sys-sage: A Unified Representation of Dynamic Topologies & Attributes on HPC Systems

Stepan Vanecek
Chair of Computer Architecture and Parallel Systems
Garching, Germany
stepan.vanecek@tum.de

Martin Schulz
Chair of Computer Architecture and Parallel Systems
Garching, Germany
schulzm@in.tum.de

ABSTRACT

HPC systems are getting ever more powerful, but this comes at the price of increasing system complexity: node architectures are deeply hierarchical and in many cases heterogeneous, and components can interact with each other in unpredictable ways. Further, current and future systems exhibit increasingly dynamic behavior, making static knowledge of their configuration alone insufficient. To use such systems efficiently, users as well as runtime systems have to be aware of the exact hardware structure at any time, i.e., the systems topology, its configuration parameters, and any side-effect a component can have on the rest of the system, and how this changes over time.

Current approaches to providing such information usually focus on a single aspect and do not consider dynamic behavior. For example, the widely used *hwloc* library, the current de-facto standard solution for retrieving hardware topology information, provides a static hierarchical view of all node hardware, but neither covers other system configuration aspects nor dynamic behavior; other systems have similar limitations.

In this paper, we propose *sys-sage*, a novel approach that overcomes these limitations and goes beyond the functionality of existing tools, including *hwloc*. It offers the ability to track dynamic changes, while unifying access to all system topology and configuration data. With that, it provides, at any point in time, a complete and updated view of the HPC system on which an application or runtime system is executing. The novelty of our approach lies in the ability to combine static hardware topology information with other relevant system data in a single API, while enabling a dynamic view and exposing system updates and reconfigurations on the fly. We show the design of *sys-sage* and demonstrate its applicability based on three separate use-cases, as well as by presenting further scenarios not easily solvable with currently available tools.

CCS CONCEPTS

• **Computer systems organization** → **Architectures**; • **Computing methodologies** → *Modeling and simulation*; *Parallel computing methodologies.*

ICS '24, June 4–7, 2024, Kyoto, Japan

ACM ISBN 979-8-4007-0610-3/24/06…$15.00
https://doi.org/10.1145/3650200.3656627

KEYWORDS

HPC System Topology, Hardware Architecture, Heterogeneous Computing, Performance Optimizations.

ACM Reference Format:
Stepan Vanecek and Martin Schulz. 2024. sys-sage: A Unified Representation of Dynamic Topologies & Attributes on HPC Systems. In *Proceedings of the 38th ACM International Conference on Supercomputing (ICS '24), June 4–7, 2024, Kyoto, Japan.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650200.3656627

1 MOTIVATION

High-Performance Computing (HPC) architectures have evolved into complex systems built of diverse and heterogeneous multi-core, vector, and GPU processing units. This design trend has led to increased architectural complexity of both chip and node designs as well as complex dynamic behavior when executing code. Fully utilizing such architectures has become a challenge and requires a deep understanding of the system's architecture, from its hardware structure to its dynamic behavior. Users and runtime systems alike have to be able to query this information – both static and dynamic – to adjust their execution behavior and to optimize for the underlying platform. Currently, however, there is only limited access to such information. Further, the access is split across many independent libraries and often restricted to static configuration data.

One of the most commonly used tools in this context is the *hwloc* library [6], the current de-facto standard for retrieving the hierarchical hardware topology of a node. Hwloc builds a tree data structure, representing the static hardware architecture of a node with its building blocks, such as cores, caches, or NUMA (Non-Unified Memory Access) regions. Hwloc's primary goal is to provide this representation in an easy-to-use way to reveal which cores share which resources, e.g., to support decisions on which applications benefit from being scheduled together or which yield better performance when running on more isolated instances. Such scheduling decisions are then made by high-performance runtime systems, like the ones shipped alongside OpenMP or MPI implementations. However, the approach is designed for static data only and limited to (CPU) topology information.

However, the growth in complexity is continuing and is far from reaching its peak, making a static and strictly hierarchical view on topology data alone insufficient. For example, optimizing the use of on-chip networks requires more complex information, such as node distance, not expressable in a static tree. Another example is that modern architectures offer features for sharing and isolating available resources, like multiple power domains, SMT capabilities, or dynamic cache and bandwidth partitioning [21],

which are runtime configurable and hence can lead to dynamic configuration changes in resource availability. Further, emerging memory technologies, such as HBM or disaggregated memory, form heterogeneous memory hierarchies with their own topologies, replacing simple DRAM blocks. On top of these complexities, many state-of-the-art systems feature heterogeneous, accelerated node architectures (nine of the top ten machines in the recent Top500 list [2]) — with GPUs being the most common, but by far not the only type of accelerator. The high complexity is even amplified with components, such as smart NICs, which have their own independent complex internal architecture, making networking more complex as well.

Overall, not only has the complexity, variety, and amount of components in an HPC system increased (and is predicted to further increase drastically), but also their usage is becoming increasingly dynamic, with architectural features being software-controllable and hence adjustable as needed for a specific workload.

As a consequence, there is a need for a new solution that goes beyond the functionality of existing tools. This solution must be able to combine, under one unified roof, the strengths of the existing tools, providing selected yet highly valuable information for subparts of the overall problem space and extend it with support for new hardware resource types and structures as well as architectural features. Further, such a solution must also be able to incorporate dynamic information — be it changing hardware resources, software configurations, or system status data indicating current resource usage and/or sharing — as well as mechanisms to influence dynamic settings. The resulting knowledge base and configuration mechanisms must then be made available to users and runtime systems to be used for optimization using a unified API concept.

In this paper, we present ***sys-sage***, a novel approach that addresses the needs stated above and is publicly available as an open-source **library implementation** on GitHub[1]. It enables — in a unified fashion — acquiring, storing, updating, and querying all relevant information about the hardware and software configuration of a node/system, including its (possibly dynamic) hardware topology, its dynamic state and current configuration, its capabilities, and other information related to the system and/or software setup, all of which are logically connected to each other. It builds upon and generalizes the concepts of hwloc to address its limitations for modern systems and workloads.[2] *sys-sage* integrates data from existing data sources, such as hwloc, provides other (possibly dynamic) frequently needed information, and enables logical connection of different topology-related pieces of information in its internal representation. We demonstrate the universality and usability of *sys-sage* by presenting **three use-cases from different areas** — CPU resource sharing, performance analysis tools, and GPU performance estimation — where *sys-sage* is employed.

[1]https://github.com/caps-tum/sys-sage

[2]We present the concepts of *sys-sage* as a standalone approach for the purpose of this paper and implement it as a separate library, but it could be integrated into approaches, like hwloc.

2 DATA DISCOVERY APPROACHES AND *SYS-SAGE*

Nowadays, there are countless tools, APIs, interfaces, and general instruments that provide some partial information about an HPC system, its components (such as nodes or CPUs), and its behavior. Some of them (such as hwloc or `nvidia-smi`) provide mainly static data describing the system.[3] Others (such as DCDB [25], PAPI [24], or LDMS [5]) provide dynamic information and metrics describing the behavior of a system or applications running on it, but then rely on other sources to map the data to static context information. A third type of information can be exposed via quantitative measurements, e.g., using targeted benchmarks exploring individual components' attributes, such as cache sizes. It can also help guide optimizations and should be included in a generic approach, yet it is not included in existing tools and must be added manually during the subsequent data analysis process.

All existing approaches focus on certain, specific areas, and extending them with additional information can be very cumbersome or impractical due to their often specialized API designs. On top of that, there are dynamic system characteristics, which also need to be considered, even if data is considered static. Examples of this are hardware resources that can be subject to change through software reconfiguration in modern systems (e.g., cache sizes, bandwidth limits, or power caps) or that exhibit a smaller effective size when shared with other workloads.

Depending on the kind of data, it needs to be collected pre-run (it does not change), during the startup (does not change during runtime), or continuously at runtime, and then all data sources need to be combined in a single data representation. Only then can we gain a full understanding of a system's state and its components. For this we need information from all these sources and life-cycle stages, we need to understand how they relate to each other, and users need to be able to access them in a unified manner to keep it manageable and provide consistency. To complicate things further, each application/runtime system/daemon/tool relying on this data is unique and needs a different subset of the above-mentioned information. Therefore, any approach needs to be flexible regarding what information is included and how to extend it, to allow easy customization based on the targeted use case and the addition of new/different data sources.

sys-sage is designed to fulfill these requirements and unifies different data sources and their respective native APIs into a single tool. With that, it supports a wide range of different use cases with diverse usage characteristics. *sys-sage* covers the acquisition, processing, and querying of hardware- and system-relevant data from an extensible set of arbitrary sources throughout the entire application lifetime, as presented in Figure 1. This is accomplished at its core using a novel approach in the form of a dynamic graph-based data store able to represent arbitrary data alongside the needed relationships between data points.

As a result, *sys-sage* is the first to enable:

(1) Building and manipulating custom topologies,
(2) Fully reflecting the dynamic properties of a system,

[3]While these tools may also provide some dynamic information, their focus and design is geared towards static data, and dynamic data is typically limited and provided via rudimentary add-ons developed later, and therefore limited in scope and usability.

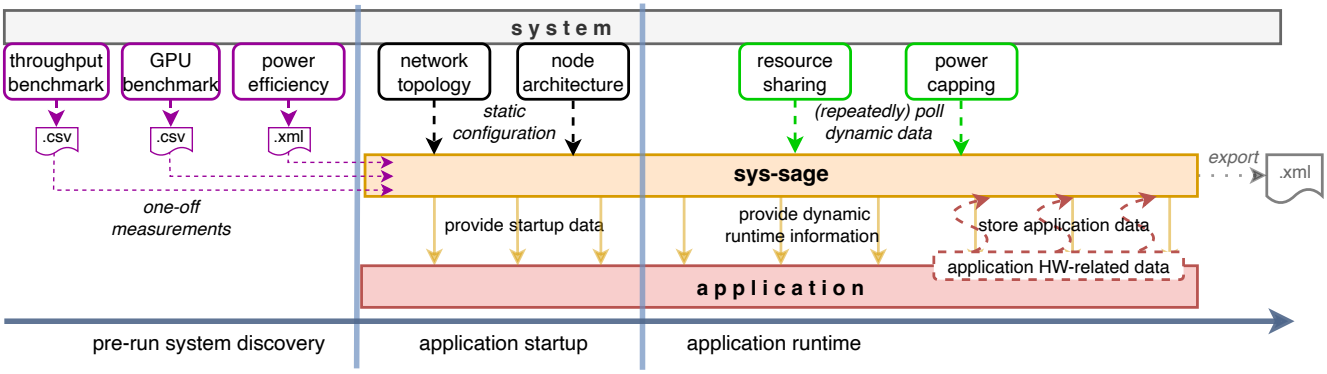


Figure 1: Workflow of integrating different data to *sys-sage*.

(3) Representing more complex information regarding relations of the system's components, and
(4) Representing heterogeneous systems, also covering GPUs and other non-CPU architectures.

2.1 Use-Case Descriptions

For the design of *sys-sage*, we consider a wide variety of use-cases across different disciplines and areas of HPC. They range from scheduling (from node- to core-granularity) through resource sharing, heterogeneous and deep memory systems, and energy efficiency to modeling and simulation tasks. To address all these problems, we require information that is very complex or impossible to retrieve using the available tools, such as hwloc, and we then provide it through a unified interface offered by the tool-facing side of *sys-sage*.

The first example is job scheduling based on energy efficiency and network topology. Prior work has been done on both topics (such as [9, 31] for network topology and [3, 7] for energy efficiency). However, each topic has been considered individually, which results in conflicting scheduling strategies. Therefore, only one option could be used. Integrating *sys-sage* will allow representation and correlation of the network topology and node-/socket-/core-based energy efficiency information. By balancing these two aspects, job schedulers can make better scheduling decisions. The job would run primarily on the efficient nodes (higher FLOPs under a given power cap) while increasing the network overhead as little as possible. As a next step, each application could weigh the parameters differently based on its cross-node communication intensity.

In the second example, we consider on-node scheduling. State-of-the-art chips have highly complex internal designs, such as mesh topologies, going far beyond the strict hierarchy. Consequently, some cores are faster when accessing a particular memory type or a GPU. Hence, we see different performance based on core allocation in some scenarios[4]. Dynamic power-capping introduces another level of complexity. An application utilizing some threads to do compute-intensive tasks, and others to write frequently to memory, needs to find optimal hardware mapping. The compute-intensive threads should be on cores with the highest power cap (increasing FLOPs), the data-intensive ones on cores with the highest memory bandwidth. To achieve that, we need to represent the system topology (hwloc), the core-to-memory data transfer capabilities (bandwidth benchmarks), and the current (core-specific) power caps (variorum); and to correlate this information (in *sys-sage*), coming from three different sources/APIs.

The following scenario considers resource sharing. On a system with GPUs shared between two (or more) jobs, we need to decide how to split the node. Each NUMA region has a different latency when moving data between the CPU and the GPU. If NVIDIA GPUs are split using Multi-instance GPU (MIG) [1], we need to find a GPU with a sufficient amount of GPU Instances (GIs) for our GPU kernel[5]. The solution is to find a GPU with sufficient GIs and assign the CPU-side workload to cores with the lowest CPU-GPU latency – to benefit both from sufficient GPU compute power and low data transfer latency. Therefore, we need to combine (again in *sys-sage*) the information from hwloc (NUMA-core mapping), benchmark results (NUMA-GPU latency), and NVML (current GI availability) into one whole to find an optimal allocation solution and achieve optimal performance.

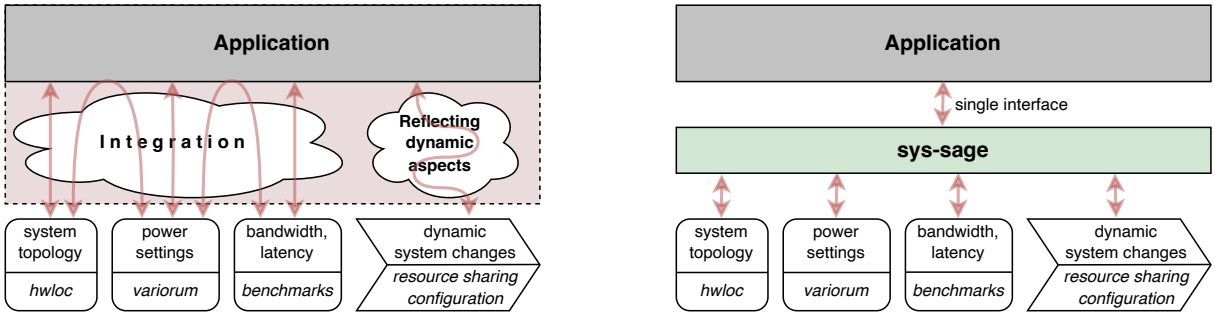
Finally, in many scenarios, users simulate or model behavior of an application on a different system. To define the arbitrary topologies or system attributes of the modeled system, they need to have a system representation that is versatile and can combine static and dynamic system topology information and properties. In this scenario, *sys-sage* can be the backend to choose to represent the modeled system topology with all the attributes relevant for the computation.

2.2 Functionality Scope

In the context of this paper, a 'user' of *sys-sage* is any resource manager, daemon, user-side application, etc., storing data to or retrieving data stored in the library. Each user of *sys-sage* has widely different requirements on what the library should provide, i.e., what hardware- or system-relevant information is needed and what is redundant. We gain the flexibility for such a broad coverage by

[4]The relevance of these differences has been shown in scenarios, such as the one by Ramos et al. [10, 28].

[5]For example, to avoid register spilling or to have sufficient memory/L2 cache size.



(a) Traditionally, an application has to take care of all the integration, storage and connection of the information.

(b) *sys-sage* serves as a backend for HW-related data and offers a single interface for the application.

Figure 2: Integrating HW-related data with and without *sys-sage*.

logically decoupling the core tasks. We split the workflow into three stages:

(1) Collecting the relevant information (from any source) and mapping it to a common representation,
(2) Maintaining the information – different kinds of information (static/dynamic, qualitative/quantitative, variable/constant, ...), collected from multiple data sources, regarding different components of the system – so that all data forms one logical structure, and
(3) Providing the data to the user in a unified fashion.

As mentioned above, many sources of HW- and system-relevant data are being used today — provided by applications, the OS or drivers, or by specially tailored benchmarks, to name a few typical examples. *sys-sage* does not aim to replace these methods; it rather uses the information they provide as one of its many data sources, and simplifies and unifies the way the data from different sources is combined, correlated, and offered to the user. Hence, mainly **(2) maintaining and (3) providing the information are in the focus of *sys-sage***, while its design allows (1) importing any relevant information from any existing source by supporting flexible mapping mechanisms into the *sys-sage* data representation.

2.3 Integration with *sys-sage*

Traditionally, an application needing hardware- or system-related information specifically has to integrate a separate tool/API for each data source or targeted component. Moreover, it also has to provide the integration and correlation of these data sources with each other in order to make sense of the combination of the needed data. This is illustrated in Fig. 2a. For instance, hwloc tells us which CPU cores map to which NUMA region. Another tool provides us with a power efficiency ranking of each core. Neither of the pieces of information alone can tell us which cores to select when scheduling one core per NUMA region; only when combined and correlated, we gain the needed insight.

sys-sage simplifies this task, as shown in Fig. 2b. Its design connects the partial information (or information regarding different system components) to present a complete view of the system. It provides a unified data and storage model that is used to store the independently obtained static and dynamic system information. The user can then retrieve all the information from this unified data model via a single interface. Once implemented, a procedure for integrating data sources into *sys-sage* can be reused by multiple applications — each possibly using a different subset of these data sources. Finally, dynamic information about system state and system changes needs to be stored somewhere, too — *sys-sage* handles this task as well. The design of *sys-sage* enables the applications to benefit from more precise information about a system, while drastically simplifying the implementation and integration process.

While certain types of information are needed and expected in many use cases, such as the number of CPU cores, other information may be completely use-case-specific, such as profiling output (cf. Sec. 5.2). To get the best out of both worlds, *sys-sage* offers the frequently-needed information out-of-the-box, while enabling the users to add more specific data via a simple API.

The novelty of our approach lies in identifying the need for providing correlated hardware information from multiple sources in a unified way. The implementation of *sys-sage* enables reusing and combining data regarding various (static and especially dynamic) aspects of different components, appearing in different stages of the application life-cycle, using a single interface. Our library builds a logical relation between the available information, simplifying the navigation through the quantity of the data while extracting the needed information.

3 DESIGN OF *SYS-SAGE*

The design of our library *sys-sage* reflects the requirements presented in the previous section, and, in particular, reflects the three-stage approach that enables the decoupling of data acquisition, storage, and delivery. Further, *sys-sage* is designed to be expandable, so future developments in HPC architectures will not make our approach obsolete.

The integration of system information and interaction of an application with *sys-sage* is presented in Fig. 3. It illustrates the two options for data integration – for data already available before the startup (purple) and for data available at runtime (green) – and how the user interacts with the library (blue arrows). The functionality encapsulated in *sys-sage* out-of-the-box is marked in full lines, and

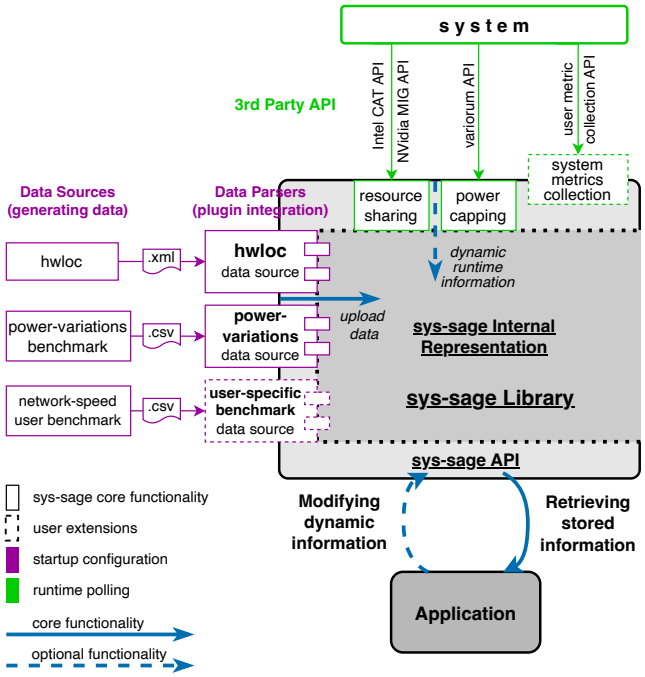


Figure 3: Integration of system information to *sys-sage*.

the possible custom extensions are dashed. In general, the user of our tool defines which sources get uploaded to the library, either by choosing from provided (default-) or by adding custom data sources. This forms the basis of the stored information. Once data is loaded into *sys-sage*, the user, which can be an application, a resource manager, a runtime system, or any other program, can query the stored data from the library. In addition, through the API, users can add or modify arbitrary attributes to reflect changes in the system or new information as it becomes available.

3.1 Conceptual Elements of *sys-sage*

For a closer understanding, we define the following conceptual elements:

Internal Representation is the central part of the library that stores information regarding the components of the system as well as their relations. It combines all data together into one logical representation, enabling easy orientation in it. More detail on the *Internal Representation* follows in Sec. 3.2.

Data Sources are files or other interfaces containing/providing the raw information to upload into the library. There are two types of data sources: 1) *Default Data Sources* and 2) *Custom Data Sources*. The former is generated through functionality (applications, benchmarks, scripts, ...) supported by the core of *sys-sage*. The goal is to provide the frequently-used information out-of-the-box so that the users do not need to implement anything themselves. *Custom Data Sources*, on the other hand, are data generated by user (custom) extensions offering measurements, observations, or findings. They are not a part of the core of *sys-sage*, but can still be a part of the workflow for uploading data to *sys-sage*.

Input Parsers are responsible for uploading the *Data Sources* into *sys-sage*. They read the *Data Sources* and transform the data into structures recognized by the library's *Internal Representation*. Analogously to *Data Sources*, there are *Default* and *Custom Input Parsers*. *Default Parsers* are a part of the library and enable a simple out-of-the-box parsing and uploading of *Default Data Sources*.

Custom Parsers are user extensions that parse their *Custom Data Sources*. For instance, results of a user-defined benchmark can be integrated via a *Custom Input Parser*. They consist of a simple function that reads the *Custom Data Source* and builds the *Internal Representation* based on its contents. Once the *Internal Representation* mapping is decided, the effort is typically minimal and focuses primarily on parsing the input from its custom format.

API – *sys-sage* exposes an API through which the *Internal Representation* is manipulated and queried. Moreover, the API is used to query 3rd party tools and APIs to obtain live information regarding the system and/or application behavior and state. The API is further described in Section 4.

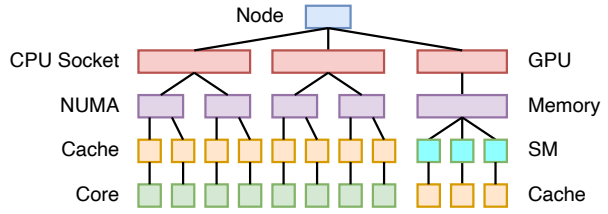
Being a user-side library, the user/application has full control over the information contained in *sys-sage*. Therefore, *sys-sage* only has access to information/APIs/data that the user can access. It operates within the process that utilizes it, and no data leaves the process unless the user purposely exposes it, to ensure security of the stored data. Finally, the user-side library format provides proper separation of concern, where each user (process) has its own *sys-sage* instance with its own set of information, even if running on the same machine, relying on the proven security mechanisms in the operating system.

3.2 Internal Representation

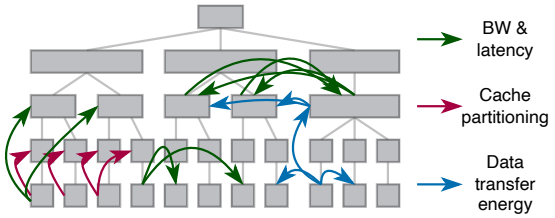
We represent an HPC system in *sys-sage* in the form of ***Components***, their hierarchical physical structure (that we call ***Component Tree***), and ***Data Paths***, which reflect additional attributes and relations between *Components* that go beyond the tree structure. *Data Paths* form a ***Data-path Graph***, which captures information orthogonal to the *Component Tree*, as the latter mainly represents the static physical composition of the system. Both the *Component Tree* and the *Data-path Graph* (presented in Fig. 4) can be modified at runtime, allowing *sys-sage* to capture the changing characteristics of a dynamic system.

For instance, *Components* can describe different caches (the capacity of each cache, its associativity, hit rate from HW counters) or the number of registers and current frequency of a CPU core. *Data Paths* bring information on relations between two *Components*, such as cache-to-core latency for different cache levels, or the percentage of cross-NUMA accesses for different NUMA regions.

Components and Component Tree. An HPC system is composed of multiple (physical or logical) elements that we call *Components*. *Components* have hierarchical relations to each other, which we capture in the *Component Tree*. It provides a structure that is easy to understand for the user, and is easy to navigate, as presented in Fig. 4a. It thereby forms the core of *sys-sage*, and all additional information (static and dynamic) is connected to and referenced from it. The concept is inspired by the way hwloc represents CPU



(a) *Component Tree* showing different *Component Types* in different colors.



(b) *Data-path Graph* with *Data Paths* carrying different information in different colors.

Figure 4: Example *Component Tree* and *Data-path Graph* over the same set of *Components*.

data, but *sys-sage* offers more freedom covering more diverse components capturing all on-node resources (hardware and system related), i.e., not limited to CPU resources.

There are multiple ***Component Types*** that are derived from different parts of computer systems so that their specific attributes and functionalities can be represented. *Component Types* are implemented in a class hierarchy and should be thought of in that way. All *Components* inherit basic attributes and API calls from the Type 'Component', which is a default generic type. Each *Component*, therefore, contains information such as

- its position in the *Component Tree* with links to the parent and child *Components* for easy navigation in the tree,
- basic attributes such as 'id' or 'name', and
- a wildcard map '`attrib`', which allows the user to add arbitrary pieces of information or data. It is a key-value store, where the key denotes the name of the attribute, and the value points to the data.

Each *Component* is of one of the following *Component Types*:[6]

- **Component**: generic type with no special attributes.
- **Topology**: the root of an HPC system.
- **Node**: HPC system nodes.
- **Storage**: a data storage device[7].
- **Memory**: main memory or a part of it (any technology).
- **Chip**: a building block of a node. It may represent a CPU socket, a GPU, a NIC or any other chip.
- **Subdivision**: a generic grouping within the system.
- **NUMA regions**: a special and frequently-occurring case of *Subdivision* grouping memory locations.
- **Cache**: different levels or kinds of caches.
- **Core**: a processing core or a block of compute units.
- **Thread**: a hardware thread, i.e., what a Linux OS considers a 'CPU' or a compute unit on other types of chips.

Each of these *Component Types* introduces a specific set of attributes or functionalities. This may be the cache level, size, and associativity for type *Cache* or frequency for type *Core*.

Data Paths and Data-path Graph. A *Data Path* is a construct that carries (often dynamic) information about the relation of two arbitrary *Components*. The union of all *Data Paths* forms a *Data-path Graph* (an example shown in Fig. 4b). Each *Data Path* has a source and a target *Component*. Apart from that, there are no other limitations. *Data Paths* may be oriented (differentiating between the source and the target) or bidirectional. Multiple *Data Paths* may connect the same pair of *Components*, enabling the representation of multiple dependencies or relationships. To differentiate between them, attribute '`dp_type`' is used to easily tell apart different kinds of information carried by *Data Paths*.

Analogously to *Components*, *Data Paths* have a set of default properties, such as bandwidth or latency, and a wildcard map '`attrib`' for all other data.

Data Paths carry all different kinds of information, including but not limited to data-transfer-, performance- or power-consumption-related, or even application-specific data, as shown in the use-cases in Sec. 5. Altogether, *Data Paths* can be used to model any property regarding the connection or relation of two arbitrary *Components*.

Data Paths are associated with the *Components* they refer to. Each *Component* contains a reference to all *Data Paths* associated with it; and each *Data Path* includes references to source and target *Components*.

Overall, *Data Paths* provide a very generic mechanism to express potentially dynamic relationships between components. This is in contrast to existing, less flexible approaches, such as the one by hwloc. Here, distance information, which we would represent as a possible relationship, is covered by separate 'distance matrices' — a concept that is not easy to work with from the user-perspective and is not flexible in its usage, as it is a limited, single purpose specification of only latency/bandwidth information.

4 *SYS-SAGE* API AND IMPLEMENTATION

sys-sage is implemented as a C++ library, which can be easily integrated into other (mainly C++-based) projects. *sys-sage* offers a rich API based on C++ object-oriented principles. The API is designed to be easy to use and easy to understand.

Components and *Data Paths* alike are instances of different C++ classes, forming the basis of the *Internal Representation*. To create, access, and manipulate them, *sys-sage* exposes an API in the form of public methods.

The *Components* are linked to each other through the *Component Tree*. *Data Paths* are linked to the source and target *Components* (as well as *Components* are linked to their *DataPaths*). Therefore, having a handle (pointer) to an arbitrary *Component* (or *Data Path*) suffices to traverse to any *Component* (or *Data Path*) and access/update its information.

[6]Different Component Types have no predefined hierarchy.
[7]Note, that for technologies such as NVRAM, the system administrator decides whether to model it as a memory or storage component (or both).

Apart from focusing on the *Internal Representation*, other parts of the API handle the *Data Parsers* and retrieval of dynamic variable information through integrated 3rd party APIs at run time (cf. Fig. 1).

4.1 API Categorization

We divide the API into categories based on the functionality they provide. The complete API description is available in *sys-sage*'s documentation via the GitHub repository.

4.1.1 ***Components.*** The *Components*' API is realized via public methods of the *Components*, i.e., the users call functionalities from a particular *Component*'s perspective. The majority of the API is available to all *Component Types*. Exceptions are *Component Type*-specific getters, setters, and constructors.

Creating/Deleting Components. The creation of individual new *Components* (and their placement in the *Component Tree*) is managed by the respective class constructors (examples being `Cache(...)`, `Node(...)`). Each *Component Type* (being a separate class) has a set of its own constructors. Deletion of *Components* is done through `void Delete(...)` or `void DeleteSubtree()` methods, deleting either the *Component* itself or the entire subtree.

Navigating in the Component Tree. Users can browse through the *Component Tree*, using *sys-sage* API to find the *Component*/*Data Path* of interest (to read or update its attributes). They traverse to a single child/ancestor filtering by different parameters (such as `Component* GetParent()`, `Component* GetAncestorByType(int)`, `Component* GetChild(int)`), or retrieve a list of *Components* matching selected criteria (`vector<Component*>* GetChildren()`, `vector<Component*> GetAllChildrenByType(int)`, or `vector<Component*> GetAllSubcomponentsByType(int)`).

Modifying the Component Tree. The user can insert and remove *Components* to dynamically modify the structure of the *Component Tree*. For this purpose, we can insert a *Component* as a new leaf element[8] (e.g., `void InsertChild(Component*)`) or between a parent and (one or more) children *Components* (e.g., `int InsertBetweenParentAndChildren(...)`). Inversely, one can also remove a *Component*'s child (`int RemoveChild(Component*)`)[9]

Aggregating the Component Tree. Aggregation operations over the *Component Tree* provide basic tree structure characteristics. `int CountAllSubcomponents()` or `int GetSubtreeDepth()` are some examples.

Attribute Getters/Setters. *Components*' attributes are retrieved through getters and updated through setters (or constructors). Some are common for all *Components* (such as `int GetComponentType()` and `int GetId()`), and others are *Component Type*-specific (such as `void SetCacheLineSize(int)` or `void SetSize(long long)`).

`attrib` *store.* To ease the access to the additional data stored in the *Components*, `map<string, void*> attrib` is a public member.

Accessing the Data Paths. *sys-sage* offers methods, such as `DataPath* GetDpByType(...)` and `void GetAllDpByType(...)`[10], to access the *Data Paths* related to a *Component*.

Helper and Output Functionality. Finally, supporting functionality for development/debug purposes is present. The users can, for example, print basic information about the *Components* in the subtree to stdout (`void PrintSubtree()`). They can also use the `int CheckComponentTreeConsistency()` method to check if all *Components* in the *Component Tree* are interlinked properly.

4.1.2 ***Data Paths.*** The APIs for *Data Paths* offer very similar functionality to *Components*, but in a limited scale, as not all operations are relevant. There are methods for **Creating/Deleting the DataPaths** (`DataPath(...)`, `void DeleteDataPath()`). Next, **Accessing the associated *Components*** is possible through `Component* GetSource()` or `Component* GetTarget()`. There are also **Getters and Setters** for *Data Paths*' attributes (such as `int GetDpType()`, `void SetLatency(double)`), and the **`attrib` store**. Finally, a `void Print()` **Helper Function** is available as well.

All in all, the exposed functionality for *Data Paths* follows the same logic as one for *Components*. It does not offer as many functionalities, as *Data Paths* do not have to manage the *Component Tree* complexities, neither do they support multiple *Data Path* types.

4.1.3 ***Data Parsers.*** The *Data Parsers* upload *Data Sources* into *sys-sage*'s *Internal Representation*, using primarily the *Component* and *Data Path* APIs in the background to do so. As such, they offer uploading a larger amount of information in one command. As they are not explicitly related to a particular *Component* or *Data Path*, they are available as standalone functions. The user only defines which part of the *Component Tree* to add the information to (such as which node or socket).

For instance, there are *Data Parsers* for hwloc CPU topology output `int parseHwlocOutput(Node* n, string fileName)` and for mt4g[11] `int parseMt4gTopo(...)` (cf. Sec. 5.3). As more and more *Data Sources* become available, this part of the API will grow in size.

4.1.4 ***Dynamic 3rd Party API.*** Finally, there is an API for managing dynamic information retrieved at the run time. The dynamic information extends the API for *Components* or *Data Paths* by introducing new calls. For example, `int UpdateL3Partitioning()` updates the current L3 cache partitioning settings of a Node *Component*. `int RefreshFreq(...)` retrieves the frequency of a particular CPU core (as a core's method) or each CPU core in a node (as a node's method). The dynamic 3rd Party API may also grow as we introduce support, for instance, for power- or GPU-related metrics, introducing additional options for the users.

4.2 Performance of Basic Operations

We evaluate the overhead of *sys-sage* in Table 1. It presents the times needed to upload and parse chosen *Data Sources* into the library, including the creation of the *Internal Representation*. The parsing times are typically in the microsecond range per *Component*/*Data Path*, including overheads of reading the source files.

[8] Constructors can also attach the new *Component* as a leaf.
[9] Removing only as parent's child, the *Component* does not cease to exist.
[10] The result is returned as a parameter to minimize data movements.
[11] Providing GPU topology. Available under https://github.com/caps-tum/mt4g

Action	Time	Time per element
Parse hwloc output	3.2 ms	30 µs/Component
Parse GPU topology information	15.6 ms	2.0 µs/element*
Get handle to all *Components*	379 µs	92 ns/Component
Find NUMA region with the largest bandwidth	242 ns	60 ns/DataPath searched
Get dynamic L3 size for the current thread	2.6 µs	
Create new *Component*	3.3 µs	

*: Created 4023 *Components* and 3840 *Data Paths*.

Table 1: Overhead of selected static API calls in *sys-sage*.

Action	Time	% 3rd party intereface
Update L3 cache partitioning	7.0 ms	97.0 %
Update Frequency of all cores	1.01 ms	78.1 %
Update MIG configuration*	22.8 ms	95.8 %

*: Measured on a AMD Ryzen Threadripper 2990WX, 32 cores, 3.0 GHz, 64 MB L3, running Linux 5.15.0 and NVidia A-100, driver v. 545.23.08.

Table 2: Overhead of integrated dynamic 3rd party interfaces in *sys-sage*.

Next, Table 1 presents the overhead of selected API calls, which range in microseconds or lower per piece of information obtained. The results are, however, highly dependent on how the calls are made, especially regarding caching effects. This and all subsequent performance measurements were performed on a Dual Socket Intel Xeon Silver 4116, 12 cores/socket, 2,10 GHz, 16,5 MB L3, running Linux 5.15.10.

4.3 Performance of Dynamic 3rd Party Operations

Table 2 presents selected 3rd party interfaces to query dynamic varying information during the runtime. The performance highly depends on the overhead of the underlying 3rd party APIs – these calls (which *sys-sage* cannot influence) form over 95 % of the overall time. The frequency read-out is an exception where *sys-sage* overhead presents 22 % of the total time. This is because the frequency is read directly from the `/proc/cpuinfo` file.[12]

In general, these results show that *sys-sage* would only introduce a small percentage of overhead to process and store the information. Even when parsing a file, the file read overhead is significantly higher than *sys-sage*'s.

4.4 Memory Footprint

Regarding memory requirements, the amount of memory *sys-sage* occupies is defined by the amount of information we are interested in. The memory footprint of a 24-core, 105-*Component* hwloc output in the *sys-sage Internal Representation* is 22 kB, i.e., 209 B/Component.

[12]We measure the file parsing as a part of *sys-sage* overhead; only opening/reading the file is measured as 3rd party interface contribution.

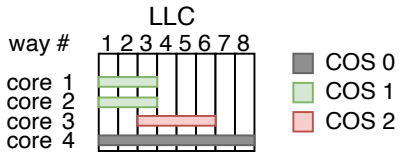


Figure 5: Intel CAT technology. Each core belongs to a COS, which defines the LLC ways it can access.

Due to low memory requirements, *sys-sage* can benefit from caching effects when used repeatedly.

The time and memory resource footprint results show that our *sys-sage* implementation is both fast to use and occupies only a small amount of memory that does not interfere with the application running on the system.

5 EVALUATION WITH USE-CASES

While the experiments in the last section show the low overhead of the library, which is a key prerequisite for any support library in HPC, the real value of *sys-sage* lies in its capability to ease the handling of topology data and in support of a wide range of use cases. The latter targets both the simplification and improvement of existing use cases, enabling the migration of functionality out of individual consumers into a shared code base, as well as its ability to support new use cases, especially ones focusing on dynamic system behavior. To illustrate this and demonstrate and evaluate the capabilities of *sys-sage*, we describe three distinct use cases from very different areas. None of the use cases can be solved using currently available tools, such as hwloc, hence showing the need for a new library such as *sys-sage*.

5.1 Cache-aware Algorithm on Dynamically Changing Cache

Modern CPUs offer means to restrict access for specific processes or cores to certain resources, such as L3 cache partitions, to more reliably support virtualization or co-scheduling techniques. Resource sharing and isolation are essential topics in cloud environments, but they are also becoming more prominent in HPC co-scheduling efforts [30]. On Intel CPUs, which we use in the following experiments, this feature is called 'Cache Allocation Technology (CAT)'.[13] When running multiple applications on one CPU, we can use CAT to provide each application with an isolated portion of shared resources, such as the L3 cache. CAT uses Classes of Service (COS) for each CPU core to assign it an available fraction of the L3 cache. For instance, in Fig. 5, Cores 1 and 2 can only use cache ways 1–3, i.e., 3/8 of the total L3 size. CAT, therefore, turns the available cache size from a static hardware property to a dynamic runtime setting. Many applications, particularly those that repeatedly access their dataset, as is for example the case for matrix operations or stencil computations, are often tuned to a particular cache size. For example, they split their domains based on the available cache sizes to profit from their lower latency.

[13]Similarly, AMD (Platform QoS) and ARM offer means to partition caches; we plan to add support through the same API in the future.

This use-case discusses a parallel OpenMP implementation of a Jacobi stencil computation. Each core performs the computations row-wise. The goal of a tiled implementation is to ensure that the neighboring cells already accessed in the given time step are not evicted from the L3 cache in the meantime. Therefore, the row-wise computation is split into parts to fit n rows in the L3 cache; hence, each value (cache line) is loaded only once. On the test system, this tiled implementation results in a speedup of 1.47 over the naive parallel implementation for larger stencils.

However, when CAT is activated, the L3 size information provided by hwloc or `/sys/devices/system` is no longer applicable, and the speedup disappears. To solve this, we must provide the true accessible L3 size available to a particular CPU core instead. *sys-sage* enables this by tracking the dynamic CAT settings on the system. The dynamic information is represented using *Data Paths* between the core (source) and the L3 cache (target).[14] Specifically, each *Data Path* carries the bitmask of the opened L3 ways (COS) as an entry in the 'attrib' key-value store. Combined with the static 'size' attribute of the L3 'Cache' *Component*, the exploitable L3 size can be calculated and provided to the user.

The implementation is sketched on Listing 1.[15] First, a *Component* of type 'Node' is created, and the hwloc output file is parsed and uploaded to this node to capture the static hardware property. Then, we obtain a handle to the currently running thread in the *Component Tree* representation. Next, *Data Paths* for all hardware threads with the current L3 CAT settings are updated. Finally, the exploitable L3 size is obtained, including L3 CAT settings.[16] These six lines of code utilizing *sys-sage* spare us the burden of 1) parsing the static topology information, 2) finding the related L3 cache sizes for a given CPU core, 3) communicating with the CAT API to obtain the bitmask for the current core, and 4) correlating all the data together. As none of these tasks is in the focus of the stencil computation code, it only makes sense to offload them to a library, i.e., *sys-sage*, thus providing a separation of concern for the application.

To evaluate the correctness and impact of this approach, we enable different cache configurations using CAT and compare the performance when running (1) the naive non-tiling algorithm, (2) tiling using the static cache information provided by existing approaches, such as hwloc, and (3) tiling using the dynamic information provided by *sys-sage*. For reproducibility of the results, the CAT settings do not change throughout the measurement and are obtained only once. We could check the CAT settings repeatedly to react to potential dynamic changes during the program execution simply by moving the last two lines from Listing 1 into the main for-loop.

[14]Note that also L2 CAT exists, which can be represented in *sys-sage* in the same way.
[15]Error handling (if return value == NULL) is omitted for simplicity.
[16]We know that all OMP threads share the same COS, hence we generalize the information from OMP Thread 0. Otherwise, we would query the available L3 size for each thread.

```
Node* n = new Node(1);
parseHwlocOutput(n,"hwloc_output.xml");
int cpu_num = sched_getcpu();
Thread* t=(Thread*)n->FindSubcomponentById(cpu_num,
      SYS_SAGE_COMPONENT_THREAD);
n->UpdateL3Partitioning();
long long L3_sz = t->GetDynamicL3Size();
```

Listing 1: Obtaining available L3 size for running core.

Speedup \ Available L3 Fraction	full L3	8/11 L3	2/11 L3
Static L3 size vs. Naive	1.47	1.20	1.03
Dynamic L3 size vs. Naive	1.47	1.43	2.11
Dynamic L3 size vs. Static L3 size	1.0	**1.19**	**2.05**

Table 3: Reducing the available L3 size renders static tiling useless. Our cache-partitioning-aware approach maintains the performance advantage of tiling.

The results are presented in Table 3. We see that the performance gain of tiling using the static L3 cache size vanishes as we limit the effective L3 size, as the tile size is too large to fit into the cache partition, similar to the naive implementation. When limiting the L3 size to 2/11 of the static hwloc-provided size, the performance gain is eliminated (1.03x). On the other hand, the implementation querying the correct dynamic L3 size via our approach, which reflects the dynamic changes in the system, maintains or even improves the observed speedup (up to **2.11x** vs. naive implementation when using 2/11 of the L3 cache).

Cache-aware algorithms are, in many cases, useful for reducing data transfer bottlenecks. This use-case clearly shows that using only the static information no longer suffices on modern systems with dynamic properties. To provide accurate information based on these dynamic properties, *sys-sage* can be integrated within a few lines of code.

5.2 Capturing Memory Access Data in *sys-sage*

A second use case for *sys-sage* is maintaining and organizing data collected for performance analysis and visualization. To demonstrate this capability, we build upon Mitos [12], a data collection tool that uses Intel's PEBS functionality to provide raw traces of an application's memory movements with detailed hardware-related information.

MemAxes [11] is a complementary tool that analyzes and visualizes the collected data so that users can easily identify memory-based bottlenecks of their applications. For the data analysis and visualization, it is necessary to augment the measured data with the system context in which they were measured and to connect the acquired memory access samples with the hardware topology information. Initially, the implementation of MemAxes internally converted the hwloc output to a rigid structure. This implementation is no longer applicable to modern systems with complex and varying hardware characteristics. We, therefore, replaced the rigid internal topology representation with *sys-sage*. This way, we gain a new flexible and future-proof approach to maintain the relation of the two inputs – the acquired memory traces and the dynamic system configuration.

For this transformation in MemAxes, the original hwloc topology information is directly imported to a *sys-sage Component Tree*. We then store this dynamic sample information in *sys-sage*'s *Data Paths*. We use the information from Mitos samples about the issuing core

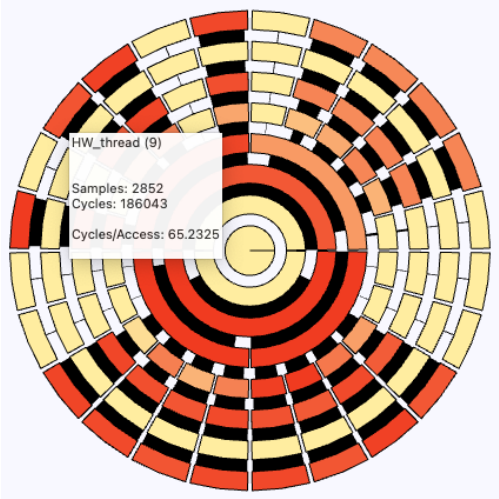


Figure 6: MemAxes Topology View visualization is based on data from *sys-sage*'s *Component Tree*. Individual blocks represent (from outside to inside): Hardware Threads, Cores, L1 Caches, L2 Caches, L3 Caches, NUMA memories, Sockets and the Node. Darker color indicates more samples collected. Mouse hover presents detailed information.

(source) and the memory component serving the request [17](target), connecting the samples with the underlying hardware structure.

Benefits of adopting *sys-sage*: This integration results in a highly flexible system representation and sample storage, which is visualized in MemAxes, as shown in Fig. 6. *sys-sage* takes over the data management and its connection, which significantly simplifies the implementation.

The original hwloc output parsing logic took over 200 lines of code, which could be replaced with a single-line *sys-sage* API call. Moreover, the original parsing logic was very naive, parsing the output line-by-line. MemAxes would no longer work in the current (2.9+) version of hwloc, as NUMA memory segments are no longer parents of the underlying caches in the XML topology export. To overcome this, the parsing logic would become much more complex and would, as a result in the ideal case, end up mimicking the hwloc parser already integrated in *sys-sage*.

More importantly, we can loosen the data backend coupling through our *sys-sage* integration. This means that MemAxes can now easily switch the HW topology backend to other tools, like, e.g., MUSA [17] to visualize MUSA-generated traces[18]. Similarly, MemAxes can also visualize AMD IBS samples (also provided by Mitos), which use different data layout, with minimal effort, allowing us to unify the access to this data across both platforms.

The transfer to *sys-sage* enables representation of systems with various cache hierarchies, multi-socket systems, or cross-NUMA data accesses, none of which was previously possible without significant rewrites in the end-user tool, which is not maintainable in the long run. The integration of *sys-sage* provides a proper separation of concern and a generalization of the functionality.

[17] L1/L2/L3 cache or which NUMA region

[18] As MUSA uses a custom topology definition format incompatible with hwloc's, we only have to add a custom *Data Parser* for MUSA.

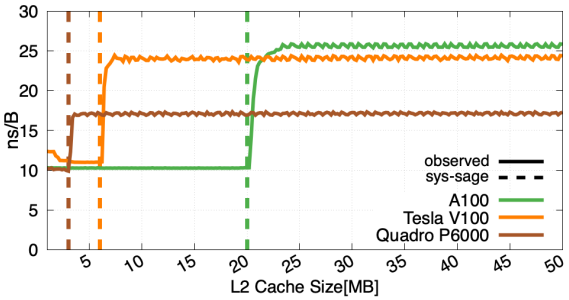


Figure 7: Speed of repeatedly reading an array of a certain size on different NVidia GPUs/microarchitectures. Vertical lines denote the L2 cache size provided by *sys-sage*.

5.3 Performance Estimation on GPUs with Dynamically Changing Memory Topology

Similarly to CPUs, GPUs use a complex set of caches and GPU kernels profit from their efficient usage. However, GPUs' compute and memory hardware architecture is hidden and hard to capture and, therefore, is often not considered by the developers. Our third use-case presents how *sys-sage* concepts can be used beyond CPU, for performance estimation and optimization on GPUs, which is crucial to achieving performance portability across different platforms. In particular, knowing the size of the caches enables users to allocate the right amount of resources or adjust the data set size to achieve optimal performance on an arbitrary system.

Some tools and interfaces provide the L2 cache size, such as ones of CUDA (`cudaGetDeviceProperties`) [26], hwloc [6], or LIKWID [19]. Information about other caches is, however, not available. *sys-sage* provides means to represent the complete GPU hierarchy, including different types of caches, SMs (Streaming Multiprocessors), warps, etc. It integrates the mt4g *Data Source*[19], which utilizes native interfaces to query the cache sizes, where available, or measures the size using carefully designed microbenchmarks that stress the memory and caching subsystem. Ultimately, *sys-sage*'s *Data Parser* for mt4g combines the information from both sources and offers it through a single API.

sys-sage stores and manages the information the same way as the matching CPU information – by constructing a *Component Tree* and respective *Data Paths*. Therefore, even accessing the data follows the same rules thanks to *sys-sage*'s unified interface, despite the widely different structure of the underlying *Data Sources'* information. Currently, *sys-sage* supports all recent NVIDIA microarchitectures, and preliminary results have been obtained for AMD GPUs.

5.3.1 Algorithmic Impact. Fig. 7 presents a measure of performance (ns/B) of repeated vector addition using vectors of different sizes, illustrating the potential impact on performance. The vertical lines represent L2 cache sizes for different GPUs provided by *sys-sage*. The plot shows a visible decrease in performance once the vector

[19] As a *Default Data Source*, it is directly linked from *sys-sage* repository.

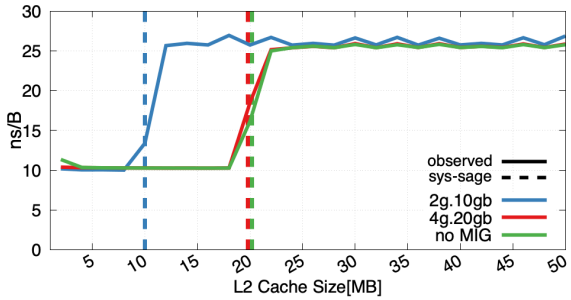


Figure 8: Speed of repeatedly reading an array of a certain size on different NVIDIA A100 using MIG. Vertical lines denote the L2 cache size provided by *sys-sage*.

size exceeds the L2 cache. Therefore, it is crucial to select a data set that fits into this L2 size limit on a given machine, for which knowledge of this information is required.

Currently, this can be done manually via separate interfaces using different APIs. Further, hwloc, likwid, and NVIDIA native APIs would report the L2 size of NVIDIA's A-100 as 40 MB. However, by analyzing the hardware architecture (e.g., in Choquette et al. [8]), one can observe two partitions on the GPU, each having 54 SMs and a 20 MB-block of L2. Since the used topology representation in *sys-sage* does not rely solely on native interfaces, but verifies it using microbenchmarking results, this feature could be recognized and correctly modeled in the *Internal Representation*, offering users correct information based on actual observed hardware properties.

5.3.2 Usage for Dynamic Multi-instance GPUs. Most recent NVIDIA GPUs (starting with the Ampere microarchitecture) feature dynamic capabilities enabling resource isolation. The functionality is called Multi-instance GPU (MIG) [1] and is intended for use in cloud and HPC environments. MIG enables isolated usage of only fractions of the GPU (called GPU Instances or GIs), consisting of several SMs and slices of L2 cache, the main memory, or available bandwidth. Using this technology, multiple programs or even users can share one GPU with clearly defined available resources and without interfering with each other.

Using MIG, therefore, renders the static information, such as the number of cores, SMs, or main memory or L2 size, useless. Again, the MIG setting can be captured in *sys-sage*, so only the available resources are returned when queried, reflecting the MIG setup. Fig. 8 presents the previously described workload on the same NVIDIA A-100 machine, this time using different MIG settings. The sudden performance decrease is again clearly visible, this time depending on the amount of resources allocated. When using 2 GIs ('`2g.10gb`', using 1/4 of L2), the change occurs around 10 MB. 4 GIs ('`4g.20gb`', using 1/2 of L2) yield the same curve as the full instance. As described above, since one SM would access only 1/2 of the L2 cache anyway, having the other half of L2 unavailable does not impact the result.

Overall, this use-case shows that *sys-sage* correctly and universally provides the L2 cache size despite the hardware design details or dynamic changes in the resource availability, thus helping to correctly estimate the performance of GPU kernels.

6 RELATED WORK

Regardless of the previously mentioned limitations, hwloc [6] is widely used and is the de-facto standard in obtaining information about hardware topology, as it is reliable, easy to use, and available on many systems. It nicely exposes hardware locality on a CPU. To address some of the new systems' complexities, follow-up works propose extensions to hwloc [13, 14, 16]. They focus on heterogeneous systems, providing information about GPUs, I/O, networking, heterogeneous memories, or capturing topologies of multi-node systems. While this is a step in the right direction, the main representation remains a static tree structure, which, even though it is helpful in many cases, has its limits – we address these limits with our approach.

As an extension to hwloc on CPUs, netloc [4, 15] targets the network topology and process mapping on HPC systems. It represents the inter-node network as a map and integrates hwloc's tree on-node topology. Even though the network representation enables more realistic system representation, it is still a static view offering little flexibility in use-cases. Since the network can be represented as *sys-sage Data Paths*, netloc's topology could be easily integrated into *sys-sage* as an additional *Data Source*.

Other approaches, such as `likwid-topology` [29] or the Linux `lscpu` tool [27], provide similar information as hwloc. There have also been smaller-scale efforts to present system topologies, such as the systopo tool [20] or Topo library [18]. However, their functionality is limited to a specific task, and they have often not been maintained.

Other works describe applications that require some system information beyond what hwloc provides, but their main goal is not to provide this information itself. They specifically implement the needed topology-related functionality as a one-off and provide it to the main program. *sys-sage* can be used there as a flexible solution to provide the needed information, making *sys-sage* a central and unified data source for these approaches. Doing so would prevent reimplementing the same functionality and enable the applications to profit from the more in-depth analysis *sys-sage* enables.

Mpibind [22] presents an algorithm for (MPI+OpenMP+CUDA) hybrid applications, which maps threads or processes to hardware based on the memory system. It combines information from several sources, such as hwloc or `nvidia-smi`, to retrieve the hardware information. Using *sys-sage* as a data backend would reduce the complexity of managing data from these different sources. Also, it would simplify the whole algorithm, as some of the computed topology information is available via *sys-sage* out-of-the-box.

Leon et al. [23] present a scheduling framework for heterogeneous architectures. It maps the tasks (processes, threads, GPU kernels) to hardware resources based on different requirements and system/application characteristics. Also in this case, *sys-sage* would be a good candidate to provide system-related information. Moreover, many tasks of the proposed Mapping coordinator, such as tracking resource utilization, could be done in *sys-sage* in a unified manner as a single backend, further reducing software complexity and improving reusability.

There are many applications requiring some information about system topology and capabilities. They often use hwloc as a basis and implement the remaining functionality manually. To our

knowledge, very little work focuses on providing detailed and comprehensive information about system topology and capabilities in a universal manner that enables providing custom information and attaching additional data to it. Adopting the *sys-sage* approach, which is specifically designed for that purpose, will benefit many such application scenarios.

7 CONCLUSIONS AND FUTURE WORK

The increasing complexity of current and future HPC systems renders a static hierarchical system topology insufficient for fully understanding and efficiently utilizing these systems, in particular for complex workloads. A wide range of APIs and tools exist to enable applications to query the needed hardware- and system-data to capture the platform's structure as well as dynamic properties. The hwloc library is the most prominent example of such an API/-tool and enables the query of static topology information; however, more information, in particular from other sources as well as dynamic information, is needed to obtain a better understanding of the system.

To address this gap, we introduced *sys-sage*, which provides both a highly detailed and fully dynamic view on HPC systems and their attributes. It combines two ways of representing system information – a mostly static *Component Tree* and dynamic *Data Paths*. The information is correlated with each other by mapping the two views onto each other, creating a holistic view capturing both static and dynamic information, and providing it to the user in a single interface. We defined the requirements for the solution so that it can be applied to a wide spectrum of problems. Based on the analysis, we presented our library implementation and its capabilities.

We brought forward three use-cases where *sys-sage* was adopted to help to understand the underlying system – in the first case, we showed how to use the library to model dynamically changing systems, in particular to track dynamic changes to cache configurations and how to use it to steer algorithmic decisions, like tiling. In the second use case, *sys-sage* was used to provide flexible storage for performance data and to connect this data with the underlying HW of the measured system, offering a simple way of extracting both measurements and context information together. Finally, the third use-case presented capabilities of *sys-sage* on GPUs with dynamically changing properties, going beyond the static and incomplete information in native interfaces. These three very different use-cases show the versatility of our solution and demonstrate the extra functionality of *sys-sage* over the state-of-the-art approaches.

In the near future, we will keep adding more default *Data Sources* to provide the users with a vast majority of the data they need out-of-the-box. We also plan to extend the scope of our approach to span multiple heterogeneous components and multiple nodes to showcase the versatility and scalability of *sys-sage*.

ACKNOWLEDGMENTS

The DEEP-SEA project has received funding from the European Union's Horizon 2020/EuroHPC research and innovation programme under grant agreement No 955606. National contributions from the involved state members match the EuroHPC funding.

REFERENCES

[1] 2020. *NVIDIA Multi-Instance GPU and NVIDIA Virtual Compute Server.* Technical Report. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/solutions/resources/documents1/Technical-Brief-Multi-Instance-GPU-NVIDIA-Virtual-Compute-Server.pdf.

[2] 2024. Top500. https://www.top500.org/. Accessed: 2024-01-10.

[3] Deva Bodas, Justin Song, Murali Rajappa, and Andy Hoffman. 2014. Simple power-aware scheduler to limit power consumption by hpc system within a budget. In *2014 Energy Efficient Supercomputing Workshop*. IEEE, 21–30.

[4] Cyril Bordage, Clément Foyer, and Brice Goglin. 2018. Netloc: a Tool for Topology-Aware Process Mapping. In *Euro-Par 2017: Parallel Processing Workshops: Euro-Par 2017 International Workshops, Santiago de Compostela, Spain, August 28-29, 2017, Revised Selected Papers 23*. Springer, 157–166.

[5] James M Brandt, Ann C Gentile, Jonathan Edwin Cook, Benjamin A Allan, Jeanine Cook, Omar Aaziz, Thomas Tucker, Naksinehaboon Nichamon, Narate Taerat, Emre Ates, et al. 2018. *Runtime HPC System and Application Performance Assessment and Diagnostics.* Technical Report. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).

[6] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. 2010. hwloc: A generic framework for managing hardware affinities in HPC applications. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*. IEEE, 180–186.

[7] Dimitrios Chasapis, Miquel Moretó, Martin Schulz, Barry Rountree, Mateo Valero, and Marc Casas. 2019. Power efficient job scheduling by predicting the impact of processor manufacturing variability. In *Proceedings of the ACM International Conference on Supercomputing*. 296–307.

[8] Jack Choquette, Edward Lee, Ronny Krashinsky, Vishnu Balan, and Brucek Khailany. 2021. 3.2 the A100 datacenter GPU and Ampere architecture. In *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 64. IEEE, 48–50.

[9] Yiannis Georgiou, Emmanuel Jeannot, Guillaume Mercier, and Adèle Villiermet. 2017. Topology-aware resource management for HPC applications. In *Proceedings of the 18th International Conference on Distributed Computing and Networking*. 1–10.

[10] Balazs Gerofi, Masamichi Takagi, and Yutaka Ishikawa. 2014. Exploiting Hidden Non-uniformity of Uniform Memory Access on Manycore CPUs. In *Euro-Par 2014: Parallel Processing Workshops*, Luís Lopes, Julius Žilinskas, Alexandru Costan, Roberto G. Cascella, Gabor Kecskemeti, Emmanuel Jeannot, Mario Cannataro, Laura Ricci, Siegfried Benkner, Salvador Petit, Vittorio Scarano, José Gracia, Sascha Hunold, Stephen L. Scott, Stefan Lankes, Christian Lengauer, Jesús Carretero, Jens Breitbart, and Michael Alexander (Eds.). Springer International Publishing, Cham, 242–253.

[11] Alfredo Giménez, Todd Gamblin, Ilir Jusufi, Abhinav Bhatele, Martin Schulz, Peer-Timo Bremer, and Bernd Hamann. 2017. Memaxes: Visualization and analytics for characterizing complex memory performance behaviors. *IEEE transactions on visualization and computer graphics* 24, 7 (2017), 2180–2193.

[12] Alfredo Giménez, Benafsh Husain, David Böhme, Todd Gamblin, and Martin Schulz. 2015. Mitos: A Simple Interface for Complex Hardware Sampling and Attribution.

[13] Brice Goglin. 2014. Managing the topology of heterogeneous cluster nodes with hardware locality (hwloc). In *2014 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 74–81.

[14] Brice Goglin. 2016. *Towards the Structural Modeling of the Topology of next-generation heterogeneous cluster Nodes with hwloc.* Research Report. Inria. https://inria.hal.science/hal-01400264

[15] Brice Goglin, Joshua Hursey, and Jeffrey M Squyres. 2014. Netloc: Towards a comprehensive view of the HPC system topology. In *2014 43rd International Conference on Parallel Processing Workshops*. IEEE, 216–225.

[16] Brice Goglin and Andrès Rubio Proaño. 2022. Using Performance Attributes for Managing Heterogeneous Memory in HPC Applications. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 890–899.

[17] Thomas Grass, César Allande, Adrià Armejach, Alejandro Rico, Eduard Ayguadé, Jesus Labarta, Mateo Valero, Marc Casas, and Miquel Moreto. 2016. MUSA: a multi-level simulation approach for next-generation HPC machines. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 526–537.

[18] Samuel Grossman. 2016. Topo. https://github.com/stanford-mast/Topo.

[19] Thomas Gruber, Jan Eitzinger, Georg Hager, and Gerhard Wellein. 2019. likwid 5: Lightweight performance tools. In *Presented on SC19 Conference, Denver, CO.*

[20] Martin Grund. 2011. systopo. https://github.com/grundprinzip/systopo.

[21] Andrew Herdrich, Edwin Verplanke, Priya Autee, Ramesh Illikkal, Chris Gianos, Ronak Singhal, and Ravi Iyer. 2016. Cache QoS: From concept to reality in the Intel® Xeon® processor E5-2600 v3 product family. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 657–668.

[22] Edgar A León. 2017. mpibind: A memory-centric affinity algorithm for hybrid applications. In *Proceedings of the International Symposium on Memory Systems*.

262–264.

[23] Edgar A León, Balazs Gerofi, Julien Jaeger, Guillaume Mercier, Rolf Riesen, Masamichi Takagi, and Brice Goglin. 2020. Application-Driven Requirements for Node Resource Management in Next-Generation Systems. In *2020 IEEE/ACM International Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*. IEEE, 1–11.

[24] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. 1999. PAPI: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, Vol. 710.

[25] Alessio Netti, Micha Müller, Axel Auweter, Carla Guillen, Michael Ott, Daniele Tafani, and Martin Schulz. 2019. From Facility to Application Sensor Data: Modular, Continuous and Holistic Monitoring with DCDB. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 64, 27 pages. https://doi.org/10.1145/3295500.3356191

[26] NVidia 2022. *CUDA Runtime API – API Reference Manual*. NVidia, https://docs.nvidia.com/cuda/pdf/CUDA_Runtime_API.pdf.

[27] Cai Qian, Karel Zak, and Heiko Carstens. 2021. *lscpu(1) – Linux manual page*. man7.org, man7.org.

[28] Sabela Ramos and Torsten Hoefler. 2017. Capability models for manycore memory systems: A case-study with xeon phi KNL. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 297–306.

[29] Jan Treibig, Georg Hager, and Gerhard Wellein. 2010. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *2010 39th international conference on parallel processing workshops*. IEEE, 207–216.

[30] Josef Weidendorfer, Carsten Trinitis, Sebastian Rückerl, and Michael Klemm. 2017. Cache-Partitionierung im Kontext von Co-Scheduling. *PARS-Mitteilungen: Vol. 34, Nr. 1* (2017).

[31] Wenxiang Yang, Cheng Chen, and Jie Yu. 2022. Topology-Aware Node Allocation on Supercomputers with Hierarchical Network. In *2022 IEEE 24th Int Conf on High Performance Computing & Communications; 8th Int Conf on Data Science & Systems; 20th Int Conf on Smart City; 8th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*. IEEE, 1095–1100.

Received 19 January 2024; revised 16 April 2024; accepted 21 April 2024