processor. Figure 1(b) shows the same display with `--perf` option appended. Coloured boxes represent the monitors.

[1] http://paje.sourceforge.net/index.html.



(a) Istopo classic output (b) Istopo performance output

Fig. 1. Graphical `lstopo` output (Color figure online)

The topology structure is a hierarchy of resources. For each hierarchy level (cores, caches, threads (PU), *etc.*), the user may set an arithmetic expression of counters meant to represent a metric associated with that level. A file containing such a description is specified to `lstopo` with the option `--perf-input` and is written with the following syntax:

```
name {hwloc_obj_type_t where to accumulate,
      algebraic formula using PAPI counters}
```

An example of such syntax is:

```
CYC_per_INS {PU, PAPI_TOT_CYC/PAPI_TOT_INS}
PER_CORE_L3_MISS {Core, PAPI_L3_TCM}
PER_NUMA_L3_MISS {NUMANode, PAPI_L3_TCM}
```

In this example, the program would count on each processor hyperthread the number of cycles (`PAPI_TOT_CYC`), the number of instruction (`PAPI_TOT_INS`) and the number of cache misses in the L3 (`PAPI_L3_TCM`). The latter is summed by Core (2nd line) and by NUMA node (3rd): in this case all the cache misses are displayed at the node level. The formers are used to compute the ratio of cycles per instruction on each hyperthread (1st line).

More complex expressions could be used to translate raw counter values into higher-level criterias such as the memory bandwidth, or another performance monitoring library could be used instead of PAPI to directly gather such information.

4.3 Implementation

The tool uses hwloc to build the topology and store counters' values into its nodes. The lstopo utility (which shows the machine topology) was updated to add a hook plugin, periodically sampling hardware counters, and changing nodes rendering as the monitors' hierarchy is updated.

```
   Data: map(hwloc object type, counters), hwloc topology, process ID
   Result: trace(timestamp, topology object, monitor value)
 1 Spawn a thread per leaf waiting in a barrier;
   /* The main thread triggers counter's collect periodically        */
 2 repeat
 3 |   forall the topology's leaves do
 4 |   |   if isRunning(leaf,pid's thread) then activate(sampling(leaf));
 5 |   end
 6 |   wait(barrier);
 7 until wait(timer);
   /* The others gather samples                                      */
 8 forall the thread do
 9 |   repeat
10 |   |   wait(barrier);
11 |   |   if thread's leaf is not activated then  continue ;
12 |   |   read(counters);
13 |   |   forall the node in leaf's parents into the monitor hierarchy do
14 |   |   |   forall the counters do
15 |   |   |   |   if counter physical location is under node then
16 |   |   |   |   |   sum counters into node;
17 |   |   |   |   |   if Current leaf is the last to update node then
18 |   |   |   |   |   compute the user input arithmetic expression of counters ;
19 |   |   |   |   else
20 |   |   |   |   |   if Current leaf is the first to update node then
21 |   |   |   |   |   Set the node and sibling counters value to the read counter ;
22 |   |   |   |   end
23 |   |   |   end
24 |   |   end
25 |   until isAlive(pid);
26 end
```

Algorithm 1. Dynamic counter aggregation algorithm.

When performance monitoring is enabled, the topology is redrawn periodically after gathering counters.

The new `lstopo` can also attach to a process to track and record a single process placement and performance values.

Performance counters are usually collected for each processing unit leaf of the topology (hyperthreads in the above example) but `lstopo` may accumulate them in parents. For counters that are not per-core, they are usually displayed higher in the hierarchy, for instance in the system or NUMA node box.

As an example, displaying the last level cache (LLC) miss count on the LLC itself would sum the total number of LLC miss performed on all cores, whereas displaying it on each Core would show the miss count performed by each Core. Algorithm 1 describes this behavior synthetically.

The aggregation currently only implements the addition of counters from all children resources, but we plan to support average, min and max operations as an optional flag in the language in the near future.

5 Analyzing Tasks Concurrency Gives a Room for Thread Placement

Here, we will illustrate a situation where a visual hint can be useful to choose thread placement. In this example the monitoring mode is the *per node* one: we monitor all the processes of the node.

5.1 Spreading or Packing Threads?

Tasks affinity is a widely studied problem in HPC because it can suit several optimizations such as, MPI-process placement, node allocation, co-scheduling, *etc.* At the node granularity, the efficiency of scheduling threads under a shared cache depends both on the pressure they put on the cache and the reuse distance of their shared data. Two solutions often discussed are: either spread threads to balance the pressure on caches, or pack them to optimize shared data access [10]. These two placement policies are widely used and implemented in most OpenMP Runtime. But whether you should use one or the other is left to the user responsibility.

5.2 A Use Case of Threads' Interference Balancing, Using the Cache Miss Ratio

Some works use the cache miss ratio as a metric to measure pressure on cache [14] and decide on thread's placement policy.

Based on this observation we build a small application able to put arbitrary pressure on the last level cache, and thanks to hardware counter information we will be able to see where are the threads pinned on the Cores, and the amount of pressure they put on the last level cache.

The application we monitor is a walk into a linked list. It basically consists in loading data and therefore should be sensitive to memory stress. Each item in the list is sized to fit into a cache line and the list is randomly linked to avoid successful prefetching by the processor.

Before doing any measure, we walk the whole list so that each attempt to resolve the next pointer will trigger a cache miss if the list size is greater than the cache size as shown in Fig. 2.

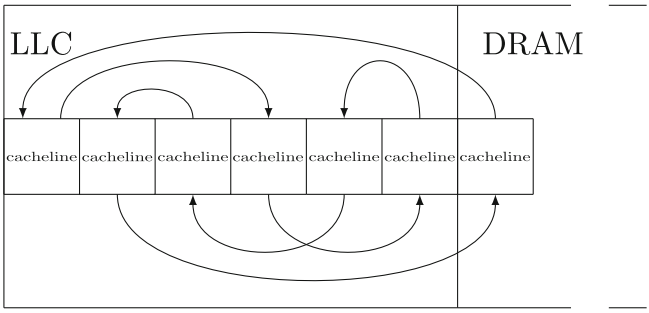


Fig. 2. Randomly linked list.

5.3 Experimental Conditions

Figure 3 shows the LLC miss count evolution when changing the list size and running n threads walking each a cloned list.

The vertical line at 2^{20} shows that walking simultaneously 4 lists of size 2^{20}KiB, results in a reasonable amount of LLC misses even if the 4 lists should fit into the last level cache.

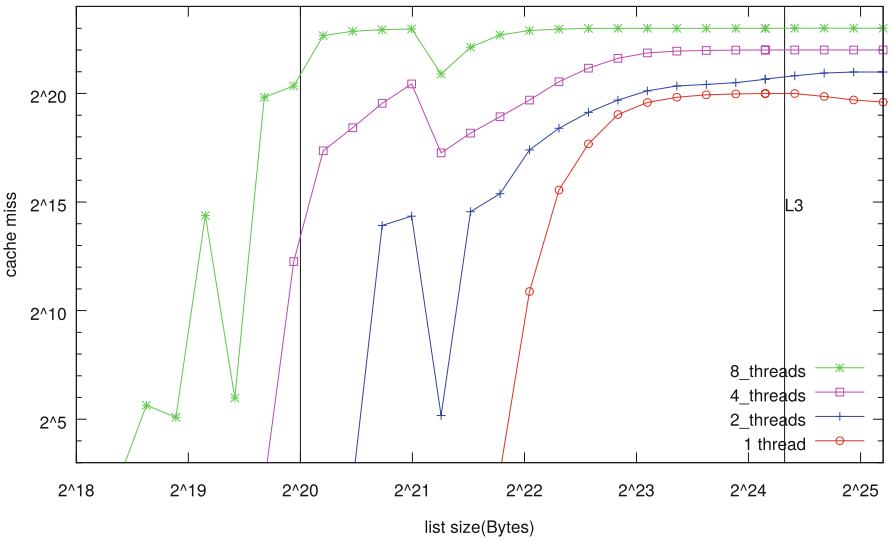


Fig. 3. Evolution of cache miss count of the node when increasing pressure on cache.

Hence we build two lists *list1* and *list2* of respective memory size $s1 = 2^{20}$KiB and $s2 = 2^{24}$KiB. The 4 threads of the former one may have their whole data set fit simultaneously in the last level cache, whereas 2 threads of the latter

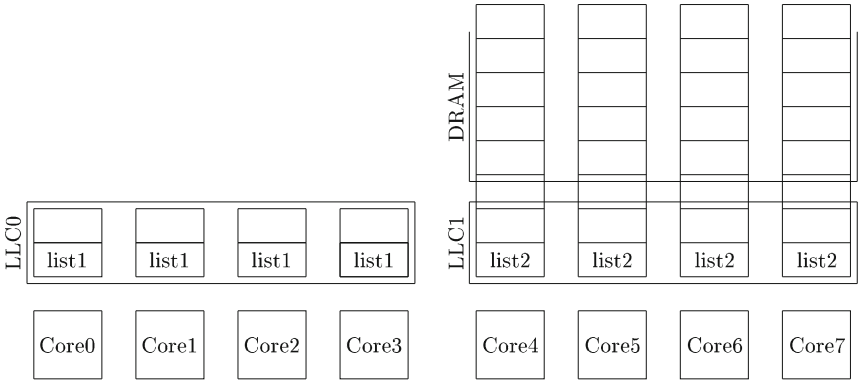


Fig. 4. Co-scheduling low-pressure threads, then high pressure threads.

would already overflow the cache. Using a topology with two 8-core processors, we co-schedule 4 threads, each one walking a clone of *list1* and 4 threads, each one walking a clone of *list2* spread across the processors so each processors hosts 4 threads. In the first scenario we schedule 2 threads of *list1* and 2 threads of *list2* on each processor whereas in the second scenario Fig. 4, 4 *list1* are walked on a processor and 4 *list2* are walked on the other.

5.4 How Lstopo Shows the Situation

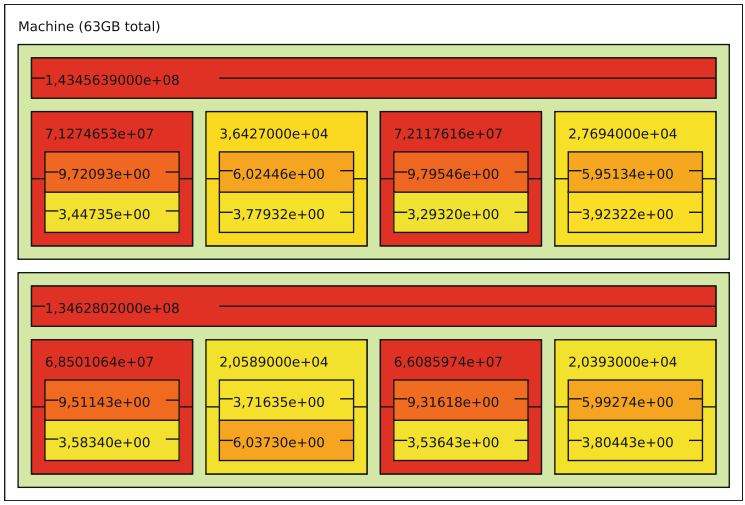
In this context, co-scheduling walks of *list1* with walks of *list2* would grow the number of miss for *list1* walks and slow the miss count for *list2* walks, whereas doing the opposite strategy would reverse the tendency. What about the total number of cache misses?

Figure 5 illustrates both scenarios from `lstopo` view. The topology has been restricted to only 4 cores per processors and cache boxes are hidden for clarity. The input configuration files used was the one described before in subsection implementation.

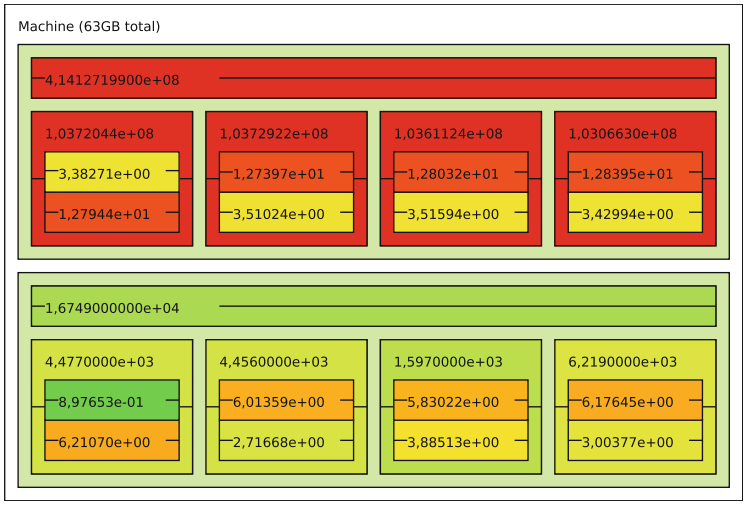
The upper most red and green boxes represent the machine NUMA nodes and we display on them the total amount of LLC miss for each node. The red, yellow and green outer boxes represent the machine cores 3,4,8,9 in logical numbering. Each single core LLC miss count is displayed on it. The red, yellow and inner boxes represent the machine hyperthreads on which is displayed the hyperthread cycles per instruction ratio.

In the first scenario Fig. 5(a), we use the scatter placement strategy, and see distinctly the pressure being balanced across the nodes. Whereas the second scenario Fig. 5(b) exhibits the packed strategy lowering considerably the pressure on the LLC for *list1* walk threads. However, the latter increases the number of cache misses on the most pressured cache by more than twice, suggesting that the first scenario is better than the second one to optimize total LLC miss count.

Actually, the execution walltime of the first scenario is 8.32 s while the second one is 11.08 s.



(a) Linked list walk, scattered threads



(b) Linked list walk, packed threads

Fig. 5. Two scenarios' comparison with lstopo (Color figure online).

6 Conclusion and Future Work

The road to exascale requires careful design of parallel runtime systems with respect to software affinities and hardware locality so that task and data can be properly colocated. Analyzing hardware performances values and matching them with topology information is crucial to understand and optimize resources

utilization. In this article, we presented a tool based on hwloc[2], able to collect performance, topological informations, and match them to deliver valuable hints. This tool extends the `lstopo` utility to display *per node* or *per process* performance counters. It is able to aggregate these counters at a given level of the topology hierarchy and to combine them through algebraic formulas. Colors and bars are used to display the values of these counters dynamically on the topology. Additionally, it is able to keep a trace of an execution to be replayed later or analyzed afterward. We showed with a low-level application and relevant performance metrics that mapping hardware counters on machine topology can bring out locality issues.

We are now working at improving the output by matching performance counters with the corresponding source code, and allowing the display of multiple counters per box. Then detection of application phases is also being investigated in the case of post-mortem analysis. Indeed, the trace may be displayed as a graph or analyzed with statistical tools to study the dynamic behavior of applications during the execution. Applications with varying heterogeneous behavior may indeed benefit from dynamic re-placement between phases.

Finally, our performance monitoring currently relies on PAPI high level abstraction and consequently inherits its strength such as simplicity of use, but also its weaknesses. For instance, PAPI does not expose advanced memory access counters (available in recent processors with Intel PEBS or AMD IBS). This limits our current abilities contrary to MemAxes which manually supports them. Depending on the support for most relevant technologies we envision support for other performance monitoring interfaces.

Acknowledgment. We would like to thanks Erwen Rohou for insightful discussion, Heike McCraw and Asim Yarkhan for providing us with useful hints about PAPI. This work is partially funded under the ITEA3 COLOC project #13024.

References

1. Barthou, D., Rubial, A.C., Jalby, W., Koliai, S., Valensi, C.: Performance Tuning of x86 OpenMP codes with MAQAO. In: Müller, M.S., Resch, M.M., Schulz, A., Nagel, E. (eds.) Tools for High Performance Computing 2009, pp. 95–113. Springer, Heidelberg (2010)
2. Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., Namyst, R.: hwloc: a generic framework for managing hardware affinities in HPC applications. In: The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing, PDP 2010, Pisa, Italy. IEEE, February 2010
3. Browne, S., Dongarra, J., Garner, N., London, K., Mucci, P.: A scalable cross-platform infrastructure for application performance tuning using hardware counters. In: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing, SC 2000, IEEE Computer Society, Washington (2000)

[2] Available from https://github.com/NicolasDenoyelle/dynamic_lstopo.

4. Geimer, M., Wolf, F., Wylie, B.J.N., Ábrahám, E., Becker, D., Mohr, B.: The SCALASCA performance toolset architecture. In: Proceedings of the International Workshop on Scalable Tools for High-End Computing (STHEC), Kos, Greece, pp. 51–65, June 2008
5. Gimenez, A., Gamblin, T., Rountree, B., Bhatele, A., Jusufi, I., Bremer, P.T., Hamann, B.: Dissecting on-node memory access performance: a semantic approach. In: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing, pp. 166–176. IEEE Press, New Orleans, LA, November 2014
6. Hursey, J., Squyres, J.M., Dontje, T.: Locality-aware parallel process mapping for multi-core HPC systems. In: 2011 IEEE International Conference on Cluster Computing (CLUSTER), pp. 527–531. IEEE (2011)
7. Jeannot, E., Mercier, G., Tessier, F.: Process placement in multicore clusters: algorithmic issues and practical techniques. IEEE Trans. Parallel Distrib. Syst. **25**(4), 993–1002 (2014)
8. Knüpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., Mickler, H., Müller, M.S., Nagel, W.E.: The vampir performance analysis tool-set. In: Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing, July 2008, HLRS, Stuttgart, pp. 139–155 (2008)
9. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2005, pp. 190–200. ACM, New York (2005)
10. Majo, Z., Gross, T.R.: Memory management in NUMA multicore systems: trapped between cache contention and interconnect overhead. SIGPLAN Not. **46**(11), 11–20 (2011)
11. Pillet, V., Labarta, J., Cortes, T., Girona, S.: PARAVER: a tool to visualize and analyze parallel code. In: Nixon, P. (ed.) Proceedings of WoTUG-18: Transputer and Occam Developments, pp. 17–31, March 1995
12. Rohou, E.: Tiptop: Hardware Performance Counters for the Masses. Research Report RR-7789, November 2011
13. Treibig, J., Hager, G., Wellein, G.: Likwid: a lightweight performance-oriented tool suite for x86 multicore environments. In: Lee, W.C., Yuan, X. (eds.) ICPP Workshops, pp. 207–216. IEEE Computer Society (2010)
14. Zhuravlev, S., Blagodurov, S., Fedorova, A.: Addressing shared resource contention in multicore processors via scheduling. SIGPLAN Not. **45**(3), 129–142 (2010)