IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 17, 2021, accepted January 25, 2021, date of publication February 9, 2021, date of current version February 24, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3058280

Optimizing Coherence Traffic in Manycore Processors Using Closed-Form Caching/Home Agent Mappings

STEVE KOMMRUSCH[1], MARCOS HORRO[2], LOUIS-NOËL POUCHET[1], GABRIEL RODRÍGUEZ[2], AND JUAN TOURIÑO[2], (Senior Member, IEEE)

[1]Department of Computer Science, Colorado State University, Fort Collins, CO 80523, USA
[2]CITIC, Computer Architecture Group, Universidade da Coruña, Campus de Elviña, 15071 A Coruña, Spain

Corresponding author: Gabriel Rodríguez (gabriel.rodriguez@udc.es)

This research was supported in part by the Ministry of Science and Innovation of Spain under Grant PID2019-104184RB-I00 / AEI / 10.13039/501100011033, in part by the Ministry of Education of Spain under Grant FPU16/00816, and in part by the U.S. National Science Foundation under Award CCF-1750399. CITIC is funded by Xunta de Galicia and FEDER funds of the EU (Centro de Investigación de Galiciaaccreditation, grant ED431G 2019/01).

ABSTRACT Manycore processors feature a high number of general-purpose cores designed to work in a multithreaded fashion. Recent manycore processors are kept coherent using scalable distributed directories. A paramount example is the Intel Mesh interconnect, which consists of a network-on-chip interconnecting ''tiles'', each of which contains computation cores, local caches, and coherence masters. The distributed coherence subsystem must be queried for every out-of-tile access, imposing an overhead on memory latency. This paper studies the physical layout of an Intel Knights Landing processor, with a particular focus on the coherence subsystem, and uncovers the pseudo-random mapping function of physical memory blocks across the pieces of the distributed directory. Leveraging this knowledge, candidate optimizations to improve memory latency through the minimization of coherence traffic are studied. Although these optimizations do improve memory throughput, ultimately this does not translate into performance gains due to inherent overheads stemming from the computational complexity of the mapping functions.

INDEX TERMS Network-on-chip, manycores, coherence traffic, distributed directories, architectural discovery, reverse engineering.

I. INTRODUCTION

Manycore processors feature a high number of general-purpose cores designed to work in a multithreaded fashion. In order to make systems scalable, current designs are usually based on replicated IP blocks connected by a high-performance fabric. An example of such an approach is the Intel Mesh interconnect (IM), first featured in the Intel Xeon Phi Knights Landing (KNL) processor [34]. The IM is the current interconnection standard in the most advanced Intel processors, including Intel Xeon Scalable servers and the High-End Desktop family of Core-X chips [1], [35].

Each IP block in an IM-based processor, called ''tile'', includes computation cores and local caches. In order to maintain memory coherence the system employs the Intel MESIF protocol, supported by a distributed directory. Each tile includes part of this distributed directory in a component called the Caching/Home Agent (CHA). The directory must be accessed each time a core requests access to a memory block which is not already locally available in the appropriate state. This distributed design increases the scalability of the coherence system by removing the bottleneck that a centralized directory would impose, but causes a non-uniform increase in the network latency due to the varying distances between a tile and the set of CHAs on the mesh. Figure 1 details this latency variation across the full mesh of an Intel Xeon Phi x200 7210 (Knights Landing), as measured by Horro *et al*. [12]. As can be observed, latency overheads are higher than 25% for extreme cases.

One key aspect of the design of the CHA-based directory in IM processors is that both the physical layout of the logical components of the processor and the mapping of memory blocks to CHAs are opaque and non-disclosed by Intel. This prevents memory latency optimizations, since the

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

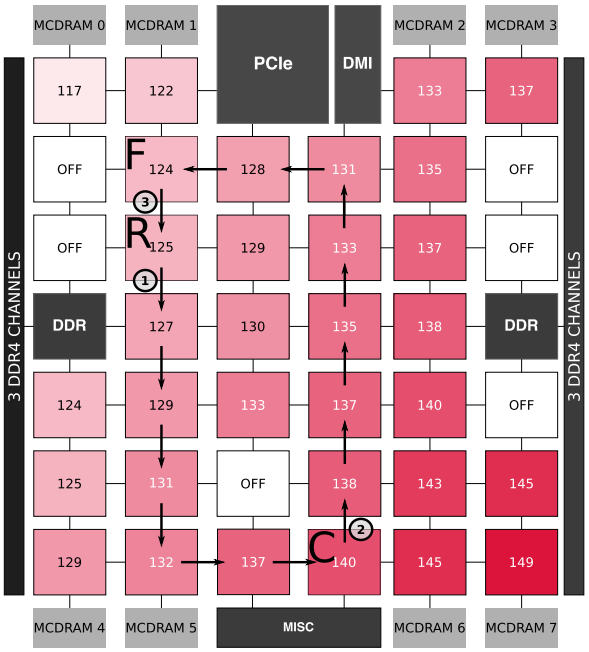


FIGURE 1. Actual access latencies from each tile in the mesh of an Intel x200 7210 processor to MCDRAM #0, for a memory block whose coherence data is contained in the tile next to the memory interface. We note differences in access latency of up to 32 CPU cycles (a 27% overhead over the minimum observed latency of 117 cycles). Section III details how these latencies are measured.

programmer has no a-priori knowledge of the latency that can be expected for each access. Furthermore, Intel advertises this architecture as UMA, since the *average* memory access latency is approximately the same for all tiles in the mesh. This article builds on a previous work [12], which reverse-engineered the physical layout of the logical components of the processor and showed how this knowledge, coupled with an inspector-executor which dynamically analyzes which CHAs are associated to each memory block access, can be used to optimize irregular codes. Leveraging this, the present work focuses on building a closed form function of the mapping of memory blocks to CHAs in order to remove the costly inspection phase, enabling new optimization strategies for IM processors. More specifically, this paper makes the following contributions:

- The mapping of memory blocks to CHAs is reverse-engineered. Binary functions which compute a target CHA from a physical memory address are exposed and shown to be pseudo-random in nature (Section IV).
- Different optimization strategies to improve memory latency by leveraging the mappings between memory and CHAs are designed. Approaches are proposed based on both dynamic and static work scheduling (Sections V and VI).
- Experiments are performed to quantify the effectiveness of the proposed optimizations. It is shown how the proposed schedulings improve the memory latency by exploiting CHA proximity. However, due to the pseudo-random nature of the block-mapping functions the implementation of these schedulings affects other performance-impacting factors, which may ultimately lead to performance degradation (Sections V-A and VI-B).

The paper is structured as follows. Section II covers the IM architecture, with a particular focus on Knights Landing processors, and provides an overview of the current work. Section III explains the approach to discover the physical layout of the logical components of the KNL processor. Section IV details the reverse engineering process that leads to the discovery of the memory-to-CHA mapping. Sections V and VI detail runtime- and compile-time-based approaches, respectively, to optimize coherence traffic and summarize the results of the experimental evaluation phase. Section VII discusses the obtained results and related work. Finally, Section VIII concludes the paper.

II. BACKGROUND AND OVERVIEW

This paper studies the Intel Knights Landing (KNL) architecture as a paramount example of the Intel Mesh interconnect. KNL [18] 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. 2. Internally, each tile contains two cores, each with its private L1 instruction and data caches (32 KiB each); and a unified L2 cache (1 MiB) shared among the local cores, but private to the tile.

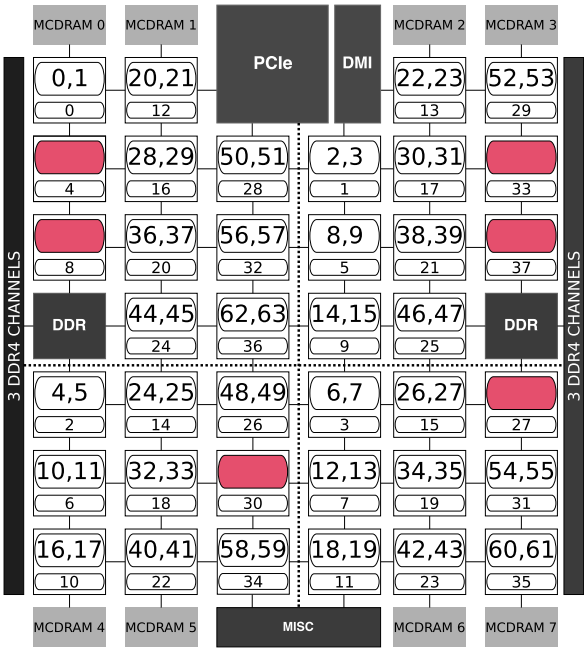


FIGURE 2. Physical location of logical entities on the KNL floorplan. Each tile contains two cores with the logical IDs shown in the enclosing rectangle. The smaller number below shows the logical ID of each CHA module. Tiles with blank boxes have their cores disabled. A simple naming algorithm can be inferred: CHAs are labeled in sequence, and the set of IDs in each quadrant yield the same value modulo 4. Logical processor IDs are also assigned sequentially, but processors with disabled cores are skipped. More details on the analysis of the logical layout of the processor can be found in [12].

The KNL processor has two different types of DRAM memory. A Multi-Channel DRAM (MCDRAM) provides high bandwidth through 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 packet 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.

KNL employs a directory-based cache coherence mechanism using Intel MESIF [11], a variant of MESI. In order to alleviate the bottleneck of centralized directories, it features a distributed system in which each tile includes a Caching/Home Agent (CHA) in charge of managing a portion of the directory. 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).

The KNL processor features specialized *sub-NUMA* clustering modes, which provide lower memory latency to NUMA-aware applications only (e.g., MPI codes). In this work we focus on the more general *Quadrant* configuration mode, which is the de-facto standard in which any processor can access any memory block. Commonly, the access time of a core to any memory block is assumed to be UMA when in Quadrant mode [18], [30]. 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 be averaged out over a sufficient number of accesses for all cores. If the fine-grained behavior of each core is analyzed, the access latency for different memory blocks is, however, not uniform. Horro *et al*. [12] measured the access latencies in Figure 1, showing that the actual communication costs from different cores to a fixed memory block are far from UMA. More precisely, the coherence traffic causes a systematic degradation of the theoretical optimal memory performance which, on average, creates the illusion of UMA behavior. They further identified the physical placement of logical entities on the processor, shown on Figure 2, which allows to optimize data and process placement to minimize traffic latencies. Finally, Horro *et al.* generated the map of the correspondence between each single block of the 16 GiB high-bandwidth MCDRAM memory to its corresponding CHA, by leveraging the performance counters provided by the architecture. Using this map, an inspector-executor approach was proposed to dynamically schedule tasks in irregular codes to processors improving both coherence and data traffic. Further details about the reverse-engineering of the mapping of the logical components of the processor to the physical dye are given in Section III.

Inspector-executor approaches present undesirable runtime overheads. If a closed form of the memory-to-CHA mapping function were known it could be exploited to devise dynamic approaches to task scheduling with lighter overhead, or provided to an optimizing compiler that generated ad-hoc schedules taking advantage of the access latency information. The following section details how the actual pseudo-random memory block mapping over the CHAs was analyzed to extract mapping functions that can be used to predict the CHA assigned to a given physical memory block. This information is then exploited in Sections V and VI to devise the proposed optimizations.

III. MAPPING THE KNL ARCHITECTURE

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. In Quadrant mode, blocks stored in a given MCDRAM interface MC can only be indexed by CHAs located in the same quadrant as MC. As such, for each block stored in MC there are only 10 (for upper quadrants) or 9 (for lower quadrants) potential CHA candidates for storing its directory information. This amounts to 76 different (MC, CHA) combinations. We initialize a sufficiently large memory buffer so that it contains at least an instance of these 76 combinations. If the distribution of memory blocks over CHAs were uniform, approximately 5 kB of memory would be sufficient. However, due to the pseudo-random nature of the distribution function, more memory may be required. In practice, we used 64 kB of data to ensure that we find at least an instance of each (MC, CHA) pair. With the results of the full CHA mapping of data found in Section IV, it can be

[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.

proved that the lower bound of this buffer size is, in general, 16 kB, or 8 kB if the buffer is 8 kB-aligned.

In order to identify the (MC, CHA) pair to which each memory block is assigned, we employ a microbenchmark that repeatedly accesses a block and flushes it from the cache N times. After these accesses, 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) to measure the number of accesses to each CHA component and MCDRAM interface.[2] After we find a block associated to (CH, MC), we repeatedly access and flush it again from each of the active tiles in the mesh, but this time we measure the access latency. Note that we only need to obtain the latency for one out of each 2 cores, since cores in the same tile will present the same out-of-tile access latency. It is inferrable from `/proc/cpuinfo` that cores $(2x)$ and $(2x + 1)$ lie in the same tile.

We discover the association of CHAs and MCDRAMs to quadrants by analyzing the missing (CH, MC) pairs in the obtained data. 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 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 [26]. 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) tiles to the physical mesh. In our Intel x200 7210, only 32 tiles have active cores. As such, we have to discover the actual location of the 32 tiles with active cores, plus the 6 disabled tiles. Taking into account that we know the associations from (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 other two 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 a heuristic approach. In the first place, 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, human-designed pattern in the location of both CHAs and cores, as shown in Figure 2. The CHAs in each quadrant are sequentially arranged in column-major order. Cores are assigned sequentially to CHAs, skipping those with disabled cores. We believe 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. Unfortunately, we have not been able to validate this assumption by experimenting on different processor units.

Physically mapping the processor mesh is the first step towards reasoning about communication latencies for IM architectures, as will be described in the following sections.

IV. REVERSE ENGINEERING THE CHA MAPPING

In hardware designs, pseudo-random mappings often make use of XOR gates, such as with Cyclic Redundancy Codes (CRCs), Linear Feedback Shift Registers, and other XOR hashes [20]. XOR mappings can be efficiently implemented in gates relative to other forms of pseudo-random mapping binary addresses, such as modulo arithmetic of the form $x = (n_1 \text{addr} + n_2) \bmod n_3$.

Horro *et al.* [12] generated a map of the correspondence between the 16 GiB of MCDRAM memory to each CHA using a similar process as the one described in Section III. The 16 GiB of MCDRAM are allocated, their virtual-to-physical correspondence checked, each memory block repeatedly read and flushed from the caches, and the MSRs configured to read information about the CHAs that are involved in that operation. In order to speed up and simplify this process, 1 GiB hugepages are used for the memory allocations. Table 1 shows the CHA mapping for the first 128 cache lines out of the 256 million mapped locations, i.e., the entire MCDRAM address space.

This section describes the analysis of this mapping data in order to generate the closed forms of the mapping functions. Since full 64-byte cache lines are stored when a CHA location is determined for the data, the address-to-CHA mapping does not make use of address bits 5:0. In a first, coarse-grained analysis of the data, we find that the CHA mapping depends only on address bits $A_{34:6}$, which allows for $2^{536,870,912}$ distinct binary functions for each of the 6 CHA bits.

Given values for CHA from 0 to 37, 6 bits are needed to represent this number, but given that 38 is not a power of 2,

[2]We employ the `PERF_EVT_SEL_X_Y` and `ECLK_PMON_CTRX_LOW/HIGH` registers to monitor CHAs and MCDRAMs, respectively [14]. We measure events `RxR_INSERTS.IRQ` and `RPQ.Inserts` [15].

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

TABLE 1. Address-to-CHA mapping for the first 128 CHA values out of 256 million. To aid in visualizing, CHA_0 and CHA_1 are shown in a box and CHA_{37} is shown in white over black.

	Address bits 9:6															
	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
Address	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
bits	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
12:10	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
000	26	9	24	11	21	6	23	4	31	12	29	14	16	3	18	1
001	27	8	25	10	20	7	22	5	14	37	28	15	33	34	19	0
010	10	25	8	27	5	22	7	20	15	36	13	30	32	35	2	17
011	11	24	9	26	4	23	6	21	30	37	12	31	33	34	3	16
100	12	31	14	29	3	16	1	18	9	26	11	24	6	21	4	23
101	13	30	15	28	2	17	0	19	8	27	34	33	7	20	37	6
110	28	15	30	13	19	0	17	2	25	10	35	32	22	5	36	7
111	29	14	31	12	18	1	16	3	24	11	34	33	23	4	37	22

we did not expect to see a straightforward XOR equation of address bits for each CHA bit. However, given the ease of computing binary functions in hardware, we did expect and found that each bit for the CHA value can be computed independently (again, as opposed to a scheme like *addr* mod 38).

The process we follow to determine the equations for CHA bits is shown in Figure 3. Our final process finds equations of the form $\text{CHA}_n = f\,\overline{g}|h$ where most of the bits are correctly predicted by the f function alone, $\overline{g}$ masks some bits to 0, and h is OR'ed in to correct some bits to 1. Following the idea that the function should be easily implementable in hardware, we first attempt to find $f = a_1 \oplus a_2 \oplus \ldots \oplus a_{n-1} \oplus z(a_n, a_{n+1}, \ldots)$; that is, f is a function which XORs certain address bits together with a binary function z which uses a small number of identifiable address bits.

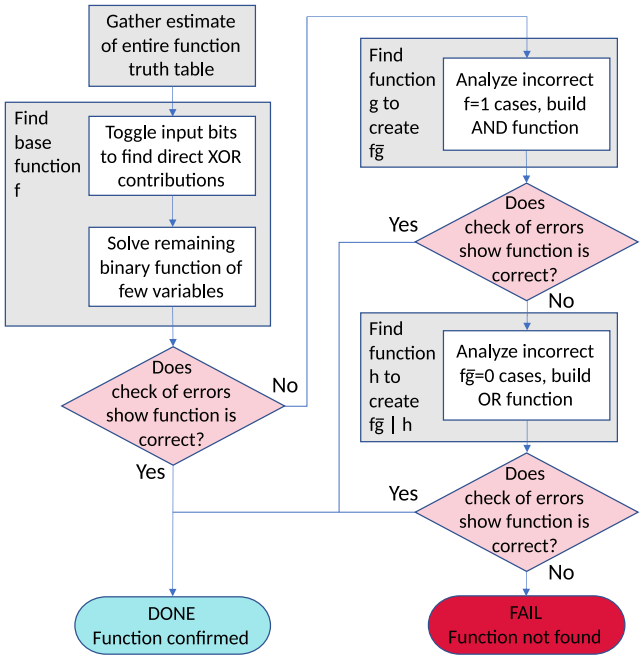


FIGURE 3. Reverse engineering hardware-friendly hash functions.

For instance, consider the toggle frequency for CHA_0 when different bits of the address are toggled shown in Table 2. As can be seen, 99.93% of the time toggling a_6 or a_8 changes the result of CHA_0, whereas toggling a_7 almost never affects its value. It can be concluded that the mapping function for bit 0 must be of the form $\text{CHA}_0 = a_6 \oplus a_8 \oplus \ldots$, where "..." is yet to be determined. The fact that the data is not 100% conclusive is either attributable to the over-simplification of the generated closed-form, or to measurement errors in the performance counter-based mapping process. The latter will be shown to be the case: the generated closed-form 100% matches the mapping, as experimentally proved below.

TABLE 2. CHA_0 toggle frequency when toggling address bits a_6 to a_{34}. In this case, values greater than 0.98 or less than 0.02 indicate errors in the CHA predicted based on performance counters, and are interpreted as 1 and 0, respectively.

Addr Bit	CHA_0 Toggles	Equation Role	Addr Bit	CHA_0 Toggles	Equation Role
a_6	0.9993	XOR	a_{21}	0.0009	Ignore
a_7	0.0001	Ignore	a_{22}	0.0011	Ignore
a_8	0.9993	XOR	a_{23}	0.9989	XOR
a_9	0.9994	XOR	a_{24}	0.0014	Ignore
a_{10}	0.9994	XOR	a_{25}	0.0014	Ignore
a_{11}	0.0002	Ignore	a_{26}	0.0013	Ignore
a_{12}	0.0002	Ignore	a_{27}	0.9954	XOR
a_{13}	0.0002	Ignore	a_{28}	0.0045	Ignore
a_{14}	0.9994	XOR	a_{29}	0.0120	Ignore
a_{15}	0.9994	XOR	a_{30}	0.9458	Function
a_{16}	0.0004	Ignore	a_{31}	0.9444	Function
a_{17}	0.9993	XOR	a_{32}	0.0555	Function
a_{18}	0.9993	XOR	a_{33}	0.0546	Function
a_{19}	0.0006	Ignore	a_{34}	0.0000	Ignore
a_{20}	0.9993	XOR			

The analysis of the toggle frequency finds that some of the bits $A_{29:6}$ are directly XOR'ed into CHA_0, while some others do not appear at all. However, the study also shows that bits $A_{33:30}$ affect the function, but not in the same categorical way. The toggle frequency is somewhere between 5% and 95%. In order to reverse engineer the role of these bits in the function, the limited input binary function of $A_{33:30}$ is analyzed to detect which combinations of these bits toggle the result of the partial XOR function built from $A_{29:6}$, as shown in Table 3. This reverse engineering process yields the functions CHA_0 and CHA_1 in Figure 4 for the 2 least significant bits of the CHA.

Although the number of CHA locations (38) is not divisible by 4, we found that CHA_{0-1} are each on for 50% of the addresses, and as seen in Figure 2 this distributes data evenly among the 4 quadrants of the die. $\text{CHA}_1 = 1$ indicates the data is in the lower half of the die; $\text{CHA}_0 = 1$ indicates the data is on the right side of the die. Given that the lower quadrants have one fewer CHA as compared to upper quadrants, this will cause an imbalance of up to 20% in the number of memory blocks mapped to different CHAs, as described in more detail in Section V.

$$\text{CHA}_0 = a_6 \oplus a_8 \oplus a_9 \oplus a_{10} \oplus a_{14} \oplus a_{15} \oplus a_{17} \oplus a_{18} \oplus a_{20} \oplus a_{23} \oplus a_{27} \oplus ((a_{30}a_{31})|(\overline{a_{30}a_{31}}(a_{32}|a_{33})))$$

$$\text{CHA}_1 = a_6 \oplus a_7 \oplus a_8 \oplus a_{12} \oplus a_{16} \oplus a_{17} \oplus a_{20} \oplus a_{21} \oplus a_{22} \oplus a_{23} \oplus a_{24} \oplus a_{25} \oplus a_{26} \oplus a_{28} \oplus a_{30} \oplus a_{33}$$

$\text{CHA}_2 = f\overline{g}$, where:

$$f = a_8 \oplus a_9 \oplus a_{12} \oplus a_{15} \oplus a_{16} \oplus a_{18} \oplus a_{21} \oplus a_{22} \oplus a_{23} \oplus a_{25} \oplus a_{26} \oplus a_{28} \oplus \overline{a_{30}}(a_{31}|a_{32}|a_{33}))$$

$$g = ((a_{11} \oplus a_{16} \oplus a_{17} \oplus a_{21} \oplus a_{23} \oplus a_{26} \oplus a_{27} \oplus a_{28} \oplus a_{29} \oplus a_{31})|(a_{10} \oplus a_{15} \oplus a_{16} \oplus a_{20} \oplus a_{22} \oplus a_{25} \oplus a_{26} \oplus a_{27} \oplus a_{28} \oplus \overline{a_{30}} \oplus a_{34}))(a_6 \oplus a_{12} \oplus a_{20} \oplus a_{21} \oplus a_{22} \oplus a_{23} \oplus a_{28} \oplus a_{32} \oplus a_{34})(a_7 \oplus a_{12} \oplus a_{14} \oplus a_{17} \oplus a_{18} \oplus a_{19} \oplus a_{22} \oplus a_{23} \oplus a_{25} \oplus a_{26} \oplus a_{27} \oplus a_{28} \oplus a_{29} \oplus a_{32} \oplus a_{34})(a_9 \oplus a_{14} \oplus a_{15} \oplus a_{19} \oplus a_{21} \oplus a_{24} \oplus a_{25} \oplus a_{26} \oplus a_{27} \oplus a_{29} \oplus a_{33} \oplus a_{34})(a_{13} \oplus a_{14} \oplus a_{18} \oplus a_{24} \oplus a_{26} \oplus a_{28} \oplus a_{29} \oplus a_{31} \oplus a_{33} \oplus a_{34})$$

$\text{CHA}_3 = f\overline{g}|h$, where:

$$f = a_8 \oplus a_{13} \oplus a_{14} \oplus a_{18} \oplus a_{20} \oplus a_{23} \oplus a_{24} \oplus a_{25} \oplus a_{26} \oplus a_{28} \oplus (a_{30}|a_{31}|a_{32})(a_{32} \oplus \overline{a_{33}})$$

$$g = ((a_{11} \oplus a_{16} \oplus a_{17} \oplus a_{21} \oplus a_{23} \oplus a_{26} \oplus a_{27} \oplus a_{28} \oplus a_{29} \oplus a_{31})|(a_{10} \oplus a_{15} \oplus a_{16} \oplus a_{20} \oplus a_{22} \oplus a_{25} \oplus a_{26} \oplus a_{27} \oplus a_{28} \oplus \overline{a_{30}} \oplus a_{34}))(\overline{a_7} \oplus a_{12} \oplus a_{13} \oplus a_{17} \oplus a_{19} \oplus a_{22} \oplus a_{23} \oplus a_{24} \oplus a_{25} \oplus a_{27} \oplus a_{31} \oplus a_{32} \oplus a_{33})(a_9 \oplus a_{14} \oplus a_{15} \oplus a_{19} \oplus a_{21} \oplus a_{24} \oplus a_{25} \oplus a_{26} \oplus a_{27} \oplus a_{29} \oplus a_{33} \oplus a_{34})$$

$$h = ((a_{11} \oplus a_{16} \oplus a_{17} \oplus a_{21} \oplus a_{23} \oplus a_{26} \oplus a_{27} \oplus a_{28} \oplus a_{29} \oplus a_{31})|(a_{10} \oplus a_{15} \oplus a_{16} \oplus a_{20} \oplus a_{22} \oplus a_{25} \oplus a_{26} \oplus a_{27} \oplus a_{28} \oplus \overline{a_{30}} \oplus a_{34}))(\overline{a_6} \oplus a_{13} \oplus a_{15} \oplus a_{16} \oplus a_{17} \oplus a_{18} \oplus a_{20} \oplus a_{21} \oplus a_{26} \oplus a_{29} \oplus a_{34})(\overline{a_7} \oplus a_{13} \oplus a_{14} \oplus a_{15} \oplus a_{16} \oplus a_{19} \oplus a_{25} \oplus a_{27} \oplus a_{34})(\overline{a_8} \oplus a_{15} \oplus a_{17} \oplus a_{18} \oplus a_{19} \oplus a_{27} \oplus a_{28} \oplus a_{29} \oplus \overline{a_{30}} \oplus a_{31} \oplus a_{32} \oplus a_{34})(a_9 \oplus a_{14} \oplus a_{15} \oplus a_{19} \oplus a_{21} \oplus a_{24} \oplus a_{25} \oplus a_{26} \oplus a_{27} \oplus a_{29} \oplus a_{33} \oplus a_{34})(\overline{a_{12}} \oplus a_{14} \oplus a_{15} \oplus a_{16} \oplus a_{17} \oplus a_{22} \oplus a_{23} \oplus a_{24} \oplus a_{31} \oplus a_{32} \oplus a_{33} \oplus a_{34})$$

$\text{CHA}_4 = f\overline{g}|gh$, where:

$$f = a_6 \oplus a_{11} \oplus a_{12} \oplus a_{16} \oplus a_{18} \oplus a_{21} \oplus a_{22} \oplus a_{23} \oplus a_{24} \oplus a_{26} \oplus a_{30} \oplus a_{31} \oplus a_{32}$$

$$g = ((a_{11} \oplus a_{16} \oplus a_{17} \oplus a_{21} \oplus a_{23} \oplus a_{26} \oplus a_{27} \oplus a_{28} \oplus a_{29} \oplus a_{31})|(a_{10} \oplus a_{15} \oplus a_{16} \oplus a_{20} \oplus a_{22} \oplus a_{25} \oplus a_{26} \oplus a_{27} \oplus a_{28} \oplus \overline{a_{30}} \oplus a_{34}))(\overline{a_7} \oplus a_{12} \oplus a_{13} \oplus a_{17} \oplus a_{19} \oplus a_{22} \oplus a_{23} \oplus a_{24} \oplus a_{25} \oplus a_{27} \oplus a_{31} \oplus a_{32} \oplus a_{33})(a_9 \oplus a_{14} \oplus a_{15} \oplus a_{19} \oplus a_{21} \oplus a_{24} \oplus a_{25} \oplus a_{26} \oplus a_{27} \oplus a_{29} \oplus a_{33} \oplus a_{34})$$

$$h = (\overline{a_{10}} \oplus a_{11} \oplus a_{13} \oplus a_{16} \oplus a_{17} \oplus a_{18} \oplus a_{19} \oplus a_{20} \oplus a_{21} \oplus a_{22} \oplus a_{27} \oplus a_{28} \oplus \overline{a_{30}} \oplus a_{31} \oplus a_{33} \oplus a_{34})(\overline{a_6} \oplus a_{12} \oplus a_{13} \oplus a_{14} \oplus a_{18} \oplus a_{20} \oplus a_{21} \oplus a_{22} \oplus a_{23} \oplus a_{24} \oplus a_{26} \oplus a_{29} \oplus a_{31} \oplus a_{32} \oplus a_{33})(\overline{a_7} \oplus a_{12} \oplus a_{13} \oplus a_{17} \oplus a_{19} \oplus a_{22} \oplus a_{23} \oplus a_{24} \oplus a_{25} \oplus a_{27} \oplus a_{31} \oplus a_{32} \oplus a_{33})(\overline{a_8} \oplus a_{12} \oplus a_{14} \oplus a_{16} \oplus a_{18} \oplus a_{19} \oplus a_{22} \oplus a_{23} \oplus a_{24} \oplus a_{27} \oplus a_{28} \oplus a_{29} \oplus \overline{a_{30}} \oplus a_{33})(a_9 \oplus a_{14} \oplus a_{15} \oplus a_{19} \oplus a_{21} \oplus a_{24} \oplus a_{25} \oplus a_{26} \oplus a_{27} \oplus a_{29} \oplus a_{33} \oplus a_{34})$$

$\text{CHA}_5 = f\overline{g}$, where:

$$f = ((a_{10} \oplus a_{15} \oplus a_{16} \oplus a_{20} \oplus a_{22} \oplus a_{25} \oplus a_{26} \oplus a_{27} \oplus a_{28} \oplus \overline{a_{30}} \oplus a_{34})|(a_{11} \oplus a_{16} \oplus a_{17} \oplus a_{21} \oplus a_{23} \oplus a_{26} \oplus a_{27} \oplus a_{28} \oplus a_{29} \oplus a_{31}))(\overline{a_7} \oplus a_{12} \oplus a_{13} \oplus a_{17} \oplus a_{19} \oplus a_{22} \oplus a_{23} \oplus a_{24} \oplus a_{25} \oplus a_{27} \oplus a_{31} \oplus a_{32} \oplus a_{33})(a_9 \oplus a_{14} \oplus a_{15} \oplus a_{19} \oplus a_{21} \oplus a_{24} \oplus a_{25} \oplus a_{26} \oplus a_{27} \oplus a_{29} \oplus a_{33} \oplus a_{34})$$

$$g = (\overline{a_6} \oplus a_{12} \oplus a_{13} \oplus a_{14} \oplus a_{18} \oplus a_{20} \oplus a_{21} \oplus a_{22} \oplus a_{23} \oplus a_{24} \oplus a_{26} \oplus a_{29} \oplus a_{31} \oplus a_{32} \oplus a_{33})(\overline{a_8} \oplus a_{12} \oplus a_{14} \oplus a_{16} \oplus a_{18} \oplus a_{19} \oplus a_{22} \oplus a_{23} \oplus a_{24} \oplus a_{27} \oplus a_{28} \oplus a_{29} \oplus \overline{a_{30}} \oplus a_{33})$$

FIGURE 4. Reverse-engineered mapping function between memory blocks and CHAs.

The functions for CHA_{2-5} are more complex than those for CHA_{0-1}, in order to reasonably distribute data among the 38 CHA values. As shown in Figure 3, finding the base function f, which matches most binary function values, sometimes does not fully match the measured function values. In such cases, we search for $\overline{g}$ functions to logically AND with f to set certain values to 0, and h functions to logically OR with $f\overline{g}$ to set certain values to 1. As an example, we will describe the process of determining the g function for CHA_2 with reference to Table 4. The f function includes $a_8 \oplus a_9 \oplus a_{12}$ which results in regular blocks of 1's and 0's when the address bits 13:6 are varied with a total of 128 1's and 128 0's in the set of 256 cache lines. However, the performance counter data implies that 6 of those 1's are actually 0's. Note that in the binary representation of 0 through 37, bit 2 is high 18/38 = 47% of the time, hence the simple 50/50 XOR equation from f needs to be masked to 0 in some pseudo-random locations resulting on $\text{CHA}_2 = f\overline{g}$. Given where the masking occurs in Table 4, we surmise the structure of the g function to be $((a_{11} \oplus \ldots)|(a_{10} \oplus \ldots))(a_6 \oplus \ldots)(a_7 \oplus \ldots)(a_9 \oplus \ldots)(\ldots)$ where "..." represents unknown functions of higher order bits. By comparing $f\overline{g}$ to the known CHA locations with partially completed g functions, we build up the complete mask functions detailed in Figure 4.

TABLE 3. For CHA_0, bits 33 to 30 do not directly get XOR'ed with other bits, but are part of a function that itself is XOR'ed with those bits.

Addr Bits 33:30	Avg CHA_0 when direct XOR low	Addr Bits 33:30	Avg CHA_0 when direct XOR low
0000	0.0019	1000	1.0
0001	0.0	1001	0.0014
0010	0.0	1010	0.0007
0011	1.0	1011	1.0
0100	1.0	1100	1.0
0101	0.0	1101	0.0
0110	0.0	1110	0.0005
0111	1.0	1111	0.9991

TABLE 4. CHA_2 for the first 256 cache lines. Given a base function which includes $a_8 \oplus a_9 \oplus a_{12}$, the 6 boxed [0] positions show where a masking function is used.

	Address bits 9:6															
	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
Address	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
bits	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
13:10	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0000	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0
0001	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0
0010	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0
0011	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0
0100	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1
0101	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1
0110	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1
0111	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1
1000	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0
1001	0	0	0	0	1	1	1	1	1	1	1	[0]	0	0	0	0
1010	0	0	0	0	1	1	1	1	1	1	1	[0]	0	0	0	0
1011	0	0	0	0	1	1	1	1	1	1	1	[0]	0	0	0	0
1100	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1
1101	1	1	1	1	0	0	0	0	0	0	0	0	[0]	1	1	1
1110	1	1	1	1	0	0	0	0	0	0	0	0	[0]	1	1	1
1111	1	1	1	1	0	0	0	0	0	0	0	0	[0]	1	1	1

Like CHA_2, CHA_5 uses a base f function and a mask-to-0 g function. Bits CHA_3 and CHA_4 had a base f function, mask-to-0 g function, and mask-to-1 h function. The h function was found by recognizing where the $f\overline{g}$ pattern itself was not producing correct predictions. The process of determining the f function by observing XOR toggle indications and solving the 4- or 5-bit binary function remaining can be automated. In theory, given a rough constraint on the types of functions to be considered, the process of finding the g and h functions could also be semi-automated by searching for mispredictions of the f function to the true result. Future architectures may vary the mapping structure, but the general approach used here to reverse engineer XOR trees would be applicable to them. However, involving a human to interpret binary results and build the final equations may remain common for this type of task. Note that this approach allows for discovery of partial equations without necessarily solving the full binary function.

After generating the closed forms, we find a small proportion of 0.03% discrepancies when comparing the original mapping data to the CHAs predicted by the generated closed-form functions. We validated the closed forms by re-running the mapping microbenchmarks for these divergent blocks, this time finding 100% agreement with the generated closed forms. Consequently, we attribute the discrepancy in the original mapping to transient measurement errors in the MSRs.

V. RUNTIME OPTIMIZATION

As mentioned in Section II, Horro *et al.* [12] developed an inspector-executor approach to the optimization of coherence traffic in KNL processors. This approach is limited to irregular codes, and consists in transforming the data layout so that the data to be accessed by each tile lie in memory blocks for which the coherence information was assigned to nearby CHAs. This approach has an important overhead during the inspection phase. First, the input data need to be physically copied to target memory blocks with the required coherence properties. Then, the associated indirection arrays need to be recomputed. Lastly, the resulting data are now spread across a much larger region of memory, in order to find suitable memory blocks, and therefore cache locality is degraded and the number of page faults increased. With the closed form of the mapping functions exposed in Section IV, it is possible to apply this approach to general codes, instead of being restricted to irregular computations. The basic idea is to encode the schedule of tasks not on the indirection arrays, but to exploit the properties of the mapping function.

Consider the general matrix-vector multiplication code depicted in Figure 5. This is an interesting problem because of its simplicity, its transversality, and because of the fact that it is memory-bound in modern processors. As such, it will benefit from increasing the memory throughput. The dominant part of the memory footprint of the computation is the access to matrix B, and therefore the following analysis will be centered on trying to optimize its access.

```
1 #pragma omp parallel for
2 for (int i = 0; i < N; ++i)
3     for (int j = 0; j < N; ++j)
4         y[i] += B[i * N + j] * x[j];
```

FIGURE 5. Scalar code for general matrix-vector multiplication parallelized using a static block schedule.

Given the complexity of the mapping functions, it is implausible to dynamically perform a very fine-grained scheduling of iterations to tiles that will actually have the required coherence information in its local CHA. Besides, this would imbalance the computation, as our mapping data shows that some CHAs manage up to 20% more memory blocks than others. This is a consequence of two different factors. First, the upper quadrants have 10 CHAs each, whereas the lower quadrants have only 9 CHAs. That will create some imbalance, given that the memory distribution over quadrants is balanced, i.e., each quadrant manages exactly 4 GiB of memory. But furthermore, distributing a power of 2 number

of memory blocks over a non-power of 2 number of CHAs creates an additional imbalance. The actual distribution of memory to CHAs is as follows:

- Upper quadrants have 10 CHAs, 8 of them manage 416 MiB each, while the remaining 2 (those with the highest IDs in each quadrant) manage 384 MiB each.
- Lower quadrants have 9 CHAs, 8 of them manage 464 MiB each, while the remaining one (that with the highest ID in each quadrant) manages 384 MiB.

Note that this does not vary across different Xeon Phi x200 models, as all units have 38 enabled CHAs, independently of the number of active cores.

In order to alleviate this imbalance we focus instead on the quadrant granularity, emulating the behavior of the sub-NUMA modes of the machine by ensuring that each tile computes data with coherence information resident on its quadrant only. In this fashion, each quadrant manages exactly 4 GiB of memory. The approach followed for scheduling iterations in this fashion is described in the following.

The quadrant mapping benefits from a convenient feature of the address-to-CHA functions. As noted in Section IV, and due to the physical placement of logical CHAs on the network-on-chip shown in Figure 2, bits $\text{CHA}_0 = c_0$ and $\text{CHA}_1 = c_1$ identify the quadrant c_1c_0 in which the CHA is located. Consider the kth memory block with address A^k aligned to a 256-byte boundary, i.e., k is a multiple of 4. Bits $A^k_{5:0}$ express an offset inside the memory block, and therefore are not used in the computation of the associated CHA. Because of the 256-byte alignment, $A^k_{7:6} = 00_b$. The address of the next memory block, $A^{k+1} = A^k + 64$, will share its most significant bits with A^k, i.e., $A^{k+1}_{63:8} = A^k_{63:8}$, and $A^{k+1}_{7:6} = 01_b$. Since A_6 participates in the XOR computation in the equations for CHA_1 and CHA_0 in Figure 4, it can be determined that the least significant bits of its associated CHA will be flipped, i.e., if the associated quadrant for A^k is c_1c_0 then the associated quadrant for A^{k+1} will be $\overline{c_1}\,\overline{c_0}$. Similarly, $A^{k+2}_{7:6} = 10_b$ and its associated quadrant will be $\overline{c_1}c_0$; and $A^{k+3}_{7:6} = 11_b$ and its associated quadrant is $c_0\overline{c_1}$. This results in the convenient organization that precisely 1 out of every 4 cache lines is in each physical quadrant, allowing parallel access routines to evenly divide up work among physical processors.

In the proposed sub-NUMA schedule a processor located in quadrant c_1c_0 will process only memory blocks with associated CHA in the same quadrant. After processing a block at address A, the next address in the same quadrant could be located at $A + 100_b$, $A + 101_b$, $A + 110_b$, or $A + 111_b$ depending on $A_{33:8}$. Determining which of the 4 addresses is next in our quadrant mathematically requires to compute the full CHA equations discovered in Section IV. However, these are complex so these computations should be performed as little as possible. The actual offset required to compute the next address in quadrant c_1c_0 has a fixed pattern for address bits $A_{12:8}$, which allows a 64-bit register to store the offsets for the next 32 cache lines. In this way, processors stepping through memory can thus avoid full computation of the mapping function 31 out of each 32 iterations.

A. EXPERIMENTAL RESULTS

In order to have full control over the executed instructions, the original code from Figure 5 is manually vectorized using AVX-512 intrinsics as shown in Figure 6. In this way, opaque optimizations that may bias the comparison of different schedules are avoided. This section focuses on single-precision floating point arithmetic only, but all obtained results are directly extrapolable to double-precision FP.

```
1  #pragma omp parallel for
2  for (int i = 0; i < N; ++i) {
3     __m512 bb, bx, accum;
4     accum0 = _mm512_setzero_ps();
5     for (int j = 0; j < N; j += 16) {
6        bb    = _mm512_load_ps(&B[i * N + j]);
7        bx    = _mm512_load_ps(&x[j]);
8        accum = _mm512_fmadd_ps(bb, bx, accum);
9     }
10    y[i] = _mm512_reduce_add_ps(accum);
11 }
```

FIGURE 6. Manually vectorized code for general matrix-vector multiplication parallelized using a static block schedule.

Both the code in Figure 6 and the equivalent sub-NUMA schedule are executed on an Intel x200 7210 running at the base frequency of 1.30 GHz, to avoid turbo-related variations. The codes were compiled using ICC 19.1.1.217, with flags `-Ofast -xKNL -qopenmp`. They are executed on 64 threads using `KMP_AFFINITY=scatter`. Heap variables are stored into 1 GiB hugepages via `hugectl -heap`, and these hugepages are guaranteed to be allocated in the MCDRAM address space using `numatcl -m 1`. The experiments are run with $N = 16384$, which makes matrix B take up 1 GiB of memory, that is, an entire hugepage.

The roofline model generated by Intel Advisor [29] for these codes is shown in Figure 7. For these experiments, the hardware prefetcher was manually turned off using Model Specific Registers (MSR) [36] in order to observe the raw effect of the proposed coherence traffic optimizations without interference. As shown in the figure, the sequential schedule achieves 50.7 GFLOPS for an arithmetic intensity (AI) of 0.25, which is approximately 65% of the roofline for that AI, whereas the sub-NUMA schedule achieves 54.7 GFLOPS for an AI of 0.22, or 81% of the roofline. The GFLOPS have increased and the AI has decreased, due to the additional memory traffic required to compute the sub-NUMA schedule, resulting in a large net increase of the percentage of peak performance that is obtained. The figure shows how executions with double-precision arithmetic achieve the same approximate results, but dividing the number of raw GFLOPS by 2.

The improvement in raw performance measured by the roofline model, however, can be deceitful. Although the sub-NUMA schedule achieves a higher FLOP count, it also

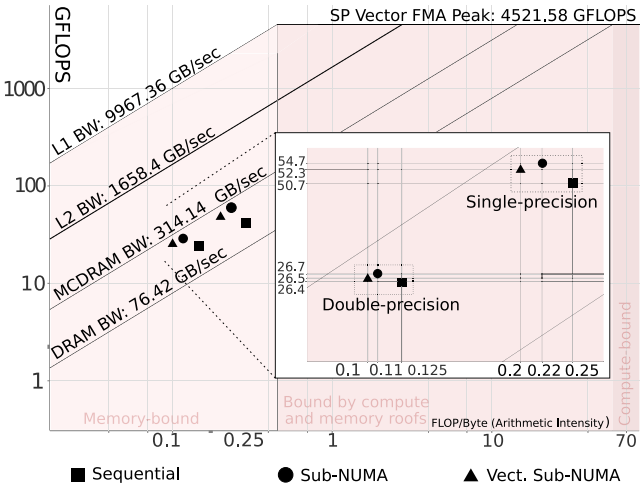


FIGURE 7. Roofline plot for the matrix-vector multiplication using both single- and double-precision arithmetic.

executes additional instructions on non-consecutive memory blocks, causing a degradation in cache behavior and ultimately execution time. In order to more closely investigate the effect of the proposed optimization, selected performance counters were measured for several different execution setups. The results are shown in Figure 8. In order to compute the sub-NUMA schedule, the number of instructions to be executed almost triples, increasing by 188%. The largest share of these are data L1 loads and stores, which grow by 145%. This increase, however, is absorbed by the L2 cache, and the L2 misses remain virtually identical. There is a very significant increase in the IPC of these codes, which goes from 18.6 in the original version to 53.15 in the sub-NUMA schedule. The memory latency, approximated by the `OFFCORE_RESPONSE_0:OUTSTANDING` performance counter, is slightly decreased by 1.8%. All these variables compound for an almost zero net effect on execution time: execution cycles are reduced by a modest 0.8%.

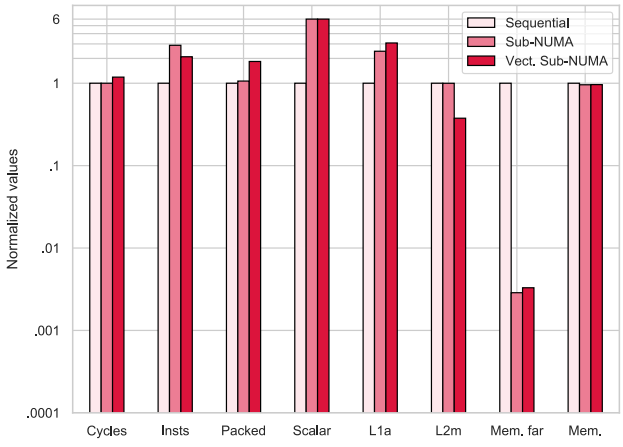


FIGURE 8. Sum of selected performance counters for all threads. Logarithmic scale is used for the Y axis. The figure shows the number of cycles, instructions issued, packed SIMD instructions, scalar SIMD instructions, L1 data accesses, L2 misses, MCDRAM "far" accesses to other quadrants in the NoC, and total number of MCDRAM accesses. Values are normalized to those of the sequential schedule.

In order to try to decrease the schedule-related computations, a modified version which employs vectorization operations for offset computation was developed. In essence, the offsets for each 32 consecutive memory blocks are now computed using AVX-512 arithmetic. This version, labeled "Vect. sub-NUMA" in Figures 7 and 8 reduces the number of instructions by 37.8% with respect to the regular sub-NUMA schedule. However, it worsens register pressure, increasing L1 accesses by a further 26%. As a result, the GFLOPS decrease to 52.3, and so does the AI to 0.20, for a grand total of 82.4% of the peak performance.

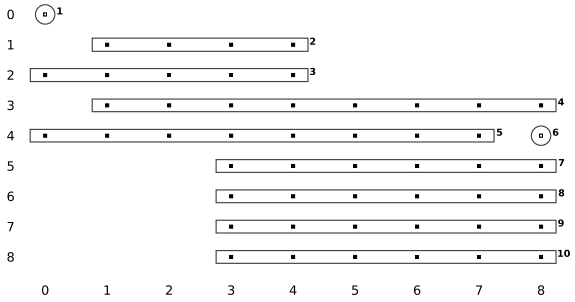
As previously mentioned, these results were executed after disabling the hardware prefetching. The reason is that the sub-NUMA schedule does not access memory sequentially, and is at a tremendous disadvantage against the sequential schedule when the prefetcher is enabled, which would absorb and eliminate any potential advantage from the sub-NUMA schedule. In fact, when enabling the hardware prefetcher the performance of the sequential schedule is improved by 1.2x, whereas it is detrimental for sub-NUMA (i.e., its performance slightly decreases by approximately 5%) as it features a pseudo-random access pattern that mimics the memory-to-CHA mapping funtions.

VI. COMPILE-TIME OPTIMIZATION

As shown by the experiments in the previous section, improving the mesh locality during runtime has an important impact on other execution parameters due to the pseudo-random nature of the memory-to-CHA mapping functions and their computational complexity. A different way to exploit this knowledge is to optimize the scheduling of completely static codes during the compilation stage.

Augustine *et al.* [2] recently proposed a data-specific code generation technique for the optimization of sparse-immutable codes, including artificial neural network inference. In essence, this approach automatically builds sets of regular subcomputations by mining for regular subregions in the irregular data structure. The resulting code is specialized to the sparsity structure of the input matrix, but does not employ indirection arrays, improving predictability and SIMD vectorizability. This section focuses on the sparse matrix-vector multiplication (SpMV) as an immediate target of this class of data-specific optimizations.

A graphical depiction of a small subset of operations performed by the sparse matrix-vector multiplication of matrix `FIDAP/ex7`, included in the SuiteSparse Matrix Collection [7] is offered in Figure 9. For many sparse matrices, this code generation approach delivers better performance than the generic, irregular alternative. Besides promoting vectorization, data-specific approaches encode the matrix structure implicitly in the program source. This does not only reduce the number of memory accesses, but collaterally stores the matrix structure in the first-level instruction cache, which



```
y[0] += A[0] * x[0];
y[1] += A[1:4] * x[1:4];
y[2] += A[5:9] * x[0:4];
y[3] += A[10:17] * x[1:8];
y[4] += A[18:25] * x[0:7];
y[4] += A[26] * x[8];
y[5] += A[27:32] * x[3:8];
y[6] += A[33:38] * x[3:8];
y[7] += A[39:44] * x[3:8];
y[8] += A[45:50] * x[3:8];
```

FIGURE 9. Sets of regular subcomputations built for the Sparse Matrix-Vector multiplication of matrix `FIDAP/ex7` in the SuiteSparse repository. The figure in the left shows the location of the nonzero points in the upper left corner of the matrix. Each identified regular subcomputation is marked as a rectangle enclosing several nonzeros, and captured as an AVX-512 operation, as shown in the pseudo-code on the right.

is classically underutilized for small irregular codes such as SpMV. The effect is similar to extending the first-level data cache: matrix structure will be stored in the instruction cache (since it is embedded in the code), whereas actual matrix values will be stored in the data cache. The immediate disadvantage is that the code grows proportionally to the matrix size. Still, for sufficiently regular sparse matrices the combined size for structure and data values (the program footprint) will be small enough as to benefit from this tradeoff.

As opposed to the dynamic approach of Section V, the static optimization has no explicit execution overhead. As such, the schedule of each computation can be carefully analyzed and planned in order to improve coherence traffic. Note that, as opposed to the dynamic approach in which the mapping functions could be applied on already-allocated memory, in this case the memory allocation must be statically known. The approach employed for this is detailed in Section VI-A. For the remainder of this section it is assumed that the physical address associated to each data block in the program is statically known.

Consider the generic SpMV statement s executed by the data-specific approach:

$$s : y_i = A_j \cdot x_k$$

Note that this statement does not include irregular indices, since the code has been generated for a specific input matrix with a fixed sparsity structure, as exemplified in Figure 9. Consequently, the compiler has static knowledge of all the memory movements that will be required for executing each specific part of the code. At a glance, the proposed compile-time approach computes an access cost for each statement in the data-specific SpMV code for each tile in the processor, and then schedules operations across the mesh following a greedy approach. Access costs are dynamically updated during the scheduling process to reflect the updated placement of each memory block in the private caches of each tile.

Consider a data block B with directory information associated to tile T_d and actual data accessed through tile T_B. The actual source of data can either be the private L2 cache of tile T_B, if the associated tile is the Forwarder for B; or T_B can be one of the tiles with an associated memory interface, which will serve B after reading it from memory. Regardless of the actual coherence status of B, in order to access the data the requestor tile will send a message to T_d, which will forward the request to T_B, which in turn will send B back to the requestor. Figure 10 illustrates this situation. Note that T_d and T_B constitute the opposite corners of a rectangle on the network-on-chip (NoC) which contains the tiles that can access B with minimum latency. Tiles outside this rectangle incur extra latency, which can be computed as $2 \times (2 \times D_x + D_y)$, where D_x and D_y are the horizontal and vertical distances from the tile to the rectangle, respectively.

Based on these access times, a scheduling system is developed, conceptually described in Algorithm 1. Each tile in the NoC is visited in order, and for each of them the subset of operations to be executed on that tile is selected in a greedy, iterative fashion, choosing the one with the smallest data movement cost at each iteration, until that tile reaches its balanced load. The upper bound of its computational complexity is $O(\mathcal{S}^3)$: the algorithm essentially distributes all of the statements in the program to the tiles in the mesh, which would present linear complexity on $\mathcal{S}$. However, the cost of the set of remaining statements has to be recomputed frequently, due to the data movements derived from the assignment decissions in each iteration of the inner loop. The cost τ of executing

Algorithm 1: Static Scheduling of SpMV Operations

Input: Set $\mathcal{S}$ of SpMV statements to be scheduled
Input: Set $\mathcal{T}$ of tiles in the NoC
Output: Schedule $\Theta(\mathcal{S}) \rightarrow \mathcal{T}$
Compute $L_T =$ total number of FLOPS in $\mathcal{S}$;
Compute $L_b = \frac{L_T}{\#\mathcal{T}}$ the number of FLOPs to be computed by each tile to balance load;
foreach *tile* $t \in \mathcal{T}$ **do**
  while $Load(t) < L_b$ **do**
    Select $s \in \mathcal{S} : \tau(s,t) \leq \tau(s',t), \forall s, s' \in \mathcal{S}$;
    Assign $\Theta(s) = t$;
    Update $\mathcal{S} = \mathcal{S} - \{s\}$;

each statement s in tile t is computed as:

$$\tau(s,t) = \tau(y_i,t) + \tau(A_j,t) + \tau(x_k,t)$$

That is, the aggregated cost of accessing memory blocks y_i, A_j, and x_k from tile t. For each individual memory block, its access cost is computed as:

$$\tau(B,t) = \lambda_B + 2 \times \big(2 \times D_x(t,R_B) + D_y(t,R_B)\big)$$

where:

- λ_B is a factor that depends on the latency to physically access B, including 12 cycles for accessing a private L2 in the NoC, and 115 cycles for accessing an MCDRAM interface.
- $D_{x/y}(t,R_B)$ is the horizontal/vertical distance from the requestor tile t to the rectangle defined by the tiles in its opposite corners T_B and T_d, containing the data and the coherence information, respectively, as described in Figure 10.

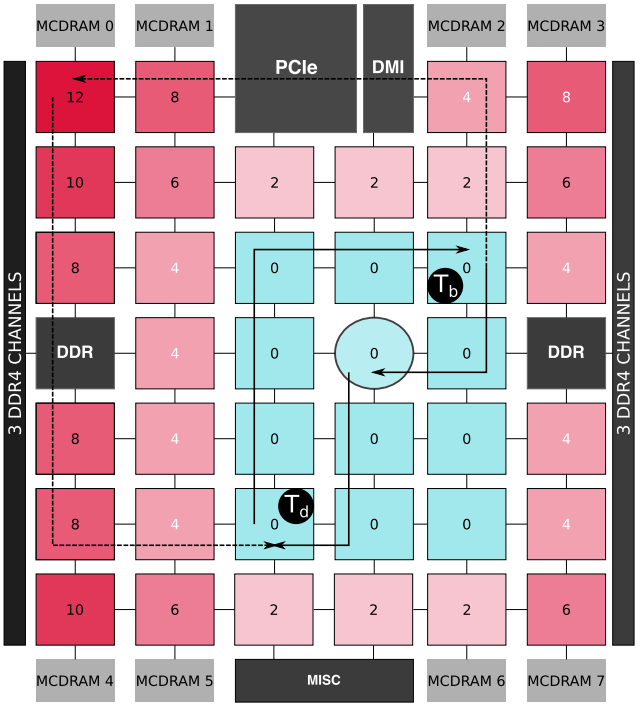


FIGURE 10. Overhead, in mesh cycles, of accessing a block of data resident in the L2 cache of tile T_B, and with coherence information resident in tile T_d. All the tiles inside the rectangle defined by T_d and T_B access data in 14 cycles with zero associated overhead. In any communication, first the CHA at T_d is queried. Then the CHA sends a *F*orward request to the source L2 cache at T_B, which sends the data back to the requestor. For any tile inside the 0-overhead rectangle, being closer to the CHA means a shorter travel time for the query, and a larger one for the response. These compensate one another, yielding zero net effect. For tiles outside the rectangle, the overhead is compounded by the extra time that the query needs to enter the rectangle, plus the extra time that the response needs to arrive back at the requestor from inside a rectangle tile.

The order in which each of the tiles is visited is carefully selected: those with worst-case trip times are selected first. For instance, the upper-left tile in the NoC has a worst-case round trip time of 32 cycles when accessing data with T_d or T_B on the bottom-right tile. However, the round trip time from a central tile to any other tile in the mesh is of at most 18 cycles.

Note that the schedules generated by this static optimization process are no longer sub-NUMA, as opposed to the dynamic approach in Section V. In this case, there is no runtime constraint enforcing quick computation of the schedule, so the system can use the full fine-grained information about memory-to-CHA mapping to decide whether accessing data on a different quadrant will be the best option from a coherence traffic point of view.

Once all operations are scheduled, the code is generated specifically for each tile. In order to reduce code sizes, affine compression may be applied to group similar operations together on regular affine loops [31]. These do not employ indirection arrays, being still fully vectorizable, while reducing the pressure on the instruction cache.

A. FIXING PHYSICAL ADDRESSES

One of the challenges of static scheduling with this class of pseudo-random functions is that it is not possible to compute the associated CHA of a virtual address, as the 34 least-significant bits of the address will be used. Even with 1 GiB hugepage sizes, the maximum supported by the architecture, only 30 bits remain unchanged during the virtual-to-physical address translation. This means that the code cannot rely simply on page alignment, as can be done for cache optimization, and must target specific physical pages.

In order to fix the physical pages that are assigned to a specific application, we employ 1 GiB hugepages. Since the MCDRAM address space has only 16 GiB in total, there will only be 16 possible pages that can be assigned to our application. The assignment order varies slightly depending on the machine state upon launch. To overcome this difficulty we employ a hybrid static/dynamic approach. During the static analysis, the code generated assumes that specific 1 GiB hugepages will be allocated to the different data structures in the program. These assumptions are registered in static constant variables in the source code. During runtime, an executor overallocates as many 1 GiB pages as possible. Then, it translates their virtual addresses to physical addresses by reading the process pagemap in `/proc`. Finally, it assigns the required hugepages to the data structures in the code by comparing the allocated physical addresses to the static constant variables assumed during the scheduling process, and frees the remaining, unused ones.

B. EXPERIMENTAL RESULTS

We generate data-specific codes for more than 20 sparse matrices selected from the range of matrices between 1 million and 10 million nonzeros in the SuiteSparse repository. The upper bound is used for tractability purposes. The lower bound to ensure sufficiently large operation. The selected matrices were the cluster centroids resulting from running k-means on SuiteSparse and using regularity and size

as the target characteristics [2]. Each of the selected matrices was processed to extract the data-specific operations required by its SpMV. Three different implementations were generated for testing:

- The generic irregular version of Figure 11.
- A data-specific version with sequential schedule, as described by Augustine *et al*. [2].
- A data-specific version containing exactly the same set of operations, but scheduled in a coherence-aware fashion using Algorithm 1.

```
1 #pragma omp parallel for private(j)
2 for(i = 0; i < n; ++i) {
3     y[i] = 0;
4     for( j = pos[i]; j < pos[i+1]; ++j)
5         y[i] += A[j] * x[cols[j]];
6 }
```

FIGURE 11. Classical, irregular SpMV code.

Codes are compiled using ICC 19.1.1.217 with `-Ofast -xKNL -qopenmp`. They are executed on an Intel x200 7210, running at the base frequency of 1.30 GHz, to avoid turbo-related variations, using 64 threads, one per core in the NoC. Ten repetitions were performed for each execution, and average values are reported for each thread after discarding outliers (identified as values x such that $|x-\bar{X}| > 3\sigma(X)$). For the generic irregular version and the sequentially-scheduled data-specific one the "scatter" thread placement is employed. For the coherence-aware version an ad-hoc assignment is employed, ensuring that each thread is executed on the appropriate statically scheduled tile. These codes are typically very large in size, explicitly containing the full set of operations to be performed for multiplying a sparse matrix by a given vector. Executable sizes vary between 39 and 206 MiB. As for dynamic scheduling, `hugectl -heap` and `numactl -m 1` are used to control the use of hugepages and memory domains. The hardware prefetcher is enabled for all the experiments in this section.

The data-specific versions were found to be 2.1x faster on average than the generic irregular version. This is a clear indication that a manycore architecture with light, vectorization-oriented processors is not well geared towards irregular codes, which feature many control flow-related instructions such as induction variable increments and branches. The data-specific versions perform, on aggregate, 4.7x less L1 accesses, but incur 1.2x more L1 data misses. The L1 instruction misses increase by 39.9x. This increase is mostly absorbed by the L2 cache and the hardware prefetcher, however, and overall the number of L2 misses is only 12.8% higher in the data-specific versions. Furthermore, these additional misses are resolved locally by the mesh, and the number of MCDRAM accesses decreases by 21.6%. In summary, the memory behavior, which is potentially the weakest runtime aspect of a data-specific version, is not significantly worsened. In exchange, the data-specific codes execute 5.4x less instructions, including 2.3x less scalar operations and 859x more vector operations. The biggest culprit in runtime difference is precisely the number of executed instructions, and the number of stalls due to missing reservation stations is 4.9x larger on irregular codes. Due to these intrinsic differences in the nature of each implementation, we drop the irregular version of SpMV in the following experiments, and focus on comparing only the sequential and coherence-aware schedules of the data-specific implementations. Figure 12 shows these detailed results.

From a performance point of view, on aggregate the coherence-aware schedule increases execution time by 3.2%. The detailed execution cycles obtained for the SpMV of each matrix are shown in Figure 13. None of the matrices

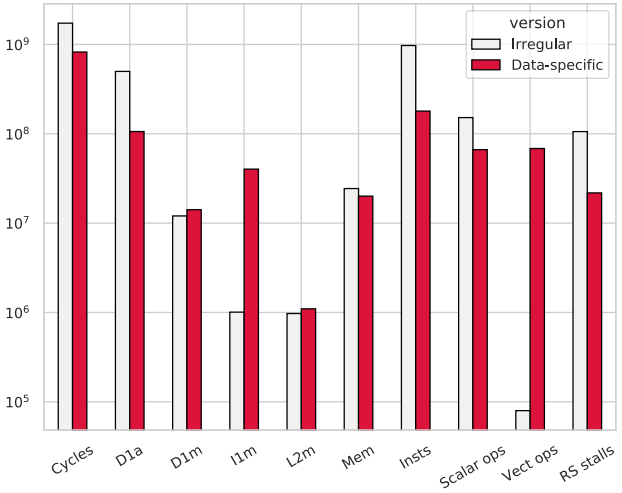


FIGURE 12. Execution cycles, data L1 accesses, data L1 misses, instruction L1 misses, L2 misses, MCDRAM accesses, executed instructions, scalar operations, vector operations, and stalls due to missing reservation stations (RS) for irregular and data-specific versions of the SpMV operation. Note that the Y axis is truncated for readability.

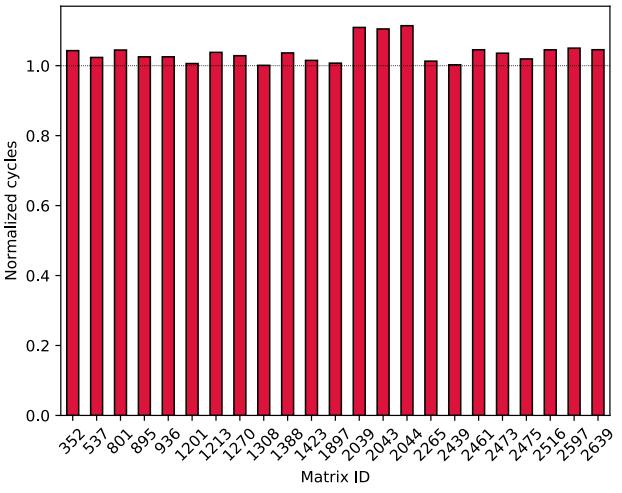


FIGURE 13. Execution cycles of the coherence-aware schedule, normalized to those of the sequential schedule.

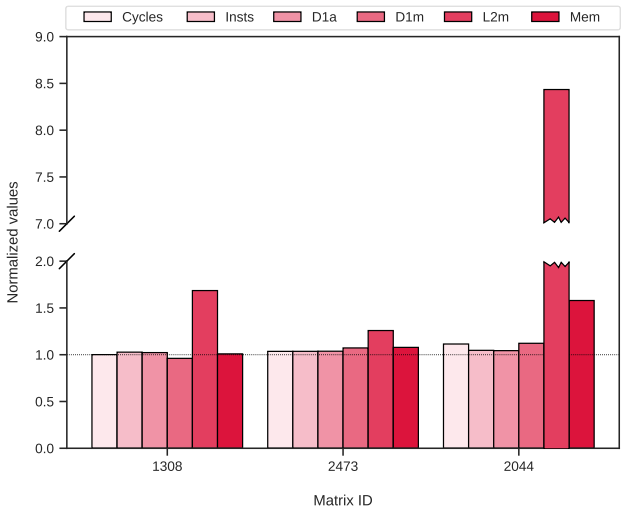


FIGURE 14. Execution cycles, executed instructions, data L1 accesses, data L1 misses, L2 misses, and MCDRAM accesses of the coherence-aware schedule for selected matrices in the experimental set: the one with the best relative performance (#1308), the one with the worst one (#2044), and the middle case (#2473). Values are normalized to those of the sequential schedule.

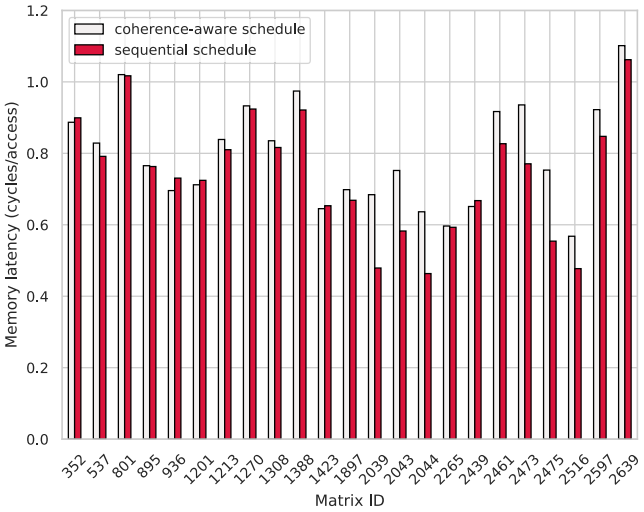


FIGURE 15. Memory latency of the coherence-aware schedule normalized to the sequential schedule baseline for all the matrices in the experimental set.

achieves a performance improvement, the best one being 0.1% slower than the baseline. For some matrices the operation is noticeably slower, the extreme case being 10.3% less performant.

In order to study in more detail the reasons for this performance degradation, three selected matrices are closely examined. Selected performance counters for these matrices are detailed in Figure 14. On careful inspection the performance is strongly correlated with the number of MCDRAM accesses incurred by each version of the code (with Pearson's correlation coefficient $R = 0.91$).

The conclusion to be inferred from these experiments is that, even with fully static scheduling, CHA locality cannot be appropriately leveraged to improve performance of data-specific sparse codes. The reason is that, due to the pseudo-random nature of the assignment between memory blocks and CHAs, rescheduling the code to promote the access of nearby CHAs to improve the cache coherence traffic patterns necessarily impacts cache locality negatively for codes benefiting from sequential data access. Even though SpMV has a varying degree of randomness in the access to the x vector, the matrix data in A can be accessed sequentially, and this is a huge advantage of the sequential schedule, particularly taking into account the hardware prefetcher. Despite the performance degradation, a careful analysis of the performance counters evidences that the coherence-aware schedule broadly improves memory latency, as shown in Figure 15, by 10% on aggregate. Average latency goes from 0.77 cyles per access in the sequential schedule, to 0.70 cycles per access in the coherence-aware schedule. The IPC is very slightly increased, going from 12.37 to 12.42.

VII. DISCUSSION AND RELATED WORK

When Horro *et al.* [12] initially explored the optimization of coherence traffic on the Knights Landing NoC, they observed a clear effect on the application performance due to affinity relationships between cores and CHAs. This work was based on a pre-computed assignment of memory blocks to CHAs. The optimized scheduling was performed dynamically in an inspector-executor fashion, which represents a very costly step that would negate any actual performance benefit in a real setting. Furthermore, the rescheduling could only be applied to irregular codes. Based on these promising results, the current work focused on reverse engineering the functions behind this mapping. The authors expected these functions to be useful to alleviate the overhead of the inspector-executor approach, in addition to being usable by architecture-specific compilers that could perform low-level optimizations of coherence traffic. However, these expectations were toned down by the actual shape of the mapping functions. Although the XOR-based functions are cheap to implement in hardware and widely used for other non-regular mappings, such as the assignment between memory blocks and LLC slices in Intel Core processors [9], [16], [23], they are costly to compute in software. This cost can be overcome if the mapping presents some kind of regularity that can be exploited by carefully optimizing the code and schedules. For instance, Scolari *et al.* [33] propose to exploit the knowledge about the hash functions that map data to LLC slices in an Intel Sandy Bridge processor to achieve performance isolation. This is possible since the hash functions which govern this mapping only employ a simple XOR of selected bits from 17 to 32 of an address, which means that blocks of 64 kB of contiguous and aligned data will be mapped to the same LLC slice. This approach is limited to processors having a power of 2 number of cores, as otherwise the mapping functions will likely be nonlinear. This is the reason for the contrasting complexity

of the equations presented in Section IV, which require XOR and negations, dramatically broadening the search space and making brute-force approaches essentially infeasible. This complexity is derived from the non-power of 2 number of tiles in the Xeon Phi x200.

When trying to exploit the mapping of coherence data to CHAs in the Xeon Phi x200 architecture, the software complexity of the XOR-based hash functions, together with their pseudo-random nature, in which sequences of four consecutive memory blocks are guaranteed, by design, to be mapped to CHAs in different quadrants, makes it impossible to apply similar, regular schemes. The approaches proposed in Sections V and VI achieve to reduce contention on the processor network, but ultimately do not achieve to improve execution performance due to the computational cost involved in the optimized scheduling.

Note that the approach followed in this paper has focused on a particular Intel Xeon Phi 7210 unit, but it is generalizable to any Xeon Phi 72xx. We have studied other units, including 7250 and 7290, and found that the memory-to-CHA mapping is identical, as is the physical placement of the CHAs on the network and the way to distribute the logical core IDs over the set of enabled physical cores. This set, however, is subject to fabrication process variations and changes for each specific unit of the processor. The process for mapping the logical components of the processor to their physical locations, based on profiling the communication latency of different logical entities in the processor, followed by a discrimination process based on mean squared error models, is applicable to other processor designs. The process presented to reverse-engineer the binary functions for the memory-to-CHA mapping could be applied when searching for hardware-friendly hash functions in general. The presented flow chart in Figure 3 may not work for a specific problem, but could provide useful information which indicate adjustments to explore or rule out certain function forms.

For all these inconveniences from the high-performance computing point of view, the approach followed by Intel has many advantages in everyday computing. It is implausible to write a code that systematically accesses only a particular set of CHAs, making them into a bottleneck. Such a bottleneck can happen with regular mappings, such as a modulo-based mapping that can suffer from systematic conflicts for certain access patterns. Furthermore, it manages to distribute memory blocks across the quadrants in the NoC in a fair fashion, ensuring that all of them have to manage the same amount of information on aggregate. This is no simple task, given the irregularity of the NoC, which features a non-power of two number of tiles, unevenly distributed across quadrants. Still, the price to pay is an all-to-all coherence traffic pattern which requires dedicated communication rings to handle.

Going forward, it would be desirable to improve this design, coupling the directory distribution that avoids bottlenecks in the NoC with a more regular and predictable mapping of the memory blocks to enable programmers, particularly in the high-performance computing domain, to have full control over coherence traffic. Horro *et al*. [13] developed a simulator for the traffic on the NoC of distributed directory architectures based on the Tejas architectural simulator [32], predicting that codes with coherence traffic control would experiment a 20% decrease in overall traffic over the NoC, yielding more than 50% latency improvement for the coherence packets. McCalpin [24], [25], [27] discusses address hashing in Intel Scalable and KNL processors, analyzing the binary permutations in address bits and their physical location in the mesh, i.e. their corresponding CHA. His work focuses on cache line distribution, rather on the actual hash functions.

In recent years, a number of papers have explored the design of scalable networks-on-chip to support manycore architectures. Daya *et al.* [8] 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*. [10] 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*. [5] 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. Numerous other techniques to efficiently schedule applications on network-on-chips have been developed. Kim *et al*. [19] focus on categorizing the memory accesses behavior of threads and employing different policies for them. Xiao *et al*. [38] parallelize applications by implementing careful load balancing between cores and minimizing inter-core communications. Liu *et al*. [21] 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. Lu *et al*. [22] propose a polyhedral model and associated optimizations to achieve data locality in these topologies. In these works, no particular consideration is given to the effect of distributed cache home agents on memory access latency, and therefore deploying such approach on a KNL may be refined with placement and subsequent inter-core communications further improved using the results we presented earlier.

Several papers have explored the performance of the KNL architecture, mainly through the analysis of well-known benchmarks, machine learning applications, and parallel workloads [3], [4], [6], [17]. None of these works undertake the analysis of the locality characteristics of the KNL interconnect. Ramos and Hoefler [30] 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.

Finally, other works have proposed ways to discover architectural features, or to automatically tune applications in highly complex systems. Yotov *et al*. [39] developed a set of microbenchmarks to measure parameters of the memory hierarchy. Wang *et al*. [37] argue that the static discovery of

optimal configuration parameters is a fundamentally flawed approach, proposing a configuration interface to specify performance constraints that should be satisfied at runtime. Mishra *et al*. [28] propose to use automatic learning systems to manage resources towards meeting specific latency and energy constraints.

VIII. CONCLUSION

Current manycore designs are usually based on replicated IP blocks connected by a high-performance fabric. An example of such an approach is the Intel Mesh interconnect (IM), first featured in the Intel Xeon Phi Knights Landing (KNL) processor [34]. The IM is the current interconnection standard in the most advanced Intel processors, including Intel Xeon Scalable servers and the High-End Desktop family of Core-X chips [1], [35].

In this work, we presented the first complete reverse-engineering of the hardware mapping functions between memory block addresses and the Cache/Home Agent on the KNL, exposing complex bitwise XOR-based functions that can then be exploited at compile-time to further improve data access latency via careful placement. We presented different optimization strategies based on both dynamic and static work scheduling. Extensive experiments quantified the merits and drawbacks of the proposed optimizations, improving memory access latency by leveraging the spatial locality of CHAs. However, our experiments clearly expose the limitations of exploiting such complex XOR-based functions in software, which may ultimately lead to overall performance degradation despite memory latency improvements.

ACKNOWLEDGMENT

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

REFERENCES

[1] 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.

[2] T. Augustine, J. Sarma, L.-N. Pouchet, and G. Rodríguez, "Generating piecewise-regular code from irregular structures," in *Proc. 40th ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Jun. 2019, pp. 625–639.

[3] A. Azad and A. Buluc, "A work-efficient parallel sparse matrix-sparse vector multiplication algorithm," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2017, pp. 688–697.

[4] 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)*, Sep. 2017, pp. 1–6.

[5] 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*, Oct. 2018, pp. 1–8.

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

[7] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, pp. 1–25, Dec. 2011.

[8] B. K. Daya, C.-H.-O. Chen, S. Subramanian, W.-C. Kwon, S. Park, T. Krishna, J. Holt, A. P. Chandrakasan, and L.-S. Peh, "SCORPIO: A 36-core research chip demonstrating snoopy coherence on a scalable mesh NoC with in-network ordering," in *Proc. ACM/IEEE 41st Int. Symp. Comput. Archit. (ISCA)*, Jun. 2014, pp. 25–36.

[9] A. Farshin, A. Roozbeh, G. Q. Maguire, and D. Kosti, "Make the most out of last level cache in Intel processors," in *Proc. 14th EuroSys Conf.*, Mar. 2019, pp. 1–17.

[10] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi, "Cuckoo directory: A scalable directory for many-core systems," in *Proc. IEEE 17th Int. Symp. High Perform. Comput. Archit.*, Feb. 2011, pp. 169–180.

[11] 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. [Online]. Available: https://researchspace.auckland.ac.nz/handle/2292/11593?show=full and http://hdl.handle.net/2292/11593

[12] 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 Autom. Conf.*, Jun. 2019, pp. 1–6.

[13] M. Horro, G. Rodríguez, and J. Touriño, "Simulating the network activity of modern manycores," *IEEE Access*, vol. 7, pp. 81195–81210, 2019.

[14] *Intel Xeon Phi Processor Performance Monitoring Reference Manual—Volume 2: Events*, Intel, Mountain View, CA, USA, 2017.

[15] *Intel Xeon Phi Processor Performance Monitoring Reference Manual—Volume 1: Registers (Rev. 002)*, Intel, Mountain View, CA, USA, 2017.

[16] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Systematic reverse engineering of cache slice selection in Intel processors," in *Proc. Euromicro Conf. Digit. Syst. Design*, Aug. 2015, pp. 629–636.

[17] 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)*, May 2017, pp. 234–243.

[18] J. Jeffers, J. Reinders, and A. Sodani, *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Burlington, MA, USA: Morgan-Kauffman, 2016.

[19] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread cluster memory scheduling: Exploiting differences in memory access behavior," in *Proc. 43rd Annu. IEEE/ACM Int. Symp. Microarchitecture*, Dec. 2010, pp. 65–76.

[20] H. Krawczyk, "LFSR-based hashing and authentication," in *Advances in Cryptology*, Y. G. Desmedt, Ed. Berlin, Germany: Springer, 1994, pp. 129–139.

[21] 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.*, Jun. 2015, pp. 1–6.

[22] Q. Lu, C. Alias, U. Bondhugula, T. Henretty, S. Krishnamoorthy, J. Ramanujam, A. Rountev, P. Sadayappan, Y. Chen, H. Lin, and T.-F. Ngai, "Data layout transformation for enhancing data locality on NUCA chip multiprocessors," in *Proc. 18th Int. Conf. Parallel Archit. Compilation Techn.*, Sep. 2009, pp. 348–357.

[23] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse engineering Intel last-level cache complex addressing using performance counters," in *Proc. 18th Int. Symp. Res. Attacks, Intrusions, Defenses*, 2015, pp. 48–65.

[24] J. D. McCalpin, "Address Hashing in Intel KNL and SKX Processors," in *Proc. Intel eXtreme Perform. Users Group Annual Fall Conf.*, 2018. [Online]. Available: https://www.ixpug.org/events/ixpug-fallconf-2018

[25] J. D. McCalpin, "HPL and DGEMM performance variability on the xeon platinum 8160 processor," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2018, pp. 225–237.

[26] J. D. McCalpin, "Observations on core numbering and 'Core ID's' in Intel processors," Texas Adv. Comput. Center, Univ. Texas at Austin, Austin, TX, USA, Tech. Rep. TR-2020-01, 2020.

[27] J. D. McCalpin, "Mapping core and L_3 slice numbering to die locations in Intel Xeon scalable processors," Texas Adv. Comput. Center, Univ. Texas at Austin, Austin, TX, USA, Tech. Rep. TR-2021-01, 2021.

[28] 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.*, Mar. 2018, pp. 184–198.

[29] K. O'Leary, I. Gazizov, A. Shinsel, R. Belenov, Z. Matveev, and D. Petunin. (2017). *Intel Advisor Roofline Analysis*. Accessed: Jul. 28, 2020. [Online]. Available: https://www.codeproject.com/articles/1169323/intel-advisor-roofline-analysis

[30] 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)*, May 2017, pp. 297–306.
[31] G Rodríguez, M. T. Kandemir, and J. Touriño, "Affine modeling of program traces," *IEEE Trans. Comput.*, vol. 68, no. 2, pp. 294–300, Feb. 2019.
[32] 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)*, Sep. 2015, pp. 47–54.
[33] A. Scolari, D. B. Bartolini, and M. D. Santambrogio, "A software cache partitioning system for hash-based caches," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 4, pp. 1–24, Dec. 2016.
[34] A. Sodani, "Knights landing (KNL): 2nd generation Intel Xeon phi processor," in *Proc. IEEE Hot Chips 27 Symp. (HCS)*, Aug. 2015, pp. 1–24.
[35] 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 Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2018, pp. 34–36.
[36] K. Viswanathan. (2014).*Disclosure of Hardware Prefetcher Control on Some Intel Processors*. Accessed: Aug. 2020. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors.html
[37] S. Wang, C. Li, H. Hoffmann, S. Lu, W. Sentosa, and A. I. Kistijantoro, "Understanding and auto-adjusting performance-sensitive configurations," in *Proc. 33rd Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, Mar. 2018, pp. 154–168.
[38] Y. Xiao, Y. Xue, S. Nazarian, and P. Bogdan, "A load balancing inspired optimization framework for exascale multicore systems: A complex networks approach," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design (ICCAD)*, Nov. 2017, pp. 217–224.
[39] K. Yotov, K. Pingali, and P. Stodghill, "Automatic measurement of memory hierarchy parameters," in *Proc. ACM SIGMETRICS Int. Conf. Meas. Modeling Comput. Syst.*, 2005, pp. 181–192.

STEVE KOMMRUSCH is currently pursuing the Ph.D. degree with the Department of Computer Science, Colorado State University (CSU). His research interests include machine learning techniques for syntactic understanding and generation of computer code, AI safety, and verifiable ways to use machine learning. Prior to beginning his computer science Ph.D. work, he worked in the field of computer hardware design attaining the level of Fellow Design Engineer with Advanced Micro Devices. He has 31 patents granted in the fields of computer graphics, system-on-chip design, silicon debugging, low-power processors, and asynchronous clocking techniques.

MARCOS HORRO received the B.S. and M.S. degrees in computer science from the Universidade da Coruña, Spain, in 2016 and 2018, respectively. He is currently pursuing the Ph.D. degree with the Department of Computer Engineering, Universidade da Coruña. His main research interests include in the area of HPC, computer architecture focused on the performance evaluation, and optimization of heterogeneous memory systems and compilers.

LOUIS-NOËL POUCHET is currently an Associate Professor of computer science with Colorado State University, Fort Collins, CO, USA, with a joint appointment with the Electrical and Computer Engineering Department. He is also working on pattern-specific languages and compilers for scientific computing, and has designed numerous approaches using optimizing compilation to effectively map applications to CPUs, GPUs, FPGAs, and System-on-Chips. His work spans a variety of domains including compiler optimization design especially in the polyhedral compilation framework, high-level synthesis for FPGAs and SoCs, and distributed computing. He is the author of the PolyOpt and PoCC compilers, and of the PolyBench benchmarking suite.

GABRIEL RODRÍGUEZ is currently an Associate Professor with the Department of Computer Engineering, University of A Coruña, where he is a member of the Computer Architecture Group and a CITIC Research. His main research interests include in the field of optimizing compilers, architectural support for high-performance computing, and power-aware computing.

JUAN TOURIÑO (Senior Member, IEEE) is currently a Full Professor with the Department of Computer Engineering, University of A Coruña, where he also leads the Computer Architecture Group. He has extensively published in the area of High Performance Computing (HPC): programming languages and compilers for HPC, high-performance architectures and networks, parallel algorithms, and applications. He is a coauthor of more than 170 papers on these topics in international conferences and journals. He has also served in the Program Committee of 70 international conferences.