Effect of distributed directories in mesh interconnects

Marcos Horro
marcos.horro@udc.es
Computer Architecture Group, CITIC
Universidade da Coruña, Spain

Mahmut T. Kandemir
kandemir@cse.psu.edu
Dept. of Computer Science and Engineering
The Pennsylvania State University, USA

Louis-Noël Pouchet
pouchet@colostate.edu
Dept. of Computer Science
Colorado State University, USA

Gabriel Rodríguez
gabriel.rodriguez@udc.es
Computer Architecture Group, CITIC
Universidade da Coruña, Spain

Juan Touriño
juan.tourino@udc.es
Computer Architecture Group, CITIC
Universidade da Coruña, Spain

ABSTRACT

Recent manycore processors are kept coherent using scalable distributed directories. A paramount example is the Xeon Phi Knights Landing. It features 38 tiles packed in a single die, organized into a 2D mesh. Before accessing remote data, tiles need to query the distributed directory. The effect of this coherence traffic is poorly understood. We show that the apparent UMA behavior results from the degradation of the peak performance. We develop ways to optimize the coherence traffic, the core-to-core-affinity, and the scheduling of a set of tasks on the mesh, leveraging the unique characteristics of processor units stemming from process variations.

1 INTRODUCTION

Mesh interconnects featuring distributed directories are becoming essential in the design of scalable manycore architectures [6, 13, 17], and reducing their coherence footprint is critical for performance [3, 10, 15]. Each time a core issues an access to a memory block not present in its local caches in a valid state, it needs to query the distributed directory in order to discover its current status and location.

A paramount example of this trend is the Intel Mesh Interconnect Architecture, first present in the Intel Xeon Phi Knights Landing (KNL) manycore [13], and more recently featured in the Xeon Scalable [14]. In this architecture each tile in the mesh contains not only processor cores, but also a network of Caching/Home Agents (CHAs) in charge of managing a distributed cache coherence directory. While this reduces the contention of the network by eliminating the need for snoops and the bottleneck of a centralized directory, it causes an increase in the network latency due to the distance between directories and memory controllers [5]. This paper analyzes the effect of the coherence traffic in distributed directory interconnects. Although we focus on one specific state-of-the-art manycore, the Intel KNL, the techniques we propose can be extended to other distributed directory architectures.

In order to optimize coherence traffic the compiler needs to have a deep low-level knowledge of the underlying network-on-chip architecture. Unfortunately, some important aspects of the KNL design are not disclosed. In particular, for this architecture two specific hidden pieces of information need to be discovered, namely, the physical location of logical components and the scattering of memory blocks across the distributed directory and memory interfaces. We expose this information through microbenchmarking [16], and leveraging it we propose mechanisms to: i) reduce the round-trip times of coherence messages across the mesh, which improves the access latency to both memory and other caches; and ii) optimize the scheduling of sets of tasks across the processor mesh. More specifically, we make the following three contributions:

- We propose a mechanism to discover the physical layout of the logical components (cores and CHAs) of any KNL unit, as well as the mapping of memory blocks across CHAs and memory interfaces.
- Leveraging the previous contribution, we analyze the impact of coherence traffic in the memory latency of distributed directory architectures. Mechanisms to optimize coherence traffic are proposed, improving CHA-to-core and thread-to-core affinity.
- We gather and analyze data generated by a large number of executions on a KNL unit, and develop optimized strategies for scheduling a set of tasks across the tiles in the mesh. We perform experiments to quantify the efectiveness of our optimizations. Our results reveal that exploiting the multiple opportunities for locality in a mesh interconnect is essential to increase the potential performance of future manycores.

The paper is structured as follows. Section 2 covers the Knights Landing architecture and introduces the proposed approach. Section 3 details how to map the components of the processor to its physical floorplan. Section 4 covers how to optimize coherence traffic. Section 5 evaluates the potential advantages of the proposed approach, and develops ways to exploit the architectural characteristics of a particular KNL unit. The related work is discussed in Sec. 6, and Sec. 7 concludes the paper.

2 BACKGROUND AND OVERVIEW

The Knights Landing [9] is the latest architecture released by Intel for high performance computing. It is a manycore processor, including from 64 to 72 cores inside a single die. The processor layout consists of a 2D mesh topology containing 38 tiles, detailed in Fig. 1. Internally, each tile contains two cores, each with its private L1 instruction and data caches (32 kB each); and a unified L2 cache (1 MB) shared among the local cores, but private to the tile.

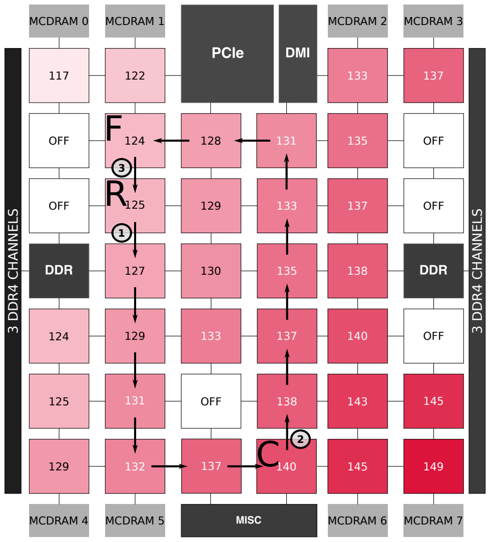


Figure 1: Floorplan of the Intel KNL architecture. Superimposed, heatmap of the measured access latency (in CPU cycles) from each tile in the mesh of an Intel Xeon Phi x200 7210 to a single block of memory associated to MCDRAM #0 and its adjacent CHA.

The KNL processor has two different types of DRAM memory. An MCDRAM (Multi-Channel DRAM) system provides high-bandwidth accesses to 3D-stacked memory through the eight interfaces in the corners of the mesh. Besides, two DDR controllers on opposite parts of the chip control three memory channels each. The MCDRAM memory has higher latency than DDR (it is approximately 10% slower), but the eight interfaces can be accessed simultaneously, providing a much higher bandwidth.

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 KNL employs a directory-based cache coherence mechanism using Intel MESIF [7], a variant of MESI. In order to alleviate the bottleneck that a centralized directory would impose, it features a distributed system in which each tile includes a portion of the directory in a module known as the Caching/Home Agent (CHA). 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 Fig. 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). This paper explores ways to exploit nearby CHAs, avoiding the overhead of accessing distant tiles of the mesh.

The KNL architecture can be configured into one of three main cluster modes, which determine the affinity between memory interfaces, CHAs, and cores: *All-to-All*, only recommended when different amounts of memory are connected to each MCDRAM interface; *Quadrant*, the de-facto standard, in which the mesh is logically divided into four different clusters; and *SNC*, targeted towards NUMA-aware MPI applications only [9]. Similarly, the KNL architecture lets the user configure the MCDRAM into one of two modes: "Flat" memory, in which the address space is explicitly exposed as an independent NUMA domain; and "Cache" mode, in which it serves as a memory-side cache. In this paper we focus exclusively on the MCDRAM subsystem, although all the proposed mechanisms and optimizations are directly extensible to the DRAM subsystem, and on the Quadrant/Flat mode.

In the related literature, the access time of a core to any memory block is assumed to be UMA when in Quadrant mode [9, 12]. 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 [12]. The challenging aspect of these measurements is that the physical locations of logical entities on the 2D mesh are not exposed to the programmer, and are variable across KNL units due to process variations. After reverse engineering these locations using the techniques detailed in Sec. 3, however, we observe 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. Fig. 1 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 (12 vertical plus 10 horizontal hops). 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 [3, 10, 15].

In order to characterize the latency and traffic across the network, we need to obtain information about: i) how the logical components of the mesh (CHAs, cores) are physically mapped; and ii) how the address space is distributed across CHAs and MCDRAM interfaces. We propose the following approach:

[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.

(1) Identify where CHAs and cores are physically located in the processor mesh by leveraging the information provided by the CPUID instruction, profiling memory access latencies, and building a minimum squared error model (Sec. 3).
(2) Dynamically modify the layout of the runtime data so that each core operates on data whose directory information lies in CHAs close to that core (Sec. 4).

3 MAPPING THE KNL ARCHITECTURE

When working in the SNC cluster mode the correspondence between logical and physical cores is explicit. 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. 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 discover core location and bind a team of threads to neighboring cores, the coherence data of the accessed memory blocks will be scattered across the full mesh. For this reason, it is not sufficient to know where each core is located in the physical mesh; we need to know where each CHA is located and how memory blocks are assigned to CHAs in order to carefully optimize the network traffic for each thread.

We reverse engineered the physical layout of an Intel x200 7210 processor 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.[2]

Once these data are collected, we analyze them to determine where each pair 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. 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 associations between cores and CHAs, and to provide 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 cores and CHAs to the physical mesh, and finally accept the one which presents the least squared error.

The obtained results present a clear pattern in the location of both CHAs and cores, as shown in Fig. 2. The CHAs in each quadrant are sequentially arranged in a vertical fashion. Cores are assigned sequentially to CHAs, skipping tiles with disabled cores. We believe

[2]We use a custom kernel module which leverages the uncore Model Specific Registers (MSRs) to measure the number of accesses to each CHA component and MCDRAM interface, inferring where coherence data is stored.

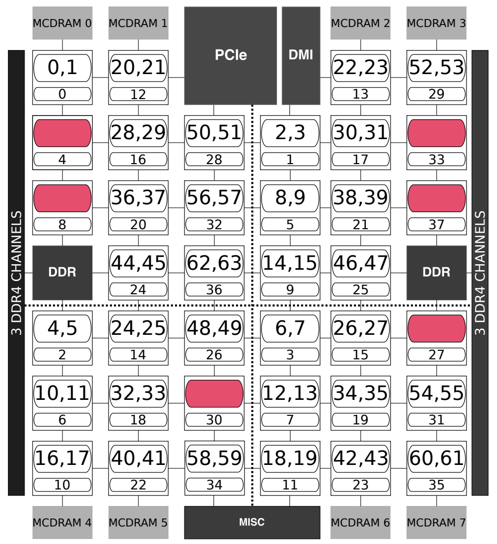


Figure 2: Result of our model. Each tile contains two cores and one CHA (their IDs are enclosed in a large and small box, respectively). Tiles with blank boxes indicate that their cores are not active.

that disabled cores vary for each particular KNL unit, depending on process variations, but that the pattern for arranging the CHAs and assigning the cores to CHAs is fixed. If this assumption is correct, it allows one to obtain the physical layout of any individual KNL unit immediately, by just checking which CHAs have disabled cores through CPUID instructions.

4 PROCESSOR AFFINITY AND DATA LAYOUT

Once the mapping of the logical components of the processor onto the physical floorplan is exposed, the next step is to take advantage of this information. There are at least two orthogonal ways in which an application might exploit locality across the mesh:

- Each thread should access data with coherence information stored in a nearby CHA as much as possible. In this way, memory access latency will be improved due to the shortest message trips across the network. Furthermore, restricting coherence data to subsets of the tile will improve the network contention when a large number of cores is active. We refer to this optimization as exploting CHA-to-core affinity.
- A core requesting data in a nearby L2 cache may not take advantage of this proximity due to the coherence data being assigned to a distant CHA, as illustrated in Fig. 1. However, once CHA-to-core affinity is improved, applications will benefit from co-locating cooperating threads. We refer to this optimization as thread-to-core affinity.

Mapping memory blocks and their associated CHAs and using them accordingly requires important changes to the compilation chain and/or the source code of an application. For any array in a computational kernel, we need to obtain a mapping of the correspondence between the memory blocks in the array and the CHAs in the mesh. Once that is done, work has to be scheduled across the available threads according to the affinity between the core executing each thread and the CHAs. There are two ways in which this can be accomplished: i) dynamically, by running an inspector/executor which finds the mapping between memory blocks and CHAs, and schedules tasks accordingly; and ii) statically, by exposing information about the memory system to the compiler.

The static option, which would get rid of any runtime overhead, would require one to discover the hash function that determines CHA location for each physical memory block. We are exploring this approach, which falls out of the scope of this paper, for future work. For the current paper, we are interested only in showing that CHA proximity plays an important role in the performance of multithreaded codes, and showcasing the potential of optimizing mesh locality. For this purpose, we map the CHA locations of all the memory blocks in the 16 GB MCDRAM memory subsystem. For each block, we note the associated CHA and MCDRAM interface, and store both in one byte. This information takes up 256 MB for the entire memory, and is incorporated into the runtime of each application. Upon execution, an inspector-executor copies the data to be accessed by each core to memory locations indexed by CHAs with high affinity to said core. This requires using arrays of indirections, and therefore this technique will likely bring performance advantages to irregular codes only. Nevertheless, the aim of this work is to: i) provide evidence of the impact of the distributed directory; ii) highlight the importance of disclosing architectural features for code optimization; and iii) serve as a basis to develop a compiler-based optimization model. All our experimental codes will use arrays of indirections, even when accessing memory sequentially, in order to fairly assess memory performance.

5 EXPERIMENTAL RESULTS

We applied the proposed approach to several commonplace computational kernels to analyze how the location of coherence data affects system behavior. The experiments were run on the Intel Xeon Phi x200 7210 mapped in Sec. 3, with 64 total cores, 192 MB of DDR, and 16 GB of MCDRAM. The codes are compiled using ICC 18.0.3 with `-O2 -xKNL`. The processor is configured in "Flat" memory mode, meaning that the address space is divided into two different regions, one for DDR and one for MCDRAM. The Quadrant cluster mode is employed. Turbo mode is disabled, i.e., the frequency is fixed to the base of 1.30 GHz. Our applications were configured to use MCDRAM exclusively for data allocation through `numactl`. All our data arrays are allocated into 1 GB hugepages. We ran three different sets of experiments. First, we analyzed the impact of the core-to-CHA affinity optimization using different affinity strengths (Sec. 5.1). Next, we analyzed the effect of the thread-to-core affinity optimization in coherence traffic, using a modified 1D stencil (Sec. 5.2). Finally, we broaden our scope to analyze the impact of optimized thread-to-core scheduling of several different workloads (Sec. 5.3).

5.1 Effect of core-to-CHA affinity on memory latency

We first measure the potential of optimizing core-to-CHA affinity to reduce memory latency. For this purpose, we employ a vector-vector reduction kernel, due to its high memory bandwidth requirements.

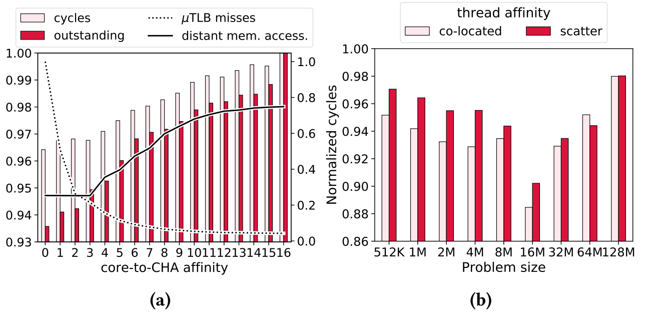


Figure 3: Effects of core-to-CHA and core-to-core affinities: 3a) execution cycles, outstanding weighted cycles, μTLB misses, and accesses to distant memory interfaces for different core-to-CHA affinities. Results are normalized to the maximum value for each series, except for accesses to distant memory interfaces, which are normalized to the total number of memory accesses. μTLB misses and distant accesses are referenced to the right axis; and 3b) execution cycles of the best-performing CHA-to-core affinity for two different core-to-core affinities: "scatter" (thread i is assigned to OS core i), and "co-located" (adjacent threads are placed in adjacent physical cores). Results are normalized to the execution cycles of the non-optimized code with scatter thread placement. The left Y axes are truncated to better reflect the differences in values.

We work with a total dataset of 128 MB to ensure that no reuse takes place through caches, and repeat the computation 100 times to average out performance differences across the full experiment.

Figure 3a illustrates how performance metrics evolve for all possible affinities between cores and CHAs. The core-to-CHA affinity (X axis) indicates the maximum distance in CPU cycles allowed from a core to the CHAs indexing the data it accesses. The figure shows a clear performance improvement from limiting the spread of coherence traffic. The reduction in outstanding weighted cycles is of 7.2%, close to the expected theoretical optimal. However, the derived speedup is only 3.1%. The culprit is the increase in the number of μTLB misses, due to the pseudo-random nature of the access to the data arrays enforced by the search for blocks associated to local CHAs, and to the lack of hugepages support on the L1 TLB. Increasing the neighborhood size reduces μTLB misses, as more memory blocks are usable by each core. This reduces the spread of the accesses, but also increases access latency, as more distant MCDRAM interfaces are accessed causing an increase in the travel times of data and coherence packets. The proportion of distant accesses eventually converges to approximately one fourth, as 2 out of the 8 MCDRAM interfaces are considered close to each core.

5.2 Effect of core-to-core affinity on coherence traffic

A second optimization enabled by our architectural analysis is the exploitation of core-to-core affinity. We aim to improve the locality of the data across the L2 caches in the mesh to reduce coherence traffic. For this purpose, we modify a jacobi-1d stencil so that neighboring cores swap their data at the end of each timestep. Note that it is futile to try to exploit core-to-core affinity without enforcing the CHA-to-core affinity first, as a thread sharing a block of data will need to traverse the mesh to query the appropriate CHA before finding out that the block lay in a neighboring core.

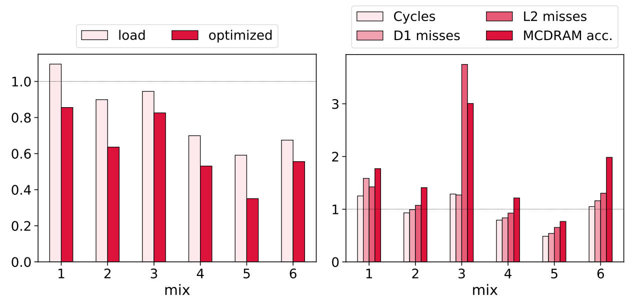
The 1D stencil was run on 64 cores using 108 different configurations, including several problem sizes, five different CHA-to-core affinities, and two core-to-core affinities: scatter and an ad-hoc "co-located" affinity in which adjacent threads are assigned to adjacent cores whenever possible. In total, more than 3,000 executions of the stencil were run. Fig. 3b shows the normalized median execution cycles after discarding outliers. As can be observed, the two optimizations target different types of workloads. For applications with small datasets, in which reuse comes from other cores in the mesh, adjusting core-to-core affinity yields important benefits. The figure clearly shows how this optimization loses effectivity when the memory footprint reaches the total combined cache size (32 MB in our Intel x200 7210 processor). On the other hand, applications with large datasets, which consume a large volume of data directly from memory, benefit more from the memory latency reduction provided by CHA-to-core affinity. This optimization also loses effectivity as footprint increases. In this case, the reason is the exponentially increasing number of μTLB misses, as covered in Sec. 5.1.

5.3 Optimized thread-to-core scheduling

In the previous section, we studied how changing the core-to-core affinity impacted performance for a particular workload. The data was always shared among consecutive threads, and therefore a simple affinity could be devised ensuring that threads sharing data were never more than two hops apart on the mesh. However, designing balanced affinities for more complex data sharing relationships in 64-threaded applications is a non-trivial problem, particularly given the irregular structure of the mesh. Instead, we focus on how to optimally schedule smaller workloads on the available cores.

In order to discover whether there are significant performance differences to be exploited from running a workload on particular cores, we run different applications with footprints ranging from 512 KB to 2 MB per thread (4 times the alloted cache space per core), and using 4, 8, 16, and 32 threads. We test a total of 103 thread-to-core schedules, including 51 4-core groups, 45 8-core groups, 14 16-core groups, and 3 32-core groups. For instance, in the case of 4 threads, we test the full set of 46 different contiguous 2-tile allocations, the default "scatter" affinity in ICC, and several random ones to act as control groups.

The benchmarks employed are the stencils and array kernels detailed in Table 1. In total, more than 45,000 executions were performed. The results were analyzed using k-means clustering to discover the factors that impact performance. We gather information about architectural trends, particularized for our processor unit: which processors are faster or slower, depending on process variations; how benchmarks with different bandwidth requirements are better located with respect to the memory interfaces; which types of benchmarks benefit from CHA-to-core and core-to-core affinity optimizations; etc.



(a) (b)

Figure 4: Results of optimized scheduling methodologies: 4a) Execution cycles of "load" and "optimized" schedules normalized to "fair" values; and 4b) Performance counters of "optimized" schedules aggregated over the sum of each mix and normalized to the values in "fair" schedules.

We validate the collected "historic" data by generating random mixes of applications and executing them using a schedule which exploits the architectural characteristics discovered during the analysis phase. We randomly generate 6 different workload mixes, each including 8 benchmarks of varying sizes, as detailed in Table 2. We then execute each mix using three different configurations: i) "fair": cores are assigned to each task proportionally to the size of their dataset; no CHA-to-core-affinity is enforced; and the binding of threads to cores is managed by the OS; ii) "load": cores are distributed across the tasks by using the historic data to estimate expected execution times; no CHA-to-core affinity is enforced; and the binding of the threads to cores is managed by the OS; and iii) "optimized": the number of cores per task is the same as in ii), but strong CHA-to-core affinity is enforced; and an optimal thread-to-core binding is computed by consulting historic data.

Figure 4a shows the performance of our optimized scheduling methodologies. The improvement obtained by the "load" scheduling depends on how well the computational load of each mix is predicted by the dataset sizes of its applications. For example, in mix #1 the initial "load" scheduling does not improve the execution time because, for this mix, our resource allocation binds threads of very small benchmarks to different hyperthreads of the same core, to better exploit the available slack in the mix. The "optimized" schedule takes into account the characteristics of the mesh to achieve further improvements. This effect is most noticeable in mix #5, in which the longest computation corresponds to an instance of `jac-1d-swap`, a benchmark which is particularly sensible to the co-location of its computing threads. Aggregating all mixes, "load" scheduling improves total execution times by 20.8%, and "optimized" increases that gain to 61.3%.

Figure 4b gives a more detailed view of different performance metrics for the "optimized" schedules. The plot aggregates the sum of all metrics for all tasks in each mix. Note that sometimes the total number of execution cycles increases with respect to the original execution cycles. Yet, as shown in Fig. 4a, the total execution time always improves. The reason is that the "optimized" schedule exploits the slacks of non-critical path tasks to better balance resource

Table 1: Benchmarks used in the experiments, characterized by the weighted averages of cache accesses and misses, memory accesses, and floating-point operations. Values are reported in millions per thread per second.

Benchmark	Description	D1 acc.	D1 misses	L2 misses	MCDRAM acc.	FLOPs
rvec	Vector reduction	14.88	10.24	8.42	8.34	190.25
rvv	Vector-vector addition and reduction	19.68	12.59	11.08	10.95	257.05
vecsearch	Search for value in vector	11.42	8.96	8.39	8.32	284.88
jac-2d	2D Jacobi stencil	249.98	12.26	9.58	9.38	334.62
avv	Vector-vector addition	124.45	9.66	11.40	11.29	635.02
jac-1d	1D Jacobi stencil	52.98	11.02	15.22	15.10	702.72
jac-1d-swap	1D Jacobi with data swap after each timestep	59.79	11.50	14.77	7.37	717.60

Table 2: Applications in each mix of workloads.

mix	apps
#1	jac-2d-512kB, rvv-512kB, jac-1d-8MB, jac-2d-8MB, rvec-8MB, rvv-32MB, vecsearch-32MB, avv-128MB
#2	avv-512kB, rvv-1MB, vecsearch-1MB, avv-2MB, avv-2MB, jac-1d-swap-4MB, jac-1d-swap-32MB, jac-2d-128MB
#3	vecsearch-512kB, jac-2d-1MB, vecsearch-1MB, jac-2d-2MB, avv-2MB, vecsearch-4MB, rvec-8MB, jac-2d-16MB
#4	rvec-512kB, rvv-512kB, vecsearch-512kB, avv-32MB, jac-1d-4MB, vecsearch-4MB, avv-32MB, jac-2d-32MB
#5	jac-1d-swap-1MB, jac-2D-1MB, rvec-2MB, vecsearch-4MB, avv-16MB, jac-1d-swap-16MB, jac-2d-16MB, jac-2d-32MB
#6	jac-1d-swap-512kB, avv-1MB, avv-2MB, rvv-2MB, rvv-8MB, avv-32MB, rvv-32MB, jac-2d-64MB

allocation. Another interesting effect is the total increase in the number of MCDRAM accesses for almost all mixes. This is caused by a benchmark with large dataset but short comparative execution time being allocated a reduced set of resources. This causes the benchmark to become memory-bound, and its execution time to increase, but keeping it out of the critical path of the mix. To avoid interference, these tasks are allocated a set of MCDRAM interfaces which are not used by other high-bandwidth demanding tasks. On the other side of the spectrum is mix #5, where the number of MCDRAM accesses is greatly reduced by the co-location of the executing threads and the increase in allocated cache resources.

6 RELATED WORK

In recent years, a number of papers have explored the design of scalable networks-on-chip to support manycore architectures. Daya et al. [5] design a NoC based on an ordered network and a snoopy coherence protocol, and show how congestion increases heavily with the number of cores. Ferdman et al. [6] propose a scalable distributed directory system to alleviate the power and performance problems of sparse and duplicate-tag directories, scaling up to 1,024 cores. Charles et al. [3] identify the importance of the coherence traffic in manycore performance, and show how the memory modes in the Intel KNL can be manipulated to achieve better performance. They neither explore software optimizations to coherence traffic, nor the actual layout of the KNL processor.

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 [1, 2, 4, 8]. These works analyze behavioral trends of real workloads, but none of them undertake the analysis of the locality characteristics of the KNL interconnect. Ramos and Hoefler [12] develop a capability model of the cache performance and memory bandwidth of the KNL, characterizing the impact of the different memory and cluster modes. However, this work does not consider the impact of the distributed directory.

Few works focus on data layout optimizations for 2D interconnects. Lu et al. [11] propose a polyhedral model and associated optimizations to achieve data locality in these topologies. Liu et al. [10] use a compiler-guided scheme to minimize on-chip network traffic by reducing the distances of cores to data, but without taking into account the effects of a distributed directory.

7 CONCLUSION

This work has explored the performance implications of the coherence traffic in distributed directory architectures, in particular when coupled with an opaque distribution of memory blocks over directory fragments, such as in the Intel Knights Landing architecture. We have presented ways to improve network performance by optimizing the coherence traffic, which for the KNL required exposing the physical topology of the processor. We have revealed novel intuitions which negate the pretended UMA behavior of this kind of architectures, and shown how the illusion of uniformity comes from a degradation of access latencies to a sub-optimal average. Furthermore, we have shown how traditional data placement optimizations are not exploitable unless the affinity between cores and the distributed directory fragments is first controlled for. We validated all these proposals by developing and testing optimized scheduling algorithms for a unique KNL processor.

Complex mesh interconnects as exemplified by the KNL architecture are a viable approach to implementing multi- and manycores in the forthcoming future, as demonstrated by their inclusion in the Intel Xeon Scalable series. While manufacturers may obfuscate information, which had to be empirically revealed in the current work, its dissemination is essential for compiler developers to deliver higher code optimization quality, as suggested in this paper.

ACKNOWLEDGMENTS

This work was supported in part by the Ministry of Economy, Industry and Competitiveness of Spain under grant TIN2016-75845-P (AEI/FEDER/EU), and by the FPU Program of the Ministry of Education under grant FPU16/00816.

The authors would like to thank John McCalpin for his invaluable insights into the KNL architecture.

REFERENCES

[1] A. Azad and A. Buluç. 2017. A Work-Efficient Parallel Sparse Matrix-Sparse Vector Multiplication Algorithm. In *IPDPS*. 688–697.
[2] C. Byun et al. 2017. Benchmarking Data Analysis and Machine Learning Applications on the Intel KNL Many-Core Processor. In *HPEC*. 1–6.
[3] S. Charles et al. 2018. Exploration of Memory and Cluster Modes in Directory-Based Many-Core CMPs. In *NOCS*. 1–8.
[4] L. Chen et al. 2017. Benchmarking Harp-DAAL: High Performance Hadoop on KNL Clusters. In *CLOUD*. 82–89.
[5] B.K. Daya et al. 2014. SCORPIO: A 36-Core Research Chip Demonstrating Snoopy Coherence on a Scalable Mesh NoC with In-Network Ordering. In *ISCA*. 25–36.
[6] M. Ferdman et al. 2011. Cuckoo Directory: A Scalable Directory for Many-Core Systems. In *HPCA*. 169–180.
[7] J.R. Goodman and H.H.J. Hum. 2009. *MESIF: A Two-Hop Cache Coherency Protocol for Point-to-Point Interconnects.* Technical Report. University of Auckland.
[8] M. Jacquelin, W. De Jong, and E. Bylaska. 2017. Towards Highly Scalable Ab Initio Molecular Dynamics (AIMD) Simulations on the Intel Knights Landing Manycore Processor. In *IPDPS*. 234–243.
[9] J. Jeffers, J. Reinders, and A. Sodani. 2016. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition.* Morgan-Kauffman.
[10] J. Liu et al. 2015. Network Footprint Reduction through Data Access and Computation Placement in NoC-Based Manycores. In *DAC*. 181.
[11] Q. Lu et al. 2009. Data Layout Transformations for Enhancing Data Locality on NUCA Chip Multiprocessors. In *PACT*. 348–357.
[12] S. Ramos and T. Hoefler. 2017. Capability Models for Manycore Memory Systems: A Case-Study with Xeon Phi KNL. In *IPDPS*. 297–306.
[13] A. Sodani. 2015. Knights Landing (KNL): 2nd Generation Intel Xeon Phi processor. In *HCS*. 1–24.
[14] S.M. Tam et al. 2018. SkyLake-SP: A 14nm 28-Core Xeon Processor. In *ISSCC*. 34–36.
[15] L. Yang et al. 2017. Task Mapping on SMART NoC: Contention Matters, Not the Distance. In *DAC*. 88.
[16] K. Yotov, K. Pingali, and P. Stodghill. 2005. Automatic Measurement of Memory Hierarchy Parameters. In *SIGMETRICS*. 181–192.
[17] C. Zhang. 2015. Mars: A 64-core ARMv8 processor. In *HCS*. 1–23.