Asgard: Empowering Memory Scalability for Terabyte-Scale Hardware-Partitioned Servers

Ángel Burgos Muñoz
EPFL, Lausanne
angel.burgosmunoz@epfl.ch

Konstantinos Chasialis
EPFL, Lausanne
konstantinos.chasialis@epfl.ch

Pedro Palacios Almendros
EPFL, Lausanne
pedro.palaciosalmendros@epfl.ch

Luis Romero Rodriguez
EPFL, Lausanne
luis.romerorodriguez@epfl.ch

***Abstract*—In the growing business of cloud services, there exist numerous solutions to rent cloud computing services to multiple clients on the same physical machine. Among these, new bare-metal cloud solutions such as Core Slicing look promising due to their increased security and performance. Additionally, new solutions such as Midgard leverage intermediate address spaces to take advantage of growing cache sizes and reduce the address translation overhead, allowing for memory scalability beyond the memory sizes efficiently supported by traditional virtual memory systems. In this paper, we argue that the combination of these solutions is not orthogonal and hardware-enforced isolation can benefit from an intermediate address space such as Midgard to implement its key features while overcoming its limitations. We present a solution that combines Midgard and Core Slicing, demonstrating improved granularity for core assignment to clients in bare-metal cloud and reduced unnecessary Last-Level Cache (LLC) communication and physical sharing. In addition, our proposed LLC partitioning scheme maintains high associativity, resulting in cache miss rates up to 63% lower than those observed with existing commercial cache partitioning techniques.**

I. Introduction

Infrastructure as a service has steadily been gaining relevance in recent years. Multiple companies have launched massive investments in cloud computing such as Amazon [1] or Microsoft [11]. In such economic-driven environments, evaluating different alternatives for managing cloud computing and its future development is not straightforward. Current trends show a growth in cache and memory sizes that may increase pressure on traditional memory management structures. Additionally, most of the market is currently dominated by the use of Virtual Machines (VMs) but other alternatives exist trying to address their shortcomings. In particular, virtualization overheads incur a performance penalty and expose clients to hypervisor-level side-channel attacks.

Multiple solutions are arising in order to address these problems. One of the possibilities to manage memory growth and requirements is the use of intermediate address translations between virtual and physical memory. An alternative to virtualization is the use of hardware-managed isolation or bare-metal cloud. Our research focuses on the observation that these solutions are not necessarily orthogonal to each other and that their combination may offer new benefits that would not be available for each solution independently. Yet again, for each one of these solutions, there exist multiple alternatives but this research will be built on top of the Midgard [5] intermediate address space and the Core Slicing [18] hardware isolation solution. Ideally, we would like to combine the strong isolation mechanisms and performance boost from the Core Slicing solution with the faster memory translation of Midgard. The main observation is that the intermediary address translation is well suited to accomplish both objectives.

II. Background

This section will summarize and explain the main points from Midgard and Core Slicing and describe the state-of-the-art and limitations of current solutions for the problems at hand.

A. Core Slicing

Core Slicing is a bare-metal cloud solution aimed at minimizing virtualization overhead and hypervisor interference while maintaining strong isolation guarantees. Whenever a new client is instantiated, they are assigned a static slice of the machine's physical resources. Two clients are not allowed to simultaneously access the same resource.
Core slicing achieves this by introducing new hardware mechanisms. In Figure 1 a system overview of Core Slicing is displayed. We briefly analyze its mechanisms:

- Slice Manager: It's responsible for both establishing and dismantling user slices, as well as managing the allocation of resources when the client is instantiated.
- Slicevisor: The trusted (both by guests and host) and privileged portion of the slice manager.
- Lockable Filter Registers: Hardware registers responsible for limiting access to resources from a specific core. Once configured and locked, these registers become read-only until the core undergoes a reset.
- Core-local Secure Reset: Secure reset performed by the slicevisor.
- Sliceloader: First code loaded and executed after core-local secure reset.

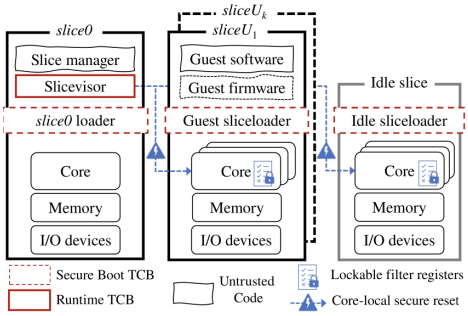

Fig. 1. Core Slicing system overview

B. Midgard

Midgard [5] addresses the problem of increasing address translation overheads that traditional TLB-based systems suffer from as memory size increases. Midgard introduces an intermediate address space between virtual and physical memory designed to be used to access the cache hierarchy, as can be seen in Figure 2. The aim is to reduce the frequency of expensive memory translation overheads from traditional TLB misses to LLC misses. This takes advantage of increasing cache sizes that benefit from lower LLC miss rates. A key feature of Midgard is the introduction of Virtual Memory Areas (VMAs), depicted in Figure 3. These are large contiguous units of memory on the Midgard address space that replace pages without incurring any fragmentation penalty on the physical memory as large pages would. Leveraging range checks on these VMAs, Midgard obtains lower translation overheads when accessing the cache through the use of a Virtual Lookaside Buffer (VLB) that replaces traditional TLBs at the cost of the introduction of a new address translation step from Midgard to the Physical address space. The benefits of Midgard increase as cache sizes increase and are able to fit primary, secondary and tertiary working sets of modern workloads.

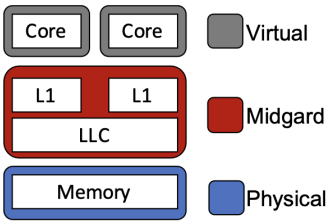

Fig. 2. Midgard memory hierarchy and address space arrangements

C. Side-channel attacks

This research project will deal with hardware isolation and therefore side-channel attacks are relevant to it. They have been rising in popularity during the last years [16], and they can be particularly devastating in a cloud computing context. These attacks exploit information that is leaked unintentionally by the hardware under normal operation and can lead to adversary clients being able to breach isolation barriers, infer

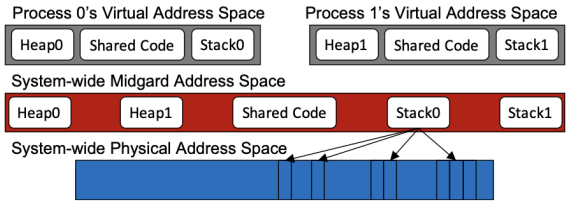

Fig. 3. Mappings between the Virtual, Midgard and Physical address spaces

sensitive data from other clients, and corrupt other clients' memory. The infamous Spectre [7] and Meltdown [9] vulnerabilities exemplify how devastating and far-reaching side-channel attacks can be.

One of our implementation requirements is for clients to be able to use several cores that can operate on the same shared memory, so some shared hardware structures must be used. Those hardware structures are potential targets for side-channel attacks. An especially important kind of side-channel attacks are Last-Level Cache side-channel attacks, which have been proven to be practical [10]. Therefore, it is imperative for any hardware-partitioning technique to also partition the shared LLC cache, and to try to minimize interference between clients in it.

D. LLC partitioning

Modern multiprocessors have large shared distributed LLC caches that are split into smaller LLC slices. Additionally, in modern systems, LLC slices are not necessarily associated with a single core, but rather multiple cores may share a single LLC slice [13, 4]. This project will consider scalable solutions that allow for different levels of isolation between cores independently of the number of cores per LLC slice, while preserving the granularity of being able to assign individual cores to clients.

As we saw in subsection II-C, cache partitioning is a vital component of hardware-partitioning. There are several approaches for performing logical cache partitioning. Core Slicing proposes using Intel's implementation [12] of the way partitioning [3] technique. Way partitioning divides the cache by ways, giving each client a subset of associativity ways of the total cache.

Page coloring [8] is a software cache partitioning technique that modifies the virtual to physical address mapping to map virtual pages to a specified range of cache sets.

Reconfigurable caches [14] is a cache-partitioning technique that partitions the cache into sets instead of ways, maintaining the associativity. In order to do so, the decoders of the SRAM memory arrays are modified to support partitioning.

PIPP [17] performs pseudo-partitioning by controlling the promotion and insertion policies, but does not provide strong isolation guarantees.

Vantage [15] is a cache-partitioning technique that provides isolation without decreasing associativity by providing statistical guarantees and dividing the cache into a managed zone and an unmanaged zone.

III. Problem Definition & Motivation

The combination of Midgard and Core Slicing to try to take advantage of the benefits from both offers a variety of design possibilities and questions. These include, for example, the possibility of using a single Midgard address space or several, the restructuring of memory permission checks and the spaces in which they are performed or the new possibilities for implementing isolation using the intermediate address space. Another example of design possibilities is page management, which is performed by the clients themselves in the Core Slicing solution, but could be changed in our solution, or modified to control how the translation pages are updated by implementing privilege checks or control measures. Other problems require compromises, such as the one between perfect isolation of cores and the capability for them to work together on shared memory instead of using messaging. This also affects, once again, the location of permission checks and raises new questions concerning cache coherence.

We establish the following requirements for our design:

- Clients should have complete control over their slice with minimal interference from a slicevisor (guaranteeing isolation between clients at the hardware level).
- Each client's slice should behave exactly the same as a normal Midgard-enabled CPU. The client should be able to take advantage of the faster address translation of Midgard and have complete control over their slice.
- A single client must be able to use multiple cores working on the same shared memory.
- Considering that complete isolation of resources is impossible due to the previous point, the system should aim at minimizing side-channel attacks that could be performed between different clients.

We argue that existing cache partitioning techniques do not fit well with the combination of Midgard and hardware-partitioning, for the following reasons:

- *Way partitioning* diminishes associativity, which in many cases leads to higher miss rates. It also limits the partitioning granularity based on the maximum associativity of the cache. Moreover, it spreads all accesses across all LLC slices, which may lead to longer cache-to-core latencies and may allow more side-channel attacks due to all clients sharing all LLC slices.
- *Software page coloring* would induce pressure on Midgard's VLB, which can only hold a few entries. The Midgard Address Space operates on Virtual Memory Areas, not pages. In order to do page coloring, a maximum size would have to be imposed on VMAs, which would limit their usefulness and may result in too many VLB misses.
- *Reconfigurable caches* require modifying the SRAM hardware structure, which we would like to avoid, as SRAM arrays are highly optimized components that are difficult to change. Moreover, it has additional hardware support for features not needed in hardware-partitioning, as the bring-up and shut-down of clients induce a very specific partition creation and destruction pattern.
- *PIPP* only provides pseudo-partitioning, which is not a strong enough guarantee for hardware-partitioned servers.
- *Vantage* provides only statistical guarantees, and requires dividing the cache into managed and unmanaged regions.

We can take advantage of the partition of the shared LLC into different LLC slices and the particularities of partition creation and destruction imposed by Core Slicing (a client's partition is never moved in cache, new client partitions are only created where already shut-down clients' partitions were) in order to create an LLC partitioning scheme better suited for both Midgard and hardware-partitioned servers.

IV. Design

A. Base design

There exist multiple viable solutions to try and profit from the benefits of both Midgard (faster address translation, support for bigger memory sizes) and Core Slicing (strict partitioning between clients with minimal external intervention, higher performance with respect to VMs). We have explored these design choices and their trade-offs, arriving at our final design.

Our proposed approach uses a system-wide hardware-partitioned Midgard space. This system-wide space allows for a single coherence domain in which all cores from the same client can communicate. A key insight is that we can assign to each client a disjoint partition of this intermediary address space that can be fully managed by each client. This property is guaranteed by prepending to each Midgard address access the client ID associated with the requesting core. This partitioning of the Midgard address space may seem too restrictive (as all partitions have the same size, regardless of the number of cores owned by each client), but the Midgard address space is a virtual address space, so its size can be increased without much penalty by just increasing the number of bits of Midgard addresses. Doing this coarse-grained partitioning by prepending the client ID bits helps us avoid the need to perform permission checks on Virtual to Midgard translation, which is a frequent operation.

This can be implemented via hardware through the use of the structures proposed in the original Core Slicing solution (lockable filter registers).

In the same way as the Core Slicing paper, we propose to partition the LLC between clients, being indexed and tagged using Midgard addresses. This partition is not trivial and will be explained in detail in subsection IV-B. Private caches are also indexed and tagged using Midgard addresses, only needing to perform Midgard to physical address translation on LLC misses.

Since the clients' address spaces are already disjoint, memory access checks are not necessary until we leave the Midgard address space on LLC misses Figure 5. This is not true for the original Core Slicing solution which performs these check on every TLB miss, which are much more frequent than LLC misses Figure 4.

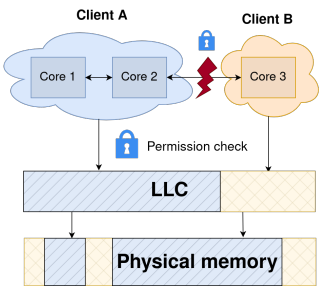

Fig. 4. High-level overview of Core Slicing's permission checks

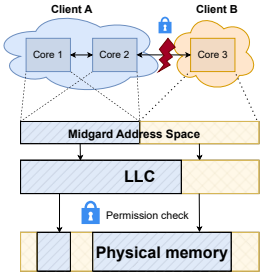

Fig. 5. High-level overview of our solution's permission checks

Note however that this permission check frequency reduction does not directly translate into performance benefits due to the existence of several buffers between the page table walker and the core, where these checks can be performed in parallel.

When going beyond the Midgard address space into the physical memory address space, our solution aligns again with the one proposed by Core Slicing, needing to perform permission checks on the physical addresses generated by each core.

B. LLC partitioning

In modern systems, the LLC is divided into LLC slices. Each LLC slice is associated to a set of cores. We will call the set of all cores associated to an LLC slice and that LLC slice a *memory node*. Those LLC slices are used together to form a shared distributed bigger LLC. Address interleaving is used to determine the home LLC slice for a given address (which is statically determined by the address).

Core Slicing uses way partitioning to divide each LLC slice amongst all clients. However, each client will see an LLC with a much smaller degree of associativity than the real LLC, which can hinder performance. Moreover, way partitioning spreads all clients' cache lines across all LLC slices, creating unnecessary distance between cache data and the cores and exposing all LLC slices to side-channel attacks.

We propose a better partitioning that takes advantage of the Midgard address space to give strong isolation guarantees while maintaining the cache associativity for each cache partition. This partitioning should satisfy the following requirements:

1) Memory used by a client should stay in the LLC as close as possible to that client's cores.
2) Uniformly distributed memory accesses should be uniformly distributed among the memory nodes.
3) We should have a reasonable per client distribution of the cache memory. This means that in a system where a client is using two thirds of the cores, he should also have an isolated partition of the cache memory representing two thirds of the total cache size.
4) We must allow dynamic reconfiguration of unused cores, as described in the project requirements. However, we will assume that the configuration of cores of an already running client won't be changed until that client is fully shutdown.

Note that way-partitioning only ensures the last two properties, it does not address the other two and reduces the cache associativity for each client. To address these issues, we have come up with new solutions and a combination of these solutions to fulfill all of these requirements for cache partitioning.

1) Inter-node cache partitioning: This solution assumes that each memory node is fully owned by a single client. The isolation is performed by assigning to each client the LLC slices that correspond to the memory nodes that the client owns.

Properties 1, 3 and 4 are trivially satisfied by any such solution. However, uniformly distributing each client's memory accesses across all their LLC slices is a challenge. In the worst case, if only the client ID is used to select an LLC slice, each client would only be able to use one LLC slice, even if the client owns all the cores of the machine. Our proposal attempts to use all the LLC slices in all the memory nodes owned by the client.

As we stated in subsection IV-A, all Midgard addresses have the client ID prepended to them, so we can use it to index a table containing the list of memory nodes owned by each client. The table also contains how many memory nodes each client owns. An example of such a table can be seen in Figure 6.

Client ID	Size	Memory node list			
1	2	3	2	-	-
2	1	4	-	-	-
3	0	-	-	-	-
4	1	1	-	-	-

Fig. 6. Example of inter-node partitioning additional hardware structure

In order to choose a memory node from all the memory nodes owned by a given client, we will use some bits from the address to index the memory node list. The address fields when accessing LLC can be found in Figure 7. The *node selection* bits are used to select a memory node among all the ones owned by the client that issues the memory request.

To ensure that memory accesses are properly distributed between nodes, we want to use the lowest bits possible in the Midgard address to determine which node each accessed block should be stored in. Nonetheless, in order to avoid distributing pages among memory nodes (which in some commercial implementations [6] may have different DDR memory controllers), the lowest bits we can choose must be taken from those following the page offset. Note that this does not ensure that VMAs are not distributed between different cache slices but this should not have any downsides.

In order to find the home memory node for a given memory access, the memory node list of the client that issues the

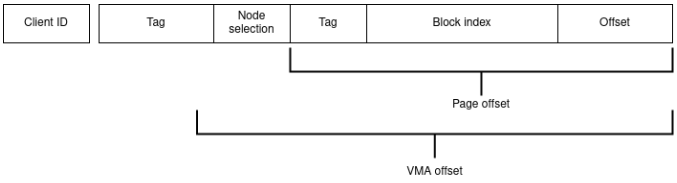

Fig. 7. Midgard address fields when accessing LLC

memory request is accessed, and the memory node at position *node selection* is selected (wrapping around the end of the list by using a modulo operation with the size of the list). More formally, the selected node would be

$$\texttt{NodeList}[\texttt{ClientID}][\texttt{NodeSel} \,\%\, \texttt{ListSize}[\texttt{ClientID}]]$$

Note that the home memory node assigned to each address is statically determined and won't change during a client's lifetime, as the core configuration of each client remains static while it is running.

This partitioning system has two main limitations:

- *Additional memory usage*. The proposed hardware structure allows retrieval of the home memory node for a given address in $O(1)$. However, assuming we can have as many clients as cores, and n being the number of memory nodes, the data structure would require n entries, each entry consisting of n pointers to a memory node, and each pointer requiring $\log_2 n$ bits. In total, the whole data structure needs $O(n^2 \log_2 n)$ bits, and the data structure would probably need to be replicated to each memory node, requiring in total $O(n^3 \log_2 n)$ bits in the whole system. To better understand the cubic growth, a system with 256 memory nodes would need $\sim$ 100Mb for the additional hardware structure and a system with 1024 memory nodes would need $\sim$ 7.5Gb. An alternative would be to store the data structure as a linked list: each memory node stores a pointer to the next memory node, which decreases the overall system memory requirements to $O(n^2 \log_2 n)$, but increases the home memory node lookup time to $O(n)$.
- *Each memory node must be entirely owned by a single client*. There are systems where there is more than one core per LLC slice. This partitioning schema would force a client to own all of the cores associated with the LLC slice. Moreover, it prevents the following memory optimization: in order to combat the cubic growth, we could group several LLC slices into a single memory node, to reduce the required memory of the additional hardware structure. However, with this partitioning scheme, a single client would have to own all of the cores of each bigger memory node, which may be counter-productive.

2) Intra-node cache partitioning: This solution aims at ensuring the partitioning by restricting the blocks each client is able to access within each memory node. That is, instead of assigning different LLC slices to different clients, it will isolate several clients inside the same LLC slice.

In order to partition the blocks within each node, the simplest way would be to replace the most significant bits from the block index with the client ID and move the replaced bits to the tag. This way, we are dividing the blocks evenly among all the clients Figure 8. Unfortunately, this does not take into account that a given client may be using more cores than another and is therefore expected to use a larger portion of the cache. In order to address this, we propose an auxiliary table replicated in each memory node associating to each client ID a variable number of fixed bits, determined when the client is initialized depending on the number of cores owned by that client. These fixed bits will replace the most significant block index bits when accessing the cache and the variable size will be managed by taking bits from the tag. An example can be seen in Figure 9. Note that this can be implemented on hardware using a simple mask on the bits from the tag and the block index.

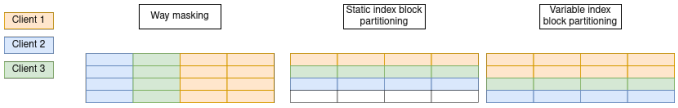

Fig. 8. Example of how a 4-way associative cache of 4 blocks would be divided between three clients with 2, 1 and 1 cores respectively

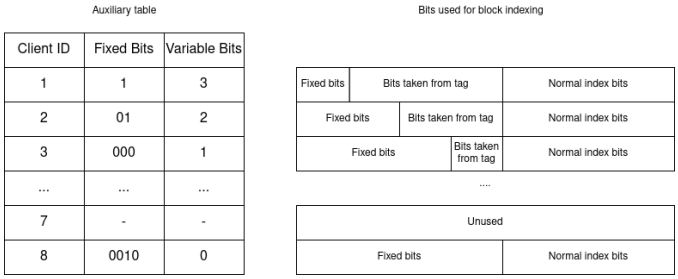

Fig. 9. Example of block indexing for a system with 8 cores using intra-node isolation

This solution is able to distribute the memory accesses among all memory nodes, preserves the associativity of the original cache when partitioning the cache nodes block-wise, is able to distribute the memory unevenly between clients and supports dynamic client reconfiguration of unused cores.

This solution has two main disadvantages:

- *Core fragmentation*: Due to the division into fixed bits and variable bits, cores can be assigned to clients only in powers of two. That is, a client may only own a number of cores that is a power of two. Moreover, repeatedly removing and adding new clients may cause additional fragmentation which would result in the inability to allocate multiple cores to a client even though there are several cores available (due to the fixed bits assignment).
- *Distance to cache elements*: As there is no control on the home LLC slice of a given address, a client may have to regularly access a remote memory node in which the client doesn't own any cores, harming performance.

3) Combination of both solutions: In order to solve some of the disadvantages of the proposed solutions, we propose a final partitioning scheme that combines elements from both of them.

In order to combine both solutions, we will define a new concept: a cluster. We will evenly divide the memory nodes into clusters. We will handle inter-cluster partitioning using the solution proposed in subsubsection IV-B1. Each cluster is then managed using the intra-node cache partitioning technique presented in subsubsection IV-B2. Each client will hold a list of clusters in which it has at least one core. In order to guarantee fair distribution across clusters (which means that a cluster in which a client owns more cores should be the home memory node of an address more frequently than a cluster in which a client owns less cores), the client needs to also store how many cores it owns in each cluster.

In order to support more efficient home cluster selection, instead of storing the number of cores owned in each cluster, we are going to store the cumulative sum (how many cores are owned in total counting the previous clusters in the list and this cluster). Then, we select the home cluster for an address by taking the node selection bits from the address, reducing them modulo the total number of cores and comparing it to the cumulative sum of each cluster. The home cluster will be the first cluster whose cumulative sum is greater than the modulo-reduced index. An example can be seen in Figure 10 for a client who has 4 cores in cluster 3, 4 cores in cluster 1, 1 core in cluster 2 and 1 core in cluster 4.

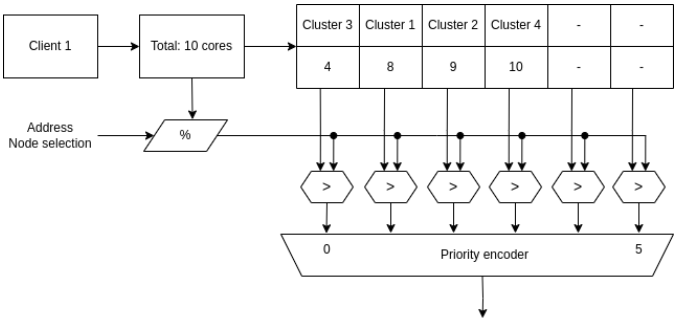

Fig. 10. Example of combined solution

This solution addresses the limitation of inter-node cache partitioning. The managing of each cluster is still intra-node cache partitioning, so there's no minimum granularity for the number of cores owned by each client. A client may own a single core in the whole system. It also addresses the memory bottleneck of the hardware structure proposed in subsubsection IV-B1, as we store a list of clusters instead of a list of nodes. The memory used by this hardware structure can be further reduced by only duplicating the hardware structure to each cluster (instead of each core), by limiting the number of clusters in which a given client can own cores and by limiting the total number of clients.

Our combined solution also addresses the limitations of intra-node cache partitioning. The average distance to LLC cache slices is reduced, as cache blocks are only stored in clusters where a client owns at least one core. Moreover, clusters where the client owns more cores have more cache space assigned to them. The fragmentation problem is also mitigated, as although a client may only own a power-of-two number of nodes inside each cluster, it may own nodes across multiple clusters. To avoid the fragmentation problem when repeatedly removing and adding new clients, each cluster can be assigned a preferred number of cores to allocate. Therefore, one cluster could be used for allocating 4 cores for each client, another for allocating 2 cores for each client and another 1 core for each client that owns a core in that cluster.

Security: Inter-node partitioning and inter-intra node partitioning techniques also provide higher security guarantees than the way partitioning technique proposed by Core Slicing. With way partitioning, all memory accesses from all clients are distributed amongst all memory nodes. Our proposed techniques encapsulate each client's memory accesses to the LLC slices associated to cores that the client owns. This not only has the direct benefit of LLC slices being only shared by clients with cores associated to that LLC slice, but also helps diminish the Network-on-Chip (NoC) traffic, as clients can only send messages on the NoC between memory nodes where they own cores, instead of sending messages between arbitrary memory nodes. This may help mitigate potential NoC Denial-of-Service attacks by malicious clients. Nevertheless, total mitigation of issues arising from these problems is outside the scope of this paper.

V. Methodology

To assess our LLC partitioning proposals, we are going to evaluate some performance metrics when running a sample workload. Those performance metrics are based on the main contributions of our LLC partitioning schemes with respect to previous work:

- Last-Level Cache miss ratio.
- Distribution of memory accesses amongst all the different LLC slices of the system.

The sample workload we have chosen is GAP PageRank from the GAP [2] graph processing benchmark with a $\sim$100 MB dataset.

Due to timing constraints, we have only been able to evaluate a few benchmarks with small datasets. However, we believe the chosen benchmark results are representative of all our experiments and therefore can be extrapolated to datasets of different sizes.

We have run the benchmark on an emulated bare-metal machine running Alpine-Linux. The underlying machine is emulated using QEMU (version 8.1, ARM64, from PARSA laboratory) running an SMP configuration with 2 cores.

The QEMU emulator is only used to generate a trace of memory accesses (which include the virtual memory address and issuing core identifier). We have written a plugin in Rust that is integrated into QEMU and generates a trace file containing each memory access.

In order to compute the previously mentioned metrics from the trace file, we have implemented our own cache simulator. Our simulator supports:

- Set-associative caches with the LRU replacement policy.
- Multi-level cache hierarchies.
- Multiple Last-Level Cache slices.
- LLC Way partitioning.
- LLC Inter-node partitioning.
- LLC Intra-node partitioning.
- LLC Inter-Intra-node partitioning.

The simulator has been written from scratch in C++ and we have hand-crafted a test suite to verify the correct implementation of the different cache simulator features.

All experiments have been run assuming a system with a private L1 and a shared L2 cache. The L1 cache is a 64KiB 4-way set-associative cache with 64 byte blocks and the parameters of the L2 vary with each experiment.

Note that although several clients may be sharing the hardware-partitioned system and own different cores, we only simulate one of the clients. That single client will have full access to the 2 cores of the QEMU system. The different cache partitioning parameters do take into account the existence of multiple clients in the system, but they are not simulated in QEMU and their memory accesses are not extracted, as our cache partitioning proposals fully isolate different clients at the logical level our cache simulator implements.

VI. Results & Analysis

We have performed three different experiments to compare our proposed LLC partitioning schemes against way partitioning.

A. Way partitioning vs. Intra-node partitioning

The goal of this experiment is to compare the miss rate of way partitioning and intra-node partitioning on an LLC with a single cache slice. The LLC is partitioned using both way partitioning and intra-node partitioning. We assume an 8-way associative cache, where the client owns one-eighth of the cache: 1 way in the case of way partitioning and one-eighth of the sets in the case of intra-node partitioning.

The GAP benchmark was executed, and all of the memory accesses were mapped to the same client. The results can be seen in Figure 11. Note that the client only has one-eighth of the capacity of the total LLC size.

We can see that for very small LLC sizes, there is not much difference (as the working set does not fit in the cache and almost every access is an eviction either way). However, as we increase the LLC size, we see how the associativity difference starts mattering more (from 8-way associativity with intra-node partitioning to 1-way associativity with way partitioning). We notice a reduction in miss rate of up to 63.34% comparing way partitioning to intra-node partitioning. Finally, as the LLC size increases even more, we notice that we can fit almost all the workload in the LLC, and associativity becomes irrelevant, so the difference between both partitioning schemes shrinks. Note that caches that big do not make sense in real systems, but we included them in our analysis for a more thorough exploration of the cache behavior.

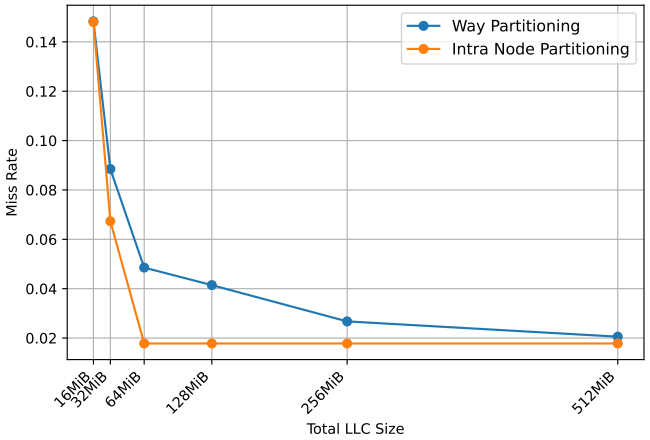

Fig. 11. Miss rate comparison of Way Partitioning and Intra Node Partitioning for different LLC sizes

Overall, we see that our partitioning system reduces substantially (up to 63.34%) the miss rate compared to way partitioning in systems with reasonable cache sizes.

B. Way partitioning vs. Inter-node partitioning

In this experiment we want to analyze the access distribution between LLC slices for both way partitioning and inter-node partitioning. We consider a system with 8 memory nodes (each one of the 8 LLC slices having 2 cores associated to it). In that system, our client owns 3 cores, 2 in one LLC slice and another in a different slice. Each LLC slice is 4 MiB and 16-way associative.

The access distribution results can be seen in Figure 12.

We can see that way partitioning distributes the address accesses across all the LLC slices. This incurs not only performance overhead (the data is further from the cores, which can be especially important for instructions, as it means the cores will be stalled for more cycles while waiting for the LLC slice to provide the instructions) but also security problems (all clients can have data in all LLC slices and a given client may overload the NOC with coherence messages between LLC slices, and all clients share all LLC slices' cache controllers).

We can see that inter-intra node partitioning behaves much better, with the accesses being restricted to the LLC slices where the client owns cores. Moreover, we can see that the proportionality of accesses is quite close to the ideal one: the client owns double the cores in the cluster associated to LLC slice 0 than in the cluster associated with LLC slice 1, so the ideal access distribution would be 66% and 33%. The empirical values are 63.93% and 36.07%, which are close to the ideal ones.

C. Way partitioning vs. Inter-Intra-node partitioning

The goal of this experiment is to compare the miss rate of way partitioning and inter-intra-node partitioning in a system

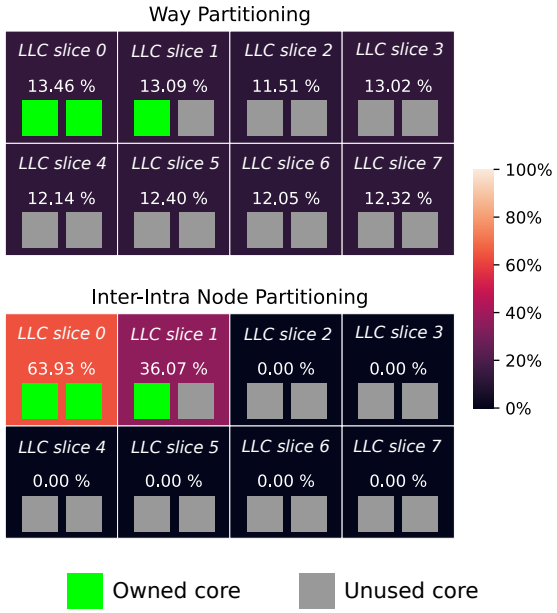

Fig. 12. Access distribution of different LLC slices with cores owned by different clients under different partitioning schemes

with 8 LLC slices, each slice being associated to a single core. For this setup, our client will have a single core, and the LLC will be 8-way associative. We will vary the LLC slice size in order to see how the miss rate changes. The GAP benchmark was executed and all of the accesses were assigned to the same client.

The results can be seen in Figure 13.

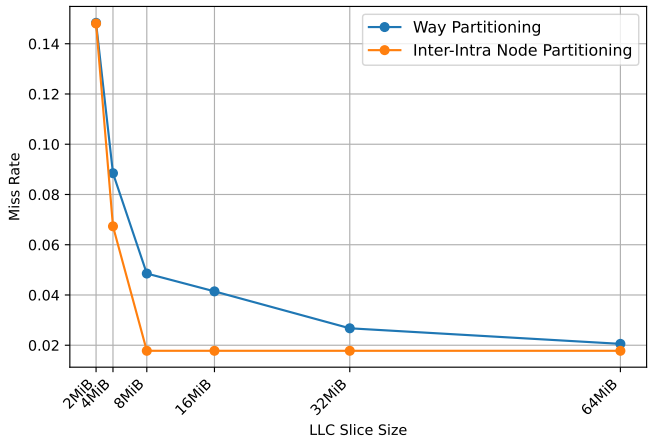

Fig. 13. Miss rate comparison of Way Partitioning and Inter-Intra Node Partitioning for different LLC slice sizes

We can see the same pattern as with the experiment shown in subsection VI-A. With a very small LLC cache slice, replacements are frequent because only a very small part of the working set fits in the cache. However, as we slightly increase the LLC slice size, we can see big differences in miss rates between way partitioning and our proposed partitioning caused by the associativity difference: 1-way with way partitioning vs 8-way with inter-intra-node partitioning. As the LLC slice size grows, it starts fitting and fitting more parts of the dataset, until it reaches a point where the associativity doesn't have any impact on the miss rate, as the cache is too large, and both way partitioning and inter-intra node partitioning achieve the same miss rates. Note that caches that big do not make sense in real systems, but we included them in our analysis for a more thorough exploration of the cache behavior.

Overall, we observe a big reduction in miss rates (up to 63%) when using inter-intra-node partitioning versus using way partitioning in the datapoints with realistic LLC slice sizes.

VII. Future Work

There are several enhancements and directions for possible future work:

- Benchmark our system with a more diverse set of workloads and dataset sizes, not just graph analytics.
- Analyze different ways to reduce the memory footprint of the inter-node partitioning table.
- Study how to integrate I/O partitioning into our system.
- Delve deeper into possible security issues when partitioning, such as side-channel attacks. An especially interesting topic would be hardware partitioning and isolation in DRAM controllers, which have to be shared by different clients.

VIII. Conclusion

Our final design effectively combines the Midgard intermediate address space and the bare-metal cloud hardware-isolation from Core Slicing. In doing so, it takes advantage of existing synergies between the two solutions to not only obtain client isolation with minimal hypervisor interference and faster address translation but also maintain high associativity in the cache and reduce stress on communication resources between LLC slices. Additionally, core assignment granularity to clients is supported at the individual core level, even supporting multiple cores associated to a single shared LLC slice, and physical interference between clients arising from shared cache controllers is reduced when compared to the original Core Slicing solution.

References

[1] *Amazon Web Services*. https://ir.aboutamazon.com/quarterly-results/default.aspx. Accessed: 2023-10-21.

[2] Scott Beamer, Krste Asanovic, and David A. Patterson. "The GAP Benchmark Suite". In: *CoRR* abs/1508.03619 (2015). arXiv: 1508.03619. URL: http://arxiv.org/abs/1508.03619.

[3] Derek Chiou et al. "Dynamic Cache Partitioning via Columnization". In: (Jan. 2000).

[4] Andrei Frumusanu. *Amazon's Arm-based Graviton2 Against AMD and Intel: Comparing Cloud Compute*. 2020. URL: https://www.anandtech.com/show/15578/cloud-clash-amazon-graviton2-arm-against-intel-and-amd/2 (visited on 12/21/2024).

[5] Siddharth Gupta et al. “Rebooting Virtual Memory with Midgard”. In: *Proceedings of the 48th Annual International Symposium on Computer Architecture*. ISCA ’21. Virtual Event, Spain: IEEE Press, 2021, pp. 512–525. ISBN: 9781450390866. DOI: 10.1109/ISCA52012.2021.00047. URL: https://doi.org/10.1109/ISCA52012.2021.00047.

[6] *Intel Xeon Scalable Sapphire Rapids*. https://www.screenhacker.com/intel-sapphire-rapids-xeon-4-tile-mcm-annotated/. Accessed: 2023-11-18.

[7] Paul Kocher et al. “Spectre Attacks: Exploiting Speculative Execution”. In: *Commun. ACM* 63.7 (June 2020), pp. 93–101. ISSN: 0001-0782. DOI: 10.1145/3399742. URL: https://doi.org/10.1145/3399742.

[8] Jiang Lin et al. “Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems”. In: *2008 IEEE 14th International Symposium on High Performance Computer Architecture*. 2008, pp. 367–378. DOI: 10.1109/HPCA.2008.4658653.

[9] Moritz Lipp et al. “Meltdown: Reading Kernel Memory from User Space”. In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018.

[10] Fangfei Liu et al. “Last-Level Cache Side-Channel Attacks are Practical”. In: *2015 IEEE Symposium on Security and Privacy*. 2015, pp. 605–622. DOI: 10.1109/SP.2015.43.

[11] *Microsoft Cloud*. https://www.microsoft.com/en-us/investor/earnings/fy-2023-q4/press-release-webcast. Accessed: 2023-10-21.

[12] K. T. Nguyen. “Usage models for cache allocation technology in the Intel Xeon processor E5 v4 family”. In: (Feb. 2016).

[13] Sung Park. *ARMs Race: Ampere Altra takes on the AWS Graviton2*. 2021. URL: https://blog.cloudflare.com/arms-race-ampere-altra-takes-on-aws-graviton2/ (visited on 12/21/2024).

[14] Parthasarathy Ranganathan, Sarita Adve, and Norman P. Jouppi. “Reconfigurable Caches and Their Application to Media Processing”. In: *SIGARCH Comput. Archit. News* 28.2 (May 2000), pp. 214–224. ISSN: 0163-5964. DOI: 10.1145/342001.339685. URL: https://doi.org/10.1145/342001.339685.

[15] Daniel Sanchez and Christos Kozyrakis. “Vantage: Scalable and efficient fine-grain cache partitioning”. In: *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. 2011, pp. 57–68.

[16] Jakub Szefer. “Survey of Microarchitectural Side and Covert Channels, Attacks, and Defenses”. In: *Journal of Hardware and Systems Security* 3.3 (Sept. 2018), pp. 219–234. DOI: 10.1007/s41635-018-0046-1. URL: https://doi.org/10.1007/s41635-018-0046-1.

[17] Yuejian Xie and Gabriel H. Loh. “PIPP: Promotion/Insertion Pseudo-Partitioning of Multi-Core Shared Caches”. In: *Proceedings of the 36th Annual International Symposium on Computer Architecture*. ISCA ’09. Austin, TX, USA: Association for Computing Machinery, 2009, pp. 174–183. ISBN: 9781605585260. DOI: 10.1145/1555754.1555778. URL: https://doi.org/10.1145/1555754.1555778.

[18] Ziqiao Zhou et al. “Core slicing: closing the gap between leaky confidential VMs and bare-metal cloud”. In: *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. Boston, MA: USENIX Association, July 2023, pp. 247–267. ISBN: 978-1-939133-34-2. URL: https://www.usenix.org/conference/osdi23/presentation/zhou-ziqiao.