MAP-*numa*: Access Patterns Used to Characterize the NUMA Memory Access Optimization Techniques and Algorithms

Qiuming Luo[1,2], Chenjian Liu[2], Chang Kong[2], and Ye Cai[1,2,3,*]

[1] National High Performance Computing Center (NHPCC), Shenzhen, China
[2] College of Computer Science and Software Engineering, Shenzhen University, China
[3] State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
`wiselcj@126.com,clarkong89@gmail.com,`
`{lqm@szu,caiye}@szu.edu.cn`

Abstract. Some typical memory access patterns are provided and programmed in C, which can be used as benchmark to characterize the various techniques and algorithms aim to improve the performance of NUMA memory access. These access patterns, called MAP-*numa* (Memory Access Patterns for NUMA), currently include three classes, whose working data sets are corresponding to 1-dimension array, 2-dimension matrix and 3-dimension cube. It is dedicated for NUMA memory access optimization other than measuring the memory bandwidth and latency. MAP-*numa* is an alternative to those exist benchmarks such as STREAM, pChase, etc. It is used to verify the optimizations' (made automatically/manually to source code/executive binary) capacities by investigating what locality leakage can be remedied. Some experiment results are shown, which give an example of using MAP-*numa* to evaluate some optimizations based on Oprofile sampling.

Keywords: Memory Access Patterns, NUMA, Benchmark, Oprofile.

1 Introduction

NUMA has been the source of performance problems for high performance computing on large scale distributed shared memory platforms for years. In these systems a processer core can access nearby memory faster than remote memory. To maximize the aggregate memory bandwidth all processes must simultaneously access data from their own local memory location. It means that multithreaded codes in NUMA platform should sustain sufficient locality of memory access and minimize access to remote data to obtain a high performance.

The importance of the data locality is well documented [1][2][3][4] and there are some OS-provided NUMA APIs to control it [5][6][7][8]. Linux traditionally had ways to bind threads to specific CPUs/Cores and NUMA API extends that to

[*]Corresponding author.

allow programs to specify on which node memory should be allocated. Some more complicated APIs are based on these basic policies, such as MAi [7] and MaMI [9].It is not an easy task to apply these API because it is much difficult to find the communication pattern in shared memory platform than message passing platform, because it is implicit and occurs through the memory accesses. Recently, some tools are available to guide a program developer on where to judiciously apply these API within a large parallel code [10][11][12]. But it is still a hard problem to find the best mapping of the access patterns, which is considered NP-Hard [13]. [12] used a heuristic algorithm to map threads and data on the machine based on the Edmonds matching algorithm [14]. [18] presents a strategy which used the Dual Recursive Bipartition algorithm for process placement to reduce communication time of parallel applications that have a steady communication pattern on clusters of multi-core machines. [19] introduced the Locality-Aware Mapping Algorithm (LAMA) for distributing individual processes of a parallel application across the processing resources in an HPC system paying particular attention to on-node hardware topologies and memory locality.

But all those works are using the traditional test tools or benchmarks to evaluate their efforts. To our best knowledge, there are no benchmark dedicate to validate the various optimizations for memory access on NUMA platform. How to qualify and characterize an optimization technology or algorithm is still need further study. One benchmark that can tell how the memory locality leakage can be remedied by an optimization would be the answer. And this benchmark should be an abstraction of typical applications which is architecture independent. This is the motivation of MAP-numa (Memory Access Patterns for NUMA, short for MAP).

The rest of the paper is organized as follows. In section 2, we discuss related work. Section 3 focuses on MAP and the memory access patterns. In section 4 we present an application example of MAP in a NUMA optimization based on memory traces (obtained via Oprofile). Finally in section 5 the conclusion is drawn and some future work is discussed.

2 Related Work

There exist plenty of researches that focus on NUMA memory access optimization. They can be classified into three categories if the verification method is considered. Some of them use well-known benchmarks for high performance computing, some use a particular application, and some other use memory benchmarks.

Memphis [11] evaluated its effectiveness by applying the NPB (NAS Parallel Benchmarks), HYCOM (a production ocean modeling application), XGC1 (a production Fortran90 particle-in-cell code that models several aspects of plasmas in a tokamak thermonuclear fusion reactor) and CAM (the Community Atmosphere Model). MAi [7] used two kernels (FFT and CG) from NPB and ICTM [15]. SPLASH2, PARSEC and Advention (a part of the Brazilian Regional Atmosphere Modeling System) were used in [13]. NPB is also used in [12][16][17]. They measured the runtime before and after the optimization with the well known scientific benchmarks to demonstrate their effectiveness.

The second category use particular applications other than using benchmarks. Gaussian computational chemistry code is used in [10]. H.264 video encoding code is used in [17]. Essentially, they are similar to the first category.

The last category try to setup some memory testing code customized to NUMA characteristic. ForestGOMP [9] made some modification to STREAM benchmark. STREAM measures sustainable memory bandwidth and the corresponding computation rate for simple vectors. The first modification is called nested-STREAM, which creates the threads by two steps. The threads in outer parallelism create a team of threads in inner parallelism. Each team works on its own set of STREAM vectors. The other modification is called twisted-STREAM. It contains two distinct phases. The first one behaves exactly as nested-STREAM. During the second phase, each team works on a different data set instead the one it was given in the first phase. ForestGOMP also considered the irregular applications with imbalanced parallelism and derived a modified version call imbalanced-STREAM.

Using the well known high performance computing benchmarks can demonstrate the effectiveness of various optimizations to sustaining the memory locality on NUMA. And so do those particular applications. But it cannot give us more details about how and what the optimization really contributes, except the reduction of running time. Nested-STREAM, twisted-STREAM and imblanced-STREAM reveal some more details about how and what ForestGOMP help to improve the memory performance in these three different circumstances. Keeping that idea in mind, we are trying to figure out one set of code that can help to understand characteristics of various optimization techniques and algorithms.

3 MAP-*numa*

MAP is designed under some guide lines or objectives. First of all, it should represent the typical memory access patterns used in today's high performance computing, or it will be useless. Secondly, it should cover a wide range of applications. The third objective is to be capable of revealing different kinds of potential memory locality leakages, and to be able to characterize an optimization in various aspects. The forth objective is to achieve platform independence. And the last one is to limit the use of cache, or the NUMA effect will be conceal under the enormous cache hits.

3.1 Data Set and Thread Affinity

Most computing programs used 1D array, 2D matrix (2D array) or 3D cubic (3D array) as their work data set. MAP uses those types of data sets too. But what is more important is the relationship between the computing threads and the data subset they access.

The first type of data set used by MAP is 1D array and the access patterns include shared, divided, interleaved and partial shared. Fig.1-(a) stands for the shared case, where all the threads share the entire array equally. The access sequence can be serialized of randomized. Fig.1-(b) stands for the divided case, where each thread accesses the dedicated memory zone separately. Fig.1-(c) stands for the interleaved case, where each thread accesses the entire memory area in an interleaved pattern. Fig.1-(d) stands for the case between shared and divided, each thread accesses its dedicated memory and shares a portion of it with its neighbor. These access patterns should have cover most applications that using 1D array as their working data set.

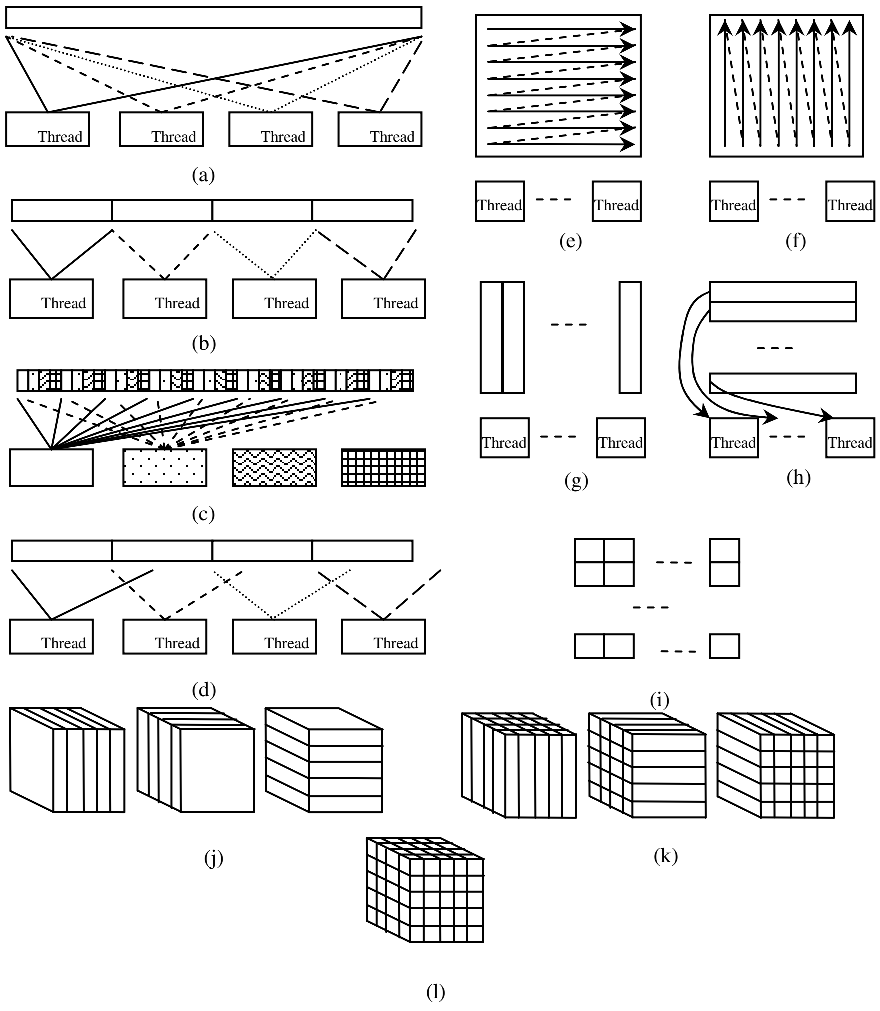


Fig. 1. Data sets in MAP

For 2D matrix cases, the data set can be accessed in more patterns. If it is shared, the access sequence may be scanning in horizontally or vertically, or randomly. As for divided cases, it can be divided in horizontal, in vertical or in 2D mesh. All the patterns are drawn in Fig. 1-(e)~(i). The 3D array in MAP is accessed in three fashions shown as Fig. 1-(j)~(l).

The STREAM benchmark uses four types of operations (COPY: a(i) = b(i), SCALE : a(i) = q*b(i) SCALE : a(i) = q*b(i), SUM : a(i) = b(i) + c(i) and TRIAD :

a(i) = b(i) + q*c(i)).While in MAP, the emphasis is NUMA memory access and the arithmetic operation should be eliminated. The basic operations (read, write and read/write) are chosen to perform without any arithmetic operations.

In order to achieve the objective of architecture independence, no architecture information is adopted in MAP. When programming MAP in C, it is completely based on the concept of one shared memory platform, and let the OS and compiler to handle the thread binding and data allocating. In order to eliminate the cache influence, each data element of the data set accessed by each thread is as big as one cache line. Each element in the array is padded by some blank field to make it long enough to occupy one cache line. Currently, MAP has two versions, which are programmed using pthreads library API and OpenMP directives respectively. The later one, OpenMP directive code, would be compiled by GCC with pthreads library or other thread library.

4 Application Example

An application example is provided in this section to demonstrate the value of MAP. Some optimizations are applied to MAP and the physical patterns are captured for evaluation.

4.1 The NUMA Hardware Platform

An HP 32-core NUMA platform is used as the test bed. Each 4 cores consists one memory node. 8 NUMA nodes are connected by HT. There are three different NUMA factors (relative latency values), 10 stands for native one, 16 and 22 stand for remove latency values.

4.2 The Tuning Steps Based on Oprofile

In this example, the optimization adopts the memory trace scheme similar to [10][13]. By analyzing the memory trace, physical patterns (contrast to the logical access patterns) can be drawn and represented in memory access matrix or communication matrix [16].

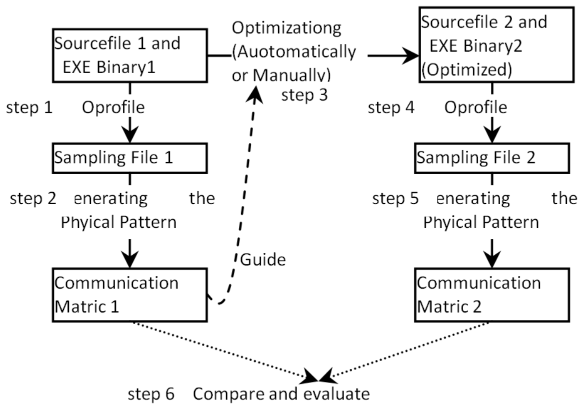

Sourcefile 1 and EXE Binary1
Optimizationg (Auotomatically or Manually)
Sourcefile 2 and EXE Binary2 (Optimized)
step 1 Oprofile
step 3
step 4 Oprofile
Sampling File 1
Sampling File 2
step 2 enerating the Phyical Pattern
step 5 enerating the Phyical Pattern
Guide
Communication Matric 1
Communication Matric 2
step 6 Compare and evaluate

Fig. 2. The optimization steps

On AMD Opteron, with the Instruction-Based-Sampling (IBS)[20], it can provide the following information for sampled instructions that *load* data or *store* data.

- The precise program counter of the instruction.
- The virtual address of the data referenced by the instruction.
- The physical address of the data referenced by the instruction.

Only the samples with correct TID are kept and analyzed. Then they are classified into load and write categories. They are further labeled by node number according their physical address. The communication matrix is generated by accumulating the item with same source node and target node. The whole process is shown by Fig.2.

4.3 The Experiment Results

Three optimization methods, Linux NUMA API functions, *numactl* command-line tools and rewriting the source code manually, are applied to MAP's 1D data set to

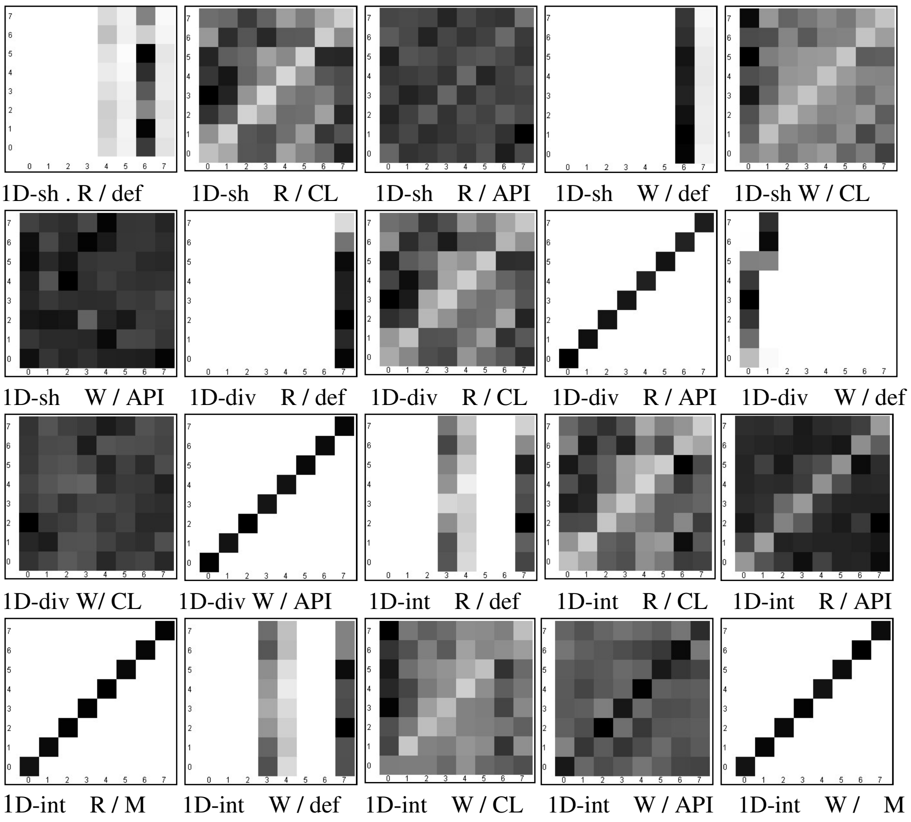
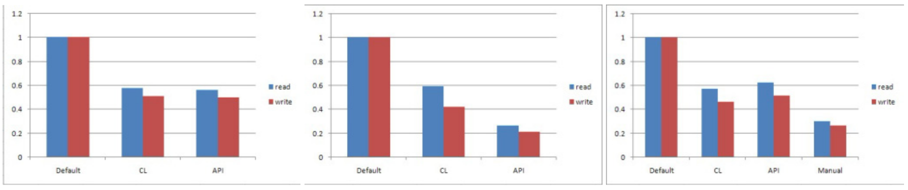


Fig. 3. Communication matrices of MAP-*numa* before and after the optimizations (1D stands for 1D data set, sh/div/int stand for shared/divided/interleave, R/W stands for read or write, def/CL/API/M stand for default/command line tools/NUMA API functions/Manually rewrite)

make comparison. The following communication matrices (vertical axis represents the nodes ID that issue the accesses and the horizontal axis represents the accessed nodes ID, the darker means more accesses) are obtained by Oprofile sampling.

For 1D shared pattern (Fig.1-a), OS allocates the data without particular policy and with influence from other processes' memory allocation (Fig. 3, 1D-sh R/def and 1D-sh W/def). Because the allocation tends to take place in one node, the communication matrix is focus on one (or a few) node. While applying numactl command-line tools (with --interleave option) to binary executable file or NUMA API functions into source file, the matrix become more flattened(Fig3, 1D-sh R/CL, 1D-sh W/CL, 1D-sh R/API and 1D-sh W/API). For 1D divided pattern (Fig.1-b), the default matrix (Fig. 3, 1D-div R/def and 1D-div W/def) still focus on one (or a few) node. Using numactl command-line tools (with --interleave option) can make the matrix more flattened (Fig.3 1D-div R/CL and 1D-div R/CL). By adding NUMA API functions into source code, the matrix can be diagonalized (Fig.3 1D-div R/API and 1D-div R/API), which means no memory locality leakage. For 1D interleaved pattern (Fig.1-c), default allocation remains. But this time, API functions have the same result as *numactl* command-line tools. They both flatten the matrix. If manually modify the code and applying NUMA API functions, a diagonal matrix can be obtained (Fig.3 1D-int R/M and 1D-int W/M). Fig. 4 give the execution time of all the cases mentioned above, which make the effectiveness of these optimization obvious.



(a) 1D-shared (b) 1D-divied (c) 1D-interleaved

Fig. 4. Execution time of default/CL/API/M methods on MAP's 1D data set

5 Discussion and Conclusion

From the experiment results, we can distinguish the abilities of different optimization methods. The OS tends to allocate memory on one node and then move to another node, which will give rise to the memory contention and become a bandwidth bottleneck. The *numactl* command-line tools can deal with 1D-shared and 1D-divided (not perfectly), but fail to handle 1D-interleaved and multi-patterns cases. NUMA API can handle 1D-shared, 1D-divided and multi-patterns. If source code can be rewritten manually, all the patterns can be handled nicely.

The change of the matrices (physical patterns) reveals the remedy of locality leakage. If a matrix with large number focus on one or a few columns was converted to a more flat matrix, which means the potential memory bandwidth bottleneck is removed. If a matrix with large number focus on one or a few rows was converted to a more flat matrix, which means the threads are distribute to more nodes and improving the parallelism. If an evenly distributed matrix is converted to a diagonalized matrix, it means the best placement is achieved where there is no remote memory access.

So, we used table 1 to summarize their ability. It is hard to compare two optimizations, if they are only applied to one particular application. Instead, MAP consists of various typical access patterns and is able to verify whether (and what) locality leakages can be remedied by an optimization method. Actually, Table 1 can have more details if all the patterns are applied.

Table 1. Abilities of *nuamctl*, NUMA API and rewrite manually methods

	default	*numactl*	NUMA API	rewrite manually
1D-shared	×	√	√	√
1D-divided	×	√	√	√
1D-interleaved	×	×	×	√
multi-patterns	×	×	√	√

MAP is applicable to the source code methods as well as the methods modify binary executable file. The optimizations made by manually coding, or with the help of compiler, don't make any difference when being evaluated. So MAP is good start point of a general purpose benchmark to evaluate all kinds of optimizations.

As future work, we intend to profile some real world parallel applications to get the memory access patterns ,which are contained by our MAP access patterns. Further more, we can use the proper way described in Table 1 to optimize the applications.

Acknowledgements. This work was supported by the project (NO. 2011A090100037) funded by Guangdong Province and Chinese Academy of Sciences, and the project funded by State Key Laboratory of Computer Architecture, ICT,CAS.

References

1. Zhang, X., Qin, X.: Performance Prediction and Evaluation of Parallel Processing on a NUMA Multiprocessor. IEEE Trans. Software Eng. 17(10), 1059–1068 (1991)
2. LaRowe Jr., R.P., Ellis, C.S., Holliday, M.A.: Evaluation of NUMA Memory Management Through Modeling and Measurements. IEEE Transactions on Parallel and Distributed Systems, 686–701 (1992)
3. Brecht, T.B.: On the importance of parallel application placement in NUMA multiprocessors. In: Proc. of SEDMS IV, Symposium on Experiences with Distributed and Multiprocessor Systems, pp. 1–18. USENIX Association (1993)
4. Holliday, M.A., Stumm, M.: Performance Evaluation of Hierarchical Ring-Based Shared Memory Multiprocessors. IEEE Trans. Computers 43(1), 52–67 (1994)
5. Drepper, U.: What every programmer should know about memory (2007), `http://people.redhat.com/drepper/cpumemory.pdf`
6. Kleen, A.: A NUMA API for linux. Technical report, Novell Inc., Suse Linux Products GmbH (2005)

7. Ribeiro, C.P., Méhaut, J.-F., Carissimi, A., Fernandes, L.G.: Memory Affinity for Hierachical Shared Memory Multiprocessors. In: 21st International Symposium on Computer Architecture and High Performance Computing, pp. 59–66 (2009)
8. Lameter, C.: Local and remote memory: Memory in a Linux/NUMA system (2006), `ftp://ftp.tlk-l.net/pub/linux/kernel/people/christoph/pmig/numamemory.pdf`
9. Broquedis, F., Furmento, N., Goglin, B., Wacrenier, P., Namyst, R.: ForestGOMP: An Efficient OpenMP Environment for NUMA Architectures. International Journal of Parallel Programming (Spring 2010)
10. Yang, R., Antony, J., Rendell, A., Robson, D., Strazdins, P.: Profiling Directed NUMA Optimization on Linux System: A Case Study of the Gaussian Computational Chemistry Code. In: 2011 IEEE International Parallel&Distributed Processing Symposium, pp. 1046–1057 (2011)
11. McCurdy, C., Vetter, J.: Memphis: Finding and Fixing numa-related performance problems on Multi-core platforms. In: Proceedings of ISPASS, pp. 87–96 (2010)
12. Cruz, E., Pousa, C., Alves, M., Carissimi, A., Navaux, P., Mehaut, J.-F.: Using Memory Access Traces to Map Threads and Data on Hierarchical Multi-core Platforms. In: 2011 IEEE International Parallel & Distributed Processing Symposium, pp. 551–558 (2011)
13. Diener, M., Madruga, F., Rodrigues, E., Alves, M., Schneider, J., Navaux, P., Heiss, H.U.: Evaluating thread placement based on memory access patterns for multi-core processors. In: 2010 12th IEEE International Conference on High Performance Computing and Communications, pp. 491–496 (2010)
14. Osiakwan, C., Akl, S.: The maximum weight perfect matching problem for complete weighted graphs is in pc. In: Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing, pp. 880–887 (1990)
15. Castro, M., Fernandes, L.G., Ribeiro, C.P., Méhaut, J.-F., de Aguiar, M.S.: NUMA-ICTM: A Parallel Version of ICTM Exploiting Memory Placement Strategies for NUMA Machines. In: PDSEC 2009: Parallel and Distributed Processing Symposium, International, pp. 1–8 (2009)
16. Cruz, E., Alves, M., Carissimi, A., Navaux, P., Pousa, C., Méhaut, J.-F.: Memory-aware Thread and Data Mapping for Hierarchical Multi-core Platforms. International Journal of Networking and Computing, 97–116 (2012)
17. Tudor, M., Teo, Y., See, S.: Understanding Off-Chip Memory Contention of Parallel Programs in Multicore Systems. In: 2011 International Conference on Parallel Processing, pp. 602–611 (2011)
18. Rodrigues, E.R., Madruga, F.L., Navaux, P.O.A., Panetta, J.: Multi-core aware process mapping and its impact on communication overhead of parallel applications. In: ISCC, pp. 811–817 (2009)
19. 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, pp. 527–531 (2011)
20. Drongowski, P.J.: Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors. Advanced Micro Devices, Inc. (2007)