Path-synchronous Performance Monitoring in HPC Interconnection Networks with Source-code Attribution

Adarsh Yoga[1,2] and Milind Chabbi[3⋆]

[1] Hewlett Packard Labs, Palo Alto, CA, USA,
[2] Rutgers University, Piscataway, NJ, USA,
adarsh.yoga@cs.rutgers.edu
[3] Scalable Machines Research, Cupertino, CA, USA,
milind@scalablemachines.co

Abstract. Performance anomalies involving interconnection networks have largely remained a "black box" for developers relying on traditional CPU profilers. Network-side profilers collect aggregate statistics and lack source-code attribution. We have incorporated an effective protocol extension in the Gen-Z communication protocol for tagging network packets in an interconnection network; additionally, we have backed the protocol extension with hardware and software enhancements that allow tracking the flow of a network transaction through every hop in the interconnection network and associate it back to the application source code. The result is a first-of-its-kind hardware-assisted telemetry of disparate, autonomous interconnection networking components with application source code association that offers better developer insights. Our scheme works on a sampling basis to ensure low runtime overhead and generates modest volumes of data. Simulation of our methods in the open-source Structural Simulation Toolkit (SST/Macro) shows its effectiveness— deep insights into the underlying network details to the developer at minimal overheads.

1 Introduction

Interconnection networks used in today's supercomputers play a vital role in the overall performance, efficiency, and scalability of scientific simulation and modeling. HPC applications achieve a paltry 5-15% of a machine's peak performance [1,2,3] on modern microprocessor-based supercomputers. A significant fraction of the loss comes from inter-node data movement.

When applications fail to make effective use of the compute resources at scale, application developers resort to profilers to understand bottlenecks. There is sufficient state-of-the-art and commercial tools for CPU profiling [4,5,6,7,8,9,10] that capture metrics such as CPU cycles, cache misses, branch mis-predictions, etc. and associate the measurements back to the application source code or application data objects [11,12].

Domain scientists can only reason about performance when the measurements are attributed back to the application source code. Unfortunately, network performance problems are a "black box" from an application developer's viewpoint. CPU-side profilers typically quantify the amount of delay waiting for a network communication but

⋆ Work done while at Hewlett Packard Labs

offer little insight into *why* an instance of network transaction was slow. Even the most sophisticated network performance analysis techniques [13,14,15,16] only reason about communication endpoints but do not capture measurements from under-the-hood workings from the autonomous interconnection hardware, which includes network interface cards (NICs), bridges, and switches.

Figure 2 in Appendix A shows the execution profile of NWChem [17]—a US Department of Energy flagship computational chemistry code—running with 1024 MPI ranks on the Dragonfly [18,19] interconnection network on the NERSC Edison [20] supercomputer. The figure shows a hotpath in the CPU profile taken using HPCToolkit [5], a state-of-the-art CPU PMU-based profiler. The figure shows a deep call stack with various layers of host-side code leading to the vendor-provided networking API `dmapp_lock_acquire` to acquire a lock on a remote node. The execution spends a significant (26%) part of execution waiting in this networking API, but the profiles cannot obtain any insights on the cause of this wait. This leaves an application developer with many unanswered questions:

1. Is there load imbalance in the code? Our conversation with the NWChem application developers eliminated this case of any load imbalance and contention for a single lock—the workload is dynamically balanced.
2. Is the network lock implementation suboptimal? Our conversation with Cray Inc. eliminated this possibility—the network lock is local spinning MCS [21] lock.
3. Is the communication network performing poorly?
4. Is there an interference from another job that affected this execution?
5. Is the observed, seemingly network problem, indeed a network bandwidth problem or delays in the local NICs to inject messages?
6. If locking is frequent, is the lock-release message getting delayed in the interconnection network? If so, can we use a separate high-priority virtual channel for such network communication that appear on the critical path?

Clearly, traditional CPU profilers cannot offer answers to these questions since they cannot measure what happens in the interconnection network hardware components. Once a network-related transaction leaves the CPU, even in a simplistic network, the following events happen. The message gets enqueued as a command to the NIC. The NIC notices the command at some later point, which introduces an arbitrary delay. Now, the NIC may initiate a DMA transfer from the local DRAM if the command is a send/put. It packetizes a put/send command into multiple MTU-sized packets and injects them one by one. The NIC may then wait for the acknowledgement of every packet (which is the case in Gen-Z [22] protocol). Different packets may take different paths in the network based on the network routing heuristics. At each router hop, a packet may be subject to different policies and arbitration delays before being forwarded to an output port. At the destination NIC, the packets may arrive out of order (which happen in Gen-Z [22]). The destination NIC may delay injecting packet-level acknowledgments. The destination CPU may get notified some time later after the entire message is reassembled and may introduce further delays before a message-level acknowledgment is generated. Finally, an acknowledgment message may be subject to the same set of uncertainties on its return journey.

With myriad autonomous, unsynchronized components, it is virtually impossible to track how a message gets affected in its roundtrip from one host to another. Prior network performance analysis efforts have conducted an event-driven simulation of the network with characteristic workloads for designing superior networks without paying attention to delivering developer insights. Production hardware has offered simple counters in network routers to collect aggregate runtime data, which offer coarse-grained statistics for system administration to spot anomalous or overloaded hardware components; these techniques are tedious, vendor specific, and often not accessible to CPU profilers.

No prior art has addressed the challenge of tracking an individual message from its source location through every hop in every hardware component in an interconnection network and associated the observed performance metrics to the source and target host codes. This level of detailed measurement in conjunction with full CPU-side context-sensitive profiling is the basis of delivering rich, end-to-end application insights. Such detailed profiling and tracing can alone answer questions that we raised previously in the NWChem example. Evidently, tracking every message and every packet in the network with this level of detailed statistics is a recipe for performance data deluge and will bring the network to a grinding halt in merely collecting the measurement data. Statistical sampling comes to our rescue in collecting detailed data with sparse sampling.

Our strategy is to "mark" network transaction to be monitored on a sampling basis at the origin (CPU) and record statistics of such marked messages at every hop along its journey in an interconnection network. By retaining both CPU-side profiles and network profiles for a sparse set of samples, we are able to observe what happens to network transactions and elevate the measurements to application source code in a manner that sheds lights on the causes of network-related problems to the application developer. The result is that the application developer, with full understanding of the problems, may,

1. Choose to refactor the source code to better utilize the network, or
2. Provision more network resources to reduce network-related bottlenecks that are caused by her application, or
3. Conclusively infer that the problem was not caused by the application but due to an interference with another job, the solution is in better network provisioning or job scheduling, or
4. Pinpoint that the problem is not in the network provisioning but in the networking algorithms, an anomalous router, or local network interface (NIC) software or hardware.

2 Related work

There is a rich literature on profiling and tracing CPU executions. Profiling provides aggregated metrics whereas tracing captures the time-varying behavior of executions. Profiling and tracing come in various flavors and granularities. It is common to instrumentation source code or binary, manually or via a tool, at function, loop, or basic-block granularities. Hardware event-based sampling is an orthogonal method where CPU PMU counter overflow triggers an interrupt that a profiler captures and attributes to application binary and in-turn to the source code [4,5,23]. None of these techniques measure data from interconnection network hardware.

MPI profilers [24,25,26] capture communication metrics at endpoints: they measure time spent in networking-related tasks by wrapping or intercepting each MPI library functions. Advanced methods [13] are able to replay execution traces to pinpoint the root causes of some performance bugs. However, none of these methods obtain measurements from networking hardware. As a result, although one might observe anomalous communication delays, there exists little evidence to isolate problems to a host-side NIC, a router, the destination NIC, or destination CPU.

Networking hardware design is often performed via low-level event-driven simulators [27,28,29]. These simulators are driven by predefined communication patterns to assess the strength of hardware designs or algorithms. A low-level simulator can simulate only a small (often milliseconds to a second) amount of real execution. High-level simulators [30,31,32] capture runtime communication traces on real execution and replay the communication traces to drive coarse-grained network simulators. Both high-level and low-level network simulators treat the CPU execution as a black box and focus only on the networking aspect and hence are incapable of offering insights to application an developer at the source code level.

There is rich literature in network profilers for Ethernet [33,34,35]. We are unaware of any tool that can a) attribute network profiles to application source code, or b) perform path-synchronous sampling to capture a specific network transaction (e.g., traversal of a specific packet) throughout its journey. Network-side monitoring schemes such as sFlow [35] and netflow [34] capture the source and destination of a packet when flowing through a component. They, however, lack the full path information of a sampled transaction and hence the hop-by-hop details of any specific packet is unavailable. sFlow can aggregate the data from many components over long periods of time and filter the data by the traffic originating from the same source going to the destination to reconstruct an "average" behavior and a "typical" path; but such schemes cannot attribute the observed behavior to the application source code because over time there can be many source code locations contributing to the same "flow". Samples from different components lack temporal correlation. This lack of temporal correlation means one can observe only aggregate behavior of traffic and not be able to pinpoint a specific anomaly to its causes. Aggregate metrics handicap the ability to pinpoint the cause of a transient anomalous behavior.

3 Methodology

A key requirement for attributing network behavior to application source code is to identify what happens to a transaction initiated by a line of source code (could be an assembly instruction) throughout its journey through the network. Such hop-by-hop tracking retains temporal correlation among performance metrics generated by unsynchronized components. We track hop-by-hop metrics of a small subset of packets as they are forwarded across a network. This is done on sampling basis because observing every transaction is infeasible both from space and time overhead viewpoint. In other words, one in N packet originating from a source is chosen to be tracked throughout its journey. The choice of N can be arbitrary or more intelligent. Each endpoint may choose the same or different value of N. Endpoints need not coordinate when they track

a packet. Sampling ensures that any event that is statistically significant will be observed with the frequency proportional to its occurrence. We propose the following extensions:

Protocol extension: every packet of the protocol carries a special Performance Monitoring (PM) tag. The PM tag may be present at a designated offset in the packet header to make it quick to inspect by the hardware. We call a packet whose PM tag is enabled as a "marked" packet. We have already incorporated a PM tag in the Gen-Z protocol [22] to enable performance tools.

Hardware Extensions:
1. The NIC exposes a special tag "track me" (TM) to the software. The software may assert the TM bit in a command it issues to the local NIC indicating the NIC to track the command.
2. The NIC propagates the TM bit from a CPU-issued command into a (one or more) packet(s) by setting the PM bit in the packets that it injects into the network on behalf of the command.
3. Every switch inspects the PM tag of each packet it routes. If the PM tag is enabled in an incoming packet, the switch logs a performance data record into its local buffer (typically an SRAM). The PM tag is propagated through the switch from an incoming packet to the corresponding out-going packet.

The fact that a marked packet's information is logged at each hop allows us to achieve the path-synchronous sampling. A key piece of information logged at each hop is the unique identity of the next hop of the packet. The next hop information allows us to, in a post mortem pass, reconstruct the full path along the journey of a marked packet. In systems with request-response protocol (e.g., Gen-Z), the PM tag is retained from request to response so that its journey is tracked in both directions. To accomplish this, the endpoint hardware (e.g., NIC) may be modified to propagate the PM tag from request to response. We assume that every network packet at least contains its source identifier (SID), destination identifier (DIS), a tag (need not be unique), and the PM tag. The log in each component contains at least the following information:

1. The arrival time of the packet or command (component local time).
2. The departure time of the packet or command (component local time).
3. The identity of the next hop (out going port) of the packet/command.
4. (Optional) In addition to the first three necessary data, a component may include any additional data: for example, anomalous condition at the time of routing the designated marked packet (e.g., ran out of credit when transmitting this packet), position of the packet in a router's input queue on arrival, conflict during router arbitration, etc.

Figure 1 depicts the workflow when the endpoint A wants to send a message to endpoint B and the packet follows the route $A \rightarrow NIC_a \rightarrow S_1 \rightarrow S_2 \rightarrow S_4 \rightarrow NIC_b \rightarrow B$ in an anecdotal network:

1. The software (profiler running on the source CPU, endpoint A) on a sampling basis chooses a transaction to be monitored. The choice can be random sampling or more intelligent, if desired.
2. The software captures its CPU calling context ($CTXT_1$) and creates a locally unique command id (CID_1) representing the network command.

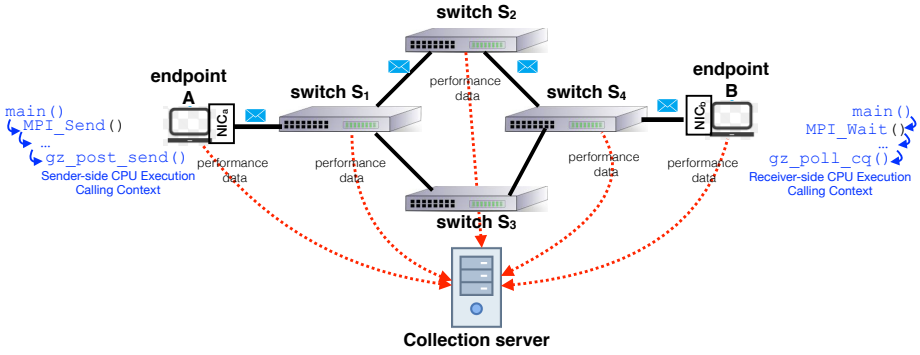


Fig. 1: An interconnection network with four switches, two endpoints and their respective NICs. A message sent from A to B traverses the path A→NIC_a→S_1→S_2→S_4→NIC_b→B. The profiler captures CPU-side contexts, marks the message to be tracked in the network and logs data to a collection server. NICs and switches that the marked packet traverses also log their data to the collection server.

.

3. The software (at time T_1) issues the network command to NIC_a passing the unique id (CID_1) setting the TM flag.
4. Software logs the tuple $\langle CTXT_1, CID_1, T_1, A, B\rangle$.
5. NIC_a at a later point (time T_2) inspects the command, generates some M network packets for the command, and by observing the TM flag, it enables the PM tag in one of (randomly chosen or otherwise) the M network packets.
6. NIC_a injects the PM-marked packet at time T_3 to the switch S_1. Let the id of the marked packet be PKID. Let the last packet corresponding to CID_1 leave at time T_4. NIC_a logs the local information tuple $\langle CID_1, A, B, PKID, S_1, T_2, T_3, T_4\rangle$.
7. The switch S_1 notices the marked packet with PKID at time T_5 and forwards it to switch S_2 at time T_6 and the logs the information tuple $\langle A, B, PKID, S_2, T_5, T_6\rangle$.
8. The switch S_2 notices the marked packet with PKID at time T_7 and forwards it to switch S_4 at time T_8 and the logs the information tuple $\langle A, B, PKID, S_4, T_7, T_8\rangle$.
9. The switch S_4 notices the marked packet with PKID at time T_9 and forwards it to NIC_b at T_{10} and the logs the information tuple $\langle A, B, PKID, NIC_b, T_9, T_{10}\rangle$.
10. NIC_b at time T_{11} assembles all packets and create an entry for the endpoint B and produces the log entry $\langle A, B, PKID, CID_2, B, T_{10}, T_{11}, TM{=}1\rangle$.
11. The CPU at endpoint B at time T_{12} in calling context $CTXT_2$ receives the full message and on noticing the TM flag, logs the tuple $\langle CTXT_2, CID_2, T_{12}, A, B\rangle$.

For brevity, we are not discussing the case of response or acknowledgment or dropped packets. In unreliable networks when a marked packet is dropped, no further logs will be available—a clear indication of a dropped packet. We do not discuss what additional information a component may log. There can be component-specific fields, which, for example, can include link-level credits.

Collection server: Hardware has a limited local buffer to log performance data. Hence, we use a management software running on each hardware component to periodically

drain the logs collected to a centralized server. The SRAM buffer on the hardware acts as a circular buffer. All modern HPC networking components have additional management hardware with Ethernet connections of $\sim$1 GBPS. The management software on each component is capable of NFS mounting a remote distributed server and dump logs from local memory to a unique file on the remote server.

Post-mortem analysis: The collection server contains logs collected from all components through which every marked packet traverses. A post-mortem analysis of the logs in the collection server allows a software tool to reconstruct the complete path traversed by each marked packet initiated at a source and associate the data with the application source code in its calling context. In the previous example, starting from the CPU-side log of the endpoint A, we can go through the following steps to reconstruct the path:

1. Endpoint A's log entry $\langle CTXT_1, CID_1, T_1, A, B\rangle$ tells that at source-code context $CTXT_1$, a command CID_1 was issued to target B.
2. Sifting through NIC_a's logs for CID_1 shows the following entry: $\langle CID_1, A, B, PKID, S_1, T_2, T_3, T_4\rangle$. T_2-T_1 is the in-node delay. The command took a total of T_4-T_2 time to get injected. The marked packet has the tag PKID and was injected at time T_3 and was sent to switch S_1.
3. Sifting through switch S_1's logs for $\langle A, B, PKID, T_3 \pm \Delta\rangle$ shows a record $\langle A, B, PKID, S_2, T_5, T_6\rangle$. The packet's delay at hop S_1 is T_6-T_5. It was forwarded to S_2.
4. Sifting through switch S_2's logs for $\langle A, B, PKID, T_6 \pm \Delta\rangle$ shows a record $\langle A, B, PKID, S_4, T_7, T_8\rangle$. The packet's delay at hop S_2 is T_8-T_7. It was forwarded to S_4.
5. Sifting through switch S_4's logs for $\langle A, B, PKID, T_8 \pm \Delta\rangle$ shows a record $\langle A, B, PKID, B, T_9, T_{10}\rangle.\rangle$. The packet's delay at hop S_4 is T_{10}-T_9. It was forwarded to the destination NIC_b.
6. Sifting through NIC_b's logs for $\langle A, B, PKID, T_{10} \pm \Delta\rangle$ shows the following entry: $\langle A, B, PKID, CID_2, B, T_{10}, T_{11}, TM{=}1\rangle$. T_{11}-T_{10} is the delay at NIC_b. It was delivered to the destination B.
7. Sifting through endpoint B's logs for $\langle A, B, CID_2, T_{11} \pm \Delta\rangle$ shows the following entry: $\langle CTXT_2, CID_2, T_{12}, A, B\rangle$. $CTXT_2$ is the receiving application calling context. The packet's journey ends here.

Full calling context with source code attribution at both endpoints along with hop-by-hop metrics for the traversal: $A(CTXT_1) \rightarrow NIC_a \rightarrow S_1 \rightarrow S_2 \rightarrow S_4 \rightarrow NIC_b \rightarrow B(CTXT_2)$ including the in-NIC delays is easily reconstructed. Since each component logs data into its local buffer, there is no need for concurrency control. There is no need for perfectly synchronized clocks across the system; but, we expect the components to be close enough in time via standard protocols such as NTP.

Alternative uses: Our approach samples a randomly chosen transaction in a window of N transactions. Alternatively, we may also sample the exact N^{th} transaction. In fact, a precise, predetermined, transaction may be sampled, if desired. Instead of the software at the source of a transaction enabling the PM tag, any component may choose to enable the PM tag and capture the partial path. Although we suggested unsynchronized sampling from endpoints, we do not preclude sampling in a synchronized manner, which is useful for debugging purposes. Our approach associates metrics to the source-code location

that initiated a transaction. We do not preclude associating metrics to some other place in the source code, e.g., a network wait event associated with a non-blocking transaction.

4 Implementation

We implemented our network performance monitoring prototype using the SST/Macro event-driven network simulator framework [32] and open sourced it [36]. SST/Macro models hardware components such as CPU, memory, NIC, switch, crossbar. The networking components of SST/Macro are mature with various, configurable network topologies, bandwidths, latencies, and algorithms of packet-based routing and arbitration, ideally suited for our evaluation. SST/Macro is easy to extend with additional hardware and software components, which was necessary for our extensions. SST/Macro is driven by "skeleton" C++ code that mimics an HPC C++ code written using MPI and needs trivial or no modifications to work with SST/Macro.

We enhanced SST/Macro in the following ways. We introduced a new flag (PM bit) in SST/Macro packet format. We extended SST/Macro NIC and switch hardware components with the additional capability to log "marked" packets to a bounded SRAM buffer. We chose a bounded buffer of 2 KB in each router and NIC. We introduced a new hardware subcomponent "drainer" in NICs and routers, which reads the performance data accummulated in a local bounded SRAM buffer and transfers it to an on-component management software. The management software NFS mounts a file on the remote data collection server and drains the incoming performance logs to the server.

Additionally, we also implemented extensions to the NIC-software interface to express the ability to track a message. We extended the NIC with the ability to mark one out of N packets with the PM bit if the command issued from the CPU carried the TM flag and append its log in a local SRAM.

We drive the profiling with a software profiler in SST that uses random sampling to determine if a message needs to be monitored. If so, it sets a special TM tag when it issues a command to its NIC. Also, the CPU profiler collects the calling context and logs the CPU metrics about the message in a per-CPU log file.

The postmortem analysis inspects the log files to reconstruct the path taken by each marked packet by each endpoint and associates performance metrics to each hop on the path as described in the previous section. The output of our postmortem analysis is a set of files containing the path information of all the marked messages. The path information also contains the performance metrics attributed to each hop along the path.

To visualize how the application behaves, we generate a heatmap and a set of stacked bar graphs from the performance metrics using a graphing software called Plotly [37]. Figure 3a in Appendix B shows the heatmap for an example NCAST program. The heatmap shows the total time taken by each marked packet to travel from the source CPU to the destination CPU. The points on the x-axis correspond to the time at which messages were initiated. The points on the y-axis correspond to the processes that sent the messages. A point on the heatmap that is darker than other points signifies the message took relatively longer to travel from the source to the destination. In addition to the heatmap, we also generate a set of stacked bar graphs, one for each process that initiated a message in the program. Figure 3b shows a bar graph for process 97 in the

ncast program. Each bar represents the cumulative time spent by the message in each network component along its path and each stack in a bar represents the time spent at each network component. A large stack in a bar shows that the message was stuck in the component for a long time. To summarize, we can use the heatmap to identify what messages were delayed, and then use the stacked bar graph corresponding to the process that initiated that message to identify network component that caused the delay.

5 Evaluation

We evaluated our prototype implementation to answer the following questions: (1) how effective is our prototype in finding performance bottlenecks in the network due to the application? (2) does our prototype monitor network traffic with low overhead? All our experiments were run on a four socket, 15-core Intel Xeon E7-4890 machine clocked at 2.8GHz and containing 1TB DRAM. Our setup simulated the NERSC Edison [20] system with the Dragonfly [18,19] topology containing 5586 compute nodes.

Effectiveness: To evaluate the effectiveness of our prototype, we executed it with an MPI skeleton program and ran with 4096 MPI ranks. In the skeleton program—NCAST—a single MPI process (rank 42) is bombarded with multiple large (4MB) messages from all the other MPI processes in the network. As a large number of messages are sent to a single node, the NIC at the destination CPU becomes a bottleneck. Also, since all packets would flow through a single network switch before reaching the destination, the switch at the last hop becomes congested. Our goal is to use our prototype implementation to precisely identify the network component that is the bottleneck in the NCAST program.

Figure 3 in Appendix B shows the graphs generated by our prototype after executing the NCAST program. The heatmap in Figure 3a in Appendix B reveals a surprising and non-obvious performance problem—the messages are all serialized; the MPI ranks are sending messages one after another, resulting in the diagonal in the heatmap. Samples from all CPUs except for CPU 42 are sparse and CPU 42 samples show that it is continuously sending messages to other processes, which is reflected in the thick horizontal line in the heatmap. The reason for serialization is the large message size sent from all other nodes. For large messages, each MPI process sends a short notification message to the target (rank 42); and the target one-by-one fetches the large message from the sources. The concurrency gets completely destroyed—a subtle anomaly invisible in the CPU-only profiles but distinctly visible in full network telemetry.

On the diagonal, we can see that the points above CPU 80 appear darker which means that those messages take relatively longer than the earlier messages. This shows that the messages that are being sent later are getting delayed at either the destination or at a network switch. Figure 3b in Appendix B shows the stacked bar graph of CPU 97. The large stack in the bar graph represents the time spent at switch 21. Switch 21 directly connects to node 42 which is receiving messages from all other nodes. Hence the stack is large since all the packets are queued at switch 21. We observe a similar pattern in the bar graphs corresponding to all other CPUs after CPU 80. This shows that all the messages are queued up at switch 21, which has become chocked.

Efficiency: We evaluate the efficiency of our network performance monitoring scheme by measuring the simulation and wall clock time on three MPI skeletons: NCAST, broadcast, and a mutiapp. We execute each application five times and report the geometric mean of the overhead. The skeletons were designed such that their simulation time was at least one second. We tracked one in every hundred NIC command at each endpoint. Our measurements showed that the hardware extensions added a negligible 0.16% mean overhead to the simulation time. The wall clock time for simulation marginally increased (4.8%) over the original execution without our extensions to SST/Macro. The average size of the log files generated for the three applications is 61MB.

6 Conclusions

Application developers better understand performance when measurements are attributed back to the source code. However, it is hard to attribute performance measurement data from myriad autonomous, asynchronously operating hardware components in an HPC system back to application source-code. Traditional profilers have either focused only on CPU-side hardware measurements for source-code attribution or focused on network-side hardware measurements without source-code attribution.

We developed a protocol extension to track the flow of packets and collect hardware performance data in the emerging memory-semantic-based communication protocol—Gen-Z. We enhanced the router and NIC hardware and management software with additional components for logging performance data. We enhanced traditional CPU profilers to unify CPU profiles with telemetry from networking hardware. Our sampling-based scheme implemented in the SST/Macro simulator shows promise of our technique in offering a unified system-wide performance insights for application developers.

Our future work involves extensively evaluating our methods on serious workloads, working with hardware development teams to incorporate our proposed extensions, and working with software profiling tools to best utilize the network telemetry.

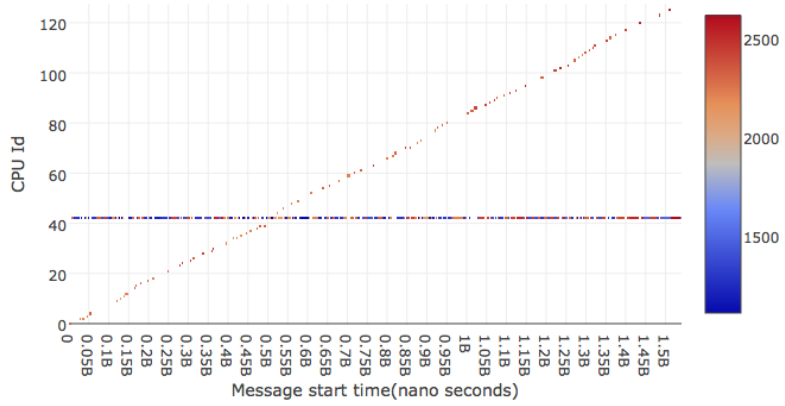
Acknowledgments

This work was supported (in part) by the US Department of Energy (DOE) under Cooperative Agreement DE-SC0012199, the Blackcomb 2 Project.

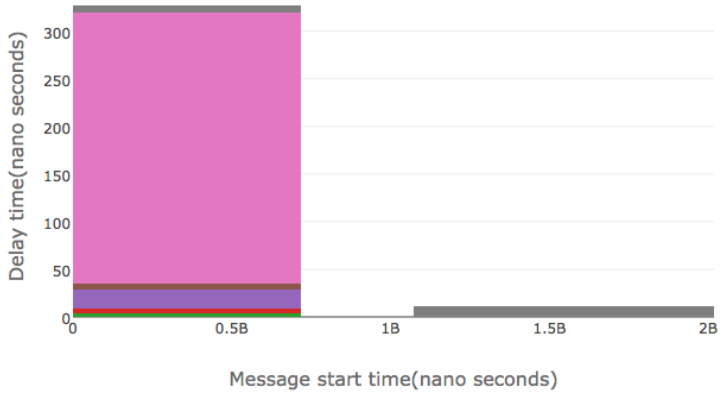
A NWChem profiles from HPCToolkit

Fig. 2: CPU execution hotspot in NWChem running on NERSC Edison with 1024 MPI ranks captured via HPCToolkit [5] profiler. 25% of execution on all MPI processes waste time waiting to acquire remote locks embedded deep inside many layers of host code. The cause of the lock waiting despite good load balance is unknown since CPU profiles do not capture networking hardware component internals.

B Profiles of NCAST program



(a) Heatmap showing the time taken by each marked packet in the NCAST program. Sample points that are darker in color correspond to messages that were delayed the most.



(b) The stacked bar graph of process 97 in the NCAST program. The colored stacks in each bar represent the delay at each hop of the packet.

Fig. 3: Figure shows the visualization graphs generated for the NCAST program running 4096 MPI ranks.

References

1. Oliker, L., Canning, A., Carter, J., Shalf, J., Ethier, S.: Scientific Application Performance on Leading Scalar and Vector Supercomputing Platforms. International Journal of High Performance Computing Applications (2006)
2. Dongarra, J., Heroux, M.A.: Toward a new metric for ranking high performance computing systems. Sandia Report, SAND2013-4744 **312** (2013) 150
3. Egawa, R., Komatsu, K., Momose, S., Isobe, Y., Musa, A., Takizawa, H., Kobayashi, H.: Potential of a modern vector supercomputer for practical applications: performance evaluation of SX-ACE. The Journal of Supercomputing (Mar 2017)
4. Intel Inc.: Intel VTune. `https://software.intel.com/en-us/intel-vtune-amplifier-xe`
5. 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 Expererience **22**(6) (April 2010) 685–701
6. Geimer, M., Wolf, F., Wylie, B.J.N., Ábrahám, E., Becker, D., Mohr, B.: The Scalasca performance toolset architecture. Concurrency and Computation: Practice and Experience **22**(6) (April 2010) 702–719
7. Shende, S.S., Malony, A.D.: The Tau Parallel Performance System. Int. J. High Perform. Comput. Appl. **20**(2) (May 2006) 287–311
8. Oracle Inc.: Oracle Solaris Studio. `http://www.oracle.com/technetwork/server-storage/solarisstudio/overview/index.html`
9. Intel Inc.: Intel Trace Analyzer and Collector. `https://software.intel.com/en-us/intel-trace-analyzer` (Oct, 2017)
10. Allinea Inc.: Allinea MAP - C/C++ profiler and Fortran profiler for high performance Linux code. `https://www.allinea.com/products/map` (Oct, 2017)
11. Liu, X., Mellor-Crummey, J.: A data-centric profiler for parallel programs. In: Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis. (2013) 28
12. Rane, A., Browne, J.: Enhancing performance optimization of multicore chips and multichip nodes with data structure metrics. In: Proc. of the 12th Intl. Conf. on Parallel Architectures and Compilation Techniques, Minneapolis, MN, USA, IEEE Computer Society (2012)
13. Böhme, D., Geimer, M., Arnold, L., Voigtlaender, F., Wolf, F.: Identifying the Root Causes of Wait States in Large-Scale Parallel Applications. ACM Trans. Parallel Comput. **3**(2) (July 2016) 11:1–11:24
14. Isaacs, K.E., Gamblin, T., Bhatele, A., Schulz, M., Hamann, B., Bremer, P.T.: Ordering Traces Logically to Identify Lateness in Message Passing Programs. IEEE Transactions on Parallel and Distributed Systems **27**(3) (March 2016) 829–840
15. Weber, M., Brendel, R., Hilbrich, T., Mohror, K., Schulz, M., Brunst, H.: Structural Clustering: A New Approach to Support Performance Analysis at Scale. In: 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS). (May 2016) 484–493
16. Isaacs, K.E., Giménez, A., Jusufi, I., Gamblin, T., Bhatele, A., Schulz, M., Hamann, B., Bremer, P.T.: State of the Art of Performance Visualization. In Borgo, R., Maciejewski, R., Viola, I., eds.: EuroVis - STARs, The Eurographics Association (2014)
17. Valiev, M., Bylaska, E., Govind, N., Kowalski, K., Straatsma, T., Dam, H.V., Wang, D., Nieplocha, J., Apra, E., Windus, T., de Jong, W.: NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations. Computer Physics Communications **181**(9) (2010) 1477 – 1489
18. Kim, J., Dally, W.J., Scott, S., Abts, D.: Technology-Driven, Highly-Scalable Dragonfly Topology. In: Proceedings of the 35th Annual International Symposium on Computer Architecture. ISCA '08, Washington, DC, USA, IEEE Computer Society (2008) 77–88

19. Alverson, B., Kaplan, L., Roweth, D.: Cray XC Series Network. http://www.cray.com/sites/default/files/resources/CrayXCNetwork.pdf
20. National Energy Research Scientific Computing Center: Edison. http://www.nersc.gov/users/computational-systems/edison/
21. Mellor-Crummey, J.M., Scott, M.L.: Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. ACM Trans. Comput. Syst. **9**(1) (February 1991) 21–65
22. Gen-Z Consortium: Gen-Z: Draft Core Specification. http://genzconsortium.org/specifications/draft-core-specification-july-2017/ (July, 2017)
23. Linux wiki: Linux perf tool. https://perf.wiki.kernel.org/index.php/Main_Page
24. Zaki, O., Lusk, E., Gropp, W., Swider, D.: Toward scalable performance visualization with Jumpshot. High Performance Computing Applications **13**(2) (Fall 1999) 277–288
25. Karrels, E., Lusk, E.: Performance analysis of MPI programs. In Dongarra, J., Tourancheau, B., eds.: Proceedings of the Workshop on Environments and Tools For Parallel Scientific Computing, SIAM Publications (1994) 195–200
26. Knüpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., Mickler, H., Müller, M.S., Nagel, W.E.: The vampir performance analysis tool-set. Tools for High Performance Computing (2008) 139–155
27. McDonald, N.: SuperSim: A flexible event-driven cycle-accurate network simulator. https://github.com/HewlettPackard/supersim
28. Carothers, C.: ROSS: Rensselaer's Optimistic Simulation System. https://github.com/carothersc/ROSS/wiki
29. Carothers, C.D., Bauer, D., Pearce, S.: ROSS: A High-performance, Low Memory, Modular Time Warp System. In: Proceedings of the Fourteenth Workshop on Parallel and Distributed Simulation. PADS '00, Washington, DC, USA, IEEE Computer Society (2000) 53–60
30. Liu, N., Carothers, C., Cope, J., Carns, P., Ross, R.: Model and simulation of exascale communication networks. Journal of Simulation **6**(4) (Nov 2012) 227–236
31. Jain, N., Bhatele, A., White, S., Gamblin, T., Kale, L.V.: Evaluating HPC Networks via Simulation of Parallel Workloads. In: SC16: International Conference for High Performance Computing, Networking, Storage and Analysis. (Nov 2016) 154–165
32. Rodrigues, A.F., Hemmert, K.S., Barrett, B.W., Kersey, C., Oldfield, R., Weston, M., Risen, R., Cook, J., Rosenfeld, P., CooperBalls, E., Jacob, B.: The Structural Simulation Toolkit. SIGMETRICS Perform. Eval. Rev. **38**(4) (March 2011) 37–42
33. So-In, C.: A survey of network traffic monitoring and analysis tools. https://www.cse.wustl.edu/~jain/cse567-06/ftp/net_traffic_monitors3.pdf
34. Cisco Inc.: Cisco IOS NetFlow. https://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-netflow/index.html
35. sFlow organization: sFlow. http://www.sflow.org/
36. Hewlett Packard Labs: Network Performance Monitoring (NWPM) Tool. https://github.com/HewlettPackard/genz_tools_network_monitoring
37. Plotly Technologies Inc.: Collaborative data science. https://plot.ly (2015)