Reducing Cache Coherence Traffic with a NUMA-Aware Runtime Approach

Paul Caheny, Lluc Alvarez, Said Derradji, Mateo Valero, *Fellow, IEEE,* Miquel Moretó, Marc Casas

Abstract—Cache Coherent NUMA (ccNUMA) architectures are a widespread paradigm due to the benefits they provide for scaling core count and memory capacity. Also, the flat memory address space they offer considerably improves programmability. However, ccNUMA architectures require sophisticated and expensive cache coherence protocols to enforce correctness during parallel executions, which trigger a significant amount of on- and off-chip traffic in the system. This paper analyses how coherence traffic may be best constrained in a large, real ccNUMA platform comprising 288 cores through the use of a joint hardware/software approach. For several benchmarks, we study coherence traffic in detail under the influence of an added hierarchical cache layer in the directory protocol combined with runtime managed NUMA-aware scheduling and data allocation techniques to make most efficient use of the added hardware. The effectiveness of this joint approach is demonstrated by speedups of 3.14x to 9.97x and coherence traffic reductions of up to 99% in comparison to NUMA-oblivious scheduling and data allocation.

Index Terms—Cache Coherence, NUMA, Task-based programming models

1 Introduction

The ccNUMA approach to memory system architecture has become a ubiquitous choice in the design-space of symmetric multiprocessor (SMP) systems of all sizes. ccNUMA architectures deliver clear benefits in the memory hierarchy such as increased capacity, bandwidth and parallelism. They realise these benefits by physically distributing the cache and memory subsystem while still offering the easily programmable abstraction of flat, shared memory to the user. As the cache and memory are both shared and distributed, memory accesses require a transaction based coherence protocol to ensure correctness. Depending on the state of the accessed cache line in the system, such coherence transactions may be costly in terms of number of coherence messages required and latency involved. This coherence traffic travelling through on- and off-chip networks within SMP architectures is responsible for a significant proportion of the system energy consumption [24]. Judicious management of data locality (either performed by the runtime system, or the application itself) is therefore crucial for both energy efficiency and performance.

The most common way to program shared memory SMP systems are thread-based programming models like OpenMP [23] or Pthreads [5], which provide basic mechanisms to handle NUMA architectures. The OpenMP 4.0 standard supports tasking and data dependencies. These two features provide the opportunity for a runtime system to automatically handle data locality in a NUMA-aware fashion. This opportunity arises from the runtime's knowledge of the system's NUMA topology, the specification of the data each task requires in the programming model and tracking where the data is allocated within the NUMA regions of the system [10], [28]. Such an approach makes data motion a first class element of the programming model, allowing the runtime system to optimise for energy efficiency and performance.

The real impact of NUMA-aware work scheduling mechanisms on the cache coherence traffic that occurs within SMP architectures is not well understood as it may be masked by other factors. For example, to effectively deploy a NUMA-aware work scheduling mechanism over an SMP NUMA system two stipulations are required: **(i)** Data must be appropriately distributed amongst all the NUMA regions the system is composed of, and, **(ii)** work must be scheduled where its requisite data resides. Then, it is not clear what proportion of the observed benefits of a NUMA-aware work scheduling mechanism are due to stipulation (i) or (ii). In this work we distinguish between the effects of stipulation (i) and (ii) on both cache coherence traffic and performance.

We directly, and in detail, characterise cache coherence traffic in a real system with workloads relevant to high performance and data centric computing, relate this traffic to performance and assess the effectiveness of combining runtime managed scheduling and data allocation techniques with hardware approaches designed to minimise such traffic. We use a large SMP architecture, a Bull bullion S server platform, to make our analysis. The bullion S platform utilises a sophisticated ccNUMA architecture composed of sets of 2 sockets grouped into entities called modules. The Bull Coherence Switch (BCS), a proprietary ASIC, manages the inter-module interface and enables scaling up to a maximum configuration of 8 modules (288 cores among 16 sockets of Intel Xeon CPUs) in a single SMP system. The BCS achieves this by providing an extra module level layer in the directory architecture managing coherence among the L3s in the system. The in-memory directory information stored as normal by the Intel architecture tracks directory information for cache lines shared within a module on a per-socket granularity. In contrast the directory information the BCS stores

- *P. Caheny, L. Alvarez, M. Valero, M. Moretó and M. Casas are with the Barcelona Supercomputing Centre (BSC) and the Departament d'Arquitectura de Computadors at the Universitat Politècnica de Catalunya (UPC), Barcelona 08034, Spain. E-mail: {firstname}.{lastname}@bsc.es*
- *Said Derradji is with Bull/Atos Group. E-mail: said.derradji@atos.net*

Manuscript received Month Day, 2017; revised Month Day, 201X.

DOI: 10.1109/TPDS.2017.2787123

about cache lines which are exported inter-module is on the less granular per-module basis. This directory information allows the BCS to filter coherence traffic from the system and thus enable scaling to larger coherent systems.

Our work uses the measurement capabilities provided by the BCS to perform a direct, fine grain analysis of the coherence traffic within the system. To the best of our knowledge, our previous work [6] contained the first study on how a hierarchical directory approach [19] to scaling cache coherence interacts with a runtime managed strategy to promote data locality in an SMP. In this work, we build on our previous work to now look at the effect of NUMA-aware work scheduling and data allocation on cache coherence traffic in a larger NUMA system which, due to its size, can easily become bottlenecked by coherence traffic.

Specifically, these are the main contributions of this paper:

- A complete performance analysis of a large SMP architecture comprised of 16 sockets arranged in 8 modules totalling 288 cores. We consider five important scientific codes and three regimes of work scheduling and memory allocation: (i) Default (NUMA-oblivious) scheduling and first touch allocation. (ii) Default (NUMA-oblivious) scheduling and interleaved allocation which uniformly interleaves memory among NUMA regions at page granularity. (iii) NUMA-aware runtime managed scheduling and allocation. We see performance improvements up to 9.97x among the benchmarks when utilising the NUMA-aware regime.
- For the three regimes, a detailed measurement of the coherence traffic within the SMP system, broken down into data traffic versus control traffic. We further decompose these traffic types into message classes e.g. data delivered to cache, write backs from cache and the different request and response classes in the control traffic. We see reductions in coherence traffic up to 99% among the benchmarks when utilising the NUMA-aware regime.
- For each benchmark and regime of work scheduling and data allocation we analyse how uniformly the executions utilise the physically distributed resources in the system, decoupling the factors underlying three distinct modes of behaviour: (i) Poorly performing, non-uniform utilisation of system resources, (ii) Uniform, but energy inefficient utilisation of system resources with limited performance scaling, (iii) Uniform and energy efficient utilisation of system resources with best performance scaling.

This paper is organised as follows: Section 2 details the types and classes of coherence traffic in our analysis. Section 3 describes the architecture of the large SMP system used to make our analysis. Section 4 introduces the programming model and runtime system that supports the three different regimes of work scheduling and data allocation. Section 5 details the benchmarks used. Section 6 presents the results of our analysis of the performance and coherence traffic. Section 7 discusses related work. Lastly, we conclude the paper in Section 8.

2 Cache Coherence Traffic

While the logically flat view of memory an SMP offers the user considerably eases the programming burden, it requires a sophisticated mechanism to enforce coherence between the physically distributed caches in the system.

In order to analyse this mechanism we categorise the coherence traffic it triggers within the SMP into two types, each consisting of different classes of messages. The first type, Data messages, contain user data (cache lines) while the second type, Control messages, are messages that signal activities in the coherence protocol and do not contain user data. For example, a message transferring a cache line from a memory controller to a cache (or vice versa) belongs to the Data traffic type while asking a certain cache for the status of a cache line or requesting data in a certain state from a memory controller are of the Control traffic type.

To understand in greater detail in what nature the software (work scheduling & data allocation regime) techniques and the BCS affect the traffic we further break down the two types of traffic into message classes. An outline of message classes and their role in the coherence protocol follows:

Data Messages: Messages that carry a single cache line payload. If the receiver is a cache the message is of the *DTC (Data To Cache)* class, the sender of such messages may be a memory controller or another cache. If a memory controller is the receiver of a Data message, the message falls into the *DWB (Data Write Back)* class, the sender of a DWB message is always a cache.

Control Messages: Messages that carry protocol signalling messages without a data payload. Request messages from a cache to a memory controller belong to the *HREQ (Home Request)* class. For example such a request could be the cache asking the memory controller for access to a cache line in a certain state. Depending on the existing state of the cache line in the requesting cache and the state the cache line is requested in, a *HREQ* may be reciprocated by a *DTC* message. *SNP (Snoop)* messages are requests from a memory controller to a cache asking it to perform some action, for example to invalidate a cache line or forward it to another cache. Depending on the exact nature of a SNP message, it may be reciprocated by a *HRSP (Home Response)* message. A HRSP message is a confirmation sent from a cache to a memory controller that the action requested by the SNP is completed. An *NDR (Non-Data Response)* message is sent from a memory controller to a cache to signal the completion of a coherence transaction, where the memory controller did not need to deliver data to the cache. This could be because, for example, the data was delivered indirectly from another cache to the requesting cache or the requesting cache already had the data but requested to change the state of the cache line.

These classes, categorised as Data or Control traffic (see Table 1), cover all possible traffic types at the LLC (Last Level Cache) to memory controller interface within the SMP and provide an insightful basis upon which to analyse the effectiveness of the three work scheduling and data allocation regimes and the BCS on the coherence traffic.

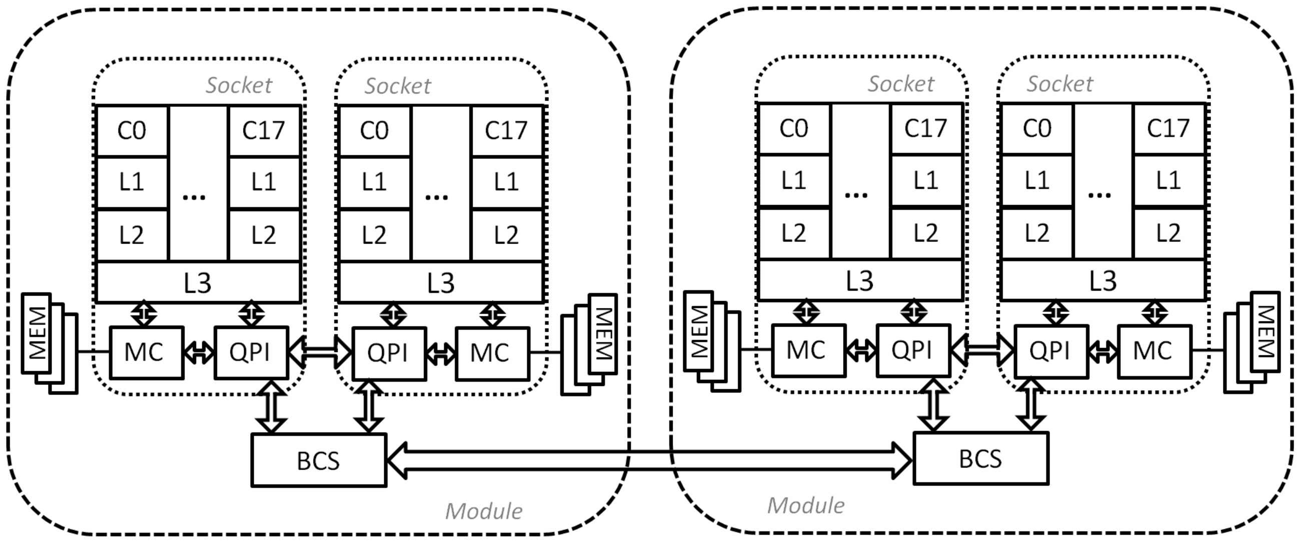


Fig. 1: Logical view of a dual module bullion S system [2]

TABLE 1: Coherence protocol message classes

Type	Class	Description
Data	DWB	Data Write Back, message from cache to mem. controller with cache line payload
Data	DTC	Data To Cache, message towards cache with cache line payload
Control	HREQ	Home Request, cache requesting cache line from mem. controller
Control	SNP	Snoop, mem. controller requesting state of, or invalidating, cache line
Control	HRSP	Home Response, cache providing state of cache line to mem. controller
Control	NDR	Non-Data Response, mem. controller signals completion without data

3 BULLION S ARCHITECTURE

Figure 1 shows a dual module bullion S system. Each module comprises two Intel Xeon sockets and their local memory connected to a single Bull proprietary ASIC, the BCS. In this work we use the largest possible configuration of the bullion S system, comprising eight modules where each module contains 2 sockets with 18 cores each, resulting in a total of 288 cores sharing memory. Our previous work [6] was performed on a much smaller system made up of 2 modules, each containing 2 sockets with 15 cores each for a total of 60 cores.

3.1 Cache Coherence in the bullion S System

The BCS is the glue for connecting multiple modules into a single SMP system. Cache coherence traffic statistics are collected from the BCS during benchmark execution. As all coherence traffic is measured within the BCS, the results we present include only coherence traffic travelling via the BCS (see Figure 1) in each module. Traffic travelling between the two CPU sockets within a single module does not travel via a BCS and is therefore not included. Measurements are recorded at each BCS in the system for both traffic incoming to the BCS (from its two local CPU sockets) and traffic outgoing from the BCS (towards its two local CPU sockets). We term this incoming or outgoing nature of the traffic its directionality. Henceforth, all references to cache refer to the LLC (labelled L3 in Figure 1) of a processor unless otherwise indicated and the coherence traffic observed represents only coherence transactions at the interface of the LLC cache and the system memory (via a BCS). The coherence agents for the system memory in Figure 1 are the memory controllers (labelled MC).

The BCS is an actor in the cache coherence protocol of the system rather than simply a routing point for inter-module messages. The BCS stores module level directory information about cache lines which have been exported from a memory in its local NUMA regions to caches in other modules. This enables the BCS to respond in place of an LLC cache or memory controller in certain inter-module cache transactions, reducing the coherence traffic required in the system. For example, for SNP and HRSP messages the BCS may filter messages from the system, where it can participate in the coherence transaction in place of a CPU. Therefore, SNP messages may appear as incoming to a BCS in one module without appearing as outgoing in any other module and vice versa for the HRSP messages. In the process of maintaining its own directory information the BCS may also initiate SNP messages to other modules, so CPUs may see SNP messages which originate at a BCS and not at any other processor in the system. Therefore, SNP messages may appear as outgoing from a BCS without having appeared as incoming to any other BCS in the system. When NDR messages are sent by a CPU they may be piggybacked on unused bits in the Data message classes as a bandwidth optimisation. Conversely, the BCS aims to optimise for latency by not piggybacking NDR messages on Data message classes. Also, when signalling the writeback or forwarding of modified data inter-module, incoming HRSP messages in a source module may need to be conveyed as outgoing HREQ message in the destination module. In light of these actions the BCS performs in the coherence protocol, there may be significant asymmetry between the incoming and outgoing traffic levels for the HREQ, SNP, HRSP and NDR message types.

By analysing the coherence traffic in the message types and classes defined in Table 1 under the different memory allocation and scheduling regimes, it is possible to provide a detailed characterisation of the effect of the different regimes and the BCS on the coherence traffic.

3.2 System Environment & Characteristics

In this work we use an eight module installation of the bullion S platform, running RHEL 6.5 with a Linux kernel version of 2.6.32. Each module (see Figure 1) is composed of two sockets of 18 core Intel *Haswell EX E7 8890 v3* processors. Each core has private 32KB L1 data and instruction caches and a private 256KB L2 unified cache. Each socket has a 45MB shared inclusive LLC and a local NUMA region comprising 512 GB of system memory. These specifications result in a total of 288 cores in 16 distinct NUMA regions sharing 8 TB of DRAM.

Information regarding the NUMA topology of a system is typically available from the firmware via the OS. The `numactl --hardware` command may be used to display the information the OS provides to the runtime regarding NUMA distances. As denoted in Table 2, the system has three levels of NUMA distance. A coherence message may travel within the local NUMA region, to the single *near* remote NUMA region, i.e. the other NUMA region in the local module, or to a *far* NUMA region in any of the remote modules. Table 2 shows the three classes of NUMA distance in the system. Besides the NUMA distances provided by the firmware we measure the real latencies and memory bandwidths available across the different NUMA distances in the system. We use Intel's Memory Latency Checker (MLC) [29] to measure the latencies and the STREAM benchmark [20] to measure the memory bandwidths. The average latencies in the system (Table 2) follow the same pattern and similar ratios to the NUMA distances in Table 2. On average there is a 42% latency penalty in accessing memory in the neighbouring NUMA region in the same module in comparison to accessing memory in the local NUMA region. There is a further latency penalty of 133% to access data in far remote NUMA regions, i.e. any remote module (or a 232% penalty for inter-module access compared to local NUMA region access).

Table 2 also shows the average memory bandwidths measured in the system for the STREAM Triad benchmark. These results use all the threads available on a single socket (18) to saturate the bandwidth to the memory of a given NUMA region. On average these figures show there is a 39% drop in bandwidth for accesses to a socket's near remote NUMA region. Accessing a far NUMA region incurs an 82% lower available bandwidth than accessing the near NUMA region. Comparing local and far accesses the bandwidth penalty is 89%.

TABLE 2: NUMA distances and their average Bandwidths and Latencies

Type	Local	Near	Far
NUMA Distance	10	15	40
Average Bandwidth (GB/s)	41.3	25	4.6
Min/Max Bandwidth (GB/s)	41.1/41.6	24.5/25.1	4.4/4.8
Latency (Avg. ns)	125.6	178	416
Min/Max Latency (ns)	124/127	177/180	410/428

4 Memory Allocation and Scheduling

In order to minimise the amount of coherence traffic required in the system for a given problem, we use a task-based data-flow programming model. Such a programming model is supported in OpenMP by new features introduced in the OpenMP 4.0 release. In task-based data-flow programming models the execution of a parallel program is structured as a set of tasks and an execution ordering among them based on data-flow constraints. The programmer identifies tasks by annotating serial code with directives. Data-flow is represented by clauses in the directives which specify what data is used by a task (called input dependencies) and produced by each task (called output dependencies). The runtime manages parallel execution of the tasks, relieving the programmer from explicitly synchronising and scheduling tasks and thus promoting programmability.

We use the OmpSs [11] task-based data-flow programming model and its associated Nanos++ runtime system to experiment with a diverse set of memory allocation and scheduling regimes. The OmpSs programming model supports task constructs in a very similar way to OpenMP 4.0. The task-based data-flow programming model supported by both OpenMP 4.0 and OmpSs provide the potential to implement NUMA-aware scheduling in the runtime system. The Nanos++ runtime system already supports NUMA-aware scheduling [4] in the release we use, version 0.10.3.

The default (NUMA-oblivious) OmpSs runtime scheduler maintains one global queue of ready tasks for the entire SMP system. Tasks are scheduled among cores without considering where their data dependencies reside. In contrast, the NUMA-aware OmpSs scheduler maintains one ready queue per NUMA region within the SMP system. Tasks are enqueued in the NUMA region in which the largest proportion of their data dependencies reside. The runtime system must already store meta information (address, whether the dependency is input/output/inout etc.) about all data dependencies in order to correctly synchronise the execution of tasks. In the NUMA aware scheduler the runtime additionally stores the location of each data dependency within the NUMA topology when the data is first tied to a physical memory location. When scheduling subsequent tasks the runtime system examines the data dependencies of each task to calculate which NUMA region contains the largest proportion of each task's data dependencies. Each task is then added to the ready queue of the NUMA region which has the largest amount of data required by the task.

If an execution suffers from load imbalance the Nanos++ runtime system overcomes this via NUMA aware work stealing [1]. Should a worker thread lay idle for a certain period of time the worker will steal work from another NUMA region. In order to minimise the NUMA cost associated with this work stealing, the worker will only steal from its nearest neighbouring NUMA region and not from any more remote regions. In the case of the system we use, this means that during work stealing a worker thread will only steal work from the other socket in the same module, and will not steal work from remote modules (and thus not impact the inter-module coherence traffic we measure at the BCS).

We use three regimes of task scheduling and memory allocation, described below, to analyse the impact of NUMA-aware scheduling on the coherence traffic classes defined in Section 2.

4.1 Scheduling & Memory Allocation Regimes

The OmpSs features described above allow us to define several execution regimes of task scheduling and memory allocation:

__D__efault scheduling & __F__irst __T__ouch allocation (__DFT__): Tasks are scheduled in a NUMA-oblivious fashion among all the utilised cores in the system, ensuring load is balanced. Data is allocated in the NUMA region of the core where the data is first touched, via the Linux first touch memory allocation policy. In a worst case scenario, all data may be allocated in a single NUMA region, leaving others entirely unused. Tasks using the allocated data are scheduled among all the utilised cores systemwide without any consideration for where their required data resides.

__D__efault scheduling & __I__nterleaved allocation (__DI__): Tasks are scheduled in a NUMA-oblivious fashion, as in DFT, ensuring load is balanced. Nevertheless, data allocation is uniformly distributed among all the NUMA regions in the system. This is achieved with the Linux NUMA interleaved memory allocation policy which distributes allocated memory among all NUMA regions at a page granularity, regardless of what core first touches the data. Data is guaranteed to be uniformly distributed among all NUMA regions in the system (at page granularity). However, tasks using allocated data run without any consideration for where their data dependencies reside.

NUMA-__A__ware scheduling & __F__irst __T__ouch allocation (__NAFT__): Tasks are scheduled in a NUMA-aware fashion. The application's memory allocating code is encapsulated in tasks by the programmer. The runtime system automatically [1] recognises tasks which first touch data (the first tasks specifying the data an an output) and schedules them in a uniformly distributed arrangement among the NUMA regions. Each task allocates all its data locally in the NUMA region it runs in due to the use of the Linux first touch memory allocation policy. Memory is therefore guaranteed to be uniformly distributed among all NUMA regions in the system (at a per first touching task granularity, determined by the programmer). Tasks using the allocated data are scheduled on cores in the NUMA region where the majority of their data dependencies reside.

A regime utilising NUMA-aware scheduling & interleaved allocation is not a valid combination as it would lead to conflicting responsibilities for distributing data allocation at both the OS and runtime system level. This would render the NUMA-aware scheduling ineffective.

5 BENCHMARKS

We chose five benchmarks for use in our evaluations, representative of important problems in both high performance and data centric computing, with significantly different data access patterns. Streamcluster is from the PARSEC benchmark suite [3], we also use benchmarks based on the Cholesky Decomposition [1], Symmetric Matrix Inversion [1], the Jacobi Method, and the computation of Integral Histograms. All benchmarks exhibit a complex data access pattern comprising both reads and writes, except Streamcluster which uses a streaming read data access pattern.

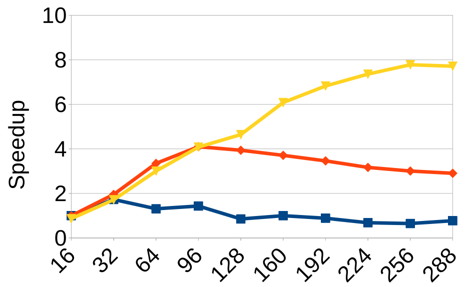
Cholesky uses a tile-based algorithm for the Cholesky factorisation problem in linear algebra which exposes fine grained parallelism and is implemented in the task-based OmpSs programming model.

Symmetric Matrix Inversion (SMI) is a larger linear algebra problem that inverts a symmetric matrix in three stages. It also follows a tile-based algorithm thus exposing fine grained parallelism. It is implemented in the task-based OmpSs programming model. In Both Cholesky and SMI We use a matrix sufficiently large (59136 rows) that the benchmark's performance does not benefit from further increasing the problem size. We use a tile size of 768 in both the Cholesky and SMI benchmarks.

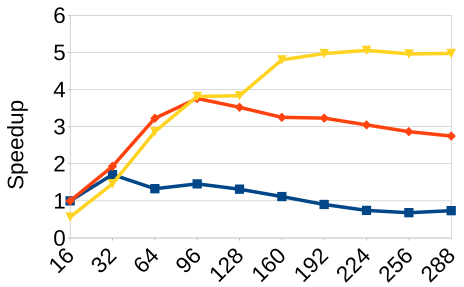
PARSEC Streamcluster is based on the online clustering problem and is part of the PARSEC benchmark suite [3]. This problem organises large volumes of continuously produced streaming data in real-time with applications in areas such as network intrusion detection, data mining and pattern recognition. The benchmark is dominated by a streaming read data access pattern and is adapted to a task-based implementation in the OmpSs programming model [11]. In order to aid scalability to such a large SMP system size as used in this work we employed a customised input set, larger than the largest ("native") input set provided by the PARSEC benchmark documentation. The custom input set parameters we use in this work are: Min. number of centres allowed: 10, Max. number of centres allowed: 20, Dimension of each data point: 128, Number of data points: 7.2 million, Number of data points to handle per step: 1.44 million, Maximum number of intermediate centres: 5000.

Jacobi is an implementation of the Jacobi algorithm for solving a system of linear equations. It is implemented using a two dimensional Jacobi five-point stencil computation. The implementation synchronises at the end of each iterative step towards convergence, however there are no synchronisation requirements among the tasks that a single iterative step is decomposed into. The algorithm decomposes each iteration's two dimensional data into blocks by row. The input problem size used in our results is 6400 blocks of double precision floating point values, each block having dimensions of 800 x 20000.

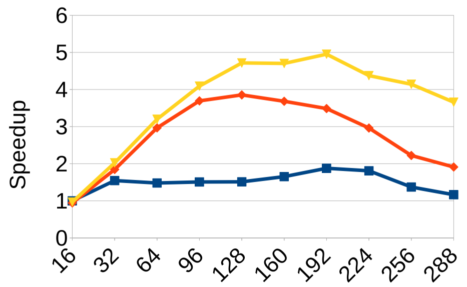
Integral Histogram computes a cumulative histogram for each pixel of an image. With this measure one can find the histogram of a Cartesian region of pixels in constant time. The histogram is used by several techniques in digital



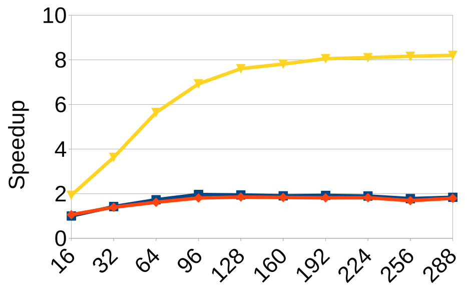
(a) Cholesky



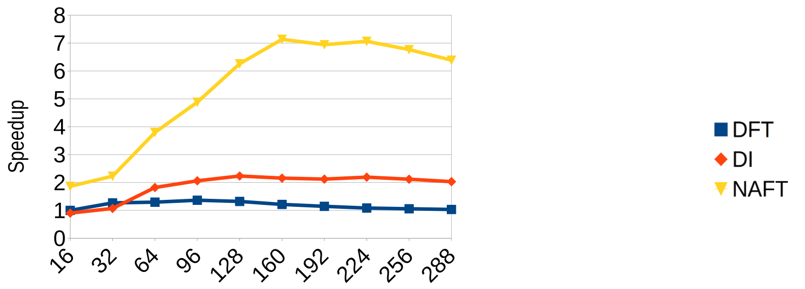
(b) SMI



(c) Streamcluster



(d) Jacobi



(e) Integral Histogram

Fig. 2: Speedup DFT, DI, NAFT regimes under increasing thread count. Threads distributed uniformly among all available NUMA regions. Figures normalised to DFT with 16 threads.

image processing and computer vision. We use an image size of 65536 x 65536 pixels with block size 512 x 512 pixels to compute a histogram with 32 bins.

All benchmarks are run on core counts ranging from 16 (1 thread per NUMA region) to 288 (18 threads per NUMA region). The utilised threads are always distributed uniformly across the 16 NUMA regions in the system. All performance and coherence traffic measurements are repeated three times and the mean values reported.

6 Results and Analysis

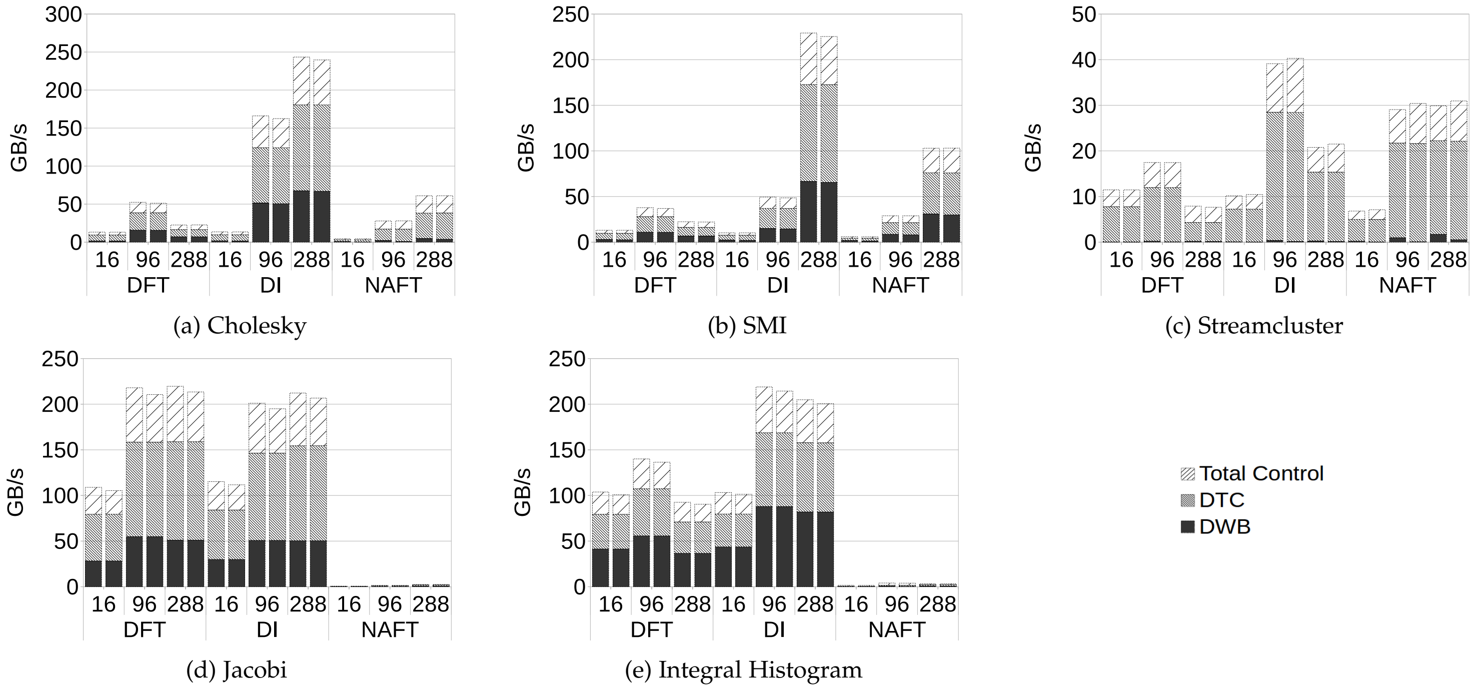
6.1 Introduction

In this section we present a detailed analysis of the coherence traffic generated by running the benchmarks presented in Section 5 on different numbers of cores on the bullion S system. We present results for the benchmarks under the three regimes detailed in Section 4.1 : DFT, DI and NAFT. First, in Section 6.2 we present a decomposition of the coherence traffic into the two Data message classes (DWB and DTC) plus aggregate Control traffic (which comprises SNP, HRSP, HREQ and NDR). Secondly, in Section 6.3 we present a detailed breakdown of the Control traffic type into its individual message classes. Thirdly, in Section 6.4 we focus on how uniformly the systemwide resources are utilised by each regime in terms of coherence traffic and how this relates to performance and energy efficiency.

For each benchmark, thread count and execution regime, we present the coherence traffic profile in two views, both measured at the BCS (see Section 3.1 for the site of measurement and precisely which coherence traffic we measure): **(1)** the bandwidth utilised by coherence traffic during benchmark execution, called *Coherence Bandwidth*, and **(2)** the total coherence traffic data moved over the entire course of the benchmark execution, called *Coherence Movement*. The coherence bandwidth and coherence movement views are related via the execution time as coherence bandwidth is the coherence movement per second of execution time. Figures 3 to 7 show the systemwide (i.e. all 8 modules aggregated) coherence traffic. In Figures 3 to 7, the traffic's directionality split (see Section 3.1) is presented as a pair of bars in the figure for each combination of thread count and regime labelled on the X axis - in all figures the left bar of the pair is the incoming traffic (from its local sockets towards the BCS) and the right bar of the pair is the outgoing traffic (from the BCS to its local sockets). The maximum achievable bandwidth through all 8 BCS in the system measured by the synthetic STREAM benchmark [20] is 275 GB/s incoming and outgoing simultaneously.

To place the coherence traffic analysis that follows in a performance context Figure 2 shows the speedup for each benchmark and regime combination at varying thread counts on the bullion S system. These speedup figures use the DFT regime at a thread count of 16 (i.e. 1 thread per socket) as the baseline (equal to 1 in Figure 2) and are discussed in Section 6.2.

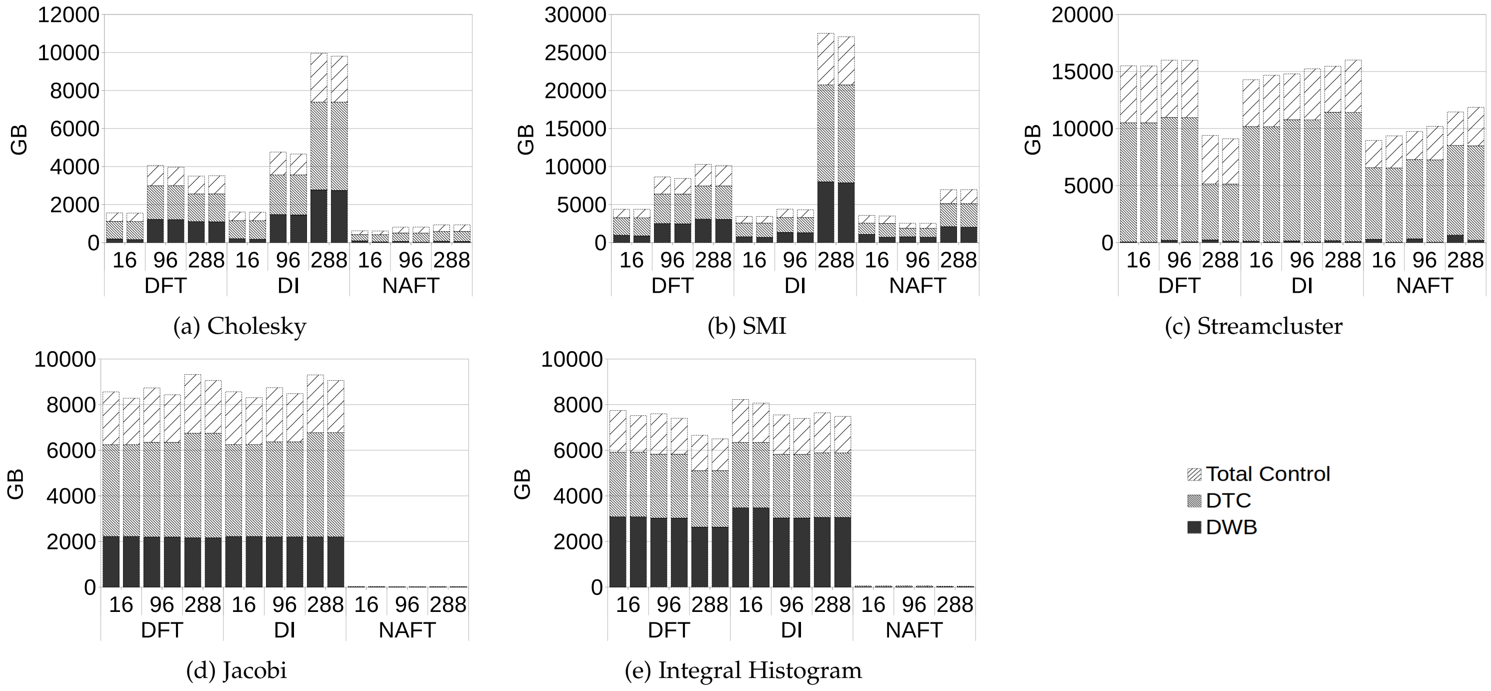
For the purpose of our results and in order to clearly distinguish between the two, we term measurements of bandwidth travelling via the BCS on the inter-module link as *coherence bandwidth* and the bandwidth to local memory within each NUMA region summed systemwide as the *memory bandwidth*. It is important to note that in a ccNUMA architecture the memory bandwidth is distributed among the NUMA regions of the system. If the number of NUMA regions in a system is R and a workload allocates memory in only K < R NUMA regions the maximum availability



(a) Cholesky (b) SMI (c) Streamcluster

(d) Jacobi (e) Integral Histogram

Fig. 3: Coherence Bandwidth: DWB, DTC and Control traffic



(a) Cholesky (b) SMI (c) Streamcluster

(d) Jacobi (e) Integral Histogram

Fig. 4: Coherence Movement: DWB, DTC and Control traffic

of bandwidth to memory will be K/R times the systemwide maximum possible. Similarly the system contains eight BCS, one in each module. Each BCS implements a module level directory with finite capacity which stores directory information for cache lines exported from that module to LLCs in remote modules. As the directory at the BCS is strictly inclusive of all cache lines exported from that module to remote LLCs, if the directory in the BCS is under capacity pressure this requires evictions of cache lines from remote LLCs in order to enable replacing an entry in the directory at the BCS. If a workload does not allocate memory among all the modules in the system, only a limited fraction of the systemwide BCS capacity resources are utilised, which may negatively impact the system in both performance and energy efficiency.

6.2 DWB, DTC and Control Coherence Traffic

Figures 3 and 4 show the measured traffic in coherence bandwidth and coherence movement views respectively, broken down into the two Data message classes, DWB and DTC, and the Control message type which comprises HREQ, SNP, HRSP and NDR.

When the two Data message classes, DWB and DTC are summed together, the systemwide level of Data traffic is symmetric in directionality. This symmetry reflects the fact that messages belonging to these classes originate at either a CPU cache or local memory within one module and travel through the inter-module interconnect (via first the local and then the remote BCS) to their respective memory or cache destination in another module. When comparing the incoming versus outgoing traffic for the two Data message classes the total aggregate DWB and DTC traffic is always symmetric in directionality.

6.2.1 Cholesky

The mix of both DWB and DTC message classes in the traffic of the Cholesky benchmark demonstrates that this benchmark requires a mixture of both read and write data accesses across modules in the system. The DWB message class represents modified cache lines being written back from the LLC of one socket to memory in the cache line's home NUMA region in a remote module, while the DTC message class represents data being delivered to a cache originating from a remote module's memory or LLC.

In terms of performance (Figure 2a) the DFT regime scales poorly and does not benefit from using any additional threads beyond 32. Between 32 and 96 threads, we see both the DI and NAFT regimes perform similarly and already significantly outperform the DFT regime. The DI regime does not continue to scale past 96 threads whereas the NAFT regime benefits from utilising up to 256 threads. As can be seen in Figure 2a we achieve a maximum speedup of 7.8x at 256 threads versus the baseline.

If we discount the poorly performing DFT regime and focus on the differences in coherence traffic between the DI and NAFT regimes we can see a significant reduction both in coherence bandwidth (Figure 3a) and coherence movement (Figure 4a) under the NAFT regime versus the DI regime.

At 96 threads both regimes are performing similarly (Figure 2a), and we see similar benefits in both coherence bandwidth and coherence movement for the NAFT regime over the DI regime: the NAFT regimes requires just 17% of both the coherence bandwidth and coherence movement required by the DI regime. Utilising all threads in the machine the NAFT regime is performing 2.7x better than the DI regime while also requiring only 25% as much coherence bandwidth and 9.4% as much coherence movement as the DI regime.

6.2.2 SMI

Like Cholesky, SMI also exhibits a significant proportion of both DWB and DTC message classes in its traffic profile. For SMI, the DFT regime scales poorly with no benefit from using more than 32 threads (Figure 2b). Between 32 and 96 threads we see both the DI and NAFT regimes performing well with a marginal edge in performance for the DI regime. However, beyond 96 threads the DI regimes performance deteriorates quickly while the NAFT regime benefits from scaling up to 192 threads. The maximum speedup we achieve is 4.9x at 192 threads versus the baseline, and performance plateaus as we use more threads out to the maximum of 288.

Again focusing on the difference between the better performing DI and NAFT regimes in both the coherence bandwidth and coherence movement views (Figures 3b and 4b respectively), at 96 threads both the NAFT and DI regimes are performing similarly (Figure 2b but the NAFT regime requires only 59% of the coherence bandwidth and 58% of the coherence movement of the DI regime. At 288 threads the NAFT regime is performing 1.8x better than the DI regime while requiring only 45% of the coherence bandwidth and 25% of the coherence movement compared to the DI regime.

6.2.3 Streamcluster

We can see from the traffic profile of the Streamcluster benchmark (Figures 3c and 4c) that the vast majority of the data traffic type is made up of the DTC coherence message class and the DWB message class represent a negligible proportion of the data traffic type. This reflects the streaming read data access pattern of the Streamcluster benchmark.

We can see from Figure 2c that the DFT regime does not scale well, never achieving a speedup beyond 1.9x the baseline. At every thread count above the baseline of 16 threads we see the DI regime outperforms the DFT regime and the NAFT regime in turn outperforms the DI regime. The DI regime performs within a small margin of the NAFT regime until 96 threads, beyond which the NAFT regime maintains a significant performance gap over the DI regime. The DI regime reaches its maximum speedup of 3.9x over the baseline at 128 threads. The NAFT regime achieves its maximum performance at 192 threads; a speedup of 5x the baseline. For both the DI and NAFT regimes performance degrades significantly as the benchmark scales to thread counts beyond the optimally performing thread count.

If we compare the coherence traffic profiles for the two best scaling regimes (DI and NAFT) we see that at 96 threads although the performance advantage for the NAFT regime over the DI regime is only 10%, the NAFT regime already requires significantly less coherence bandwidth (Figure 3c) and less coherence movement (Figure 4c), utilising only 75% and 66% respectively of the traffic levels required by the DI regime at this thread count.

Using all of the 288 threads available in the system, we see that NAFT is outperforming DI by 1.9x. At this thread count the NAFT regime consumes more coherence bandwidth than utilised by the DI regime. However because the NAFT regime's performance is so much better than the DI regime and its execution time is significantly shorter, it completes the benchmark with only 74% of the coherence movement of the DI regime.

6.2.4 Jacobi

Jacobi's traffic profile contains a significant proportion of both DWB and DTC message classes. Both the DI and DFT regimes scale very poorly for this benchmark never achieving more than a 2x speedup versus the baseline, regardless

TABLE 3: Speedup and reduction in coherence movement with 288 threads.

(a) NAFT regime in comparison to DFT regime

Benchmark	Speedup	DWB	DTC	Control	Total
Cholesky	9.97x	-94%	-65%	-63%	-73%
SMI	6.76x	-32%	-30%	-34%	-31%
Streamcluster	3.14x	137%	63%	-23%	26%
Jacobi	4.46x	-99%	-99%	-99%	-99%
IntHist	6.19x	-99%	-99%	-99%	-99%

(b) NAFT regime in comparison to DI regime

Benchmark	Speedup	DWB	DTC	Control	Total
Cholesky	2.66x	-98%	-89%	-86%	-91%
SMI	1.81x	-74%	-76%	-72%	-75%
Streamcluster	1.91x	233%	-29%	-27%	-25%
Jacobi	4.57x	-99%	-99%	-99%	-99%
IntHist	3.15x	-99%	-99%	-99%	-99%

of thread count. The NAFT regime significantly outperforms both the DFT and DI regimes at all thread counts. The NAFT regime shows a 1.9x speedup at the lowest thread count of 16 rising to a 6.9x speedup at 96 threads and a 8.2x speedup at 288 threads versus the baseline.

At all thread counts the NAFT regime achieves a huge reduction in the coherence bandwidth and coherence movement required by the benchmark. In fact, at all thread counts presented, the NAFT regime requires at most 1.1% of the coherence bandwidth and 0.3% of the coherence movement of either the DFT or DI regimes. This result shows that for this benchmark the NAFT regime achieves an extremely strong co-location of computation with its requisite data and thus causes negligible inter-module coherence traffic.

6.2.5 Integral Histogram

Integral Histogram again shows a significant amount of both DWB and DTC message classes in its traffic profile. In this benchmark the DFT regime does not scale well at all and never achieves a speedup beyond 1.4x the performance baseline. The DI regime scales only marginally better than DFT achieving a speedup of 2.2x at 128 threads. The NAFT regime scales much better, achieving a speedup of 4.9x at 96 threads and continuing to scale well to 160 threads where it reaches its highest speedup versus the baseline of 7.1x. Although not the optimal configuration, at 288 threads the speedup versus the baseline is still 6.4x.

Similarly to the Jacobi benchmark, the NAFT regime is able to achieve an extremely strong locality of data accesses with the Integral Histogram benchmark. The NAFT regime never requires more than 1.8% of the coherence bandwidth and 0.8% of the coherence movement of either the DFT or DI regimes at any thread count.

6.2.6 Summary

In all five benchmarks the DFT regime is outperformed by both the DI and NAFT regimes once more than 32 threads are used. This reflects the fact that under the DFT regime, memory is allocated in an unorchestrated fashion among the 16 NUMA regions in the system. This may result in a non-uniform distribution of the allocated memory among the NUMA regions and therefore an underutilisation of resources such as bandwidth to memory and the BCS directory capacity (as explained earlier in Section 6.1), one or both of which may bottleneck the performance. This unbalanced data distribution leads to an unbalanced computation as tasks executing in all the remote modules compete for the limited bandwidth available into the one or few modules that contain the majority of the data, thus incurring high latencies. This results in the DFT regime both performing poorly and being able to use only a fraction of the total inter-module bandwidth available across the system.

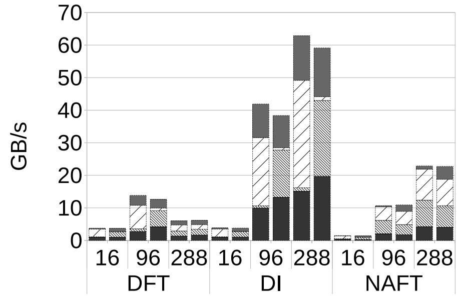
Neither the DI nor NAFT regime suffers from these issues as both these regimes guarantee that their data is allocated in a uniform manner among all NUMA regions in the system. Considering the performance and coherence traffic results we show for these two regimes we can see that the capacity of the inter-module link now becomes a limiting factor in the behaviour of these two regimes.

The maximum coherence bandwidth we see used in any of the results we present is for the SMI benchmark and DI regime at 288 threads, which uses 455 GB/s of coherence bandwidth (sum of Incoming and Outgoing traffic). This figure is close to the expected maximum capacity of the inter-module link in the system.

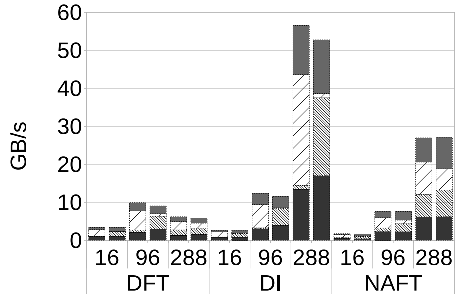
Across all five benchmarks we see a significant benefit in both performance and coherence movement for the NAFT regime over the DI regime as we scale the thread count. For Cholesky, SMI and Streamcluster (Figures 2a, 2b and 2c respectively) the NAFT regime performs similarly to the DI regime until we reach 96 threads. As can be seen from Figures 3a, 3b and 3c, the coherence bandwidth used by the DI regime for these three benchmarks at 96 threads is not yet nearing the capacity of the inter-module link. This suggests that at and below 96 threads neither the DI or NAFT regime are limited by the capacity of the inter-module link. Rather, they are limited by the relatively low (in relation to the dimensions of the inter-module and memory links) number of available computational threads in the system. As we move beyond 96 threads the much lower coherence bandwidth required by the NAFT regime enables it to outperform the DI regime. This is because the DI regime's bandwidth requirement approaches the capacity of the inter-module link at high thread counts.

In the cases of both the Jacobi and Integral Histogram benchmarks the NAFT regime outperforms both the DFT and DI regimes right from the beginning of the scaling analysis at 16 threads. We see for these two benchmarks their coherence bandwidth requirements grow more quickly under the DI regime than the other three benchmarks. As can be seen from Figures 3d and 3e, the DI regime is already using around 400 GB/s of coherence bandwidth at just 96 threads. The large amount of inter-module coherence bandwidth required by these two benchmarks at the relatively low thread count of 96 means the NAFT regime can outperform the DI regime even at low thread counts due to its significantly lower coherence bandwidth requirements.

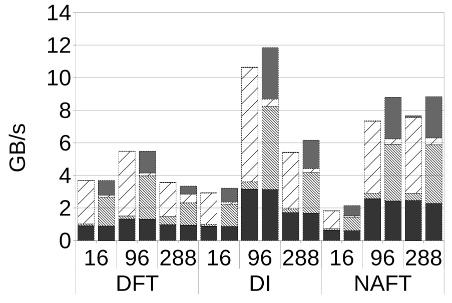
A summary of the speedups and reductions in coherence



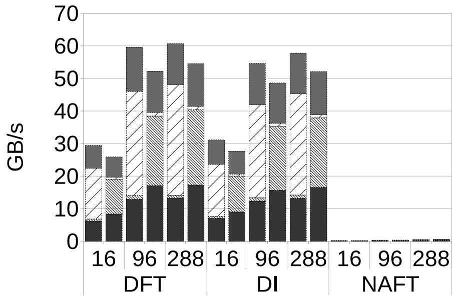
(a) Cholesky



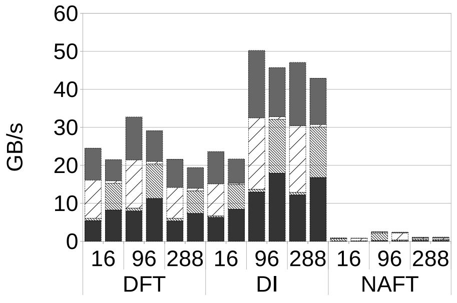
(b) SMI



(c) Streamcluster



(d) Jacobi



(e) Integral Histogram

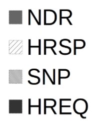
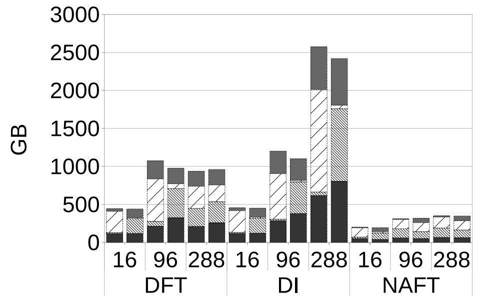
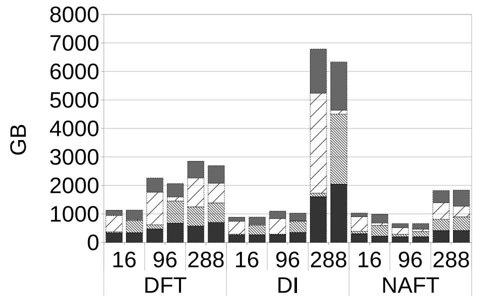


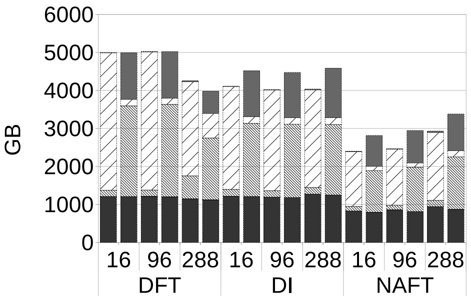
Fig. 5: Coherence Bandwidth: Control traffic by message class



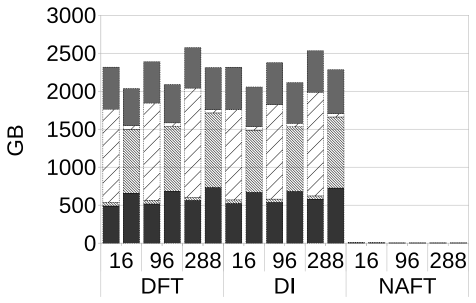
(a) Cholesky



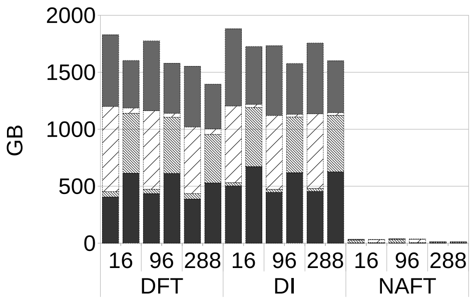
(b) SMI



(c) Streamcluster



(d) Jacobi



(e) Integral Histogram

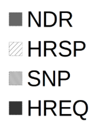


Fig. 6: Coherence Movement: Control traffic by message class

traffic is presented in Tables 3a and 3b. Table 3a shows the results for the NAFT regime in comparison to the DFT regime, while Table 3b shows the results for the NAFT regime in comparison to the DI regime.

In all cases in Tables 3a and 3b we see the NAFT regime outperforming the baseline it is compared against. Only in Streamcluster comparing NAFT to DFT do we see an increase in total coherence traffic (26%). In this case, despite the slightly increased coherence traffic, the NAFT regime is outperforming the DFT regime by 3.1x. We explore the reasons for the very poor performance of the DFT regime further in Section 6.4. Comparing the two best performing regimes (Table 3b), we see reductions in coherence movement ranging between 25% and 99%. The ability of the NAFT regime to reduce coherence traffic depends strongly on the spatial contiguity of the data dependencies of the tasks. While the NAFT regime will schedule a task local to the NUMA node containing the largest proportion of its required data, depending on the number, size and layout of the data dependencies of the task, some degree of remote NUMA accesses may be required, as is the case in Streamcluster and to a lesser extent SMI.

The large reductions in coherence traffic achieved by the NAFT regime and summarised in Tables 3a and 3b have important implications for energy efficiency which is a crucial factor in the way future computing architectures and software stacks are designed [7], [12], [15], [27].

6.3 Control Coherence Traffic Decomposition

Figures 5 and 6 show the coherence bandwidth and coherence movement for all three benchmarks for the Control coherence traffic type in isolation, now decomposed into the individual Control message classes HREQ, SNP, HRSP and NDR. In broad terms we see a similar relationship between the levels of Control traffic under the different regimes as we did in the overall traffic profile in Figures 3 and 4 (which includes the Data message types).

In the Control message classes we see asymmetry between the incoming and outgoing traffic for all four message classes. In the HREQ, SNP, HRSP and NDR Control message classes we observe significant asymmetries between incoming and outgoing traffic in certain instances, the reasons for which we already outlined in Section 3. We see very obvious asymmetries between incoming and outgoing traffic for both the SNP and HRSP message classes across all five benchmarks. In all cases the asymmetry between incoming and outgoing traffic for the SNP message class manifests itself as a much larger level of SNP messages in the incoming traffic (incoming to the BCS from its local CPU sockets) compared to the outgoing traffic (outgoing from the BCS towards its local CPU sockets). In the case of HRSP (which is a reciprocal message to SNP in a coherence transaction) we see the reverse case, that is, we see a much larger amount of HRSP traffic in the outgoing traffic compared to the incoming traffic.

These asymmetries in the SNP and HRSP traffic represent cases where the action of the BCS in the coherence protocol is having a significant impact on the SNP and reciprocal HRSP traffic. The BCS is often able to filter SNP messages it receives from its local CPUs from the system and answer the SNP message directly, without forwarding the SNP message on to any remote CPU in the system. Hence the SNP message appears as incoming at some BCS but never appears as outgoing at another BCS. We see the effect of this in our results where generally we see the level of incoming SNP traffic in the system is often much larger than the level of outgoing SNP traffic. We see the exact reverse in the HRSP traffic. In this case the BCS answers the SNP message itself with a HRSP and hence this HRSP appears as outgoing from a BCS. Because the HRSP has originated at the BCS (and not a remote CPU) it appears as an outgoing HRSP message which does not have any matching incoming HRSP message elsewhere in the system.

In Figures 5 and 6 we also see asymmetry between the incoming and outgoing traffic for the NDR message class. This is particularly pronounced across all thread counts and regimes in the Streamcluster benchmark but is also apparent in the results for the other benchmarks to some degree. This is for the reason explained in Section 3, namely that when sending NDR messages the BCS optimises for latency by always sending explicit NDR messages while the CPU optimises for bandwidth by piggybacking NDR messages on unused bits in Data messages.

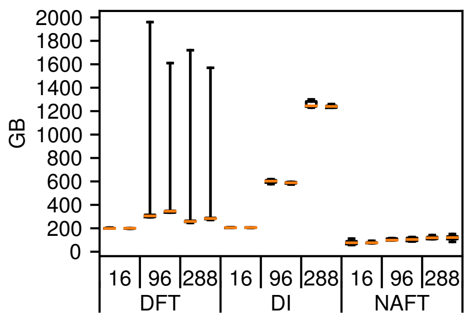
6.4 Traffic Symmetry Over Modules

Figure 7 shows the distribution of the coherence movement among the modules for each benchmark at each of the three thread counts and three regimes of data allocation and work scheduling (split into a pair of bars, left for incoming and right for outgoing traffic, as presented in the previous coherence traffic figures). Each bar in these boxplots is constructed from 8 datapoints, namely the total per-module traffic for each of the 8 modules in the system. Each bar in the boxplot features a low and high whisker for the lowest and highest traffic recorded among the 8 modules and the median value among all 8 modules marked in orange inside a box. The lower and upper edges of this box represent the first and third quartiles among the 8 modules respectively.

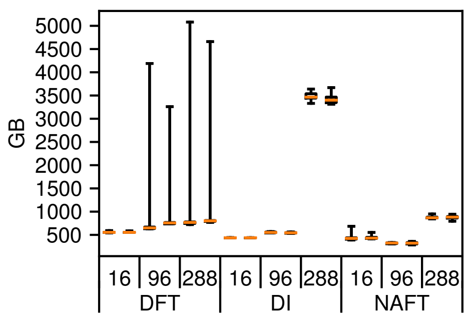
Under the DI and NAFT regime we see generally across the benchmarks that the boxplot extends through a very narrow range from the lower whisker to the upper whisker. Under these two regimes, by definition the data required by the benchmark is allocated in a uniform manner among all the NUMA regions (and therefore all the modules) in the entire system. This uniform data allocation is handled at the OS level at a page allocation granularity under the DI regime. Under the NAFT regime it is handled at a per first touching task granularity by the runtime system. These allocation mechanisms are explained in more detail in Section 5. Due to the uniform allocation of data around the NUMA regions of the system under both these regimes, we see the amount of coherence traffic per module spans a very narrow range across the 8 modules in the system, i.e. it is uniformly distributed.

All applications (except Jacobi) share a common pattern of behaviour under the DFT regime. Under DFT, we see at 96 and 288 threads there is one outlier module which has much higher traffic than the other 7 modules in the system. The 7 modules other than the one outlier remain closely clustered around a common value. This is due to the fact that under the DFT regime the data allocation of the benchmark is not orchestrated in any manner, and depends largely on the interaction between the benchmark application code, runtime system and the underlying hardware topology. Four benchmarks, Cholesky, SMI, Streamcluster and Integral Histogram, show a large range in levels of coherence traffic across the 8 modules in the system under the DFT regime. For these benchmarks a very large proportion of the entire data allocation required by the benchmark occurs within a single module, and very little data is allocated in the other 7 modules in the system. This is demonstrated by the fact that in the relevant boxplots, we see one outlier at the top whisker of the boxplot bar which represents the traffic occurring in the module where most of the data was allocated. The other 7 modules have a much lower and quite uniform level of traffic (lower whisker, 1st quartile, median and 3rd quartile values are all in a very narrow range).

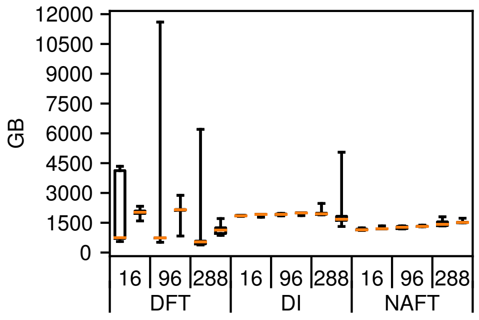
We also note that for the four benchmarks excepting Jacobi this large range is evident only under the two



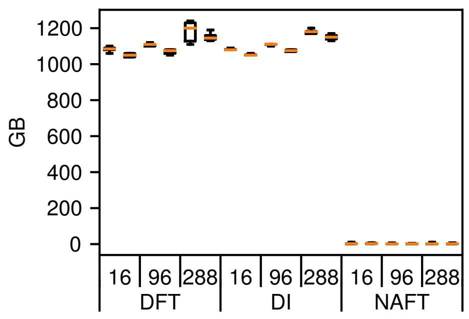
(a) Cholesky



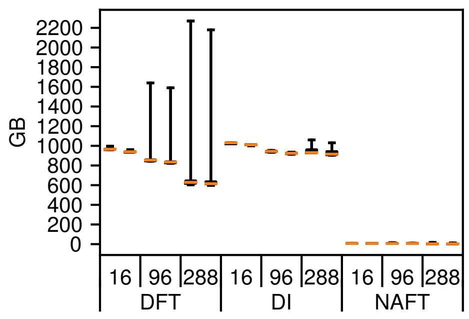
(b) SMI



(c) Streamcluster



(d) Jacobi



(e) Integral Histogram

Fig. 7: Coherence Movement distribution by module

larger thread counts of 96 and 288 threads and is not as pronounced when using 16 threads. The reason for the disparity in behaviour among the different thread counts within the DFT regime is that at 16 threads, there is only 1 thread executing in each NUMA region. In this thread configuration the relative lack of available threads causes tasks that allocate data to be forced to distribute themselves among the modules in the system in order to find an available thread on which to run. This results (by chance) in a relatively uniform distribution of data among the NUMA regions in the machine. Under the two higher thread counts of 96 threads and 288 threads, there are 6 and 18 threads per NUMA region respectively. These two configurations provide enough available threads in just one module (12 or 36 threads in these cases) in which to execute the majority of the data allocating tasks, resulting in the non-uniform distribution of data among the modules.

As outlined earlier in Section 6.1 if there is an uneven allocation of data among the NUMA regions in the system, this results in an underutilisation of resources which are distributed in the system such as capacity in the BCS directory and bandwidth to the system memory. In both these cases the total systemwide resource is distributed 1/8th in each of the 8 modules of the system. Therefore, as Figure 7 demonstrates in certain cases under the DFT regime almost the entire data allocation for the benchmark occurs within a single module. In such cases the execution is limited to using 1/8th of the systemwide bandwidth to memory and BCS capacity available. Lack of either of these two resources can very quickly limit performance as the thread count utilised across the machine is scaled up.

7 Related Work

Intel's recent Xeon CPUs use the Intel Quick Path Interconnect (QPI) [16] specification to connect Caches and Memories. The Coherence Protocol utilised by QPI is the MESIF protocol which is an extension of the well known MESI protocol [25, p. 362]. The microarchitectural details of Intel's MESIF protocol remain unpublished, however Molka and Hackenberg et al. gave insight [21] [13] into such details via sophisticated synthetic benchmarking. Molka et al's work differs from ours in that it presents aggregated total memory bandwidth and latency figures utilising synthetic benchmarks whereas we characterise the traffic and memory bandwidth utilised by real world benchmarks at the level of individual coherence protocol message types.

There has been much recent work on simplifying cache coherence systems to make them perform or scale better or be more energy efficient. Choi et al. [9] proposed restraining the shared memory programming model to enable improvements in power, performance, simplicity and verifiability in the coherence system. Manivannan et al. [17] [18] showed how the runtime system and hardware coherence substrate could co-operate to provide performance benefits by optimising particular data access and sharing patterns in task-based programming models. Hammond et al. [14] proposed changing the memory consistency model to a transactional model which allows for a less complex coherence system. All these papers utilised a simulation based approach to evaluate the impact of their designs on cache coherence whereas in our work we make a detailed and direct characterisation of the impact of a runtime managed approach to reducing coherence traffic in a real system.

Regarding NUMA-aware data distribution and scheduling Al-Omairy et al. [1] measured the performance benefits of NUMA-aware scheduling for both the Cholesky and SMI benchmarks versus the best state of the art implementations that are widely used in modern production environments. Muddukrishna et al. investigated [22] the performance impacts of NUMA-aware scheduling and data distribution for multiple real world benchmarks. Our work differs from both these as it directly and in detail quantifies the effect of NUMA-aware scheduling on coherence traffic and data motion within the system, rather than performance. Other recent work such as the Runnemede [7], SARC [26] and Runtime Aware Architecture [8] [27] proposals follow a hardware/software co-design approach to relaxing hardware provided cache coherence and move responsibility for dynamically managing disjoint memory spaces to software.

To the best of our knowledge our work is the first to study the effects of NUMA-aware scheduling and data allocation directly on cache coherence traffic in such a large, real, SMP NUMA system.

8 Conclusions

NUMA architectures continue to dominate the SMP design space and are likely to grow in prevalence and complexity alongside the trend towards higher core counts and memory capacity requirements and more functional heterogeneity sharing memory. These developments bring challenges for both computer architecture and software alike. In the architecture design space the nature of the coherence traffic required to implement the ccNUMA design is an important factor in balancing the demands of energy efficiency and performance in the design.

In this work we directly characterise the coherence traffic within a modern large SMP design at the granularity of individual classes of coherence traffic using five important benchmarks from the domains of high performance and data centric computing. We show the balance between the different types of coherence traffic (Data and Control traffic) and further break these two types down into individual message classes.

We show that a NUMA-aware regime of work scheduling and data allocation managed by the runtime system, while entirely absolving the programmer of NUMA concerns, can provide very significant benefits in terms of both performance and energy efficiency through reduced data movement. This is true whether the baseline compared against is an entirely NUMA-oblivious work scheduling and data allocation regime, or a regime which is NUMA-aware in data allocation but not in work scheduling.

Our results show that the NUMA-aware work scheduling and data allocation NAFT regime is able to reduce the coherence movement between 25% and 99% over and above the DI scheme which allocates data in a NUMA-aware fashion but does not perform NUMA-aware work scheduling. This is an important result with a view to energy efficiency and the way future ccNUMA hardware architectures and the system software stacks they run are designed. Indeed, as the trend for larger ccNUMA designs continues apace the potential for our approach to benefit energy efficiency and performance increases.

Acknowledgments

This work has been supported by the Spanish Government (Severo Ochoa grants SEV2015-0493), by the Spanish Ministry of Science and Innovation (contracts TIN2015-65316-P), by the Generalitat de Catalunya (contracts 2014-SGR-1051 and 2014-SGR-1272), by the RoMoL ERC Advanced Grant (GA 321253) and the European HiPEAC Network of Excellence. The Mont-Blanc project receives funding from the EU's H2020 Framework Programme (H2020/2014-2020) under grant agreement n^o 671697. M. Moretó has been partially supported by the Ministry of Economy and Competitiveness under Juan de la Cierva postdoctoral fellowship number JCI-2012-15047. M. Casas is supported by the Secretary for Universities and Research of the Ministry of Economy and Knowledge of the Government of Catalonia and the Cofund programme of the Marie Curie Actions of the 7th R&D Framework Programme of the European Union (Contract 2013 BP_B 00243).

References

[1] R. Al-Omairy, G. Miranda, H. Ltaief, R. Badia, X. Martorell, J. Labarta, and D. Keyes, "Dense matrix computations on NUMA architectures with distance-aware work stealing," *Supercomputing Frontiers and Innovations*, vol. 2, no. 1, 2015.

[2] B. Atos. (2015) bullion s2 factsheet. [Online]. Available: https://atos.net/wp-content/uploads/2017/06/bullion_s2_e7v3.pdf

[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proc. 17th Int. Conf. Parallel Architectures Compilation Tech. (PACT'08)*, 2008, pp. 72–81.

[4] J. Bueno, X. Martorell, R. Badia, E. Ayguadé, and J. Labarta, "Implementing ompss support for regions of data in architectures with multiple address spaces," in *Proc. 27th Int. Conf. Supercomputers (ICS'13)*, 2013, pp. 359–368.

[5] D. R. Butenhof, *Programming with POSIX Threads*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[6] P. Caheny, M. Casas, M. Moretó, H. Gloaguen, M. Saintes, E. Ayguadé, J. Labarta, and M. Valero, "Reducing cache coherence traffic with hierarchical directory cache and numa-aware runtime scheduling," in *Proc. 25th Int. Conf. Parallel Architectures Compilation Tech. (PACT'16)*, 2016, pp. 275–286.

[7] N. Carter, A. Agrawal, S. Borkar, R. Cledat, H. David, D. Dunning, J. Fryman, I. Ganev, R. Golliver, R. Knauerhase, R. Lethin, B. Meister, A. Mishra, W. Pinfold, J. Teller, J. Torrellas, N. Vasilache, G. Venkatesh, and J. Xu, "Runnemede: An architecture for ubiquitous high-performance computing," in *Proc. 19th Int. Symp. High Perf. Computer Architectures*, 2013, pp. 198–209.

[8] M. Casas, M. Moretó, L. Alvarez, E. Castillo, D. Chasapis, T. Hayes, L. Jaulmes, O. Palomar, O. S. Unsal, A. Cristal, E. Ayguadé, J. Labarta, and M. Valero, "Runtime-aware architectures," in *Proc. 21st Int. Conf. Parallel and Distributed Computing (Euro-Par'15)*, 2015, pp. 16–27.

[9] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. Adve, V. Adve, N. Carter, and C. Chou, "Denovo: Rethinking the memory hierarchy for disciplined parallelism," in *Proc. 20th Int. Conf. Parallel Architectures Compilation Tech. (PACT'11)*, 2011, pp. 155–166.

[10] A. Drebes, K. Heydemann, N. Drach, A. Pop, and A. Cohen, "Topology-aware and dependence-aware scheduling and memory allocation for task-parallel languages," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 3, pp. 30:1–30:25, 2014.

[11] A. Duran, E. Ayguadé, R. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: A proposal for programming heterogeneous multi-core architectures," *Parallel Process. Lett.*, vol. 21, pp. 173–193, 2011.

[12] Y. Durand, P. Carpenter, S. Adami, A. Bilas, D. Dutoit, A. Farcy, G. Gaydadjiev, J. Goodacre, M. Katevenis, M. Marazakis, E. Matus, I. Mavroidis, and J. Thomson, "EUROSERVER: Energy Efficient Node for European Micro-Servers," in *Proc. 17th Euromicro Conf. Digital System Design (DSD'14)*, 2014, pp. 206–213.

[13] D. Hackenberg, D. Molka, and W. Nagel, “Comparing cache architectures and coherency protocols on x86-64 multicore smp systems,” in *Proc. 42nd Int. Symp. Microarchitecture (MICRO’09)*, 2009, pp. 413–422.

[14] L. Hammond, V. Wong, M. Chen, B. Carlstrom, J. Davis, B. Hertzberg, M. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, “Transactional memory coherence and consistency,” in *Proc. 31st Int. Symp. Computer Architecture (ISCA’04)*, 2004, pp. 102–113.

[15] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel, “Rigel: An architecture and scalable programming interface for a 1000-core accelerator,” in *Proc. 36th Int. Symp. Computer Architecture (ISCA’09)*, 2009, pp. 140–151.

[16] R. Maddox, G. Singh, and R. Safranek, *Weaving high performance multiprocessor fabric: architectural insights into the Intel QuickPath Interconnect*. Intel Press, 2009.

[17] M. Manivannan, A. Negi, and P. Stenstrom, “Efficient forwarding of producer-consumer data in task-based programs,” in *Proc 42nd Int. Conf. Parallel Processing*, 2013, pp. 517–522.

[18] M. Manivannan and P. Stenstrom, “Runtime-guided cache coherence optimizations in multi-core architectures,” in *Proc. 28th Int. Parallel Distributed Processing Symp.*, 2014, pp. 625–636.

[19] M. M. K. Martin, M. D. Hill, and D. J. Sorin, “Why on-chip cache coherence is here to stay,” *Commun. ACM*, vol. 55, no. 7, pp. 78–89, 2012.

[20] J. McCalpin, “Memory bandwidth and machine balance in current high performance computers,” *IEEE TCCA Newsletter*, pp. 19–25, 1995.

[21] D. Molka, D. Hackenberg, R. Schone, and M. Muller, “Memory performance and cache coherency effects on an intel nehalem multiprocessor system,” in *Proc. 18th Int. Conf. Parallel Architectures Compilation Tech. (PACT’09)*, 2009, pp. 261–270.

[22] A. Muddukrishna, P. Jonsson, V. Vlassov, and M. Brorsson, “Locality-aware task scheduling and data distribution on NUMA systems,” in *Proc. 9th Int. Workshop on OpenMP (IWOMP’13)*, 2013, pp. 156–170.

[23] “OpenMP: Application program interface, version 4.0,” 2013. [Online]. Available: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

[24] A. Patel and K. Ghose, “Energy-efficient mesi cache coherence with pro-active snoop filtering for multicore microprocessors,” in *Proc. Int. Symp. Low Power Electronics Design (ISPLED’08)*, 2008, pp. 247–252.

[25] D. Patterson and J. Hennessy, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1990.

[26] A. Ramirez, F. Cabarcas, B. Juurlink, A. M, F. Sanchez, A. Azevedo, C. Meenderinck, C. Ciobanu, S. Isaza, and G. Gaydadjiev, “The SARC architecture,” *IEEE Micro*, vol. 30, no. 5, pp. 16–29, 2010.

[27] M. Valero, M. Moretó, M. Casas, E. Ayguadé, and J. Labarta, “Runtime-aware architectures: A first approach,” *Supercomputing frontiers and innovations*, vol. 1, no. 1, 2014.

[28] R. Vidal, M. Casas, M. Moretó, D. Chasapis, R. Ferrer, X. Martorell, E. Ayguadé, J. Labarta, and M. Valero, “Evaluating the impact of OpenMP 4.0 extensions on relevant parallel workloads,” in *Proc. 11th Int. Workshop on OpenMP (IWOMP’15)*, 2015, pp. 60–72.

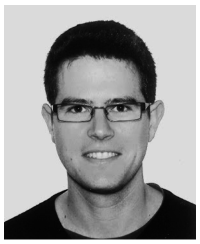
[29] V. Viswanathan, K. Kumar, and T. Willhalm. (2013) Intel memory latency checker v2. [Online]. Available: https://software.intel.com/en-us/articles/intelr-memory-latency-checker

Paul Caheny is a researcher at the Barcelona Supercomputing Center and PhD candidate at the Computer Architecture Department of UPC. Prior to joining BSC Paul was a researcher in HPC applications at Fujitsu Labs of Europe in London. Paul received his BSc. degree in Computer Applications from Dublin City University and MSc. degree from University of Edinburgh. His research interests include cache coherence and the co-design of parallel architectures and runtime systems.

Lluc Alvarez is a postdoctoral researcher at the Barcelona Supercomputing Center (BSC). He received his B.Sc. degree from the Universitat de les Illes Balears (UIB) in 2006 and his M.Sc. and Ph.D. degrees from the Universitat Politècnica de Catalunya (UPC) in 2009 and 2015. His main research interests are parallel architectures, memory systems and programming models for high-performance computing.

Said Derradji is a hardware architect at Bull. Said has been working first on several custom ASIC design interconnecting processors and focusing on cache coherency. He also worked on board design and participated on TERA-100 system delivery in 2011 (ranking 9 in Top500.org). Since 2012, he is working in the hardware architecture team at Bull, which specified recently the open exascale supercomputer, code-named SEQUANA. His areas of expertise are on ASIC/FPGA design, HPC servers architecture and on high performance interconnect technology such as the recently announced BXI (Bull Exascale Interconnect). He is BULLs representative at PCI-SIG (PCI Special Interest Group) and IBTA (InfiniBand Trade Association) consortiums.

Mateo Valero is full professor at Computer Architecture Department, UPC and director at BSC. He has published 700 papers and served in organization of 300 international conferences. His main awards are: Seymour Cray, Eckert-Mauchly, Harry Goode, ACM Distinguished Service, ”Hall of Fame” member IST European Program, King Jaime I in research, two Spanish National Awards on Informatics and Engineering. Honorary Doctorate: Universities of Chalmers, Belgrade, Las Palmas, Zaragoza, Complutense of Madrid, Granada and University of Veracruz. Professor Valero is a Fellow of IEEE, ACM, and Intel Distinguished Research Fellow. He is a member of Royal Spanish Academy of Engineering, Royal Academy of Science and Arts, correspondent academic of Royal Spanish Academy of Sciences, Academia Europaea and Mexican Academy of Science.



Miquel Moretó is a senior researcher at the Barcelona Supercomputing Center (BSC). Prior to joining BSC, he spent 15 months as a post-doctoral fellow at the International Computer Science Institute (ICSI), Berkeley, USA. He received the B.Sc., M.Sc., and Ph.D. degrees from UPC. His research interests include studying shared resources in multithreaded architectures and hardware-software co-design for future massively parallel systems.

Marc Casas is a senior researcher at the Barcelona Supercomputing Center. Prior to this, he spent 3 years as a post-doctoral fellow at the Lawrence Livermore National Laboratory (LLNL). He received his B.Sc. and M.Sc. degrees in mathematics in 2004 from the UPC and the PhD in Computer Science in 2010 from the Computer Architecture Department of UPC. His research interests are high performance computing, runtime systems and parallel algorithms.