SnuMAP: an Open-Source Trace Profiler for Manycore Systems

Camilo A. Celis Guzman, Younghyun Cho, and Bernhard Egger

Computer Systems and Platforms Laboratory,
Department of Computer Science and Engineering
Seoul National University, Seoul, South Korea
{camilo, younghyun, bernhard }@csap.snu.ac.kr

Abstract. In this paper, we introduce SnuMAP, an open-source trace profiler for multi/many-core systems. The proposed profiler is light-weight, requires no source-code instrumentation and does not degrade performance of the target parallel application, but instead it provides useful information and insights for application developers and multi/many-core resource managers. SnuMAP collects the execution trace of every thread from a multi-threaded application inside the Linux kernel and queries the trace information to visualize it in user-space. Since the trace information is collected in the Linux kernel, SnuMAP can provide insights for multi/many-core resource management. For example, the resource manager can understand how parallel applications are executed and behave on a platform, especially when other workloads are executed simultaneously. This feature is gaining importance as today's multi/many-core systems co-schedule multiple parallel workloads to increase system utilization. In this paper, we suggest several use cases of the profiler and present interesting results through SnuMAP on our two multi-core platforms, a 32-core AMD Opteron server, and a 36-core Tile-Gx36 processor. SnuMAP is an open source project, more information is available at http://csap.snu.ac.kr/software/snumap

Keywords: Trace visualization, profiler, multi/many-core systems, NUMA

1 Motivation

In recent years computer systems have undergone a trend shift from increasing CPU frequencies to multi/many-core CPUs. Modern data center and high-performance computing (HPC) centers operate distributed multi/many-core computer nodes. Because of this architectural shift, software developers have been forced to write parallel programs that can take advantage of multiple cores efficiently. However, writing efficient parallel programs requires a thorough understanding of the underlying architecture and the hardware resources, as these can heavily affect the performance of a parallel application.

This difficulty motivated the creation of novel tools that help developers understand exactly how a program executes on a target architecture, and also

assist the development by pointing out performance and bottlenecks in the system [24]. Trace visualization—as the core vs. time graph—is one of the easiest and most direct ways to understand the behavior of a (parallel) program. It allows programmers to have a clear and general picture of the execution of an application, and the interaction with different resources in a system. A variety of trace visualization and profiler tools have been introduced and are widely used by parallel application developers especially in HPC [2, 3, 22, 25].

Recently, however, parallel programs are rarely executed in isolation on modern multi/many-core platforms. For example, Mesos [13], a data-center OS, runs several big-data application frameworks across distributed computers for flexible execution with varying resource requirements. In addition, general-purpose servers and HPC centers often execute several parallel workloads [7] simultaneously to increase system utilization and energy efficiency.

In such an environment, understanding how parallel applications are executed with external workloads on a given multi/many-core platform is gaining attention for both the application designers and the multi/many-core resource managers. Yet, prevalent trace profilers [5, 12, 8, 24, 19] are not adequate to provide such information because most of them assume standalone executions and do not collect compelling scheduling information.

In this paper, we introduce SnuMAP: SNU Multi/Many-core Applications Profiler, an open-source trace profiler framework for multi/many-core systems. SnuMAP is light-weight, requires no source-code instrumentation and does not incur any performance degradation of the target (parallel) application. The execution trace of every thread in a (multi-threaded) parallel applications is collected inside the Linux kernel, using timing information at every context switch for all the threads of the target application. Since the scheduling information is collected by the Linux scheduler, SnuMAP is able to capture performance interference and scheduling conflicts between different (parallel) workloads and provide insights for application developers and platform resource managers. This feature is gaining relevance as today's multi-core systems co-schedule multiple parallel program instances to increase system utilization. Additionally, SnuMAP visualizes NUMA-related performance information such as cache miss rate or local/remote memory access which are amongst the most important performance features in modern multi/many-core architectures. The proposed tool is not limited by the programming language as we capture the creation and termination of all threads during an application's lifetime.

This paper is structured as follows. In Section 2, we introduce related profiler tools for SnuMAP. Section 3 describes the software structure of SnuMAP, and Section 4 explains additional components in the SnuMAP framework. In Section 5, we suggest several use cases and present interesting case studies. We conclude this paper in Section 6.

2 Related Work

Trace visualization and profilers are common tools for performance analysis and debugging of high-performance systems. In this section we will explore some of these tools, observe what their benefits are, and compare them to our contribution.

During the years many tracing tools have been proposed to facilitate the detection of performance problems and bottlenecks. Two of the most well know are Dtrace [10] and KProbes [15]. The Linux kernel itself comes with useful but limited set of tracers such as SystemTap [23], and Ftrace [11] which are usually used for debugging and analysis via script-like queries to define specific event(s) tracing. Three of the most recent profilers for Linux focus on NUMA architectures, performance analysis, and visualization of system invariants. Memprof [16] is a profiler that targets NUMA architectures, and analyzes the interaction between threads and objects in an executing program. The collection of memory objects uses a similar approach as SnuMAP; via library interpositioning to intercept calls to the dynamic memory allocator. Similarly, Memprof also uses a kernel module (as SnuMAP and other approaches do [9] to communicate with the Linux kernel) for tracking the life cycle of code sections and global static variables. Moreover, instead of relying on performance counters to measure memory accesses, it uses Instruction Base Sampling (IBS) [14] to periodically select a random instruction and record its performance information.

Aftermath [2] is an open-source graphical tool for performance analysis and debugging. It allows the joint visualization of task duration, hardware counters and data exchanges. As opposed to Memprof, and using a similar approach as SnuMAP, it utilizes performance counters directly from the target architecture via a software interface, PAPI [1]. Lastly, HPCToolKit is a framework for analyzing the performance and scalability bottlenecks of parallel programs. It enables powerful profiling without code instrumentation; similar to SnuMAP. It also allows the discovery and quantification of performance bottlenecks via binary analysis, top-down performance analysis and scalability of measurements. Their approach for data collection is fundamentally different that the one from SnuMAP, using statistical sampling, event triggers and stack unwinding.

Most of the surveyed tools focus on the performance analysis of single isolated programs. Multiple co-located workloads are no considered, and the interaction between the target application, runtimes, and operating systems is ignored. It is often the case that performance bottlenecks raise due to complex and tangled types of interactions between resource managers and running applications. That is why one of SnuMAP key features enables the visualization of multiple executing programs interacting with the underlying resource managers and architecture. A similar approach was taken by Lozi *et. al* [17] when exploring different types of performance bugs in the Linux scheduler. For this, they used generalized data collection techniques to gather information from the kernel without targeting a specific application.

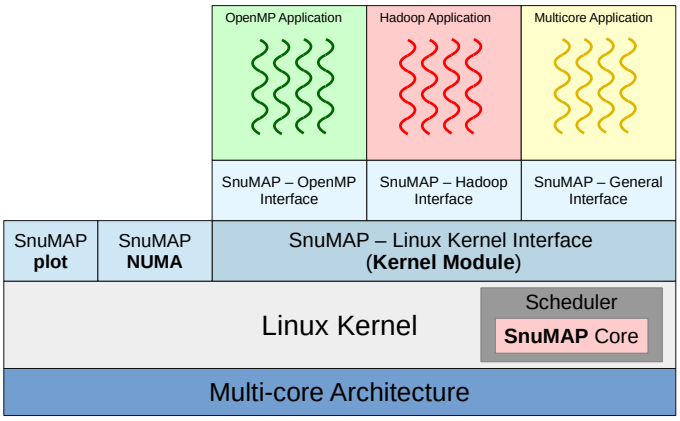


Fig. 1. SnuMAP architecture overview.

3 SnuMAP Core

In this section we present the architecture of SnuMAP and its implementation.

3.1 Overview

SnuMAP is is composed of three layered components (see Figure 1):

- SnuMAP enabled Linux kernel.
- SnuMAP kernel module to communicate the kernel for specific actions.
- SnuMAP library interface for binaries to interact with the kernel module.

We decided to free the programmer from code instrumentation when using SnuMAP by intercepting calls to `main` and `exit` routines via library interpositioning[1] in order to start, stop, and dump the results of profiling. Similarly, we used library interpositioning for programs using the OpenMP [20] programming model in order to clearly distinguish between different parallel and sequential sections. Additionally, we included a kernel module that is the channel between the SnuMAP library interface and the kernel to communicate what actions to take. In Figure 1 we describe the overall architecture of SnuMAP.

SnuMAP supports two different scenarios. The first one does not require any code instrumentation; as mentioned earlier it uses dynamic library interpositioning. This scenario is called **dynamic**, and it is executed using the main SnuMAP executable (`snumap-main <target binary>`). As soon as the target program starts, SnuMAP begins collecting data for any new thread the program creates, and after it finishes SnuMAP will move the profile data into user space. Note that SnuMAP also supports profiling multiple concurrent (parallel) programs.

[1] Note that this only applies to languages like C and C++, other languages like Java need special modifications, see section 3.2.

Meaning that SnuMAP can be executed along each application separately, generating separate trace files. Because we use a shared global variable that stores the number of ticks as a measurement of time, we can later plot the combined executions that happened concurrently, using `snumap-plot` (see section 4 for further details). Similarly, if SnuMAP detects that the target application uses the OpenMP programming model it will capture calls to every (independent) parallel section and generate an extra log file with this time-stamped information. Then, using this extra trace file we are able to plot a similar graph generated by `snumap-plot` with the addition of clearly delimiting different parallel sections with different colors.

The second scenario SnuMAP supports is for applications that might require explicit calls to the SnuMAP library interface to communicate with the kernel module when to start or stop profiling. This scenario is called **static** and is useful for profiling applications that do not support library interpositioning. For example, we have tested SnuMAP to profile Hadoop map-reduce jobs, and we used the Java Native Interface (JNI) to interact with the SnuMAP interface library; see section 3.2 for further details.

3.2 Implementation

In this section we describe to some level of detail the different components that make up the core of SnuMAP.

SnuMAP-enabled Linux Kernel In order to create high-definition plots, we opted to allow small changes to the Linux kernel to support the collection of data. The principal source of information for SnuMAP relies on the elapsed time at each context switch. In the Linux kernel each core has a list of runnable tasks. When a task performs a context-switch, SnuMAP stores the clock tick[2] at which a context-switch occured. We define two different events for a task in the system: (a) paused, or (b) running. A program thread which is *paused* refers to a thread that has been context-switched by another thread in the same core. Similarly, a running task relates to a thread that is currently running on the system. Note that we store this information for every core using a link list which has a hard limit on its size to avoid overflowing main memory. This modifications are minor and mainly involves the scheduler methods that handle the creation and destruction of a task.

Kernel Module The kernel module is the glue that interacts between the Linux kernel and the SnuMAP interface via `IOCTL` calls and directly accesses to kernel data structures (`task_struct`). It defines multiple methods to start, stop profiling and also dump the profile data. We use a simple flag per `task_struct`

[2] Since our target architecture are many/multi-cores systems, we use `jiffies` instead of the core's own clock tick as it allows us to have a global synchronized view of time.

to determine if this process should be profiled or not. Thus, all the initialization of data structures used for profiling, and communication mailboxes is done here.

SnuMAP Interface This API allows executables written in C/C++ and other programming languages (like Java) to communicate with the SnuMAP profiler. As mentioned before, in a *dynamic* scenario SnuMAP activation happens without the need of including explicit calls to this interface; rather it is done implicitly via library interpositioning. Nevertheless, this interface is necessary to communicate with the kernel, via `IOCTL` calls, to trigger two main predefined methods: start profiling for the current target application, or stop and dump profiling results.

Special Case: JNI It is not entirely/easily possible to *dynamically* profile executables from different programming languages other than C/C++. Since we can not directly call the SnuMAP library interface from a Java program, we have to use something else. The solution is the Java Native Interface; which allows Java programs to execute native C code via a dynamic linker. However, it requires a very particular and error prone format. In our open source repository, we include multiple examples that show how to profile Hadoop map-reduce jobs, like `wordCount`, using a collaboration between JNI and the SnuMAP interface.

Without going into too many details, we create a shared library with specific function names so that the Java dynamic linker can find the SnuMAP interface methods, and call them from a Java program. Due to this unusual format, we could not create a general solution for Java applications. Instead extra modifications to match a target Java application hierarchy (package name, and class name) in the shared library function names are required[3]. Additionally, it is also required to load the shared library statically, and explicitly call the SnuMAP interface methods from within the target Java program to enable profiling.

4 Additional Components

Besides the SnuMAP profiler whose main job is data collection, our framework also includes tools to plot the results of the profiler using a core vs. time graph, and a NUMA-architecture-specific version of SnuMAP that not only collects time data, but also gathers NUMA-related performance counters to create an additional plot of last-level cache (LLC) accesses (see Figure 1 SnuMAP *plot* and *NUMA* components).

4.1 SnuMAP-NUMA

As the number of cores increases in multi-core platforms, insufficient memory bandwidth has become an increasingly important issue. Modern multi/many-core architectures comprise several memory nodes (NUMA) and distribute memory accesses to the memory nodes. Nevertheless, in NUMA architectures, parallel

[3] Note that modifying only the method name is sufficient.

application performance is highly affected by how the NUMA features are used. It is often the case that if an application incurs multiple remote accesses to non-local memory this will highly affect performance.

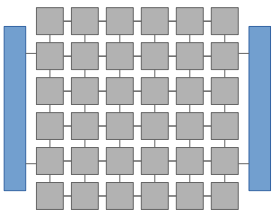
The SnuMAP framework has an online NUMA performance analyzer that collects NUMA-related performance information such as the cache miss rate or the remote/local memory access rate that occurred in a core/thread. To obtain such information SnuMAP queries hardware performance counters via the Linux `perf`. The collected information is time stamped using Linux `jiffies` and synchronized with the trace information (time vs. core graph) when we plot the NUMA performance information via `snumap-plot-numa`.

4.2 SnuMAP-plot

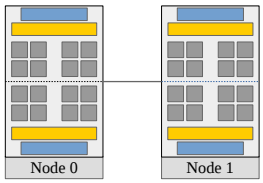
SnuMAP's `plot` is an additional tool that helps with the visualization of the tracing information after profiling a target application. It is based on Gantt charts and it shows, on the y-axis, every core of the target architecture, and, on the x-axis, the time (increment in `jiffies`). With this setup, SnuMAP can plot the elapsed time an application's thread executed on each core (core vs. time plot). Also, SnuMAP is able to generate two extra graphs for specific scenarios. In the case of NUMA architectures, SnuMAP plots the Local DRAM access ratio for each NUMA node. As for programs that use the OpenMP programming model, SnuMAP can produce a core vs. time plot that colors each parallel section differently. This makes it easier to diagnose the root cause of a possible performance bottleneck, as each colored parallel section is identified by its function's memory address (see Section 5 for an example).

5 Use Cases

5.1 Target Architectures



Tile-Gx8036



AMD Opteron 6276

Architecture	AMD32	Tile-Gx36
Processor	Opteron 6276	Tile-Gx8036
Clock frequency	1.4GHz	1.2GHz
Memory Size	32 GB	32 GB
Total Cores	32	36
# NUMA nodes	2	2
Linux Kernel	3.19	3.14

Target Architecture Descriptions

Fig. 2. Target Architectures

We performed the evaluation of multiple use cases using two difference systems, a 32-core AMD Opteron 6276 server platform [4] and the Tile-Gx36 platform [18]. The AMD Opteron server represents a NUMA system with multi-core

and multi-socket capabilities, and the Tile-Gx38 represent a mesh many-core processor. A detailed breakdown of the features of these two platforms is shown in Figure 2.

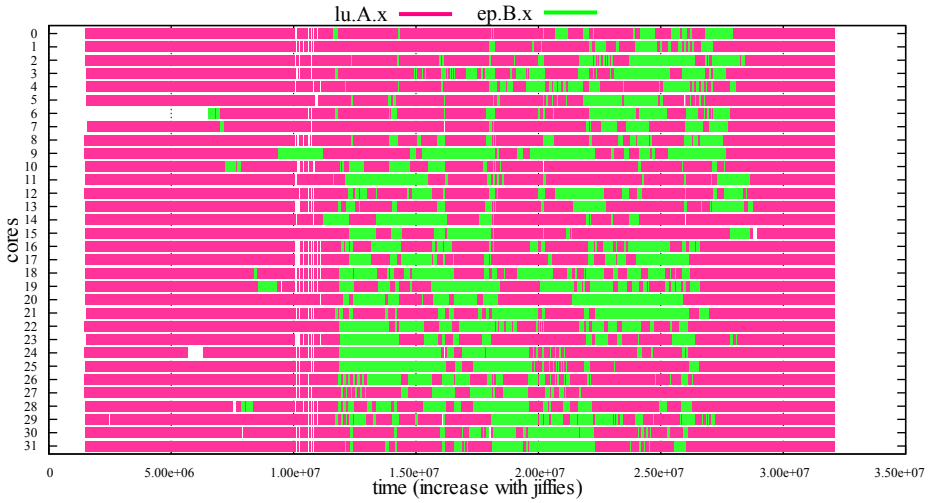


Fig. 3. SnuMAP plot for two-concurrent applications (LU class A, and EP class B) on the AMD 32 platform.

5.2 SNU-NPB3.3

We selected several OpenMP-C applications (BT, EP, LU, MG and CG) from the SNU-NPB3.3 [21] implementation of the NAS parallel benchmark [6] to showcase the capabilities of SnuMAP. In this scenario, BT, MG and LU represent memory intensive scalable applications, whereas EP is CPU intensive and CG issues irregular memory accesses. Additionally, each application computes over a specific problem size, also known as class size. We executed all possible permutations of pairs of existing applications on the SNU-NPB3.3 benchmark on both architectures; however, due to space limitation we present a subset of the profiling results.

Figure 3 shows the core vs. time plot generated by SnuMAP after profiling the concurrent execution of LU (class size A) and EP (class size B) on the AMD 32 architecture. By observing the graph, it is easy to note the interaction between these two applications in a many/multi-core system. The effects of the Linux scheduler and the load balancer are clearly visible. Additionally, note how the load is balanced among all the cores, and how the scheduler seems to favor LU at the beginning of the execution, to later start making room for EP. Similarly, it is clear how, for example, core number six is not given any *work* until after a short time period of time where it interleaves *work* from EP and LU. An interesting

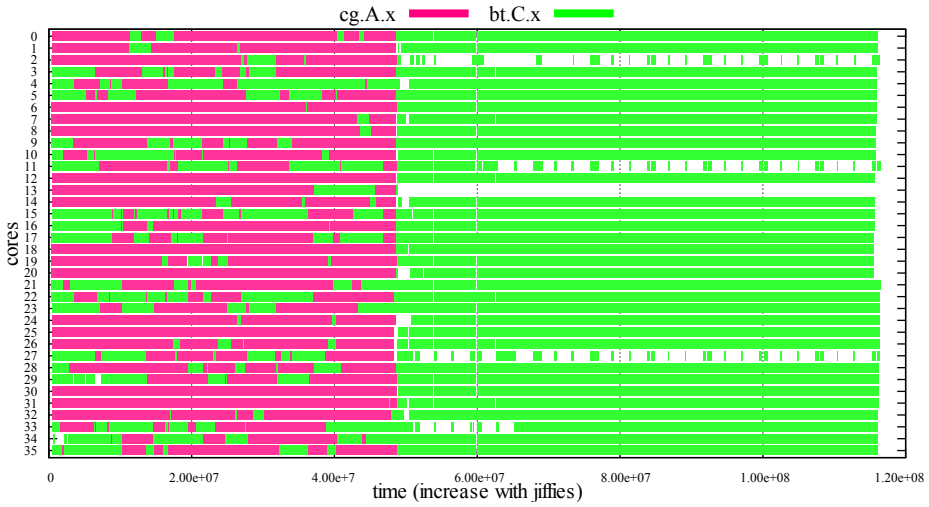


Fig. 4. SnuMAP plot for two concurrent applications (CG class A, and BT class C) on the Tile-Gx36 architecture.

observation is how useful the SnuMAP-plots are for visualizing synchronization bottlenecks. At the center-left of the graph we observe that LU seems to *stop* executing for short periods of time, this is due to the many synchronizations generated by many small parallel sections.

Figure 4 shows the plot generated by SnuMAP after profiling the concurrent execution of CG (class size A) and BT (class size C) on the Tile-Gx36 architecture. This is an interesting result because it shows how the architecture and the scheduler allocates cores to both applications. The overall allocation of CG interleaves the allocation of BT in sets of three cores. The result of such an allocation is an equal division of the processor in half. This might be the case so that each application is *closer* in hop distance to the closest memory bank (see Figure 2). Moreover, this plot clearly shows near the center of the plot, two empty vertical lines, which represent two synchronization points of the BT application.

Figure 5 shows the plot generated by SnuMAP after profiling the concurrent execution of EP (class size B) and LU (class size C) on the AMD 32 architecture. These are the same trace data used for Figure 3. However, in this case we isolated the profile data of the EP application, and by using the extra trace file generated for this OpenMP application, SnuMAP is able to plot individual parallel sections using different colors. Note how it now becomes clear which code sections execute at a given time. By looking at the results it seems that the parallel sections labeled `sequential` and `0x401260` are the longest running sections. Also, interaction between parallel sections is minimal as they appear to execute in batch (one after another).

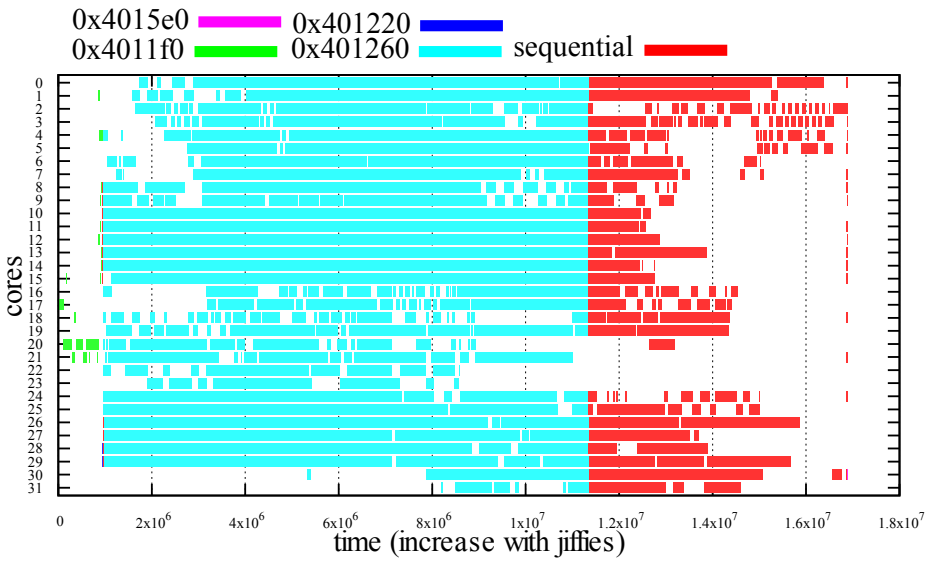


Fig. 5. SnuMAP plot distinguishing individual parallel sections of a single application (EP class B) in a concurrent applications scenario (EP class B, and LU class A) on the AMD 32 architecture.

Finally, Figure 6 illustrates the different memory access rate patterns during the concurrent execution of MG and EP (both class size C) on the AMD 32 architecture. In the plot generated by SnuMAP-NUMA shows the different access rates for both NUMA nodes while executing the concurrent workload. An important observation of this result is how one of the nodes appears to serve significantly more memory requests compared to the other node.

6 Conclusion

In this work, we have introduced SnuMAP, a light-weight trace profiler for (multiple) parallel applications. We have validated the functionalities of the profiler framework on two different Linux-based multi-core systems. The case study shows that the profiler can be efficiently used for many interesting scenarios. For example, application developers can see application behavior on a target multi-core platform, and the platform resource manager can understand what/which applications are efficiently or poorly co-executed on a same platform.

This project is open source and publicly available. Find more information at `http://csap.snu.ac.kr/software/snumap`. We expect this tool to be widely used and extended for various domains that use multi/many-core platforms.

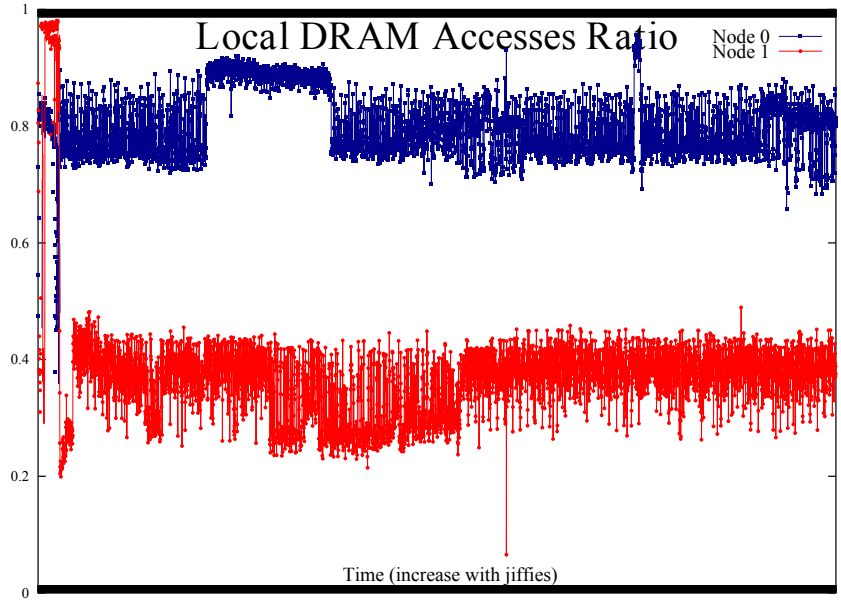


Fig. 6. SnuMAP-NUMA remote memory access rate plot for two concurrent (MG and EP) applications on the AMD 32 architecture.

Acknowledgments

We thank Heesik Shin for his effort and work on an early version of SnuMAP.

This work was supported in part by BK21 Plus for Pioneers in Innovative Computing (Dept. of Computer Science and Engineering, SNU) funded by the National Research Foundation (NRF) of Korea (Grant 21A20151113068), the Basic Science Research Program through NRF funded by the Ministry of Science, ICT & Future Planning (Grant NRF-2015K1A3A1A14021288), and by the Promising-Pioneering Researcher Program through Seoul National University in 2015. ICT at Seoul National University provided research facilities for this study.

References

[1] Performance application programming interface. `http://icl.cs.utk.edu/papi`, [onlile; accessed 15-Jun-2016]

[2] Aftermath : A graphical tool for performance analysis and debugging of fine-grained task-parallel programs and run-time systems. 7th workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG-2014) (1), 1–13 (2014)

[3] Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., Tallent, N.R.: HPCTOOLKIT: Tools for performance analysis of optimized parallel programs. Concurrency Computation Practice and Experience 22(6), 685–701 (2010)

[4] AMD: AMD Opteron 62000 Series Processors Reference Document. https://www.amd.com/Documents/Opteron_6000_QRG.pdf, [online; accessed 28-Jun-2016]
[5] B., M., AD., M., H-C., H., F., S., G., H., J., H., S., S.: A performance monitoring interface for openmp (2002)
[6] Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Dagum, L., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Schreiber, R.S., et al.: The NAS parallel benchmarks. International Journal of High Performance Computing Applications 5(3), 63–73 (1991)
[7] Breitbart, J., Weidendorfer, J., Trinitis, C.: Automatic co-scheduling based on main memory bandwidth usage. In: Proceedings of the 20th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP). JSSPP '16 (May 2016)
[8] Caubet, J., Gimenez, J., Labarta, J., Rose, L.D., Vetter, J.S.: A Dynamic Tracing Mechanism for Performance Analysis of OpenMP Applications, pp. 53–67. Springer Berlin Heidelberg, Berlin, Heidelberg (2001), http://dx.doi.org/10.1007/3-540-44587-0{_}6
[9] Choi, W., Kim, H., Song, W., Song, J., Kim, J.: epro-mp: A tool for profiling and optimizing energy and performance of mobile multiprocessor applications. Sci. Program. 17(4), 285–294 (Dec 2009), http://dx.doi.org/10.1155/2009/838745
[10] Dtrace, http://dtrace.org
[11] Ftrace, https://www.kernel.org/doc/Documentation/trace/ftrace.txt
[12] Graham, S.L., Kessler, P.B., Mckusick, M.K.: Gprof: A call graph execution profiler. SIGPLAN Not. 17(6), 120–126 (Jun 1982), http://doi.acm.org/10.1145/872726.806987
[13] Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A.D., Katz, R.H., Shenker, S., Stoica, I.: Mesos: A platform for fine-grained resource sharing in the data center. In: NSDI. vol. 11, pp. 22–22 (2011)
[14] J., D.P.: Instruction-based sampling: A new performance analysis technique for amd family processors (2007), [online: accessed 7-Aug-2016]
[15] KProbes, https://www.kernel.org/doc/Documentation/kprobes.txt
[16] Lachaize, R., Lepers, B., Quéma, V.: MemProf: A Memory Profiler for NUMA Multicore Systems. Proceedings of the 2012 USENIX Conference on Annual Technical Conference p. 5 (2012), http://dl.acm.org/citation.cfm?id=2342821.2342826
[17] Lozi, J.P., Lepers, B., Funston, J., Gaud, F., Quéma, V., Fedorova, A.: The Linux scheduler: a decade of wasted cores. Proceedings of the Eleventh European Conference on Computer Systems - EuroSys '16 pp. 1–16 (2016), http://dl.acm.org/citation.cfm?doid=2901318.2901326
[18] Mellanox: Tile-Gx Processor Family. http://www.mellanox.com/related-docs/prod_multi_core/PB_TILE-Gx36.pdf, [online; accessed 30-Jun-2016]
[19] Nagel, W.E., Arnold, A., Weber, M., Hoppe, H.C., Solchenbach, K.: VAMPIR: Visualization and Analysis of MPI Resources. Supercomputer 12, 69–80 (1996)
[20] OpenMP API for parallel programming, version 2.0, http://openmp.org
[21] Seo, S., Jo, G., Lee, J.: Performance characterization of the nas parallel benchmarks in opencl. In: Workload Characterization (IISWC), 2011 IEEE International Symposium on. pp. 137–148 (Nov 2011)
[22] Sigelman, B.H., Barroso, L.A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., Shanbhag, C.: Dapper, a large-scale distributed systems tracing infrastructure. Tech. rep., Google, Inc. (2010), http://research.google.com/archive/papers/dapper-2010-1.pdf
[23] SystemTap, https://sourceware.org/systemtab/

[24] Vetter, J.: Dynamic statistical profiling of communication activity in distributed applications. In: Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems. pp. 240–250. SIGMETRICS '02, ACM, New York, NY, USA (2002), http://doi.acm.org/10.1145/511334.511364

[25] Zhao, X., Zhang, Y., Lion, D., Ullah, M.F., Luo, Y., Yuan, D., Stumm, M.: lprof: A non-intrusive request flow profiler for distributed systems. In: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). pp. 629–644. USENIX Association, Broomfield, CO (Oct 2014), https://www.usenix.org/conference/osdi14/technical-sessions/presentation/zhao