An Approach to Visualize Remote Socket Traffic on the Intel Nehalem-EX

Christian Iwainsky[1], Thomas Reichstein[1], Christopher Dahnken[2], Dieter an Mey[1], Christian Terboven[1], Andrey Semin[2], and Christian Bischof[1]

[1] Center for Computing and Communication
RWTH Aachen University
{iwainsky,reichstein,anmey,terboven,bischof}@rz.rwth-aachen.de
[2] Intel GmbH
Dornacher Str. 1
85622 Feldkirchen bei München

Abstract. The integration of the memory controller on the processor die enables ever larger core counts in commodity hardware shared memory systems with Non-Uniform Memory Architecture properties. Shared memory parallelization with OpenMP is an elegant and widely used approach to leverage the power of such systems. The binding of the OpenMP threads to compute cores and the corresponding memory association are becoming even more critical in order to obtain optimal performance. In this work we provide a method to measure the amount of remote socket memory accesses a thread generates. We use available performance monitoring CPU counters in combination with thread binding on a quad socket Nehalem EX system. For visualization of the collected data we use Vampir.

1 Introduction

With the ever increasing demand for compute power to satisfy the demand of scientists and engineers, hardware vendors assemble larger and larger systems. Considering the Top 500 list [1], current clusters are ranging in the tens of thousands of compute nodes, typically in a dual socket Non Uniform Memory Architecture (NUMA) configuration focused on distributed memory computation. At the same time "fat" commodity nodes with up to eight sockets and up to 128 logical threads become available. They satisfy the need of heavily communicating workloads, shared memory constrained applications and memory requirements up to 2TB [2].

Developing efficient algorithms for such platforms poses a serious problem for developers, particularly if one takes into account that the sockets of these systems might not even be fully connected, i.e. one requires a number of hops between sockets to deliver a piece of data from one processor core to the other[1]. Based

[1] Notice that the terms cores, CPUs, sockets and nodes are often used in a confusing fashion, mostly due for historical reasons. Here we refer to a core as compute core (which a OS will denote as a CPU). A CPU we refer to as a single package compute device mounted in a socket.

M.R. Guarracino et al. (Eds.): Euro-Par 2010 Workshops, LNCS 6586, pp. 523–530, 2011.

on todays most widely supported programming models, OpenMP and MPI (or a hybrid version of both), one has to develop highly parallelizable algorithms taking into account the memory location across as well as within the NUMA nodes in order to minimize communication and synchronization times. A variety of software tools are available that help developers to detect various inefficiencies of their software by addressing important issues like hotspots or thread correctness checking (examples: [3,4,5]). None of these tools, however, specifically helps developers to optimize for the now ubiquitous NUMA platforms.

In this work we address this shortcoming by presenting a technique to directly measure and visualize the used bandwidth between sockets on a four socket NUMA platform based on the Intel Xeon 7500 Series processor. The remainder of this work is structured as follows: First we describe the features and the layout of our test system. We then detail on our methodology and implementation approach. After a brief evaluation of a test code we apply our method to a Jacobi solver kernel to evaluate our approach.

2 Test System Description

We conduct our research on a Xeon X7560 4-socket system, with the CPUs being clocked at 2.26 GHz nominal frequency. The Xeon 7560 CPU features 8 cores capable of symmetric multi-threading (SMT) resulting in 16 possible hardware threads per CPU. The core details are not of particular interest here since we are mostly concerned with the memory subsystem. As for the cache hierarchy, there are three levels on each die as follows: 32kB 1st and 256kB 2nd level cache, both 8 way set associative and individual to each core, and a 24MB 24-way set associative 3rd level cache that is shared between the cores.

An important design principle of this CPU is a fundamental differentiation between the processing cores and the remaining environment, like memory controllers, caches, etc., called *uncore*. Memory accesses that cannot be satisfied by the LLC are either routed to the local or remote memory via the uncore routing facility (R-Box), depending on the physical memory location. In this setup, given a homogeneous memory layout with an equal number of DIMMs per memory channel, the cores have only direct access at native speed to a fraction of the total memory of the system via four separate memory channels. The remaining memory then has to be accessed through the QPI links on a 64-byte cache line basis.

Due to the rather complex memory architecture there are several latencies for a single piece of data, depending where that data resides and if the neighboring caches, i.e. caches on one of the CPUs of another socket, have to be queried or not. Our initial research on a quad socket Nehalem EX system shows up to 6 different access times that could occur for a single memory access:

- LLC hit that does not need snooping (non-shared line): order of 40 cycles
- LLC hit requiring snooping (shared line), clean response: order of 65 cycles
- LLC hit requiring snooping (shared line), dirty response: order of 75 cycles
- LLC miss, answer from local DRAM: order of 180 cycles

– LLC miss, answer from remote LLC: order of 200 cycles
– LLC miss, answer from remote DRAM: order of 300 cycles

Evidently, there are large differences between local and remote memory responses, which strongly underlines the need to optimize multi-threaded applications for memory affinity [6].

As each of the CPUs of each socket has four QPI links, a quad socket system can be build fully interconnecting each of the packages to each other. Therefore three of the QPI links are used to connect a single package to all its neighbors. This leaves one link free for I/O or other purposes.

3 Measurement of Cross Socket Traffic

3.1 Performance Monitoring

Performance monitoring is an ubiquitous feature of modern CPUs of all flavors, not only providing the CPU designers with means to debug hardware units, but also benefiting the user with feedback on the quality of the execution[7]. Generally the configuration of a performance monitoring unit (PMU) is quite simple, although low level. In order to obtain the number of occurrences of a given event in a given time one has to touch three types of model specific registers (MSR)[2] on the CPU: a control register which globally enables the tracking of events, a configuration register telling the CPU to which event to accumulate and a counter register which gets incremented each time the event occurs.

The Nehalem EX provides a manifold of hardware events to track a wide variety of observables on a per core basis, such as executed floating point operations, cache misses, branch-mispredictions, etc[8]. Seven core events can be measured at the same time, three of which are fixed events and four can be programmed for general purpose[9].

In contrast to earlier CPUs, however, the Nehalem CPUs also include special counter registers not associated with specific cores, but with the uncore itself. It is important to notice that, unlike the conventional core PMU, not all events on the uncore-PMU can be programmed in every register reflecting the internal layout. For this work the events and counters of the routing facility (R-Box) are of interest, as there the requests for cache-lines are routed to either the local memory or dispatched to the adjunct sockets for fulfillment.

3.2 Programming the Uncore MSRs

Even though there is no direct counter to measure the request for remote-cache lines it is possible to count the responses with the *NEW_PACKETS_RECV* event. This event can be configured to track many different incoming message types, like snoop message replies (compare [10, p2-92]), but also to count all incoming data responses being received through a specific QPI-link. The number

[2] MSRs can generally change with every new generation of a CPU, although some of them are *architectural*, being guaranteed to be found on every new generation.

Table 1. Uncore counter initialization sequence

1: U_MSR_PMON_GLOBAL_CTL[29]=1	Reset global uncore counters
2: R_MSR_PMON_GLOBAL_CTL_7_0[7-0]=1	Enable R-Box counters for ports 0-3 (counters 0-7)
3: R_MSR_PORT0_IPERF_CFG0[3,6,7]=1	Configure event "Data Response Any" for port 0
4: R_MSR_PMON_CTL0=0x01	Set the control of counter 0 to monitor the event set in R_MSR_PORT0_IPERF_CFG0
5: U_MSR_PMON_GLOBAL_CTL[0,28]=1	Globally enable performance monitoring

of incoming packets along with the payload of each packet can then be used to compute a close estimate of the actually used uni-directional bandwidth of that link. As the router of the CPU (R-Box) has eight separate counter registers, it is possible to track the traffic of each QPI-link separately, facilitating the measurement of all QPI-links within a multi-socket system for performance analysis.

We detail on the programming of the uncore registers. The configuration sequence in (Tab. 1) sets exactly one counter to monitor the packages received at a particular QPI port on one socket. The counter register *R_MSR_PMON_CTR0*, which is controlled by *R_MSR_PMON_CTL0*, will now start incrementing each time a package is received that represents a data response[3].

Ideally one would like to directly measure whether a given LLC-miss was serviced from local or remote memory on a per-instruction level to get the most accurate information on that instructions impact to the whole application performance. Although this would be feasible in a statistical sampling approach, the implementation of a sampling driver is certainly beyond the scope of this work. Here we therefore measured the counters on an OpenMP-construct level, similar to ompP[5]. This results in a more coarse measurement interval, but helps to reduce the overhead. This also helps to associate the data with the parallelism described by the OpenMP constructs.

Unfortunately one is only able to measure on a per socket basis. Therefore a method to map the remote access to the thread with the LLC miss has to be found. This again splits up into two distinct issues.

On the one hand the operating system is altering the scheduling on the system, potentially moving processes and threads from one socket to another. As there are no hooks to intercept such occurrences it is not possible to measure the traffic up to that point and associate this with the new core the thread is running on. To circumvent this issue one can use process/thread-binding to pin the threads of the program to specific cores. Therefore a given thread will always execute on a specified core and no special runtime handling is necessary.

[3] Notice that we haven't configured any interrupt firing upon overflow of this register, so that we currently have to make sure that an overflow does not occur.

On the other hand there is no further mechanism available to us, to directly identify which core caused the cache miss on a given socket. For this work, our approach therefore schedules only one thread per socket, in order to directly associate its misses with the QPI-traffic. For two or more threads per socket this would not be possible and is not handled.

3.3 Implementation and Visualization

In order to obtain the counter data from an execution of an OpenMP program we used the OPARI tool[11]. It is a source-to-source instrumenter and can be used to insert measurement hooks for all OpenMP constructs. These hooks then provide the means to attach a measurement library to the program. With this we are able to instrument any given OpenMP program without any manual modification of the code.

Within the backend-library we gathered the 12 (one for each link on each of the four CPUs) *NEW_PACKETS_RECV* counter values at the beginning and end of each OpenMP construct directly from the R-Box of each Nehalem CPU. As currently no generic API, like PAPI[12], exists that supports the uncore counters of the Nehalem EX, we used the MSR interface of current Linux Kernels to directly interface with the CPU. This interface provides special files (/dev/cpu/x/msrm, where 'x' denotes the OS CPU number) by the msr kernel module which allow direct access to the model specific registers.

After obtaining the traffic through each of the 12 QPI links responsible for inter socket communication the open trace format (OTF)[13] is used to time-stamp and store the data to a trace-file on the hard-drive for post-mortem analysis. We selected the OTF due to its scalability, ease of use, wide acceptance and availability of a library implementation.

Because the QPI-links and messages through these links bear some resemblance to message passing we visualized the traces with Vampir[14], a renowned MPI-performance analysis tool. Vampir uses time-line visualization, i.e. it displays the message passing behavior as horizontal bars displaying function calls, one for each process, and vertical lines representing messages (Fig. 1A). This approach can be adapted to the view of our Nehalem EX system. To achieve this, we mapped in a post-mortem processing step the sockets to (MPI-)processes and the transfer of data within a region through a QPI link to a (MPI) message from the source socket to the target socket with the same time frame.

However as we always observed some traffic through any given QPI-link this would lead to the creation of a message for every QPI-link at every possible location. This would result in message-lines at all possible locations resulting in an overwhelming magnitude of messages to analyze. To account for this, we utilized Vampir's capability to filter its display with regard to the communicator of a message. We sorted the measured QPI bandwidths into different groups of ascending bandwidth-ranges (<100MiB/s, <200MiB/s, <1GiB/s, ...) and assigned to each bandwidth-group a different communicator. With this approach the user is later able to use Vampir's communicator filtering mechanism to select which group of "messages" he would like to investigate, with the ability to "on the

fly" suppress any noise or message ranges of low and uninteresting bandwidths. This will in the end enable the user to focus on regions where the cache traffic exhibits interesting behavior.

4 Results

4.1 Basic Functionality Test

For the initial test we used an for this purpose designed OpenMP test code. This code executes with one thread per socket and allocates and initializes one continuous block of 1GiB memory with the first OpenMP-thread. As the whole block is initialized by the first thread of the process, the memory location for that block will be on the memory of the socket that the thread was running on according to the first touch policy of Linux. Afterwards we read this block of memory once per thread, whilst using an OpenMP-single construct to limit the access to that memory to one single thread at a time. We also ensured that each time a different OpenMP thread entered the single construct. This code was then executed on our four socket system exclusively. We used four OpenMP threads, where each of the underlying system-threads was bound to the first core on each of the different sockets using the "taskset" command of Linux. With our approach we were able to measure and identify which socket each thread was running on and which QPI link was used to access the remote memory. As the QPI can transmit 64 bytes per transfer, we confirmed at least 16 million hits for 1 GiB (the QPI communicates always a whole cache line) of remote memory accesses.

4.2 Jacobi Kernel

Besides our test code, we also tested our approach on a simple implementation of a Jacobi-solver kernel. To evaluate our analysis method we started with a simple NUMA unaware implementation.

We applied our analysis technique to this code with the aforementioned setup (4 threads, binding, etc.) and a $28k \times 28k$ matrix size. The resulting program execution took $7.6s$ in total runtime, which corresponds to $0.08s$ for the compute- and $0.18s$ for the copy-back loop per iteration. As predicted the initial display of our measurement-data showed traffic-lines (messages) between all packages at all times. Using the described communicator filtering approach, we filtered any traffic below 100MiB/s out, which we concluded to be "background noise". The resulting display (Fig. 1A) showed the bulk of the inter-socket communication. From the regularly structured message-lines we concluded that a significant portion of the used memory was not evenly distributed to all sockets resulting in increased QPI-traffic (and memory latency). We confirmed this with Vampirs "communication matrix"-view for the whole program as well as for a single Jacobi-iteration (see fig. 1B). The numbers showed that 4 multiples of 1/4th of the matrix was transfered from package 1 to each other socket and 1 multiple

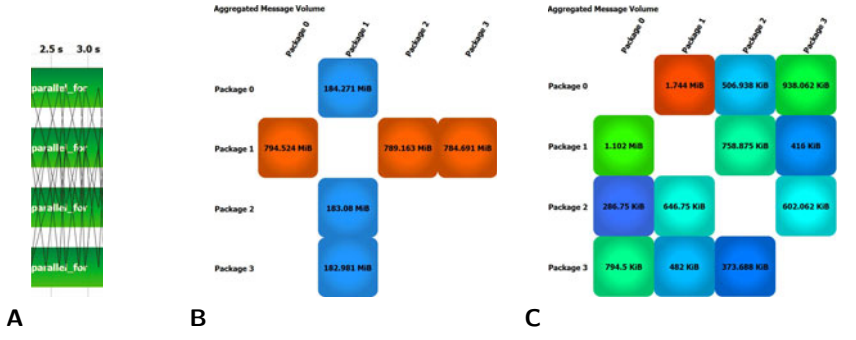

Fig. 1. A) Excerpt of the Vampir-timeline; Visualization of the QPI-message-volume of a typical iteration for **B**) the non optimized and **C**) NUMA optimized Jacobi

was collected back to the initial thread socket. This indicated that all the Jacobi matrix data was residing only on socket one. For this case the highest total transferred data observable was 11.2 GiB.

To alleviate this potential "performance issue" we included the initialization of the matrix into the parallel region and used an OpenMP-for construct in combination with the first-touch policy to distribute the memory to the sockets where it would later be used in the iterations. Re-evaluation of the program showed a 45% improvement in runtime. The improvement for the compute loop was $0.67s$ and for the copy-back loop $0.15s$ (22% and 19% improvement respectively). The change in behavior was easily observed in the graphical presentation of the performance data. In contrast to the NUMA unaware implementation the highest observable bandwidth was well below 200 MiB/s. After again filtering out the background noise, the timeline showed only one occurrence of barely increased traffic. As this did not reoccur with any of the other Jacobi iterations we concluded this to be an artefact. Investigating the "communication matrix" we observed a significant lower and more distributed amount of transferred data, peaking at 69.9 MiB. Matching the improvements of the total message volume, the highest traffic for a single Jacobi iteration was typically in the range of 0.4 to 1.74 MiB (Fig. 1C).

5 Conclusion

Analyzing an OpenMP code to adapt it to modern NUMA hardware architectures is currently a difficult task. Obtaining and interpreting information about the codes memory behavior on current NUMA-platforms is a key part when striving for good performance. In this work we introduced a novel approach to visualize the remote socket cache traffic of an shared-memory parallelized application on the Intel Nehalem EX platform. Granted the current prototypical approach with its restrictions in the setup is in its current form of limited use as it cannot map the gathered data to each core within a socket, it is a viable

approach to obtain and analyze initial data for shared memory tuning on a NUMA platform.

Further research and investigation of the counters of modern CPUs may overt ways to obtain even more detailed information possibly enabling even a per core mapping of data. In addition to the source-to-source implementation sampling may also be used to gather the necessary data within a region to provide a better time resolution enabling even better analysis. Detailed case studies and research may also provide an automatic hot-spot indicator and best-threshold detection mechanism as it is already done for MPI in tools like Scalasca[15].

References

1. Top500.org: Top 500 List June 2010 (July 2010), http://www.top500.org/
2. HP: HP ProLiant DL980 G7 Server Data Sheet
3. Intel(R): Intel(r) thread checker, http://software.intel.com/en-us/intel-thread-checker/
4. Sun Microsystems: Thread analyzer user's guide, http://dlc.sun.com/pdf/820-0619/820-0619.pdf
5. Fürlinger, K., Gerndt, M.: A profiling tool for OpenMP. In: OpenMP Shared Memory Parallel Programming, Dresden, Germany. Springer, Heidelberg (2008)
6. Terboven, C., an Mey, D., Schmidl, D., Jing, H., Wagner, M.: Data and thread affinity in OpenMP programs. In: Memory Access on future Processors: A solved problem? In: ACM International Conference on Computing Frontiers, Ischia, Italy (May 2008)
7. Jarp, S., Jurga, R., Nowak, A.: Perfmon2: A leap forward in performance monitoring. In: International Conference on Computing in High Energy and Nuclear Physics. Journal of Physics: Conference Series, vol. 119, p. 042017 (2008)
8. Intel: Intel 64 and IA-32 Architectures Optimization Reference Manual (2009)
9. Intel: Intel 64 and IA-32 Architectures Software Developer's Manuals Volume 3B (2010)
10. Intel(R): Intel(R) Xeon(R) processor 7500 series uncore programming guide (2010), http://www.intel.com/Assets/pt_BR/PDF/designguide/323535.pdf
11. Mohr, B., Malony, A.D., Shende, S., Wolf, F.: Design and prototype of a performance tool interface for OpenMP. J. Supercomput. 23(1), 105–128 (2002)
12. Terpstra, D., Jagode, H., You, H., Dongarra, J.: Collecting performance data with papi-c. In: Proceedings of the 3rd Parallel Tools Workshop (2010) (to appear)
13. Knüpfer, A., Brendel, R., Brunst, H., Mix, H., Nagel, W.E.: Introducing the open trace format (OTF). In: Alexandrov, V.N., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS 2006. LNCS, vol. 3992, pp. 526–533. Springer, Heidelberg (2006)
14. 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 HLRS Parallel Tools Workshop, Stuttgart, Germany (July 2008)
15. Wolf, F., Wylie, B.J.N., Ábrahám, E., Becker, D., Frings, W., Fürlinger, K., Geimer, M., Hermanns, M.-A., Mohr, B., Moore, S., Pfeifer, M., Szebenyi, Z.: Usage of the scalasca toolset for scalable performance analysis of large-scale parallel applications. In: Proceedings of the 2nd HLRS Parallel Tools Workshop, Stuttgart, Germany (July 2008)