IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received May 23, 2019, accepted June 10, 2019, date of publication June 19, 2019, date of current version July 3, 2019.

*Digital Object Identifier 10.1109/ACCESS.2019.2923855*

# Simulating the Network Activity of Modern Manycores

**MARCOS HORRO, GABRIEL RODRÍGUEZ, AND JUAN TOURIÑO, (Senior Member, IEEE)**
Computer Architecture Group, CITIC, Universidade da Coruña, A Coruña 15071, Spain

Corresponding author: Marcos Horro (marcos.horro@udc.es)

This research was supported by the Ministry of Economy, Industry and Competitiveness of Spain, Project TIN2016-75845-P (AEI/FEDER/EU) and by the Ministry of Education under Grant FPU16/00816.

**ABSTRACT** Manycore architectures are one of the most promising candidates to reach the exascale. However, the increase in the number of cores on a single die exacerbates the memory wall problem. Modern manycore architectures integrate increasingly complex and heterogeneous memory systems to work around the memory bottleneck while increasing computational power. The Intel Mesh Interconnect architecture is the latest interconnect designed by Intel for its HPC product lines. Processors are organized in a rectangular network-on-chip (NoC), connected to several different memory interfaces, and using a distributed directory to guarantee coherent memory accesses. Since the traffic on the NoC is completely opaque to the programmer, simulation tools are needed to understand the performance trade-offs of code optimizations. Recently featured in Intel's Xeon Scalable lines, this interconnect was first included in the Knights Landing (KNL), a manycore processor with up to 72 cores. This work analyzes the behavior of the Intel Mesh Interconnect through the KNL architecture, proposing ways to discover the physical layout of its logical components. We have designed and developed an extension to the Tejas memory system simulator to replicate and study the low-level data traffic of the processor network. The reliability and accuracy of the proposed simulator is assessed using several state-of-the-art sequential and parallel benchmarks, and a particular Intel Mesh Interconnect-focused locality optimization is proposed and studied using the simulator and a real KNL system.

**INDEX TERMS** computer architecture, cache coherence, distributed cache directory, high-performance computing, architectural simulator

## I. INTRODUCTION

Hardware trends towards exascale systems point to multi-level massive parallelism [1]: thousands of interconnected nodes where each node contains several tens of cores. The number of cores per die has steadily increased over the past decade as a response to the end of Dennard's scaling and the slowdown of Moore's law, and manycore architectures represent a big share of the current HPC market.

The development of memory systems has not followed the same pace. The gap between the performance of modern memories and processors is known as the memory wall, and represents maybe the biggest current challenge in computer architecture. New emerging technologies, such as 3D-stacked or non-volatile memories (NVM), try to mitigate this issue. Modern architectures feature heterogeneous memory systems, providing simultaneous access to different technologies that allow applications to leverage their distinct characteristics and trade-offs.

However, versatility represents a big challenge for programmers, who need to understand and leverage the complex trade-offs involved. Architectural models and simulators become fundamental to analyze the effect of candidate code transformations and optimizations. The development of these models and simulators, however, is far from trivial, due to the scarce documentation available from manufacturers. It becomes necessary to undertake reverse engineering of live systems to fully develop architectural models. This discovery process, based on performance measurements of real systems, allows to develop configuration models for architectural simulators. These tools are more flexible than hardware counters, allowing to analyze the fundamental

The associate editor coordinating the review of this manuscript and approving it for publication was Junaid Shuja.

reasons for performance differences between different system configurations and code optimizations.

This work undertakes the analysis of the Intel Mesh Interconnect (MI), first present in the Intel Xeon Phi Knights Landing (KNL) [2], and later introduced in the Xeon Scalable line, both in Skylake server processors [3] and in the latest Cascade Lake architecture [4]. In an MI chip memory is kept coherent using a distributed directory, and both the memory system and the affinity of coherence nodes and processors are configurable to tune the architecture to the application demands. We build a model of this complex architecture by analyzing the available documentation and filling the gaps using reverse engineering and statistical techniques. We include this model into Tejas [5], an open-source architectural simulator. This work makes the following contributions:

- We propose a mechanism to discover the physical layout of the logical components (cores and CHAs) of an MI-based processor, as well as the mapping of memory blocks across CHAs and memory interfaces.
- We incorporate a model of the memory accesses of the MI architecture into the Tejas architectural simulator. This extension has been made publicly available in [6].
- We validate the model by comparing the behavior of the simulated implementation against a real Intel Knights Landing processor using sequential and parallel benchmarks.
- We develop a new, MI-focused optimization to minimize the coherence traffic through the NoC, analyzing the reasons for its performance advantages using the simulator.

In this paper, we focus on the analysis of the Intel KNL, the first architecture to feature the Intel Mesh Interconnect and the one which includes the largest number of cores in a single die: up to 72, for 56 in Cascade Lake, and 28 in Skylake-SP. Although the KNL architecture was recently discontinued, this interconnect is still the Intel's choice for its HPC product lines. The techniques and software developed in this work are directly reusable for analyzing MI-based processors.

The rest of the paper is structured as follows: Section II motivates the work. Section III delves deeper into many-core architectures and the Intel KNL. Section IV presents the reverse engineering process to map logical components of an MI-based architecture to its physical layout. Section V introduces the Tejas simulator. Section VI shows how to incorporate the developed model into the simulator. Section VII presents the experimental validation of the model. In Section VIII we present a case study on optimizing the coherence traffic of KNL taking into account the effect of the distributed cache directory. Related work is discussed in Section IX, and Section X concludes the paper.

## II. OVERVIEW AND MOTIVATION

The design of the Intel Knights Landing [2], described in more detail in Section III, features a 2D mesh as depicted in Figure 1. It includes two different types of DRAM: a Multi-Channel DRAM (MCDRAM) to provide high-bandwidth using eight interfaces in the corners of the mesh, and two DDR controllers on opposite parts of the chip. In order to keep memory coherent, a distributed directory is employed. Each tile includes a Caching/Home Agent (CHA) in charge of managing a portion of the directory. Whenever a core requests a memory block that does not reside in the local tile caches, the distributed directory is queried. The flow of this request is depicted in Figure 1 and is described in detail in Section III-D.

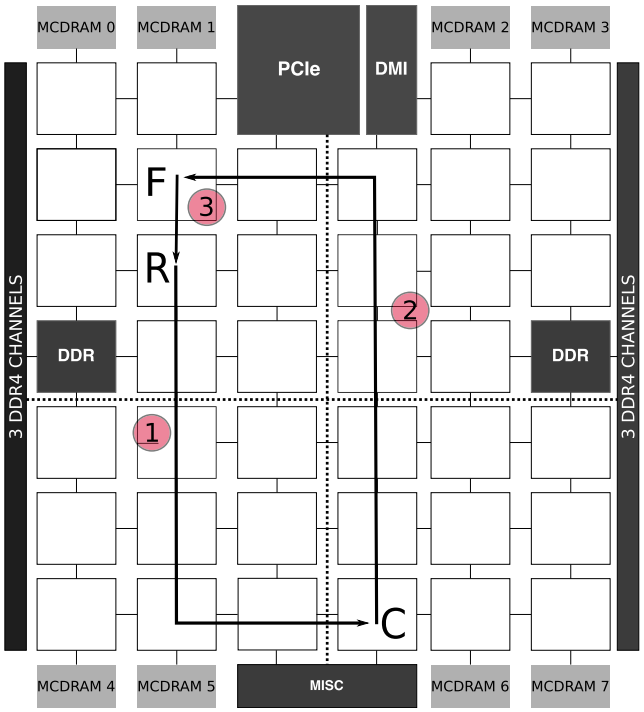



**FIGURE 1. Floorplan of the Intel KNL architecture. Each tile (square in the figure) contains two cores and their local caches.**

The architecture is generally assumed to be UMA when in Quadrant mode [2], [7] (see cluster modes in Section III-E). This is a reasonable assumption, given that memory blocks will be uniformly interleaved across the CHAs and memory interfaces using an opaque, pseudo-random hash function. As a result, the access latency will average out over a sufficient number of accesses for all cores. This is the behavior reported by works which do not consider the CHA location as a blocking factor in their experiments [7]. A challenging aspect is that the physical locations of logical entities on the 2D mesh are not exposed to the programmer, and will change across KNL units due to process variations. After reverse engineering these locations using the techniques detailed in Section IV, however, we observed that actual access latencies from different cores to a fixed memory block are far from UMA. More precisely, the coherence traffic causes a systematic degradation of memory performance which, on average, creates the illusion of UMA behavior. Figure 2 shows the actual access latencies from each tile in the mesh of a particular Intel x200 7210 processor to MCDRAM #0, for a memory block whose coherence data is contained in the

tile next to the memory interface. We note differences in access latency of up to 32 CPU cycles (a 27% overhead over the minimum observed latency of 117 cycles), which matches the theoretical time for a round trip around the mesh. In addition to the latency gap caused by the round trip, contention is generated on the network when all cores are continuously accessing all the CHAs in the mesh. Confining cooperating threads and associated coherence data to isolated regions of the mesh would reduce network footprint, a critical parameter for NoC performance [8]–[10]. For this reason, we employ the methodology presented in [11], further described in Section VIII, which demonstrates the potential of using an ad-hoc data distribution that takes into account the distance between cores and CHAs, i.e. the core-to-CHA affinity. Figure 3 shows the potential impact of controlling the maximum core-to-CHA distance when performing a `jacobi-1d` computation from the PolyBench/C suite [12]. The $x$ label represents the maximum distance, in CPU cycles, between the core that emits an access request and the tile containing the coherence information. As can be seen in the figure, there are different complex trade-offs at play: the cycles in which there are outstanding requests on the network decrease, but the instrumentation required to improve affinity increases $\mu$TLB misses. The net effect in total execution cycles is positive in this case, but this will not be the case for all workloads. The reasons for the performance offered by this and other traffic optimizations remain hidden, as the hardware offers no performance counters or other means to analyze the traffic over the NoC in detail. Besides, `jacobi-1d` is a relatively simple application. More complex workloads require a more detailed low-level view of the system in order to be fully understood.

In the following sections we fully describe the simulated architecture, as well as the Tejas simulator and the modifications required to adapt it to the Intel KNL design.

## III. MANYCORE ARCHITECTURES: INTEL KNIGHTS LANDING

The increasing demand of computational resources over the last decade has shifted architectural paradigms. The relationship between energy consumption and frequency, known as Dennard's scaling, is not linear anymore [13]. The "multicore crisis" in the past decade was partially a response to this problem. The idea behind manycore processors is to comply with thermal limitations by integrating a higher number of simpler, slower processors, capable of taking advantage of embarrassingly parallel applications or to execute many smaller workloads at the same time. Even though cores are simpler, they can communicate more efficiently since they are integrated inside a single processor die and connected to a network-on-chip (NoC).

Manycore organizations present a challenge for the memory system. Since more data-hungry cores coexist now inside a single die, the memory wall grows higher. Modern architectures propose to use heterogeneous memory hierarchies, which combine different memory technologies with their own characteristics and trade-offs. Traditional memory hierarchies are augmented with these new technologies as shown in Figure 4, contributing to reduce the gap between processor and memory speeds.

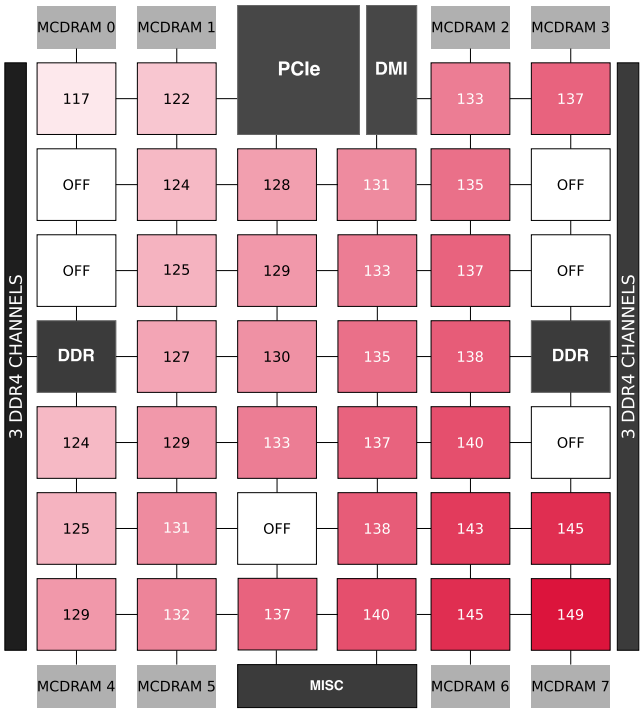



**FIGURE 2.** Heatmap of the measured access latency (in CPU cycles) from each tile in the mesh of an Intel Xeon Phi x200 7210 to a single block of memory associated to MCDRAM #0 and its adjacent CHA.

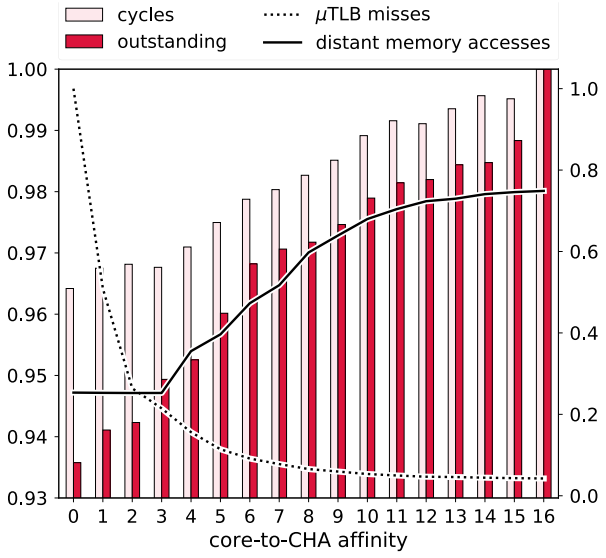



**FIGURE 3.** Effects of core-to-CHA affinities: Execution cycles, outstanding weighted cycles, $\mu$TLB misses, and accesses to distant memory interfaces for different core-to-CHA affinities. Results are normalized to the maximum value for each series, except for accesses to distant memory interfaces, which are normalized to the total number of memory accesses. $\mu$TLB misses and distant accesses are referenced to the right axis.

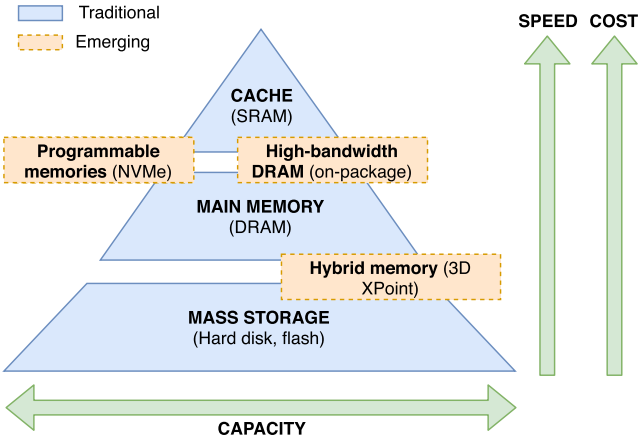



**FIGURE 4.** Traditional memory hierarchy updated with new emerging technologies.

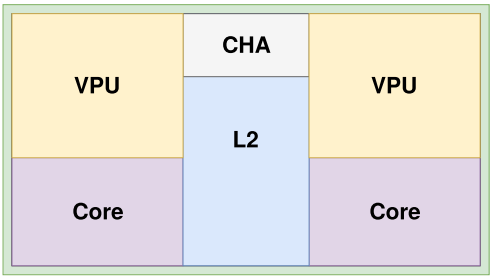



**FIGURE 5.** High-level tile organization in KNL.

This work focuses on the Intel Xeon Phi x200 architecture, codenamed Knights Landing (KNL), released in 2016 and discontinued in mid-2018. However, the distinguishing characteristics of its NoC, on which this work is focused, live on in the newer Intel developments for HPC, namely the Xeon Scalable processors, whose second generation codenamed Cascade Lake was announced in April 2019 [4]. Intel KNL is presented as a standalone x86 processor. In contrast, its predecessor, Knights Corner, was a co-processor which required a host processor. KNL is therefore not limited by the on-board memory size or the PCIe bus bandwidth. Moreover, the use of the x86 ISA enables KNL to execute different operating systems, legacy libraries, and general purpose applications. As such, KNL is more versatile than its predecessor and than current GPUs, which need applications to be rewritten following specific paradigms such as CUDA [14].

The rest of this section discusses the main characteristics of the Intel KNL: its internal organization, core architecture, on-die interconnect, memory system, and cluster and memory modes.

### A. INTERNAL ORGANIZATION

KNL integrates up to 72 cores organized in a 2D mesh of 38 tiles. Each tile comprises two cores, two vector processing units (VPUs), a shared L2 cache and a Caching/Home Agent (CHA), as shown in Figure 5. Depending on the particular processor model, the number of tiles with enabled cores varies between 32, 34 and 36, featuring 64, 68 and 72 enabled cores, respectively. Note that the CHA is always enabled for all tiles.

### B. CORE ARCHITECTURE

Intel KNL cores are two-wide out-of-order derived from low-power Silvermont cores, designed for Intel Atom processors but modified to make them suitable for HPC [2]. They have been enhanced with AVX-512-capable VPUs. Furthermore, each core has a private 32-kB L1 data cache, backed up by a 1-MB L2 shared with the other core in the same tile but private to it. The CHA manages a portion of the distributed cache directory, which stores the status and location of the most up-to-date copy of a memory line, queried when an L2 miss occurs.

### C. NETWORK-ON-CHIP

KNL features a 2D mesh NoC, replacing the ring topology used in Knights Corner, as depicted in Figure 1. Messages traverse the mesh using a simple YX routing protocol: a transaction always travels vertically first, until it hits its target row. Then, it begins traveling horizontally until it reaches its destination. Each vertical hop takes 1 clock cycle, while horizontal hops take 2 cycles. The mesh features 4 parallel networks, each customized for delivering different types of packets. The point of connection of each tile to the mesh is the CHA.

### D. MEMORY SYSTEM

KNL integrates two different types of DRAM memories (see Figure 1). Up to 16 GB of on-package 3D-stacked Multi-Channel DRAM (MCDRAM) provides high-bandwidth accesses through its eight concurrent interfaces. Besides, there are two more DDR interfaces controlling three DRAM channels each, adding up to 192 GB of memory. A distributed cache coherence mechanism using Intel MESIF [15] is employed. Each time a core requests a memory block that does not reside in the local tile caches, the distributed directory is queried. A message is sent to the appropriate CHA (message (1) in Figure 1). If the block already resides in one of the L2 caches in the mesh in Forward state,[1] the CHA will forward the request to the owner, which will send the data to the requestor in turn (messages (2) and (3) in the figure). In other cases, the data must be fetched from the appropriate memory interface. The data flow shown in the figure exemplifies one of the performance hazards inherent to the KNL architecture: although the data for the requested block lies in the forwarder tile *F*, just above the requestor *R*, the coherence data is stored far away in tile *C*. As it is, 18 cycles are required to transfer the data (10 vertical and 4 horizontal hops). But, if the directory information were stored either in the requestor or in the forwarder, the round trip time of data packets would be of only 2 cycles (2 vertical hops on the mesh).

### E. CLUSTER MODES

The affinity between memory interfaces, CHAs and cores can be configured in KNL through the so-called cluster modes:

[1] A cache containing a block in Forward state is in charge of serving said block upon a request. The requestor acquires the block in Forward state, while the sender changes it to Shared.

- *All-to-all*: data has no affinity at all. This is the most inefficient mode. It should only be used when memory modules are unevenly distributed across memory interfaces.
- *Quadrant*: the mesh is virtually divided into four clusters. The memory blocks managed by a CHA are guaranteed to be accessed through a nearby memory interface. The hash functions which assign memory blocks to CHAs and memory interfaces are not publicly disclosed. This mode is the de-facto standard for KNL operation.
- *Sub-NUMA cluster*: contiguous memory blocks are assigned to each cluster, interleaving cache lines among the memory channels in that cluster. The idea is to create different NUMA nodes isolating traffic within them. It is recommended for MPI and NUMA-aware applications only.

### F. MEMORY MODES

The MCDRAM memory may be configured into one of two modes: ''Flat'' memory, in which the address space is explicitly exposed as an independent NUMA domain and is available to the programmer; and ''Cache'' mode, in which MCDRAM serves as a transparent memory-side cache.

## IV. MAPPING THE KNIGHTS LANDING PROCESSOR

When working in the Sub-NUMA cluster mode the correspondence between logical and physical cores is explicit. Considering 64 cores, four different NUMA memory domains are created, and each core is associated to one of them: cores {0–15}, {16–31}, {32–47} and {48–63} belong to the four different logical clusters in the mesh. This allows one to carefully select the affinity for a team of processes executing an MPI application, knowing that the memory allocated to a processor will be guaranteed to lie in the local interfaces to each cluster, as will the associated directory information. It is not possible to exploit this paradigm using a multithreaded code (e.g., OpenMP) without simulating the distributed memory nature of multiprocess parallelism.

In the default Quadrant mode there is no indication as to the neighborhood relationships between different logical core IDs. This makes it impossible to reason about core affinities. Furthermore, even if we discovered core location and bound a team of threads to neighboring cores, each time a cache block is not locally available the requestor will need to query the associated CHA to discover the status and location of the block. This coherence data may reside in any part of the mesh. For this reason, it is not sufficient to know where each core is located in the physical mesh; we would also need to know where each CHA is located in order to carefully plan the memory accesses for each thread.

We reverse engineered the physical layout of an Intel x200 7210 processor, with 64 enabled cores, by profiling memory access latencies, building potential layout candidates, and iteratively discarding the ones which present a larger squared error with respect to the observed behavior. For this purpose, we systematically measure the access latency from each logical core ID to cache blocks located in each of the 8 MCDRAM interfaces and each of the 38 CHAs in the mesh. Note that, in Quadrant mode, blocks stored in a given MCDRAM interface can only be indexed by CHAs located in its same quadrant. We created a routine which, given a tuple $(C, CH, MC)$ containing core, CHA and MCDRAM IDs, locates a cache block stored in $MC$ and indexed by $CH$, and measures the latency of accessing it from $C$. This routine is detailed in Algorithm 1. We first initialize a sufficiently large region of memory (buffer $B$) so that we are reasonably sure that it will contain instances of all possible $(CH, MC)$ associations. Given that the hash function assigning blocks to CHAs and MCDRAM interfaces is reasonably uniform, this memory does not need to be extremely large (a few 4-kB memory pages are enough). Then, we test each cache block looking for one which is indexed by $CH$ and stored in $MC$. To do so, we access each block and flush it from the cache $N$ times in a loop. After the loop ends, we check which MCDRAM and CHA pair has at least $N$ accesses by using a custom kernel module which leverages the uncore Model Specific Registers (MSRs).[2] After we find a block associated to $(CH, MC)$, we repeatedly access it again, but this time we measure access latencies and calculate the average.

In the manner described above, we find the average access latency for all the valid $(C, CH, MC)$ tuples in the mesh. Note that we only need to obtain the latency for one out of each 2 cores, since cores in the same tile share the same CHA, and it is inferrable from `/proc/cpuinfo` that cores $(2x)$ and $(2x + 1)$ lie in the same tile. By analyzing the missing $(CH, MC)$ pairs, we discover the association of CHAs and MCDRAMs to quadrants. In particular, we find that data in MCDRAM interfaces $(2y)$ and $(2y+1)$ are indexed by CHAs $z$ such that $(z \mod 4 = y)$, e.g. MCDRAMs 0 and 1 are associated to CHAs $0, 4 \ldots 36$; MCDRAMs 2 and 3 to CHAs $1, 5 \ldots 37$; and so on.

Once these data are collected, we analyze them to determine where each pair $(C, CH)$ of cores and CHA is located on the physical mesh, taking into account the public KNL specifications. The floorplan includes 38 physical tiles, some of which have their cores disabled depending on the processor model.[3] Note that, despite having disabled cores, all tiles have fully functional CHAs and mesh interconnects. The actual location of the tiles with disabled cores is believed to change for each processor unit, depending on process variations. However, the CPUID instruction can be used to discover the actual $(C, CH)$ associations between cores and CHAs. It also provides the list of CHAs which do not have enabled cores. Armed with this information, and with our measured core-to-CHA-to-MCDRAM latencies, we build a squared error model for each candidate assignment of $(C, CH)$ pairs to

[2]We employ the `PERF_EVT_SEL_X_Y` and `ECLK_PMON_ÃĎÅăTRX_LOW/HIGH` registers to monitor CHAs and MCDRAMs, respectively [16]. We measure events `RxR_INSERTS.IRQ` and `RPQ.Inserts` [17].

[3]The exact count is 6 tiles with disabled cores in Intel x200 7210 and 7230 series, 4 in the 7250 series, and 2 in the 7290 series.

**Algorithm 1:** Measures Latencies for a $(C, CH, MC)$ Tuple

**Input**: Core $C$, CHA $CH$ and MCDRAM $MC$
**Output**: Avg. access latency from $C$ to $MC$ via $CH$

```
bind_to_core(C);
allocate_buffer(B);
for each cache block b in B do
    n = 0;
    start_cha_counters();
    start_mcdram_counters();
    while n < N do
        access(b);
        flush_cache_block(b);
        n = n + 1;
    end
    stop_mcdram_counters();
    stop_cha_counters();
    mc_b ← read_mcdram_counters();
    if mc_b == MC then
        cha_b ← read_cha_counters();
        if cha_b == CH then
            n = 0;
            start ← get_time();
            while n < N do
                access(b);
                flush_cache_block(b);
                n = n + 1;
            end
            stop ← get_time();
            L = start − stop;
            return L/N;
end
return error;
```

the physical mesh. In our Intel x200 7210, only 32 tiles have active cores. As such, we have to discover the actual location of these 32 tiles, plus the 6 disabled tiles. Taking into account that we know the associations of $(C, CH)$ pairs and quadrants, as detailed above, there are only $10! \times 10! \times 9! \times 9!$ different combinations, as two quadrants have 10 tiles while the remaining two quadrants have only 9 tiles each. This information allows us to reduce the possible combinations by a factor of $10^{21}$ with respect to the original 38! possible candidates. To reduce even further the number of possibilities we employ heuristics. First, we locate feasible candidates for the corner tiles, i.e. those contiguous to each MCDRAM interface. For this purpose, we identify the minimum experimental memory latency $L$ (117 cycles in our tests), and search for $(C, CH, MC)$ tuples with an access latency of at most $L$ plus a configurable error margin. In this way, we reduce the possible combinations for the 8 corners to under 200. Next, for each of these candidates, we build mean squared error models for placing the remaining tiles, and finally accept the one which shows the least squared error.

The obtained results present a clear pattern in the location of both CHAs and cores, as shown in Figure 6. The CHAs in each quadrant are sequentially arranged in a vertical fashion. Cores are assigned sequentially to CHAs, skipping those tiles with disabled cores. This technique allows to obtain the physical layout of any individual KNL unit immediately, by just checking which CHAs have disabled cores through CPUID instructions.

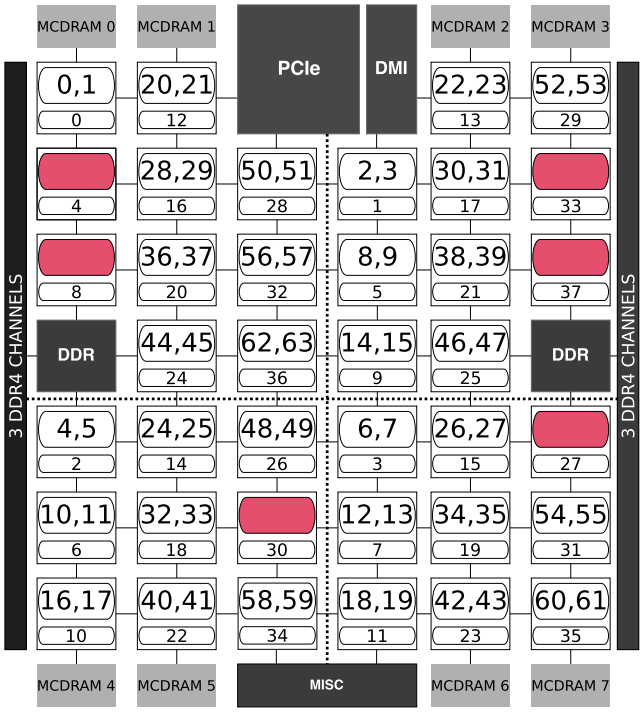



**FIGURE 6.** Result of our model. Each tile is formed by two cores (their IDs are enclosed in a large box) and a CHA (its ID enclosed in a small box). Tiles with blank boxes indicate that their cores are not active.

## V. TEJAS SIMULATOR OVERVIEW

Tejas is an open-source architectural simulator written in Java and C++ [5]. The simulation model is semi-event-driven: predictable activities follow an interactive cycle-level approach, such as advancing micro-instructions in the pipeline, whereas unpredictable tasks, such as load/store operations, follow an event-driven approach based on priority queues. Tejas decodes binaries dynamically during their execution in order to recreate the micro-operations that are used to feed simulation components, such as processor pipelines. Tejas also employs the McPAT [18] and Orion2 [19] power models in order to get statistics about energy consumption.

This framework has been validated against real hardware in terms of cycles, reporting acceptable mean errors varying between 11.45% for serial workloads and 18.77% for parallel workloads [20]. The simulator can be seen as two separate modules or layers: the emulator (front-end) and the simulation engine (back-end). The front-end instruments the code being executed, ensuring a fully functional execution. It is ISA-dependent, and in our case the x86 implementation is used. The back-end receives events from the front-end and

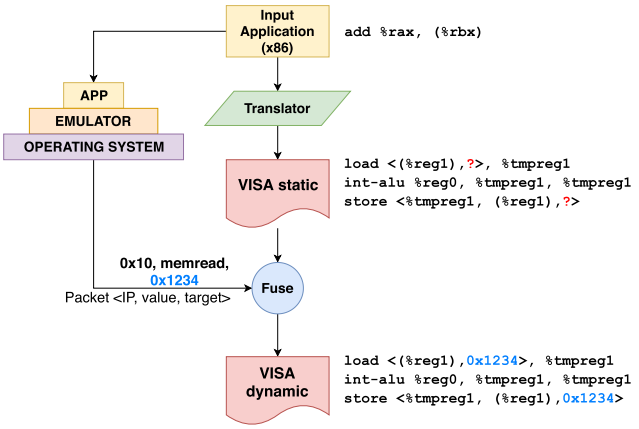



**FIGURE 7.** Translation process in Tejas.

simulates them in the configured architectural model. The following subsections describe each of these components in more detail.

### A. FRONT-END: THE EMULATOR

The emulator translates the trace of binaries into virtual micro-operations. This process is done in two steps (see Figure 7):

1) *Riscify* process: converts CISC x86 instructions to a simplified ISA called Virtual ISA (VISA). This process is done statically and does not include source or target memory addresses.
2) *Fuse* process: dynamically fills the missing memory addresses of the VISA instructions created in the previous step.

This module of Tejas is written in C++ using Pin [21] and, therefore, is basically an instrumentation of the code that is being executed. What the emulator collects is used to feed the simulation engine (the back-end), written in Java, which is in charge of starting all the pipelines and processing elements, and collecting all the statistics at the end of the execution. The communication between the emulator and the simulation engine is done through a shared memory region.

The main limiting factor of the front-end is the translation between the real micro-instructions and the VISA. Some CISC micro-instructions are translated into functionally equivalent, but not fully equivalent, VISA instructions. An example are SSE instructions, which are translated into equivalent floating-point ones. But, in other cases, some instructions may be ignored. This is the case for instructions such as AVX-512CD (released by Intel for the KNL architecture). This limitation is imposed due to the complexity of CISC architectures and in order to keep the simulated pipelines simple. Tejas' developers chose to focus on the most common x86 micro-instructions, usually obtaining a coverage of more than 99% of the binary instructions [5].

### B. BACK-END: THE SIMULATION ENGINE

The simulation engine can be seen as a set of interconnected components, sending and receiving messages through their

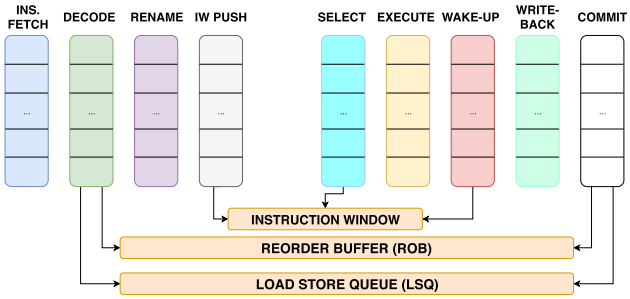



**FIGURE 8.** Stages and main registers of the Out-of-Order pipeline in Tejas.

ports. For the scope and interest of this work, we will briefly review how cores, memory system and interconnection networks, as well as their corresponding configuration parameters, are implemented in this framework.

#### 1) CORES

A Tejas core is a wrapper entity containing a pipeline and both data and instruction private caches. Two pipelines have been implemented in Tejas: Multi-issue in-order and Out-of-Order (OoO or O3). We used the O3 implementation, depicted in Figure 8. It consists of nine stages: instruction fetch, instruction decode, rename, instruction window push, instruction select, execute, wake-up, write-back and commit. The sizes of each functional unit and registers can be configured, allowing for a wide range of possibilities.

#### 2) MEMORY SYSTEM

The Tejas memory system is composed of a set of memory controllers and caches. The memory hierarchy can be seen as an inverted tree of caches where its root is the main memory. Regarding the software implementation, the cache coherence directory inherits from the cache class. A centralized directory is implemented, which is queried by last-level caches to discover the state and location of accessed memory lines.

The memory system is flexible, allowing to configure each cache as private or shared among a set of cores. Thus, buffer sizes, cache lines, MSHR (Miss Status Holding Registers), write mode and number of cache ports can be changed, opening a wide range of possible configurations.

The flow of a simulated cache request is depicted in Figure 9. When the request is received, it is first checked whether it is a hit or a miss. In the latter case, a new entry is added to the MSHR, allowing to perform other tasks while miss requests are handled by other parts of the mesh, either other caches or memory interfaces. When a lower level response is received, the event is removed from the MSHR and the response is processed: it can be a miss response to fulfill the missed request, a write ACK in order to mark the line as dirty, or an evict ACK in order to mark the line as invalid. When there is a miss in the last-level cache, the request is forwarded to the centralized cache directory, which will point to the location of the line, satisfying the request. The directory also handles coherence, using a simplified version of the MESI protocol. For instance, when a line is in shared state,

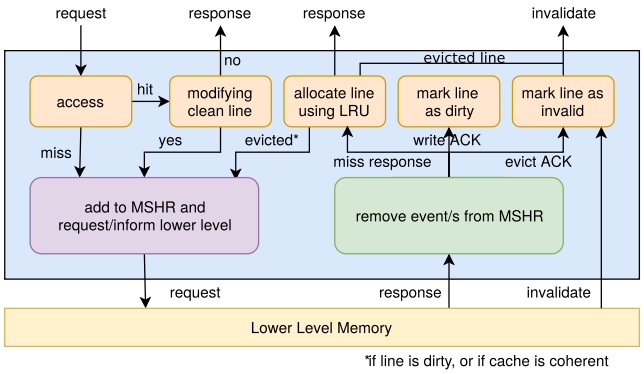



**FIGURE 9.** Cache behavior in Tejas (extracted from [5]).

the directory selects one of the sharers without taking into account priority or proximity.

#### 3) NETWORK-ON-CHIP

Tejas implements a generic NoC which can be configured in different manners. It is a crucial component, since it connects all the different elements present in an architecture. Many topologies (bus, ring, mesh, torus…) and routing algorithms (west-first, simple-XY, north-last…) are available.

Both the shape and the low-level parameters of the NoC are highly customizable. In the same manner, the dimensions and type of the topology, as well as the router latencies and capacities can also be modified. In Tejas, each router connects elements to the network and different elements can be connected to the network using the same interface.

## VI. MODELING KNL IN TEJAS

This section covers the approach we followed to implement the KNL architectural model in Tejas. This approach can be extrapolated to other architectures featuring Intel Mesh Interconnect such as the recent Cascade Lake. We particularly focus on the distinguishing features of the mesh interconnect: the 2D structure with tiles integrating cores and CHAs, employing a distributed directory for inter-tile coherence, and with distributed memory interfaces. The Tejas extensions described in this section have been made publicly available in [6].

Each of the following subsections focuses on one relevant aspect of the implementation, describing the modifications applied to the simulator using a high level of abstraction.

### A. TILES AND CORES

A tile is essentially a wrapper of cores, VPUs, caches and the CHA (see Figure 5). We implement that abstraction as two cores, a shared L2 and a CHA (described in more detail in Section VI-B) connected to the same router, and therefore sharing the same logical position in the network. The detailed Tejas configuration for each tile is shown in Table 1.

### B. MEMORY SYSTEM

The memory subsystem has suffered the most important modifications with respect to the original version of Tejas.

**TABLE 1.** Tejas configuration for modeling tiles in the KNL architecture.

| | Tejas structure | KNL parameters | Details |
|---|---|---|---|
| **L1** | Size<br>Associativity<br>Write mode<br>Latency | 32 kB<br>8-way<br>Write-back<br>4 cycles | x2 reads x1 write simultaneously |
| **L2** | Size<br>Associativity<br>Write mode<br>Latency | 1 MB<br>16-way<br>Write-back<br>13 cycles | |
| **TLB** | DTLB<br>iTLB | 256 entries<br>48 entries | 1-cycle access<br>4-cycle penalty |
| **Pipeline buffers** | ROB<br>Memory reservation station<br>LSQ | 72 entries<br>12 entries<br>12 entries (load buffers)<br>16 entries (store buffers) | |
| **Integer unit** | ALU<br>Mult<br>Div | 1 cycle<br>4 cycles<br>20 cycles | 2 units<br>12-entry RS |
| **FP unit** | ALU<br>Mult<br>Div | 2-3 cycles<br>6 cycles<br>25 cycles | x2 VPUs per core<br>20-entry FP operation RS |
| **Branch predictor** | Mode<br>BHR<br>Miss penatly | GSkew<br>12 bits<br>11-13 cycles | |

KNL introduces: 1) different memory domains, 2) a different cache coherence protocol (MESIF), and 3) a distributed cache directory.

#### 1) ACCESSING DIFFERENT MEMORY DOMAINS

In KNL, programs can choose from two different memories to allocate data: MCDRAM or DDR. Memory addresses above `0x3040000000` lie on MCDRAM, while addresses below that are bound to the DDR system. However, Tejas works with virtual memory addresses only, making it difficult to distinguish between different memory domains.

Using configuration files, we provide the ranges of physical memory that correspond to different memory domains. During emulation, we instrument memory allocations and find the mapping between virtual and physical pages. The virtual-to-physical page translation is passed to the simulation engine, in charge of the architectural modeling. When a last-level cache miss occurs, the simulation engine will employ the appropriate memory controller to fulfill the request.

#### 2) MESIF PROTOCOL

The main difference between MESI (included in Tejas) and MESIF is the inclusion of an F (Forward) state. As mentioned in Section III-D, a cache in F state will be in charge of serving requests. The requestor will acquire the block in F state and the sender will change to S (shared) state.

Figure 10 illustrates how the MESI state diagram is modified to include the new F state. From the implementation

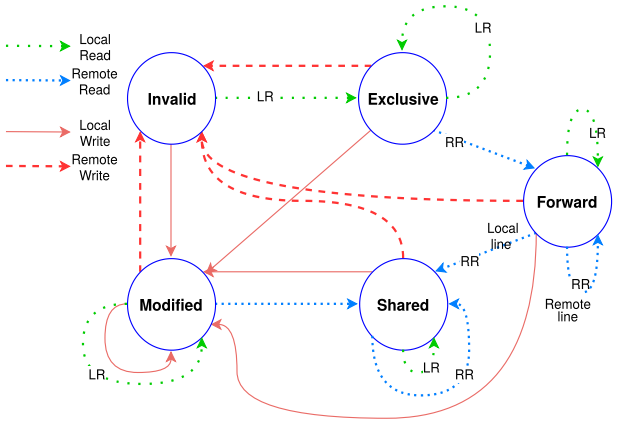



**FIGURE 10.** MESIF protocol implementation based on MESI.

point of view, this requires changes to the `CacheLine` class, which now stores the forwarder CHA in addition to all sharers.

#### 3) CACHING/HOME AGENT (CHA)

The original Tejas implementation of a centralized tag directory requires it to be placed in a fixed location in the mesh. Unlike Tejas, the Intel Mesh Interconnect architecture features a distributed system in which each CHA holds a portion of the full directory. We have implemented a `CHA` class as an extension of the `Directory` class, which is modified to include cache lines in Forward state. This new component is replicated once per tile. The last-level cache on each tile will send a miss request to its own CHA, which will respond if the data is locally available or will forward the request to the appropriate CHA for resolution.

In order to accurately simulate the distribution of memory lines across CHAs, we have mapped the association between the full set of physical memory lines and the 38 CHAs in the KNL floorplan using performance counters and a strategy similar to the one employed in Section IV for reverse engineering the physical location of logical components.

### C. INTERCONNECTION NETWORK

Although Tejas already implements a 2D mesh, it does not include an YX routing option such as the one used by the Intel Mesh Interconnect. Our implementation is basically a modification of the already included XY routing. All parameters are described in Table 2.

Furthermore, in Tejas the NoC decides which memory controller is chosen when a last-level cache miss occurs. Our implementation delegates this decision to the CHA containing the coherence information, as per Intel's design.

### D. OTHER CONSIDERATIONS

Note that not all of the architectural aspects of KNL have a direct translation to Tejas objects. For instance, Tejas employs a 9-stage pipeline, whereas KNL cores feature 5-stage pipelines. Additionally, Tejas does not support vectorization simulation.

**TABLE 2.** Tejas configuration for modeling the KNL NoC.

| Tejas structure | KNL parameters |
|---|---|
| Topology | 2D mesh |
| Flit size | 32 bytes |
| Routing algorithm | YX |
| Latency | 2 cycles (x-direction)<br>1 cycle (y-direction) |

These shortcomings can make the simulation of the cores' behavior inaccurate, but note that our main goal is the analysis of the traffic on the interconnect network and the behavior of the memory system. Thus, our simulator constitutes a very powerful tool for the architectural and behavioral analysis of the Intel Mesh Interconnect.

## VII. VALIDATION

We validated our KNL model in Tejas by comparing its behavior against a real KNL system. We have chosen two benchmark suites: PolyBench/C [12] and Parboil [22]. Both of them include benchmarks from very different domains: linear algebra computations, image processing, physics simulations, dynamic programming, data analytics... These algorithms are widely used in both scientific and commercial applications. The main difference between both suites is that PolyBench/C benchmarks are polyhedral, single-core oriented kernels, whereas Parboil includes more complex multithreaded applications. Using this mix allows us to analyze the accuracy of a single core working alone with the memory system, and also how the results vary when the traffic density in the system increases as more cores become active. This experimental design pursues a wide scope of the validation process.

### A. EXPERIMENTAL SETUP

Benchmarks were compiled using GCC 4.8.5 with the `-O1` optimization flag. This ensures a good dynamic coverage of binary codes by Tejas, which is lost when using more recent GCC versions or when enabling `-O2`. As mentioned

**TABLE 3.** Experimental setup.

| Parameters | | KNL PolyBench/C | Parboil |
|---|---|---|---|
| **Core** | Number | 1 | 64 |
| | Frequency | 1.3 GHz | |
| | Cluster mode | Quadrant | |
| **Memory** | Type | MCDRAM (16 GB) | |
| | Mode | Flat | |
| | Page size | 4 kB | |
| **Compiler** | Version | GCC v4.8.5 | |
| | Flags | -O1 -static -fopenmp | |

in previous sections, vectorization is disabled as AVX-512 is not supported by Tejas. The `-static` flag is used to link system libraries statically and improve static coverage. We fixed the core frequency at 1.3 GHz, disabling DVFS in order to minimize experimental variability. The most important KNL configuration parameters are summarized in Table 3.

We have used the PAPI/C library for measuring the events in the real machine, an Intel Xeon Phi x200 7210 processor. PolyBench/C integrates a set of macros for easily handling the configuration of this library. We have extended these macros to use PAPI/C in multithreaded applications [23], in order to conveniently measure events in Parboil codes.

Table 4 summarizes the equivalence between the events we want to measure in KNL and Tejas. Some Tejas events have been adapted to those measurable in KNL. For instance, there are no PAPI events or IMC counters in KNL for measuring the number of total MCDRAM accesses, but only MCDRAM responses to L2 read misses.

In order to test the memory system modeled in Tejas, we have ensured that all the benchmarks handle a workload that does not fit in the last-level cache, i.e. the footprint of each benchmark must exceed 1 MB for sequential workloads and 32 MB for parallel workloads. This guarantees that all components of the memory hierarchy play a part in the compound behavior of the system.

### B. RESULTS

We present the results for each event using two different metrics. The first one, shown below in Equation 1, is the relative error of the Tejas simulation with respect to the performance counters read in the KNL system (which will be referred to as *REL*). This metric has the disadvantage of not contextualizing the error in the scope of the execution, as the error increases exponentially when the event count tends to zero. For example, if a benchmark only makes a few hundred accesses to main memory, then a difference of tens of accesses will cause a large relative error, but this may have almost no effect in the context of the full execution if the data being read from memory is then heavily reused. In order to better contextualize errors, we also provide the number of errors per memory instruction, as shown in Equation 2 (which will be referred to as *INS*). This metric has advantages for our purposes: it is zero-centered and it does not suffer exponential variations for linear changes in its inputs. Both metrics become zero when the simulation exactly matches the behavior of the real system, and they are positive when the simulation overestimates a parameter and negative when it is underestimated. Note that both metrics become the same for the L1a event (number of L1 accesses).

$$\forall k_i \in V_{knl}, \quad t_i \in V_{tejas}$$

$$REL_i = \frac{t_i}{k_i} - 1 \tag{1}$$

$$INS_i = \frac{t_i - k_i}{k_{L1a}} \tag{2}$$

In order to account for experimental variabiliy we have executed each benchmark 10 times. The mean for each measured event is reported. Furthermore, when analyzing the results of multithreaded workloads we report the mean values across all cores.

#### 1) POLYBENCH/C RESULTS

Figure 11 shows the number of errors per memory instruction (*INS* metric) for the PolyBench/C codes. The maximum error found is 2.98 errors per each 100 memory instructions for the L2 accesses of the `syr2k` benchmark. For all events the mean error values are at or below one error per each 500 memory accesses. These results demonstrate that the simulated system is closely capturing the behavior of the real system, and the largest source of inaccuracy is the behavior of the local caches, which is known to be hard to simulate as it depends on other factors such as the interference or noise caused by the operating system, opaque techniques like hardware prefetching, etc.

Table 5 displays the results of Figure 11 (*INS* metric) in numerical form, together with the relative error (*REL* metric). As can be seen, sporadic high relative error values are much lower when put in context by the number of total memory accesses. For instance, the 30.57% relative error achieved for

**TABLE 4.** Equivalence between the event to measure, the PAPI event, and the event programmed in Tejas.

| Event ($V$) | KNL PAPI ($V_{knl}$) | Tejas simulator ($V_{tejas}$) |
|---|---|---|
| Cycles | CPU_CLK_UNHALTED:THREAD_P | Cycles taken |
| Instruction count | INSTRUCTIONS_RETIRED | Instructions executed |
| L1 data accesses (L1a) | MEM_UOPS_RETIRED:ANY_LD+ MEM_UOPS_RETIRED:ANY_ST | Memory Requests |
| L1 data misses (L1m) | MEM_UOPS_RETIRED:LD_DCU_MISS | L1[core] Misses |
| L2 accesses (L2a) | LLC_REFERENCES | L2[tile] Accesses |
| L2 misses (L2m) | LLC_MISSES | L2[tile] Misses |
| MCDRAM responses (MC) | OFFCORE_RESPONSE_0:MCDRAM | $\sum_{MC=0}^{7} Responses_{MC}$ |
| MCDRAM far (MCfar) | OFFCORE_RESPONSE_0:MCDRAM_FAR | $\sum_{MC \notin Q(c)}{}^{*} Responses_{MC}$ |
| MCDRAM near (MCnear) | OFFCORE_RESPONSE_0:MCDRAM_NEAR | $\sum_{MC \in Q(c)}{}^{*} Responses_{MC}$ |

* $Q(c)$ returns the number of quadrant for a core $c$

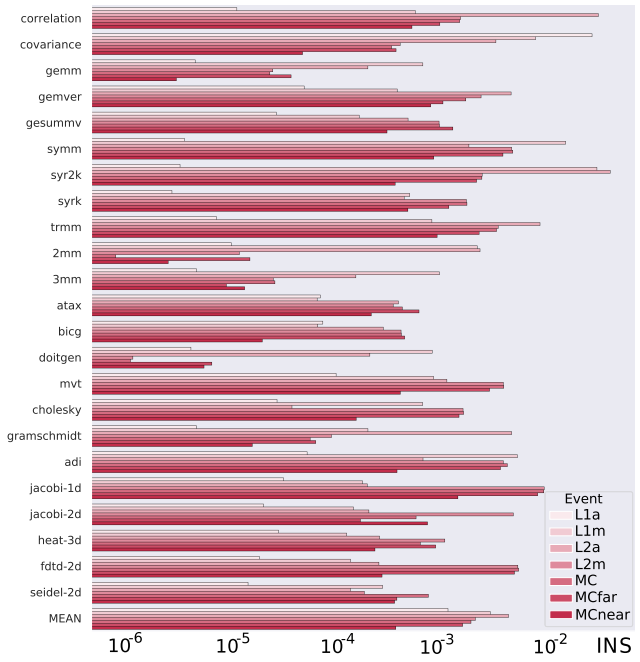



**FIGURE 11.** *INS* error metric for PolyBench/C benchmarks (logarithmic scale).

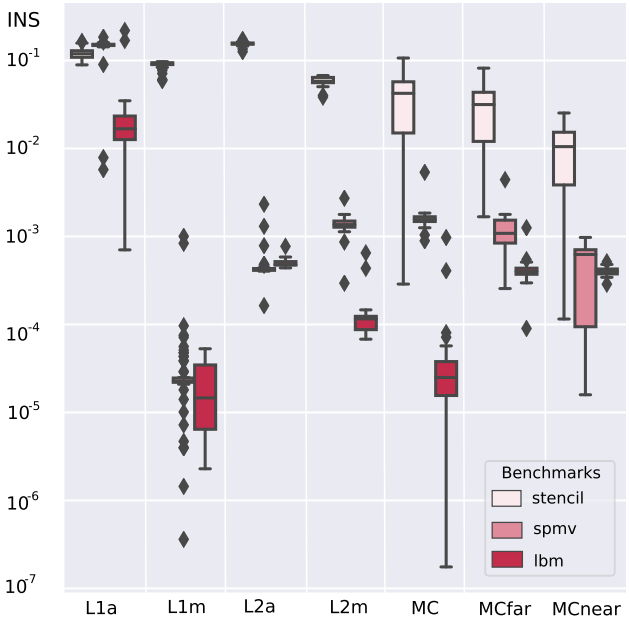



**FIGURE 12.** *INS* error metric for Parboil benchmarks. Boxplots represent the variability of the results obtained by each core (logarithmic scale).

the number of L2 misses of `trmm` might seem high, but it only accounts for 300K misses out of 56.8M total L2 accesses, i.e. the difference in L2 miss rate is only 0.5%. High relative errors usually occur for benchmarks with a memory footprint very close to the total available cache space. The simulator tends to underestimate the number of misses in this case, as it does not take into account the impact on cache occupancy of small low-level routines, such as those being run by the operating system.

The results presented in this section do not include the values for execution cycles and instruction count. The simulation of these events turns out to be very inaccurate with respect to the real execution, achieving relative errors above 50%. This is to be expected, since Tejas is not simulating the execution pipeline but translating CISC instructions to the Tejas ISA (VISA), and this is not the focus of our simulation.

### 2) PARBOIL RESULTS

Among the different benchmarks included in the Parboil suite, we have selected those with a significant number of accesses to main memory with respect to the number of instructions executed, in order to stress the simulator. Figure 12 shows the number of errors per memory instruction (*INS* metric) for these benchmarks. Boxplots are used instead of static bars, as the benchmarks are multithreaded and there is a different number of errors for each simulated core. Table 6 shows the relative (*REL*) and normalized (*INS*) errors expressed as the mean of all cores. In general, both metrics increase with respect to the single-threaded PolyBench/C benchmarks. The reason is that the number of instructions, being divided across 64 threads, is now much smaller, and the errors introduced by the emulation system represent a larger share of the total. Note how the most significant distortion in the results corresponds to the number of L1 accesses, which is completely dependent on the way in which the emulator translates CISC instructions into VISA in multithreaded codes. This in turn results in larger errors in L1 misses and L2 accesses. The memory hierarchy gradually "absorbs" these errors, and results become more accurate as we descend towards slower memories.

## VIII. CASE STUDY: ANALYSIS OF COHERENCE TRAFFIC OPTIMIZATIONS

Section II described how the location of the coherence data in the mesh is of great importance for memory latency in this architecture. As depicted in Figure 1, the time to access a memory line that resides in a nearby tile can be high if the coherence data is stored in a farther one. In order to alleviate this effect, the maximum distance between cores and the coherence data of their assigned memory can be controlled. This optimization shows potential, as evidenced by the effect of the core-to-CHA affinity shown in Figure 3 of Section II, but also a complex trade-off between different factors such as the number of TLB misses, the reduction in access latencies, and the contention on different areas of the mesh.

We modified a `jacobi-1d` stencil from the PolyBench/C suite so that cores $i$ and $i + 2$ swap their data at the end of each timestep. The reason is to reuse data from adjacent threads in other to quantify the impact of physical distance between a requestor (core) and the responder (CHA). We used two different maximum core-to-CHA distances, 16 and 0, so that all cores write to memory lines whose coherence information resides in any tile in the mesh (maximum distance $= 16$) or in their local tile (maximum distance $= 0$). We then ran the experiment using 64 threads and two different mappings: scattered and co-located. The first one is a direct

**TABLE 5.** *REL* and *INS* error metrics for each event analyzed in the PolyBench/C kernels. The mean absolute value of all benchmarks for each event and metric is shown in the last row.

| | L1a | | L1m | | L2a | | L2m | | MC | | MCfar | | MCnear | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | REL | INS | REL | INS | REL | INS | REL | INS | REL | INS | REL | INS | REL | INS |
| correlation | $\simeq 0$ | $\simeq 0$ | 0.0009 | 0.0004 | -0.0454 | -0.0231 | -0.1137 | -0.0012 | -0.1118 | -0.0012 | -0.0968 | -0.0007 | -0.1593 | -0.0004 |
| covariance | 0.0201 | 0.0201 | 0.0121 | 0.0059 | 0.0051 | 0.0025 | 0.0358 | 0.0003 | 0.0296 | 0.0003 | 0.0432 | 0.0003 | -0.0174 | $\simeq 0$ |
| gemm | $\simeq 0$ | $\simeq 0$ | 0.0211 | 0.0005 | 0.0063 | 0.0002 | 0.0753 | $\simeq 0$ | 0.0703 | $\simeq 0$ | 0.1572 | $\simeq 0$ | 0.0389 | $\simeq 0$ |
| gemver | $\simeq 0$ | $\simeq 0$ | 0.0034 | 0.0003 | -0.0384 | -0.0035 | -0.1095 | -0.0018 | -0.0812 | -0.0013 | -0.0659 | -0.0008 | -0.1497 | -0.0006 |
| gesummv | $\simeq 0$ | $\simeq 0$ | -0.0042 | -0.0001 | -0.0119 | -0.0004 | 0.0241 | 0.0007 | 0.0243 | 0.0007 | 0.0434 | 0.0010 | -0.0314 | -0.0002 |
| symm | $\simeq 0$ | $\simeq 0$ | 0.0415 | 0.0114 | -0.0049 | -0.0014 | 0.2466 | 0.0035 | 0.2541 | 0.0036 | 0.2734 | 0.0029 | 0.1837 | 0.0007 |
| syr2k | $\simeq 0$ | $\simeq 0$ | -0.1890 | -0.0224 | -0.2370 | -0.0298 | 0.1352 | 0.0019 | 0.1341 | 0.0019 | 0.1603 | 0.0017 | 0.0830 | 0.0003 |
| syrk | $\simeq 0$ | $\simeq 0$ | 0.0020 | 0.0004 | -0.0017 | -0.0003 | -0.2331 | -0.0013 | -0.2353 | -0.0013 | -0.2142 | -0.0009 | -0.2657 | -0.0004 |
| trmm | $\simeq 0$ | $\simeq 0$ | 0.0013 | 0.0006 | -0.0130 | -0.0065 | -0.3057 | -0.0026 | -0.2980 | -0.0026 | -0.2768 | -0.0018 | -0.3305 | -0.0007 |
| 2mm | $\simeq 0$ | $\simeq 0$ | -0.0566 | -0.0017 | -0.0598 | -0.0018 | 0.0235 | $\simeq 0$ | -0.0016 | $\simeq 0$ | 0.0394 | $\simeq 0$ | -0.0201 | $\simeq 0$ |
| 3mm | $\simeq 0$ | $\simeq 0$ | 0.0029 | 0.0007 | -0.0005 | -0.0001 | -0.0452 | $\simeq 0$ | -0.0467 | $\simeq 0$ | -0.0220 | $\simeq 0$ | -0.0977 | $\simeq 0$ |
| atax | -0.0001 | -0.0001 | 0.0034 | 0.0001 | -0.0191 | -0.0003 | 0.0177 | 0.0003 | 0.0217 | 0.0003 | 0.0414 | 0.0005 | -0.0441 | -0.0002 |
| bicg | -0.0001 | -0.0001 | 0.0034 | 0.0001 | -0.0138 | -0.0002 | 0.0210 | 0.0003 | 0.0212 | 0.0003 | 0.0303 | 0.0004 | -0.0042 | $\simeq 0$ |
| doitgen | $\simeq 0$ | $\simeq 0$ | 0.0207 | 0.0006 | 0.0053 | 0.0002 | 0.0025 | $\simeq 0$ | 0.0024 | $\simeq 0$ | 0.0185 | $\simeq 0$ | -0.0469 | $\simeq 0$ |
| mvt | -0.0001 | -0.0001 | 0.0046 | 0.0007 | -0.0061 | -0.0009 | 0.1463 | 0.0030 | 0.1476 | 0.0030 | 0.1419 | 0.0022 | 0.0613 | 0.0003 |
| cholesky | $\simeq 0$ | $\simeq 0$ | -0.0159 | -0.0005 | -0.0010 | $\simeq 0$ | 0.0867 | 0.0012 | 0.0878 | 0.0013 | 0.1060 | 0.0011 | 0.0345 | 0.0001 |
| gramschmidt | $\simeq 0$ | $\simeq 0$ | -0.0006 | -0.0002 | -0.0134 | -0.0035 | 0.2682 | 0.0001 | 0.1540 | $\simeq 0$ | 0.2450 | 0.0001 | 0.1937 | $\simeq 0$ |
| adi | $\simeq 0$ | $\simeq 0$ | -0.0634 | -0.0040 | -0.0041 | -0.0005 | 0.0845 | 0.0030 | 0.0928 | 0.0032 | 0.1062 | 0.0028 | 0.0341 | 0.0003 |
| jacobi-1d | $\simeq 0$ | $\simeq 0$ | -0.0056 | -0.0001 | -0.0031 | -0.0002 | 0.1670 | 0.0072 | 0.1651 | 0.0071 | 0.1942 | 0.0062 | 0.1032 | 0.0011 |
| jacobi-2d | $\simeq 0$ | $\simeq 0$ | -0.0059 | -0.0001 | -0.0042 | -0.0002 | 0.1044 | 0.0037 | -0.0128 | -0.0004 | 0.0051 | 0.0001 | -0.0653 | -0.0006 |
| heat-3d | $\simeq 0$ | $\simeq 0$ | -0.0024 | -0.0001 | -0.0037 | -0.0002 | 0.0300 | 0.0008 | 0.0178 | 0.0005 | 0.0330 | 0.0007 | -0.0266 | -0.0002 |
| fdtd-2d | $\simeq 0$ | $\simeq 0$ | -0.0019 | -0.0001 | -0.0034 | -0.0002 | 0.0742 | 0.0040 | 0.0756 | 0.0041 | 0.0919 | 0.0038 | 0.0158 | 0.0002 |
| seidel-2d | $\simeq 0$ | $\simeq 0$ | -0.0186 | -0.0002 | -0.0094 | -0.0001 | 0.0130 | 0.0001 | -0.0519 | -0.0006 | -0.0350 | -0.0003 | -0.1000 | -0.0003 |
| MEAN | 0.0009 | 0.0009 | 0.0209 | 0.0022 | 0.0222 | 0.0033 | 0.1027 | 0.0016 | 0.0929 | 0.0015 | 0.1061 | 0.0012 | 0.0916 | 0.0003 |

**TABLE 6.** *REL* and *INS* error metrics for each event analyzed in the Parboil benchmarks expressed as the mean of all cores.

| | L1a | | L1m | | L2a | | L2m | | MC | | MCfar | | MCnear | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | REL | INS | REL | INS | REL | INS | REL | INS | REL | INS | REL | INS | REL | INS |
| stencil | 0.1137 | 0.1137 | -0.2747 | -0.0905 | -0.3370 | -0.1555 | -0.3679 | -0.0593 | 0.9477 | 0.0378 | 0.9501 | 0.0284 | 0.9479 | 0.0095 |
| spmv | 0.1459 | 0.1459 | -0.0010 | $\simeq 0$ | -0.0153 | -0.0004 | 0.0584 | 0.0014 | 0.0687 | 0.0016 | 0.0696 | 0.0012 | 0.0693 | 0.0004 |
| lbm | 0.0178 | 0.0178 | 0.0016 | $\simeq 0$ | -0.0069 | -0.0005 | -0.0038 | -0.0001 | -0.0003 | $\simeq 0$ | -0.0005 | $\simeq 0$ | 0.0001 | $\simeq 0$ |

thread-to-core mapping, i.e. thread $i$ is assigned to core $i$. Since cores are cyclically distributed among the quadrants as shown in Figure 6, cooperating threads will be scattered across the mesh. In the second mapping we take advantage of the reverse-engineered floorplan in Figure 6 to optimally co-locate cooperating threads in neighboring tiles.

We analyzed the behavior of the benchmark for both mappings and both core-to-CHA distances (16 and 0), measuring different events with performance counters. Results are shown in Table 7. As can be seen, setting the maximum core-to-CHA distance to zero provides a performance benefit of 6.25%, and controlling the location of threads using the co-located mapping increases this benefit further to 7.76%. However, simply controlling the location of threads on the mesh, but not the core-to-CHA distance has a negligible effect. The counters offer no evidence of any significant difference in cache misses or memory accesses which may account for the performance difference between the configurations mentioned above. On the contrary, the number of memory accesses increases for the fastest configurations (those with zero core-to-CHA distance), due to the increase in TLB misses. We need to analyze the low-level traffic of the mesh in order to understand the reasons for the difference in performance.

The idea behind these optimizations is to eliminate the majority of the coherence traffic through the mesh. If most of the blocks requested by a core have their coherence data stored in the local CHA, the coherence traffic should be significantly reduced. We employed our simulator to analyze the traffic passing through each router, as shown in the heatmap of Figure 13. The total reduction in the number of packets traveling through the network is 18.7% when optimizing both the location of the coherence data (i.e., maximum core-to-CHA distance = 0) and the mapping of the cooperating threads (co-located mapping). The reduction in total traffic might be surprising, taking into account that the number of L2 misses increases very slightly. In order to understand the low-level behavior of the system, we studied how the reduction in traffic is distributed across the four different types of packets: *query* (requests for data directed to a CHA), *forward* (a request from a CHA to an L2 cache or MCDRAM interface to forward a block to another L2), *data* (a response to a forward request containing the requested block), and *eviction* (a write-back message from an L2 cache to an MCDRAM interface containing an evicted data block). Data packets are not reduced, as we are not changing cache locality whatsoever. Query data is reduced by 20.4% and forward data by 343%. Furthermore, the average time from the generation of a query packet to the delivery of the corresponding data packet is reduced from 76.84 to 50.87 cycles using both optimizations (maximum distance 0 and co-located mapping), i.e. the number of hops performed by each coherence packet is drastically reduced. Note that an L2 miss being resolved in its local CHA will generate a query packet and a data packet

**TABLE 7.** Selected events for the executions of the modified `jacobi-1d` stencil by thread 0. The rows present two different maximum core-to-CHA distances (16 and 0 cycles). The columns show the counters being measured and the thread mapping employed for each execution. Values are reported in millions.

| | cycles | | L1a | | L1m | | L2m | | MC | | MCfar | | MCnear | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| distance | scat | co-loc | scat | co-loc | scat | co-loc | scat | co-loc | scat | co-loc | scat | co-loc | scat | co-loc |
| 16 | 84.127 | 83.986 | 21.775 | 21.803 | 0.571 | 0.571 | 0.561 | 0.561 | 0.019 | 0.016 | 0.014 | 0.013 | 0.005 | 0.004 |
| 0 | 78.865 | 77.597 | 22.409 | 21.868 | 0.571 | 0.571 | 0.569 | 0.581 | 0.053 | 0.043 | 0.023 | 0.016 | 0.022 | 0.026 |

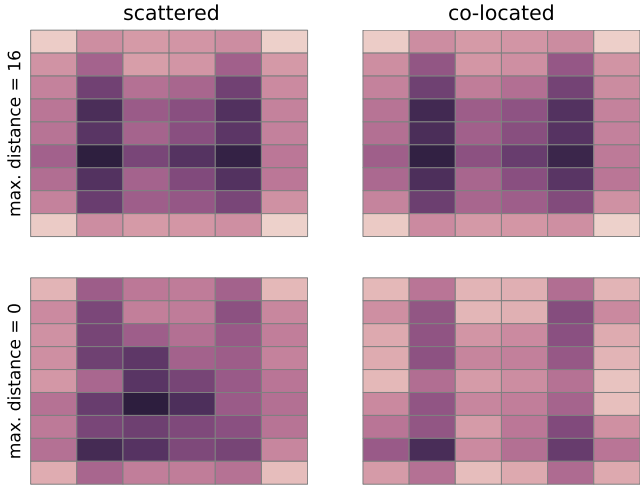



**FIGURE 13.** Heatmap of the number of packets across the mesh for two different core-to-CHA distances and thread mapping. When the maximum distance is 16 cycles, a block assigned to a core can have its coherence data residing in any tile across the mesh. On the other extreme, when the maximum distance is 0 cycles, the coherence data of the block must reside in its local tile. Darker shades indicate higher traffic density.

in the local router, but no forward packets. This explains the significant reduction in forward traffic as compared to query traffic.

Figure 14 details the breakdown of the network traffic into each packet type. The drastic reduction in the forward traffic is clear from the figure. It can be observed how data traffic across different quadrants is almost nonexistent when applying both optimizations. The eviction traffic in the central region of the mesh is similarly reduced. The reason is that each tile works now with data residing in the local MCDRAM interfaces of its quadrant, and therefore most eviction packets will not cross inter-quadrant boundaries. As such, traffic density is higher towards the center of each quadrant, instead of towards the central region of the mesh, as it happens with no optimizations. This is a result of the YX routing protocol: a drastic reduction in the number of collisions, i.e. situations in which a packet is stalled while in transit to its destination because the next router in its path has no available buffer space. Figure 15 shows the collision density on the network-on-chip. Again, the collisions are mostly confined to the center of each quadrant when applying the optimizations. The total number of collisions shows a reduction of 76.1%. The number of collisions per hop is reduced from 0.18 in the original code to just 0.05 in the optimized version.

## IX. RELATED WORK

Several recent papers have explored the performance of the Knights Landing architecture, mainly through the analysis of well-known benchmarks, machine learning applications, and parallel workloads [24]–[26]. This type of works analyze the scalability of the processor and provide the observed trends in terms of performance of real workloads, which are then compared against the theoretical performance. Furthermore, these works provide a reference when comparing this architecture with others. Nevertheless, none of these works undertake the analysis of the particular characteristics of the KNL interconnect.

Other papers offer a more in-depth look at the KNL internals. Rho et al. [27] analyze and compare the behavior of each of the offered cluster and memory modes by analyzing the behavior of MPI workloads. They propose an approach to optimize the scheduling at different granularities dynamically based on the characteristics of each workload. Jacquelin et al. [28] study the optimization of an ab initio molecular dynamics application in a Quadrant/Flat configuration, and develop its associated roofline model. Azad and Buluc [29] optimize an SpMSpV multiplication kernel on the KNL, among other architectures. Ramos and Hoefler [7] develop a capability model of the cache performance and memory bandwidth of the KNL. They perform an in-depth analysis of the memory system, and characterize the impact of the different memory and cluster modes on memory bandwidth and latency. Based on this model, they propose novel ways to optimize the communications in MPI applications. However, this work does not consider the impact of the distributed directory.

Other works have proposed ways to discover architectural features, or to automatically tune applications in modern, highly complex systems. Yotov et al. [30] develop a set of microbenchmarks specifically designed to measure memory hierarchy parameters, such as cache associativity, block size, capacity, or TLB parameters. Wang et al. [31] identify the increase in complexity associated with modern computational systems, in particular the trend to include a very large number of different architectural configurations. They argue that static discovery of optimal configuration parameters is a fundamentally flawed approach, and propose a configuration interface to allow users to specify performance constraints that should be satisfied at runtime. Mishra et al. [32] propose to use an automatic learning system to manage resources towards meeting specific latency and energy constraints. The resource allocation is performed in two different steps:

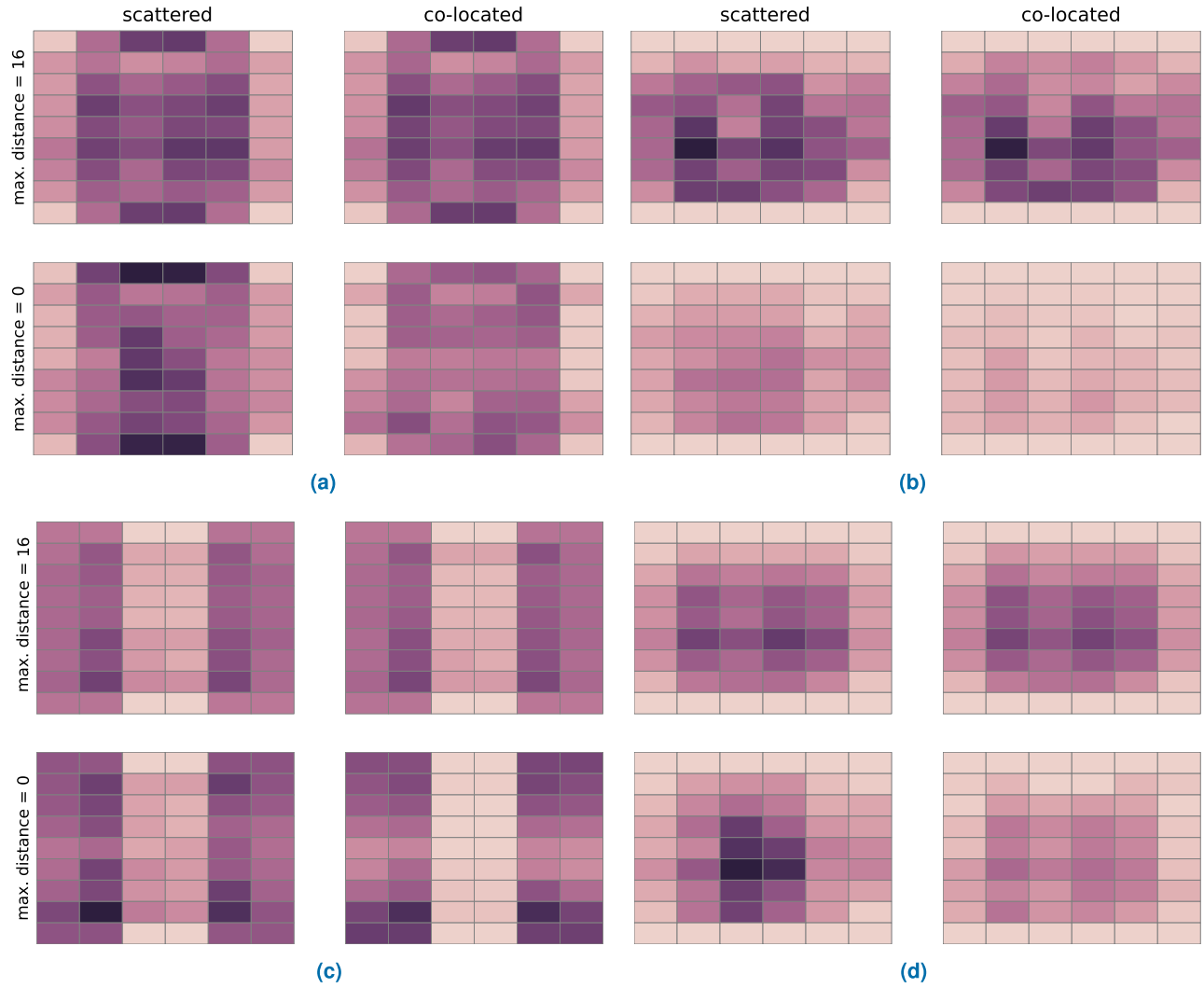



**FIGURE 14.** Breakdown of the different packet types across the network-on-chip. Darker shades indicate higher traffic density. (a) Query traffic. (b) Forward traffic. (c) Data traffic. (d) Eviction traffic.

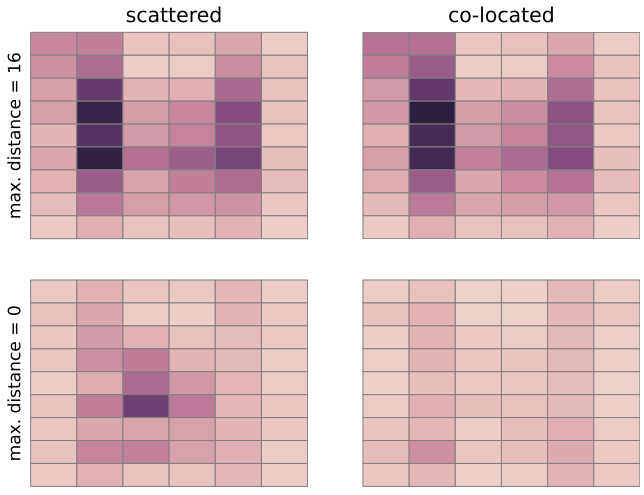



**FIGURE 15.** Density of collisions on the network. Darker shades indicate higher number of collisions.

learning how allocation affects parameters and controlling them during runtime.

Finally, there are many architectural simulators available nowadays. Some of them have very detailed pipelines, such as gem5 [33], some others are able to simulate up to thousands of cores, such as ZSim [34], PrimeSim [35], Graphite [36], McSimA+ [37] or Sniper [38]. In this work we have chosen to build upon the Tejas simulator due to the convenient compromise between the pipeline detail, the scalability of the simulation, and the low-level implementation of the distributed cache directory for cache coherence. Furthermore, it is an actively developed software project with a very responsive community.

## X. CONCLUSION

In this paper we have analyzed in detail the Intel Mesh Interconnect architecture through the study of one of its foremost examples, the Intel Knights Landing. We have performed reverse engineering of opaque architectural characteristics in order to understand the layout of the interconnection network. Armed with this knowledge, we have created an architectural model and implemented it on the Tejas simulator, which

enables an in-depth analysis of the behavior of this highly complex network-on-chip. We have provided a rigorous validation of the memory system using PolyBench/C and Parboil benchmarks. We have also presented a case study analyzing the low-level behavior of the interconnection network. Thus, we have posed optimizations taking advantage of the physical location of cores and cache blocks holders, i.e. core-to-CHA distance, and also the thread mapping for parallel applications.

We are currently working on reverse engineering the functions that map memory blocks to particular CHAs on the mesh. This will enable us to build ad-hoc compiler transformations that do not require the use of array indirections in order to achieve coherence traffic optimizations.

## REFERENCES

[1] M. A. Heroux, J. Carter, R. Thakur, J. Vetter, L. C. McInnes, J. Ahrens, and J. R. Neely, "ECP software technology capability assessment report," Oak Ridge National Lab., Oak Ridge, TN, USA, Tech. Rep. ECP-RPT-ST-0001-2018, 2018.

[2] J. Jeffers, J. Reinders, and A. Sodani, *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. San Mateo, CA, USA: Morgan Kaufman, 2016.

[3] S. M. Tam, H. Muljono, M. Huang, S. Iyer, K. Royneogi, N. Satti, R. Qureshi, W. Chen, T. Wang, H. Hsieh, S. Vora, and E. Wang, "SkyLake-SP: A 14 nm 28-core Xeon processor," in *IEEE ISSCC Dig. Tech. Papers*, San Francisco, CA, USA, 2018, pp. 34–36.

[4] M. Arafa, B. Fahim, S. Kottapalli, A. Kumar, L. P. Looi, S. Mandava, A. Rudoff, I. M. Steiner, B. Valentine, G. Vedaraman, and S. Vora, "Cascade lake: Next generation intel Xeon scalable processor," *IEEE Micro*, vol. 39, no. 2, pp. 29–36, Mar./Apr. 2019.

[5] S. R. Sarangi, R. Kalayappan, P. Kallurkar, S. Goel, and E. Peter, "Tejas: A java based versatile micro-architectural simulator," in *Proc. 25th Int. Workshop Power Timing Modeling, Optim. Simulation (PATMOS)*, Salvador, Brazil, Sep. 2015, pp. 47–54.

[6] M. Horro, G. Rodríguez, and J. Touriño. (2019). *Tejas KNL Simulator*. [Online]. Available: http://github.com/markoshorro/tejas_knl

[7] S. Ramos and T. Hoefler, "Capability models for manycore memory systems: A case-study with Xeon Phi KNL," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, Orlando, FL, USA, May/Jun. 2017, pp. 297–306.

[8] J. Liu, J. Kotra, W. Ding, and M. Kandemir, "Network footprint reduction through data access and computation placement in NoC-based manycores," in *Proc. 52nd Annu. Design Autom. Conf. (DAC)*, San Francisco, CA, USA, 2015, pp. 181:1–181:6.

[9] L. Yang, W. Liu, P. Chen, N. Guan, and M. Li, "Task mapping on SMART NoC: Contention matters, not the distance," in *Proc. 54th ACM/EDAC/IEEE Design Automat. Conf. (DAC)*, Austin, TX, USA, Jun. 2017, pp. 1–6.

[10] S. Charles, C. A. Patil, U. Y. Ogras, and P. Mishra, "Exploration of memory and cluster modes in directory-based many-core CMPs," in *Proc. 12th IEEE/ACM Int. Symp. Netw.-Chip (NOCS)*, Torino, Italy, Oct. 2018, pp. 1–8.

[11] M. Horro, M. T. Kandemir, L.-N. Pouchet, G. Rodríguez, and J. Touriño, "Effect of distributed directories in mesh interconnects," in *Proc. 56th Annu. Design Automat. Conf. (DAC)*, Las Vegas, NV, USA, Jun. 2019, pp. 51:1–51:6.

[12] L.-N. Pouchet. (2012). *PolyBench: The Polyhedral Benchmark Suite*. [Online]. Available: https://sourceforge.net/projects/polybench/

[13] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proc. 38th Annu. Int. Symp. Comput. Archit. (ISCA)*, San Jose, CA, USA, Jun. 2011, pp. 365–376.

[14] D. Kirk, "NVIDIA CUDA software and GPU parallel computing architecture," in *Proc. 6th Int. Symp. Memory Manage. (ISMM)*, Montreal, QC, Canada, 2007, pp. 103–104.

[15] J. R. Goodman and H. H. J. Hum, "MESIF: A two-hop cache coherency protocol for point-to-point interconnects," Univ. Auckland, Auckland, New Zealand, Tech. Rep., 2009.

[16] *Intel Xeon Phi Processor Performance Monitoring Reference Manual—Volume 1: Registers (Rev. 002)*, 2017.

[17] *Intel Xeon Phi Processor Performance Monitoring Reference Manual—Volume 2: Events*, 2017.

[18] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, New York, NY, USA, Dec. 2009, pp. 469–480.

[19] A. B. Kahng, B. Li, L.-S. Peh, and K. Samadi, "ORION 2.0: A fast and accurate NoC power and area model for early-stage design space exploration," in *Proc. Conf. Design, Automat. Test Eur. (DATE)*, Nice, France, 2009, pp. 423–428.

[20] S. R. Sarangi, R. Kalayappan, P. Kallurkar, and S. Goel, "Tejas simulator: Validation against hardware," 2015, *arXiv:1501.07420*. [Online]. Available: https://arxiv.org/abs/1501.07420

[21] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 190–200, 2005.

[22] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-M. W. Hwu, "Parboil: A Revised benchmark suite for scientific and commercial throughput computing," Univ. Illinois Urbana-Champaign, Urbana, IL, USA, Tech. Rep., 2012.

[23] M. Horro, G. Rodríguez, and J. Touriño. (2019). *Papi_Wrapper*. [Online]. Available: http://github.com/markoshorro/papi_wrapper

[24] C. Byun, J. Kepner, W. Arcand, D. Bestor, B. Bergeron, V. Gadepally, M. Houle, M. Hubbell, M. Jones, A. Klein, P. Michaleas, L. Milechin, J. Mullen, A. Prout, A. Rosa, S. Samsi, C. Yee, and A. Reuther, "Benchmarking data analysis and machine learning applications on the Intel KNL many-core processor," in *Proc. IEEE High Perform. Extreme Comput. Conf. (HPEC)*, Waltham, MA, USA, Sep. 2017, pp. 1–6.

[25] L. Chen, B. Peng, B. Zhang, T. Liu, Y. Zou, L. Jiang, R. Henschel, C. Stewart, Z. Zhang, E. McCallum, Z. Tom, J. Omer, and J. Qiu, "Benchmarking harp-DAAL: High performance Hadoop on KNL clusters," in *Proc. IEEE 10th Int. Conf. Cloud Comput. (CLOUD)*, Honolulu, HI, USA, Jun. 2017, pp. 82–89.

[26] N. A. Gawande, J. B. Landwehr, J. A. Daily, N. R. Tallent, A. Vishnu, and D. J. Kerbyson, "Scaling deep learning workloads: NVIDIA DGX-1/Pascal and intel knights landing," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPS)*, Orlando, FL, USA,/Jun. 2017, pp. 399–408.

[27] S. Rho, G. Park, J.-S. Kim, S. Kim, and D. Nam, "A study on optimal scheduling using high-bandwidth memory of Knights Landing processor," in *Proc. IEEE 2nd Int. Workshops Found. Appl. Self Syst. (FASW)*, Tucson, AZ, USA, Sep. 2017, pp. 289–294.

[28] M. Jacquelin, W. De Jong, and E. Bylaska, "Towards highly scalable Ab Initio molecular dynamics (AIMD) simulations on the intel knights landing manycore processor," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, Orlando, FL, USA, May/Jun. 2017, pp. 234–243.

[29] A. Azad and A. Buluç, "A work-efficient parallel sparse matrix-sparse vector multiplication algorithm," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, Orlando, FL, USA, May/Jun. 2017, pp. 688–697.

[30] K. Yotov, K. Pingali, and P. Stodghill, "Automatic measurement of memory hierarchy parameters," in *Proc. ACM SIGMETRICS Int. Conf. Meas. Modeling Comput. Syst. (SIGMETRICS)*, Banff, AB, Canada, 2005, pp. 181–192.

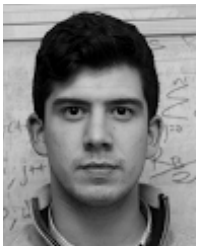
[31] S. Wang, C. Li, H. Hoffman, S. Lu, W. Sentosa, and A. I. Kistijantoro, "Understanding and auto-adjusting performance-sensitive configurations," in *Proc. 23rd Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, Williamsburg, VA, USA, 2018, pp. 154–168.

[32] N. Mishra, C. Imes, J. D. Lafferty, and H. Hoffmann, "CALOREE: Learning control for predictable latency and low energy," in *Proc. 23rd Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, Williamsburg, VA, USA, 2018, pp. 184–198.

[33] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011.

[34] D. Sanchez and C. Kozyrakis, "ZSim: Fast and accurate microarchitectural simulation of thousand-core systems," *ACM SIGARCH Comput. Archit. News*, vol. 41, no. 3, pp. 475–486, 2013.

[35] Y. Fu and D. Wentzlaff, "PriME: A parallel and distributed simulator for thousand-core chips," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Monterey, CA, USA, Mar. 2014, pp. 116–125.
[36] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A distributed parallel simulator for multicores," in *Proc. 16th Int. Symp. High-Perform. Comput. Archit. (HPCA)*, Bangalore, India, Jan. 2010, pp. 1–12.
[37] J. H. Ahn, S. Li, S. O, and N. P. Jouppi, "McSimA+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Austin, TX, USA, Apr. 2013, pp. 74–85.
[38] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*, Seatle, WA, USA, Nov. 2011, pp. 52:1–52:12.



**MARCOS HORRO** received the B.S. and M.S. degrees in computer science from the Universidade da Coruña (UDC), Spain, in 2016 and 2018, respectively. He is currently pursuing the Ph.D. degree with the Department of Computer Engineering, UDC. His main research interests include the areas of high-performance computing and computer architecture, focusing on the performance evaluation, and optimization of heterogeneous memory systems.

**GABRIEL RODRÍGUEZ** is an Associate Professor of computer engineering with the Universidade da Coruña, where he is a member of the Computer Architecture Group. His main research interests include optimizing compilers, architectural support for high-performance computing, and power-aware computing.

**JUAN TOURIÑO** (M'01–SM'06) received the B.S., M.S., and Ph.D. degrees in computer science from the Universidade da Coruña (UDC), Spain. In 1993, he joined the Department of Computer Engineering, UDC, where he is currently a Full Professor and the Head of the Computer Architecture Group. He has extensively published in the areas of high-performance computing and computer architecture. He is a coauthor of more than 160 papers in these areas. He has also served in the PC for 70 international conferences.