NUMAscope: Capturing and Visualizing Hardware Metrics on Large ccNUMA Systems

Daniel J Blueman
Principal Software Engineer
Numascale AS
Oslo, Norway
daniel@numascale.com

Foivos Zakkak
Department of Computer Science
The University of Manchester
Manchester, United Kingdom
foivos.zakkak@manchester.ac.uk

Christos Kotselidis
Department of Computer Science
The University of Manchester
Manchester, United Kingdom
christos.kotselidis@manchester.ac.uk

Abstract—**Cache-coherent non-uniform memory access (ccNUMA) systems enable parallel applications to scale-up to thousands of cores and many terabytes of main memory. However, since remote accesses come at an increased cost, extra measures are necessitated to scale the applications to high core-counts and process far greater amounts of data than a typical server can hold. In a similar manner to how applications are optimized to improve cache utilization, applications also need to be optimized to improve data-locality on ccNUMA systems to use larger topologies effectively. The first step to optimizing an application is to understand what slows it down. Consequently, profiling tools, or manual instrumentation, are necessary to achieve this. When optimizing applications on large ccNUMA systems, however, there are limited mechanisms to capture and present actionable telemetry. This is partially driven by the proprietary nature of such interconnects, but also by the lack of development of a common and accessible (read open-source) framework that developers or vendors can leverage.**

In this paper, we present an open-source, extensible framework that captures high-rate on-chip events with low overhead (<10% single-core utilization). The presented framework can operate in `live` or `record` mode, allowing both *real-time* monitoring or capture for later post-workload or offline analysis. High-resolution visualization is available either through a *standards-based* (web) interactive graphical interface or through a convenient textual interface for *quick-look* analysis.

Index Terms—**performance, architecture, monitoring, profiling, cache coherency**

I. Introduction

Cache-coherent non-uniform memory access (ccNUMA) architectures enable applications to scale up to hundreds of cores and terabytes of memory without the need of explicit synchronization or communication between the different boards of the system. ccNUMA systems essentially provide a single system image (SSI) to the software, abstracting away the memory hierarchy and thus easing development. To achieve this, ccNUMA systems rely on cache-coherent interconnects and memory-coherence protocols. This approach however may come at a cost if applications do not account for the overheads of remote accesses on a ccNUMA system.

Although memory-coherence is implemented in hardware and is transparent to the software, it comes with overhead which need to be taken into account when optimizing an application; the memory hierarchy is such that main memory access time differs depending on the memory segment being accessed and the core that performs the access. Figure 1 demonstrates how distant memory accesses relate to the latencies of accessing different levels of the memory hierarchy. As a result in a similar way that parallel applications should avoid cache line sharing (so as to avoid excessive coherence traffic), they should minimize writing cache lines frequently read among many NUMA nodes [1].

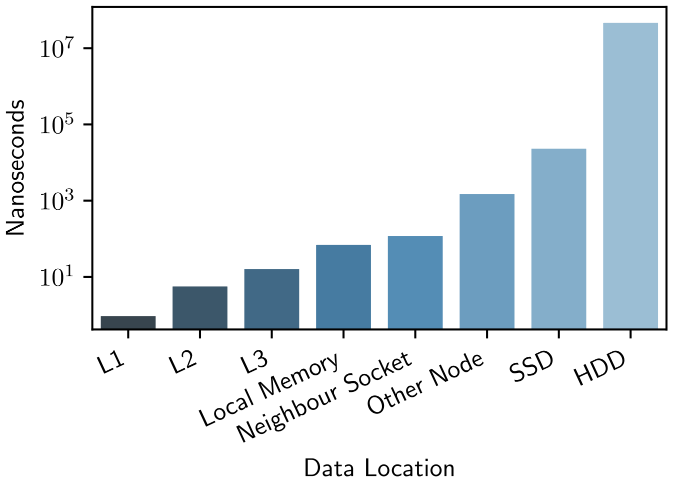


Fig. 1. Latency of memory accesses depending on the distance between the core performing the access and the memory being accessed

The first step in avoiding high latency accesses is to quantify them by inspecting the system's performance counters. However, when developing and tuning software applications on systems large enough to warrant cache-coherent interconnects, existing mechanisms to collect resource usage are limited to the system's core and uncore performance counters. For instance, existing NUMA-aware profilers can attribute increased wait cycles to last level cache (LLC) misses and even help developers understand whether those misses are served by the local or remote NUMA-node. They fail to give any further insight regarding the remote NUMA-nodes that are on a different board.

Furthermore, when developing cache-coherent interconnects outside a simulation environment, there are no universal mechanisms available to capture interconnect resource utilization. Consequently, high-rate capture of fine-grained on-chip interconnect counters would present useful telemetry both to guide interconnect and application development and tuning. As an example of the first case, on-chip buffering is generally expensive; understanding how much buffering is needed to cover the interconnect round-trip for optimal throughput with appropriate counters, would allow allocation of just enough resources. This mismatch occurs since relatively crude activity patterns can be simulated in chip development, and latency isn't modeled at all, with many approximations for simulation speed. In the latter case, the impact of algorithmic adjustments can be understood at a cache-coherent and high-level ccNUMA viewpoint.

In this paper, we demonstrate the lightweight capture of very high-rate on-chip events (up to 2×10^8 events per second), and visualization in an interactive graphing environment, along with a convenient textual interface for *quick-look* analysis. We present NUMAscope, a framework that enables tools that capture events across different cache-coherent interconnects in large ccNUMA systems. NUMAscope is open-source and based on a modular software architecture, that makes it easily extendable. NUMAscope not only enables the user to see at a high-level and quantify suboptimal application (or the system as a whole) behavior, but it also enables identification of *why* it is occurring, from the system's perspective, e.g. because some buffer queues are full. This information enables the observer to understand whether a high rate of events observed using the tool also happens to saturate some hardware resource, potentially leading to different scheduling decisions or even different hardware configurations.

In detail, in this paper we make the following contributions:

- We introduce NUMAscope, a framework that enables tools that perform real-time and *postmortem* monitoring of cache-coherent interconnects, using a mechanism for high-rate event capture; NUMAscope provides both a graphical and a textual interface that visualize the captured events.
- We present how using NUMAscope, we were able to detect a performance issue in the GNU C Compiler 4.8.5 OpenMP implementation and compare its performance before and after applying a fix.
- We evaluate NUMAscope, on a 6-board ccNUMA system, using the NAS Parallel Benchmarks (NPB) [2] EP benchmark, reporting its overhead for various sample rates, starting from a high sample rate at 1000Hz and going down to 1Hz; our evaluation shows that the overhead ranges from 0.27% at 1Hz sampling, to 5% at 1000Hz sampling.

II. Related Work

A. Hardware Counters Access

Various tools and libraries that enable gathering metrics from hardware counters are readily available [3]–[5]. However these tools and libraries are typically able to gather metrics only within a single board, failing to support large ccNUMA systems. Furthermore, to get access to NUMA-related metrics, e.g. number of remote accesses, the tools rely on processor-specific events that users need to pass as parameters to the tool. Identifying and understanding the events that are of interest is not trivial; it requires careful studying of the architecture's manual and understanding of the architecture itself [6, §3]. To make matters worse, stable OS distributions (e.g. CentOS) usually ship with older kernel versions. As a result, `perf`, the most popular profiling tool shipped with Linux, usually lacks support for the performance counters added to newer processors. For Intel processors `PMU-tools` [7] have been developed to mitigate this issue. NUMAscope compared to `perf` is more easily extendable and does not rely on the latest kernel to work, nor on kernel patches to extend it. NUMAscope can also take advantage of the community support for those tools and libraries and use them for gathering metrics.

B. Gathering Metrics About ccNUMA-interconnects

Prior work has also focused on gathering microarchitectural events from ccNUMA interconnects [8], [9]. Both these works focus, each, on a single proprietary hardware interconnect and are closed-source. NUMAscope, in contrast to these approaches [8], [9], is open-source and designed to be modular in order to support different hardware interconnects by implementing a simple software interface.

C. System Monitoring Tools

NUMAscope being able to monitor large ccNUMA systems is also related to system monitoring tools. Most system monitoring tools focus on higher-level metrics such as processor, disk, and network usage [10], as well as even higher-level metrics including service availability. NUMAscope, providing *lower-level* metrics, can complement such tools aiding developers to understand the root-cause of deficiencies in their system.

D. Profiling Tools

By using instruction-based and event-based sampling, prior works have also built profiling tools that are able to correlate code segments with NUMA-related events [6], [11]–[14]. The main goal of these profilers is to help developers detect bottlenecks in their applications, caused by cache hierarchy effects. However, these tools rely on processor core counters and lack support for large ccNUMA systems. As a result, profiling applications on large ccNUMA systems gives limited insight into what is happening outside a single board. Existing NUMA-aware profilers can only tell whether an LLC miss is served by the local NUMA-node or a remote one, with lack of insight regarding accesses to remote NUMA-nodes that are on different boards. Such information is valuable, since different boards may have different utilization, and a holistic view often aids in optimizing imbalanced workloads. NUMAscope can be used in conjunction with such tools to determine the source of excessive memory access time; that would occur when the cycle misses in the local last level cache and the physical address corresponds to a remote NUMA-node. Without NUMAscope, core profiling would simply reveal a memory load taking significantly longer.

PMU-tools also feature the `toplev` tool that implements the top-down method [15]. The top-down method groups performance counters to help developers identify which part of the processor, e.g. front-end, back-end, etc. is being stressed by their application and find potential bottlenecks. NUMAscope and measurements obtained from the ccNUMA interconnects can be used to extend the top-down methodology, and more specifically the *External Memory Bound* group.

Overall, NUMAscope differentiates itself from related work by: *a)* capturing more diverse events and data, *b)* being easily extendable, enabling the profiling of different platforms, *c)* being a framework that enables the creation of specialized monitoring tools on a per case base, and *d)* being open-source and readily available to the community. Furthermore, NUMAscope can be used to complement existing NUMA-aware profiling tools as well as system monitoring tools to aid developers trace the root cause of potential deficiencies in their applications and deployments by exposing low-level metrics obtained from ccNUMA-interconnects.

III. Background

In this Section we shortly discuss large ccNUMA systems, the importance and the role of ccNUMA interconnects, how they affect performance, and the challenges in profiling applications on systems using them.

A. Large ccNUMA Systems

Figure 2 illustrates the layout of a large ccNUMA system which consists of a number of boards or discrete servers; sometimes also called *nodes* (4 in the illustration). In general, by *large* ccNUMA systems we refer to systems larger than what the processor natively supports without a cache-coherent NUMA interconnect (see Section III-B). . Each board typically comprises 2–4 processor sockets (2 in the illustration) and a number of NUMA-nodes (4 in the illustration). Each socket comprises a number of cores and hardware threads (4 and 8 respectively in the illustration). Typically each hardware thread has it's own level 1 cache and might share higher level caches with other hardware threads on the same chip. As we move to higher levels in the cache hierarchy, the communication means changes along with the access times (see Figure 1). Additionally, each processing unit in a NUMA-node has fast access to the local memory of its NUMA-node, while it can still access the rest of the system's memory with varying latency depending on the remote memory location (see Figure 1).

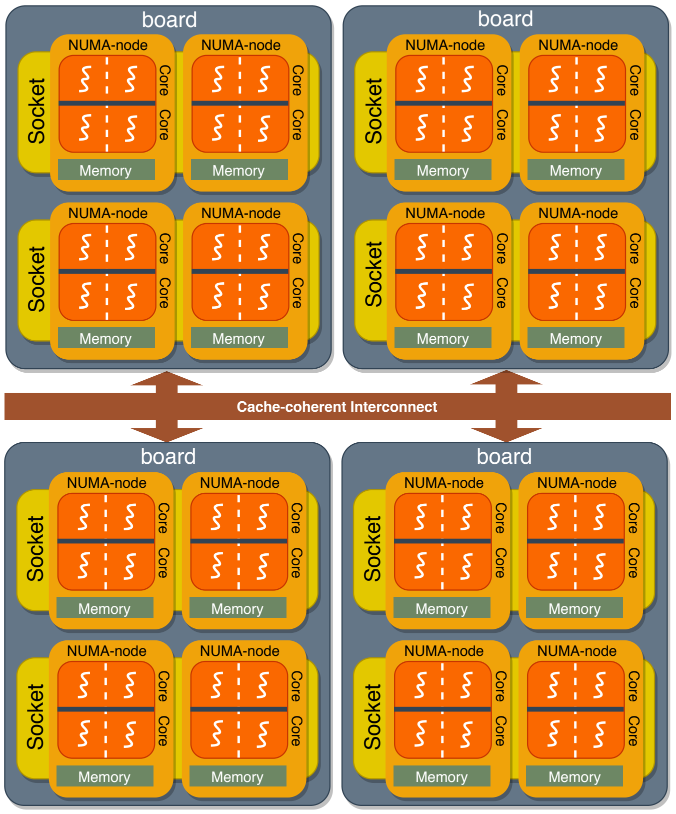


Fig. 2. An illustration of a ccNUMA system

B. Cache-coherent NUMA interconnects

For processors to be optimized for maximum efficiency, they need a common upper limit on the number of sockets they support. This dictates how many bits are allocated to each entry in a *cache directory*, to track location and state of cached lines. This is typically 8 NUMA-nodes (3 bits per cache line) [16], [17]. In addition, in some AMD processors an optimization called HT Assist reserves a fixed amount of the level 3 cache as a *cache directory*, which tracks which NUMA-node holds what cache line in what state [18]. In the AMD Opteron 6000 series the reserved, by HT Assist, space amounts to 2MB out of the 8MB of level 3 cache. Intel processors have a similar amount of level 3 cache partitioned when in multiprocessor mode.

When there is access to a cache line not tracked in the processor's cache directory, an eviction may have to occur to first invalidate a selected cache line to make space for a new entry. Since it is not known which cache holds state for the newly-tracked cache line, a *probe broadcast* must be sent to all NUMA-nodes. As the number of NUMA-nodes in a system increases, this broadcast incurs an increasing overhead, reducing application scaling [19].

In large ccNUMA systems that span multiple boards, cache-coherent NUMA interconnects are used to deliver the guarantees of coherency across the whole system. ccNUMA interconnects overcome the above limitations by being optimized for a considerably larger working set. This is achieved by tracking many more cache lines and acting as a filter, able to respond to the processor with *directed* probe responses. This improves scalability, and allows interconnecting multiple smaller (e.g. 2–4 NUMA-node) topologies to offer more processing.

ccNUMA interconnects are typically integrated onto the motherboard, or off-board in a midplane that connects to the processor fabric. This results in higher access times, compared to intra-board access times, yet significantly lower than accessing storage. In addition to the cost of visiting a remote memory due to the longer distance and the slower links in the network, the number of cocurrent remote accesses that is supported by the interconnect can also constrain the performance of the system. If the outstanding remote accesses are less than the processor can generate, applications running on ccNUMA systems may observe increased cache-coherency imposed overheads if not carefully tuned.

C. On-chip counters

On-chip counters are provided by chip designers to aid software developers to debug and optimize their applications. These counters offer measurements regarding the events occurring on different hardware modules, such as caches, branch predictors and hardware prefetchers. For instance, on-chip counters can measure the number of last-level cache misses which indicates the number of memory accesses that end up going off-chip and thus result in higher latency. Similarly, ccNUMA interconnects offer a variety of on-chip counters that count events specific to the interconnect, such as the number of outstanding remote accesses, the interconnect traffic, etc. This provides useful feedback to application developers that execute their application at larger scales than previously accessible, for example compared to a dual-socket development system. On-chip counters may also drive configuration changes in the OS, BIOS and/or firmware in the interconnect after the chip is finalized. Additionally, they provide crucial feedback to the interconnect developers on how on-chip resources should be allocated in future products.

D. Challenges

As discussed in Sections III-A and III-B, minimizing remote accesses is important for improving performance. Additionally, as discussed in Section III-B it is also important to understand resource utilization in ccNUMA interconnects so as to understand if it presents a bottleneck to the workload and how this could be reduced.

The most common mechanism of accessing on-chip counters is to access registers, directly or indirectly mapped into the application's address space. Access cycles are generated against this mapping which decode to one of the interconnect chips. Since there are many interconnect chips within the same logical system, the physical address space must have space reserved for access to all chips. Since the cache-coherent interconnects in large ccNUMA systems are proprietary, there are no standard register layouts, data formats, or even list of counters. Any tool that implements such accesses is driven by underlying implementation and design-specific details. As such, it is up to each vendor to develop an interface, documentation, or write a specific tool to access these internal counters. Finally, it is worth noting that outside the processor cores, physical addresses are always used, preventing understanding where DRAM accesses relate to.

In summary the challenges in profiling applications on large ccNUMA systems are: *a)* getting access to all the on-chip counters, both from each board and the interconnects; *b)* interpreting and normalizing the samples per unit time and clock frequency; *c)* doing the above without imposing significant overhead; *d)* exposing a way of capturing the data; and *e)* allowing useful mining and interaction with the collected data, e.g. visually. Our work tackles these challenges and provides a framework that eases the creation of tools able to monitor large ccNUMA systems.

IV. NUMAscope

NUMAscope is a framework that enables the creation of tools that gather and visualize hardware events from different modules on large ccNUMA systems. To enable extensibility, NUMAscope has been designed in a modular way, decoupling operating modes, subsystems and capturing of hardware interfaces' metrics, though doesn't need any kernel support. Figure 3 illustrates the software architecture at a high-level. As shown, NUMAscope comprises five modules.

The core module is called the *events module*, responsible for capturing hardware and software events through sampling. The rest of the modules are optional depending on the usage. In a typical session, users would first decide if they want to record the data for later analysis or not. If so, the `record` mode would be used. Inspired by the usefulness of the UNIX `vmstat` command, NUMAscope offers the `stat` mode which outputs the selected events to the terminal, allowing the users to readily observe selected metrics. Finally, the `live` mode can be used to start a web server to allow interactive graphing of the data in real-time. The *storage module* is responsible for storing the captured events when using the `record` mode. The *CLI renderer module* is responsible for formatting the captured events on the command line when using the `stat` mode. The *web server module* is responsible for interacting with the *HTML5 client* with data in real-time when using the `live` mode. Finally, the *HTML5 client* front-end is responsible for providing an interactive graphical interface to visualize in real time the data transmitted by the *webserver module*, or to visualize the data previously captured in `record` mode. In the rest of this section, we further discuss each of these modules in more detail.

A. Events module: High-rate event capture

By design, in NUMAscope, event capture is a *hot path*, wherein all state has been setup to allow reading event counters. NUMAscope can be extended to support a variety of events, either from software or hardware. Out of the box, NUMAscope is capable of capturing kernel virtual-memory events, which are not exposed via standard UNIX tools, such as `iostat`, `sar`, `vmstat` and related, or Linux-specific ones such as `perf`. These events aid in the identification of additional overhead when the executing process needs to conduct work in the kernel to take some action; for example, when accessing a page not yet faulted in from a file, or reclaiming pages to satisfy an allocation. The kernel outputs the event counters in a structured way when reading `/proc/vmstat`. NUMAscope parses the structured text, using one *lseek* and one *read* systemcall, and stores names and values in a dynamic hashtable to allow for $O(1)$ lookup. When running in `live` mode with one HTTP client, NUMAscope's memory footprint is about 14MiB of pages, 5MiB of which is shared libraries.

1) Data Biasing and Interference: To avoid interfering with applications, NUMAscope pins its threads to the first NUMA-node. NUMAscope uses a thread-pool to handle asynchronous events. When running in `live` mode with one HTTP client, NUMAscope uses 20 threads irrespective of what events are enabled or modules used. To avoid data-races, mutexes are used to ensure one thread doesn't request access while another thread is changing state of the events. Additionally, to avoid affecting the counters relating to coherent accesses, all accesses performed by NUMAscope to remote interconnects for counter collection are only non-coherent register access; this therefore doesn't evict data in the cache hierarchy. The rate of non-coherent requests is therefore in the order of thousands per second. The volume of remote non-coherent accesses running can be quantified by examining the metric *packets with less than a full cache line sent*; that is because IO accesses to registers are always 32-bit. During benchmarking, NUMAscope records a rate of $1.90{\times}10^5$ per second as compared to millions of cache-coherent cycles per second, therefore the bias is insignificant and approaching the noise threshold.

Listing 1. NUMAscope's interface for adding hardware support
```
type Sensor interface {
   // human-readable name of hardware
   Name() string
   // checks if hardware is present
   Present() bool
   // maximum sample value for percentages
   Rate() uint
   // number of hardware elements detected
   Sources() uint
   // supported events
   Events() []Event
   // activated enabled events
   Enable(discrete bool)
   // gets names of enabled events
   Headings(mnemonic bool) []string
   // returns samples
   Sample() []int64
   // used to prevent hardware access races
   Lock()
   Unlock()
}
```

2) The `Sensor` *Interface:* Regarding accessing hardware counters provided by chip developers, NUMAscope relies on the `Sensor` interface listed in Listing 1 to get access to them. Hardware counters may be platform specific, so there is no way to make NUMAscope able to access the counters of all potential hardware platforms without some form of extension. Since the design of NUMAscope is intentionally modular, adding support for new hardware counters is achieved by adding a new sub-module to NUMAscope's events module that implements the given interface for each target hardware platform.

`Name()` simply returns a human-readable description of the sensor, used in the user interface. `Present()` is used to check whether the `Sensor` is supported on the platform NUMAscope is running on. If present, it also gets set up, which typically involves the memory-mapping of the corresponding hardware registers into the process address space. `Rate()` returns the maximum number of events per second, used to normalize counter rates to percentages. `Sources()` returns the number of hardware elements detected and set up by `Present()`. `Events()` returns an array of `Events` supported by the `Sensor`, for example cycles during the sampling window and number of cache lines of data moved. `Enable()` updates internal state to activate the events to be sampled. The boolean parameter passed to it indicates whether NUMAscope should sample events per-board or sum over all boards, to increase the brevity of the CLI renderer, or the amount of data transmitted over websockets. `Headings()` returns either short event mnemonics or full descriptions useful to the developer, based on the value of the boolean parameter passed to

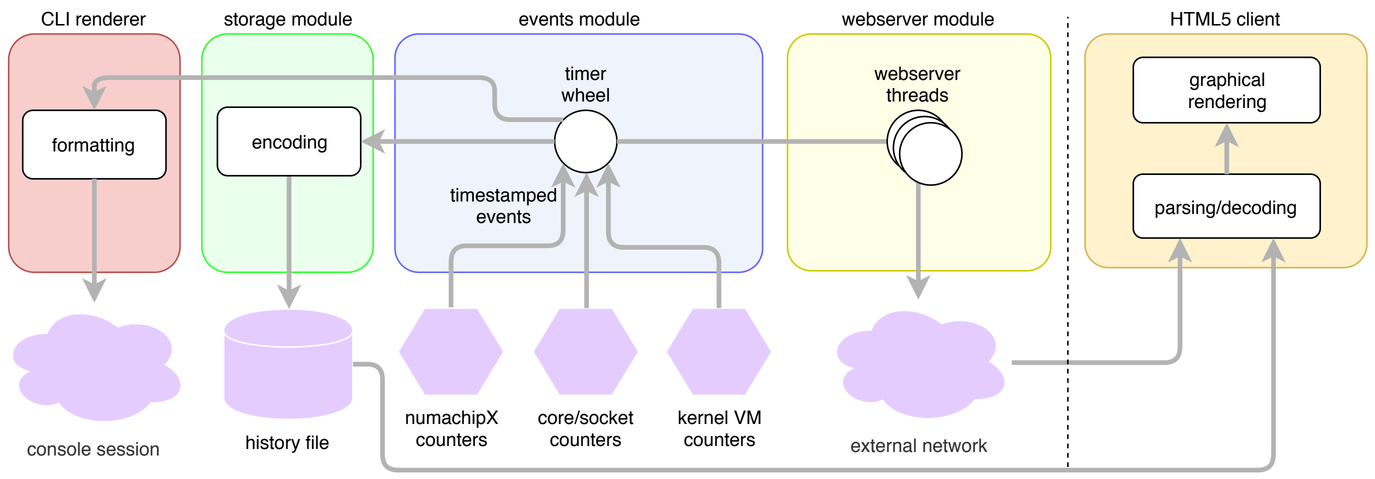


Fig. 3. High-level architecture of NUMAscope

it. Full descriptions aim to help developers in selecting which events to capture. `Sample()` reads out the enabled `Sensor` events, subtracting from the state held by the previous call to get a rate over time, and returns a slice of the enabled event rates. Signed 64-bit integers are used, since certain events can legitimately have a negative rate. For example kernel virtual memory *swapped-pages* count will increase with VM pressure, giving a positive rate, and decrease when swapped pages are read back in, giving a negative rate.

3) Controlling NUMAscope Dynamically: The events module, can also be programmatically controlled by writing commands into a UNIX FIFO at `/run/numascope-ctl`. This enables controlling NUMAscope with ease from both shell scripts and programs themselves for finer-grained profiling. The commands currently supported by NUMAscope are:

1) `record <filename>` closes any existing recording file and starts recording to given filename,
2) `label <string>` inserts a label onto the graph or textual output to mark a notable event for later analysis,
3) `pause` suspends recording events without suspending capturing,
4) `resume` continues recording events, and
5) `interval <number>ms` adjusts the sampling rate of sensors in milliseconds.

.

B. Storage module: Storing data for offline analysis

Since data may be stored and used even years after gathering them [20], using a simple and flexible but standards-compliant data storage format has very strong rationale. As such, both CSV and JSON serve the purpose well, since they are mature and well-supported by many frameworks and tools. Furthermore, both formats are not dependent on additional information such as a schema as may be expected with XML. JSON has three major advantages over CSV though. Firstly, it can encode structured data, which may be advantageous if the framework is expanded in the future. Secondly, it is native to JavaScript, so has highly optimized support without requiring external JavaScript libraries. Finally, this allows it to be consumed directly by graphing frameworks. In order to avoid making disk IO a bottleneck, the storage module writes samples in a buffer which is periodically flushed to disk by the OS; on Linux, this is every 30 seconds. Sampling at 10Hz, 28KiB of JSON test are stored per second.

Due to the modular approach taken, additional storage modules may be developed to send the data to other storage mechanisms or databases, such as InfluxDB, TimescaleDB, or Apache Cassandra.

C. CLI renderer: Command-line data presentation

The `stat` mode of NUMAscope outputs the selected events to the terminal, allowing the users to readily observe selected metrics, or pipe the output and process it on-the-fly. The CLI also enables easy monitoring of remote systems without any external dependencies. Listing 2 shows a snapshot of NUMAscope's CLI in action. In this snapshot, we see three interconnect events being monitored; `n2DirPrbRecv`, `n2CachelinesSent`, and `n2CacheRolloutRmpe`. Each line prints the values obtained by a single sample. Over time new lines are printed with the latest samples. In this example, we observe how many *cacheline probes* (snoops) and cache lines are sent over the interconnect per second, and how many level-4 (internal to the interconnect) cache line evictions or *rollouts* occurred per second; this it useful to see how much remote activity the workload is generating to understand how the *working set* interacts with the level-4 cache

Listing 2. Snapshot of NUMAscope's CLI

```
$ numascope stat
n2DirPrbRecv n2CachelinesSent n2CacheRolloutRmpe ...
          41             7357               6689 ...
         103            21357              19092 ...
       11560           115101              78575 ...
      258872           944292              72099 ...
      387750           598625               6843 ...
      375612           583840               8220 ...
```

D. web server module: Live data streaming for interactive graphing

NUMAscope provides a standalone graphical interface, directly served via the built-in HTTP web server to allow immediate live updates to be observed. Additionally to HTTP webserving, data is exposed to clients via websockets, on top of which runs a simple yet custom protocol. This way, two-way interaction between clients and the web server is possible, overcoming the one way limitation with HTTP. One example where this is useful, is given multiple live sessions capturing data from NUMAscope, to control options that are global to all clients. This is chiefly the rate at which NUMAscope latches the on-chip counters and reads them out. Since these can't be interpolated, all clients must use the same sample interval. Websockets allow the client to send data back to the web server, which in turn informs all clients of a configuration change. Since events can be captured at a higher rate than the network round-trip would easily accommodate when in live mode, events are batched and sent down all client websockets at 400 millisecond intervals; this is a good trade-off of network and processing overhead, particularly as the client which needs to re-render graph traces frequently.

E. HTML5 client: Interactive graphing of data

For the interactive visualization of the measurements, NUMAscope offers a graphical user interface (GUI) realized through web technologies. Given the ubiquity of web technologies, how standardized and optimized they are today, developing a native client application would present comparatively an intractable engineering and support burden. Since the *rate* of events is graphed, the *cycles* counter is used to measure the time period that sampling was running; the values are divided by the time period to get an accurate rate over the unit of time. This data is transferred either via websocket or JSON file, and optionally reducd over all the boards to conserve bandwidth or space. Finally, it is passed to the graphing framework, which renders the graph traces.

Figure 4 shows a screenshot from NUMAscope's GUI in action. At the top of the interface there is a panel with controls allowing the user to adjust the visualization of the data. The user may control the recording of the data, the sampling rate, and the visualization itself. Additionally, there is a load button for loading data from a file instead of the web server. Directly below the control pane, there is an interactive plot visualizing the gathered events in a time-series. The typical representation of such time-series data is by using one or more line graphs. The interactive plot allows the user to select which events to be visualized, to focus on a specific time window, and even see the exact value of an event by hovering over the corresponding line on the plot. Finally, the user can toggle the visibility of traces in the graph by clicking them in the legend.

TABLE I
HARDWARE AND SOFTWARE CONFIGURATION

CPUs	18 (3 per board) AMD Opteron 6328
NUMA-nodes	36 (6×6 per board)
Memory	2304GiB (384GiB×6 per board)
Boards	6
CC-interconnect	NumaConnect2
OS	CentOS 7.6
Linux Kernel	4.14.146-NUMASCALE [21]
GCC	4.8.5 (w/ and w/o patch)

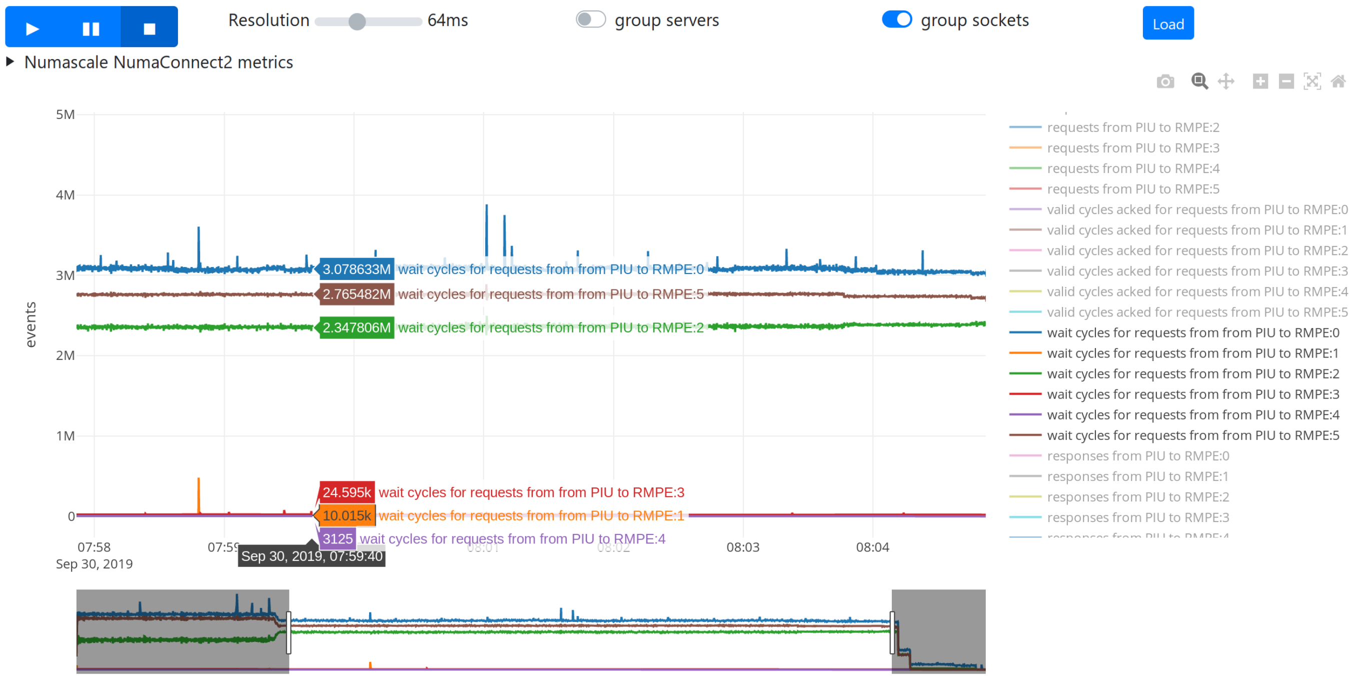
V. Evaluation

To evaluate NUMAscope, we deploy it on a 6-board, 18-socket AMD Opteron system with 6 ccNUMA interconnects and use the *Embarrassingly Parallel* (EP) benchmark from the NASA Parallel Benchmark (NPB) suite [2] to:

1) Demonstrate how NUMAscope helped us find a performance deficiency in GNU C compiler's (GCC) OpenMP implementation.
2) Assess the overhead imposed by NUMAscope.

Each board on the system comprises three AMD Opteron 6328 processors, each featuring 8 cores. The NUMA topology is shown in Figure 5a. The boards are connected using Numascale's NumaConnect2 cache-coherent interconnect, in a point-to-point configuration for single-hop latency, as shown in Figure 5b. The links between the NUMA-nodes in the processors supports 6.4GT/s (Giga Transfers) peak throughput; the links between the processors and interconnect supports 3.2GT/s. Table I summarizes the hardware and software configuration of the evaluation platform.

In our benchmarking, we use *thread pinning* to minimize non-determinism so results are stable and repeatable, and to prevent the kernel scheduler from waking threads up in different areas of the cache hierarchy. Part of the reason the latter occurs, is from the increased clock jitter due to different boards in the larger ccNUMA system that are driven by different on-board clocks. One strategy to partially mitigate this is to boot Linux with the `relax_domain` parameter set to 3 [22]; this restricts the kernel's process scheduler to limit the search for an unused core to a particular level in the hierarchy (a



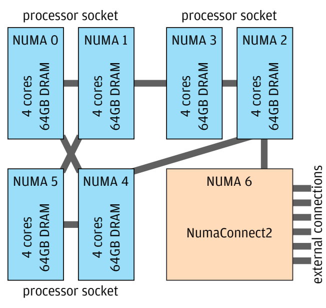
Resolution 64ms

group servers

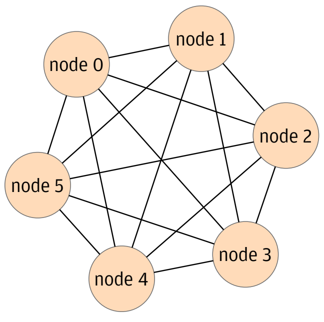
group sockets

Load
Numascale NumaConnect2 metrics
events
5M
4M
3M
2M
1M
0
07:58
Sep 30, 2019
07:59
08:03
08:04
Sep 30, 2019, 07:59:40
3.078633M wait cycles for requests from from PIU to RMPE:0
2.765482M wait cycles for requests from from PIU to RMPE:5
2.347806M wait cycles for requests from from PIU to RMPE:2
24.595k wait cycles for requests from from PIU to RMPE:3
10.015k wait cycles for requests from from PIU to RMPE:1
3125 wait cycles for requests from from PIU to RMPE:4
requests from PIU to RMPE:2
requests from PIU to RMPE:3
requests from PIU to RMPE:4
requests from PIU to RMPE:5
valid cycles acked for requests from PIU to RMPE:0
valid cycles acked for requests from PIU to RMPE:1
valid cycles acked for requests from PIU to RMPE:2
valid cycles acked for requests from PIU to RMPE:3
valid cycles acked for requests from PIU to RMPE:4
valid cycles acked for requests from PIU to RMPE:5
wait cycles for requests from from PIU to RMPE:0
wait cycles for requests from from PIU to RMPE:1
wait cycles for requests from from PIU to RMPE:2
wait cycles for requests from from PIU to RMPE:3
wait cycles for requests from from PIU to RMPE:4
wait cycles for requests from from PIU to RMPE:5
responses from PIU to RMPE:0
responses from PIU to RMPE:1
responses from PIU to RMPE:2
responses from PIU to RMPE:3

Fig. 4. Screenshot of NUMAscope's GUI



(a) NUMA, one board



(b) Interconnect, all boards

Fig. 5. Topology at different levels

super-set of the NUMA hierarchy). To pin the application threads linearly to the core numbers with OpenMP, the environment variable `OMP_PIN_THREADS` is set to `TRUE`. Since the evaluation is performed on a dedicated system, in conjunction with thread pinning, the only measurement noise comes from OS housekeeping activities.

A. Numascale NumaConnect2 support

The Numascale NumaConnect2 ccNUMA interconnect allows interconnecting multiple AMD Opteron 6300 processor boards, providing a level-4 cache and full directory to reduce global traffic. It allows point-to-point, 3D torus or arbitrary fabric topology which it load-balances over, using open-source firmware [23]. NumaConnect2 lists all the types of cycles entering and leaving the card, and has counters at most interfaces between functional blocks to accrue how many clock cycles were consumed by waiting. This includes cache line probes, cache line transfers, IO transfers, as well as level-4 cache hits, misses, evictions and time spent waiting for resources internally in the interconnect. To expose to NUMAscope the hardware events gathered by NumaConnect2 we implement the `Sensor` interface described in Section IV-A as a new submodule.

The NumaConnect2 counters profiled by NUMAscope fall into two categories. Firstly, *extrinsic events*, which include the types of traffic going into the NumaConnect2 from both the processor and interconnect fabric sides. Understanding this traffic may reveal if an application is exploiting or thrashing the cache hierarchy, which is useful in algorithm design and application tuning. Workloads thrashing the cache hierarchy would result in considerably more access over the interconnect, for example random access to a working set somewhat larger than the caches. Applications exploiting the cache hierarchy would generate considerably less activity on the interconnect either by accessing a working set that fits in the caches, or by accessing local data. Secondly, the *intrinsic events* which includes internal resource usage, or counting cycles for which a request is stalled waiting for resources. This helps understand if and when bottlenecks occur in the interconnect.

B. Workloads and metrics

The NASA Parallel Benchmarks [2] present a range of numerical problems with varying characteristics. Additionally, varying problem sizes can be configured and the number of threads of execution can be arbitrarily specified, so as to make full use of resources.

Using the NPB EP benchmark allows us a means to validate NUMAscope. The EP class D benchmark generates 2×10^{36} pairs of pseudorandom numbers with a Gaussian distribution in a 1.1GiB distributed memory array. In this paper, we analyze the amount of time the interconnect is waiting for internal resources.

From a whole-system perspective, it would be intuitive to measure the number of remote outstanding transactions at the interconnect. Since this is limited in hardware to 16, we can not get insight into the end-to-end queuing occurring from the processor cores to the remote DRAM. Instead, a linear measure of queuing is "wait cycles for requests from SIU to RMPE"; this measures the number of clock cycles spent for cache-coherent requests traveling from the NumaConnect2 *Scalable Coherent Interface* (SCI) *Interface Unit* to the *Remote Memory Processing Engine*; this shows exactly how long memory access was stalled because the interconnect was busy.

C. GCC's OpenMP implementation deficiency

During execution of the NASA Parallel benchmarks, we observed using NUMAscope, that the benchmarks were causing an unexpectedly high number of cycles waiting for resources in the interconnect, even with the NPB *Embarrassingly Parallel* (EP) benchmark. As implied by its name, the EP benchmark has no *data dependencies* among parallel threads, so minimal communication is expected to take place, and memory accesses should be mostly local or cached. Figure 6a plots, on the y-axis, the number of cache-lines transferred over the interconnects (blue line), along with the percentage of the "wait cycles for requests from SIU to RMPE" over the theoretical maximum of the system (brown line); the x-axis is time in seconds. Given the 200MHz core clock speed of NumaConnect2 the maximum number of "wait cycles for requests from SIU to RMPE" that can be observed is 0.2×10^{9} per interconnect. To calculate the plotted percentage we use the following equation:

$$100\times\sum_{n=1}^{\#interconnects}\frac{\text{wait\_cycles}_n}{0.2e9} \tag{1}$$

where wait_cycles_n is the "wait cycles for requests from SIU to RMPE" measurement obtained from interconnect n. As shown in Figure 6a, there is a constant rate of about 99% (out of the maximum 600%) of the total number of clock cycles spent waiting for resources across all six boards. This observation suggests that one of the six interconnects is acting as the bottleneck, at constant saturation.

Disabling the grouping of the boards in the user interface, we reveal the wait cycles per board, as shown in Figure 7. In a well-balanced workload like EP, we would expect similar values for each board. However, from Figure 7, it can be seen that the wait cycles are almost all from the second board. The second board sees about 97% time spend in wait cycles, while the first board sees about 2.5%, with the other boards experiencing negligible contribution. This illustrates a *highly imbalanced* access pattern, perhaps one that would fit with a software placement bug. This ultimately led to the discovery that GNU C compiler's OpenMP thread pinning was working, but was touching each thread's stack in the parent thread. This resulted in stack pages being page-faulted on the parent-thread's NUMA-node. Since the stack is intensively used, this generates considerable cross-interconnect traffic. To mitigate this a workaround was prepared to identify the correct NUMA-node and pin the stack explicitly [24].

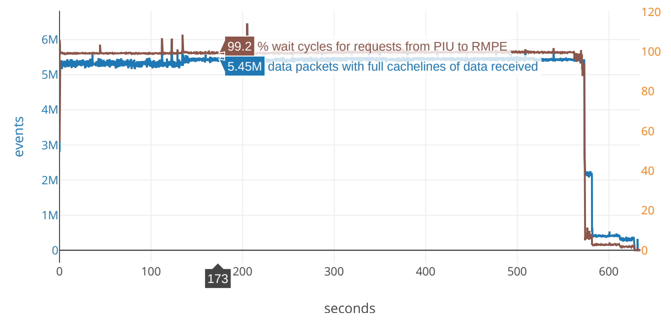
Recompiling EP with the patched GCC and rerunning the benchmark, run-time dropped from 634.4s to 112.1s; a factor of 5.7× speedup. Correspondingly, the interconnect traffic dropped from a total of 17.8GiB to 1.14GiB; a factor of 15.6× less interconnect traffic. Figure 6b shows the corresponding plot. We see that in the patched run, number of cache-line transfers reaches 6×10^{6} per second during initialization and when the parallel work starts they drop to 3.80×10^{5} steady-state. Correspondingly, the wait time drops to about 2%.

D. Tool overhead

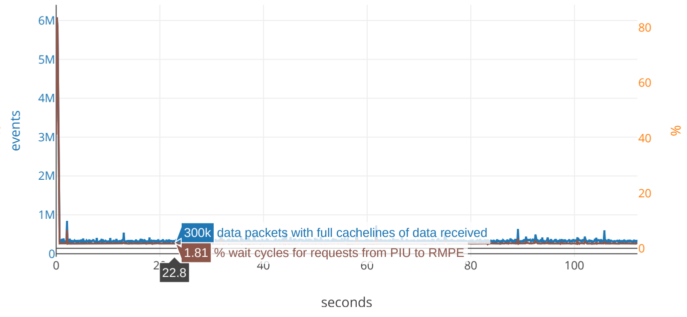
To determine the overhead of NUMAscope, we execute the NPB EP class D benchmark without using NUMAscope, and again using NUMAscope in `record` mode. We demonstrate the overhead in `record` mode since this is the only mode expected to be used for long periods of time, e.g. in production. Figure 8 plots the slowdown of the benchmark as we increase the sampling interval from 1ms to 1s. We observe that the maximum overhead on average is about 5% with a sample interval of 1ms (1KHz), while the minimum is 0.27% with a sample interval of 1s (1Hz). In typical use scenarios, for short period, a sample interval of 100ms gives sufficient resolution for helping to understand application and interconnect behavior, and gives low overhead due to the noise mitigation steps detailed previously, and the access to remote NumaConnect adapters being with non-coherent cycles, therefore *cache oblivious*. Note that the sampling interval can be dynamically changed the `/run/numascope-ctl` UNIX FIFO. That said NUMAscope can be initially configured to sample at a 100ms interval and if something alerting is observed the sampling interval can be further increased to investigate.

VI. Conclusions

NUMAscope is an open-source [25] framework that enables the creation of tools that allow the monitoring of large ccNUMA systems. NUMAscope is designed to be modular and hardware agnostic, to support a number of ccNUMA interconnects in the market. By implementing



(a) Without stack fix



(b) With stack fix [24]

Fig. 6. NPB EP class D pinned OpenMP wait cycles

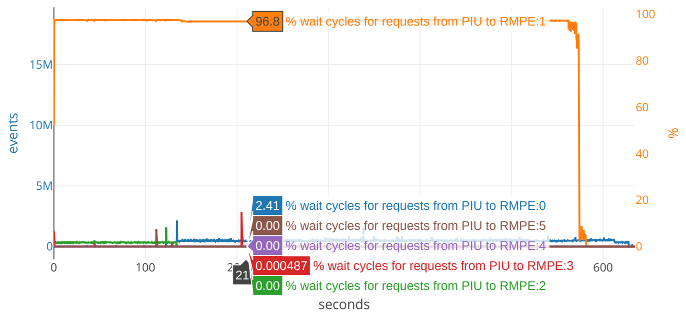


Fig. 7. NPB EP class D pinned OpenMP wait cycles per board

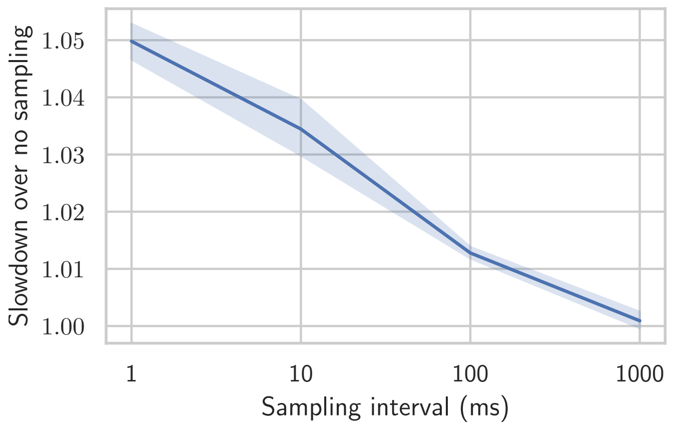


Fig. 8. NUMAscope overhead with various sampling rates

a simple interface, vendors can add support for newer interconnets and expose measurements from hardware counters. NUMAscope can then capture various events, from kernel events, to hardware events and visualize them in an interactive GUI. As demonstrated in this work, tools based on NUMAscope can help developers understand their applications' bottlenecks and fix them. NUMAscope's overhead is shown to be negligible when used with a high sampling interval (1s), and it can go up to about 5% for a sampling interval of 1ms. That makes NUMAscope-based tools production-ready.

References

[1] D. Hackenberg, D. Molka, and W. E. Nagel, "Comparing cache architectures and coherency protocols on x86-64 multicore smp systems," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2009, pp. 413–422.

[2] D. Bailey, T. Harris, W. Saphir, R. Van Der Wijngaart, A. Woo, and M. Yarrow, "The nas parallel benchmarks 2.0," Technical Report NAS-95-020, NASA Ames Research Center, Tech. Rep., 1995.

[3] "Perf tools." [Online]. Available: https://github.com/torvalds/linux/tree/master/tools/perf

[4] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with papi-c," in *Tools for High Performance Computing 2009*, M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 157–173.

[5] "Likwid." [Online]. Available: https://github.com/RRZE-HPC/likwid

[6] C. McCurdy and J. Vetter, "Memphis: Finding and fixing numa-related performance problems on multi-core platforms," in *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, March 2010, pp. 87–96.

[7] A. Kleen, "Pmu tools." [Online]. Available: https://github.com/andikleen/pmu-tools

[8] U. Prestor and A. L. Davis, "An application-centric ccnuma memory profiler," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, Dec 2001, pp. 101–110.

[9] J. S. Gibson, "Memory profiling on shared-memory multiprocessors," Ph.D. dissertation, Stanford University, 2002.

[10] K. Fatema, V. C. Emeakaroha, P. D. Healy, J. P. Morrison, and T. Lynn, "A survey of cloud monitoring tools: Taxonomy, capabilities and objectives," *Journal of Parallel and Distributed Computing*, vol. 74, no. 10, pp. 2918–2933, 2014.

[11] X. Liu and J. Mellor-Crummey, "A tool to analyze the performance of multithreaded programs on numa architectures," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '14. New York, NY, USA: ACM, 2014, pp. 259–272. [Online]. Available: http://doi.acm.org/10.1145/2555243.2555271

[12] R. Lachaize, B. Lepers, and V. Quema, "Memprof: A memory profiler for NUMA multicore systems," in *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX, 2012, pp. 53–64. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/lachaize

[13] M. Selva, L. Morel, and K. Marquet, "numap: A portable library for low-level memory profiling," in *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, July 2016, pp. 55–62.

[14] X. Liu and B. Wu, "Scaanalyzer: A tool to identify memory scalability bottlenecks in parallel programs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2807591.2807648

[15] A. Yasin, “A top-down method for performance analysis and counters architecture,” in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2014, pp. 35–44.

[16] J. P. Shen and M. H. Lipasti, *Modern processor design: fundamentals of superscalar processors*. Waveland Press, 2013.

[17] A. M. Devices, “Opteron family 15h bios and kernel developer guide.” [Online]. Available: https://www.amd.com/system/files/TechDocs/42301_15h_Mod_00h-0Fh_BKDG.pdf

[18] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes, “Blade computing with the amd opteron™ processor (" magny-cours"),” in *2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE, 2009, pp. 1–19.

[19] N. Agarwal, L. Peh, and N. K. Jha, “In-network coherence filtering: Snoopy coherence without broadcasts,” in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2009, pp. 232–243.

[20] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, “Profiling a warehouse-scale computer,” in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA ’15. New York, NY, USA: ACM, 2015, pp. 158–169. [Online]. Available: http://doi.acm.org/10.1145/2749469.2750392

[21] D. J. Blueman, “Numascale kernels.” [Online]. Available: https://resources.numascale.com/kernels/

[22] ——, “Numascale wiki.” [Online]. Available: https://wiki.numascale.com/tips/os-tips

[23] ——, “Numaconnect firmware.” [Online]. Available: https://github.com/numascale/firmware

[24] S. Persvold, “Gcc 4.9 openmp off-stack fix.” [Online]. Available: https://resources.numascale.com/gcc49-local-stack.patch

[25] D. J. Blueman, “Numascope.” [Online]. Available: https://github.com/numascale/numascope