Reducing Cache Coherence Traffic with Hierarchical Directory Cache and NUMA-Aware Runtime Scheduling

Paul Caheny[†§]
paul.caheny@bsc.es

Marc Casas[†§]
marc.casas@bsc.es

Miquel Moretó[†§]
miquel.moreto@bsc.es

Hervé Gloaguen[‡]
herve.gloaguen@atos.net

Maxime Saintes[‡]
maxime.saintes@atos.net

Eduard Ayguadé[†§]
eduard.ayguade@bsc.es

Jesús Labarta[†§]
jesus.labarta@bsc.es

Mateo Valero[†§]
mateo.valero@bsc.es

Barcelona Supercomputing Center[†]
Barcelona, Spain

Departament d'Arquitectura de Computadors[§]
Universitat Politècnica de Catalunya
Barcelona, Spain

Bull Atos Technologies[‡]
Les Clayes-sous-Bois
France

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 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 1.23x to 2.54x and coherence traffic reductions between 44% and 77% in comparison to NUMA-oblivious scheduling and data allocation. Furthermore, we show that the NUMA-aware techniques we employ at the runtime level are crucial to ensure the added hierarchical layer in the directory coherence protocol does not introduce significant coherence traffic to the system.

Keywords

Cache Coherence; NUMA; Task-based programming models

DOI: http://dx.doi.org/10.1145/2967938.2967962

1. INTRODUCTION

ccNUMA architectures have become a ubiquitous approach in the design of SMP systems that need to scale to large numbers of cores and amounts of memory capacity in pursuit of increased thread level parallelism. ccNUMA architectures provide clear advantages by increasing provision of resources such as capacity, bandwidth and parallelism in the memory hierarchy through physically distributing the cache and memory subsystem while maintaining a logically shared view of memory for the user. Since the cache and memory are both shared and distributed, many memory accesses result in a coherence transaction. 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]. Minimising this data movement by using runtime systems and parallel codes that judiciously manage data locality 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 [6], 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 performance optimal NUMA-aware fashion based on knowledge of the NUMA topology of the system, the concrete 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 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. **(1)** Data must be uni-

formly distributed amongst all the NUMA regions the system is composed of, and, **(2)** 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 (1) or (2). In this work we distinguish between the effects of (1) and (2) on both cache coherence traffic and performance.

In this work 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 [1] 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 (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 while the directory information tracked in the BCS is at a less granular per-module basis for cache lines which are exported inter-module. The BCS caches module level directory information about cache lines which are transferred between the different modules in the system. This directory information allows the BCS to filter coherence traffic from the system interconnect in certain cases 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, this paper contains the first study on how a hierarchical directory approach [19] to scaling cache coherence interacts with a runtime managed strategy to promote data locality. Indeed, we show how the extra layer of the coherence hierarchy implemented by the BCS is most fully exploited in combination with runtime managed NUMA-aware regimes for reducing coherence traffic among multiple NUMA regions.

Specifically, these are the main contributions of this paper:

- A complete performance analysis of a large SMP architecture considering three important scientific codes and three regimes of work scheduling and memory allocation: 1) Default (NUMA-oblivious) scheduling and first touch allocation 2) Default (NUMA-oblivious) scheduling and interleaved allocation which uniformly interleaves memory among NUMA regions at page granularity. 3) NUMA-aware runtime managed scheduling and allocation. We see performance improvements up to 2.54x among the benchmarks when utilising the NUMA-aware regime versus the baseline.

- 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 77% among the benchmarks when utilising the NUMA-aware regime versus the baseline.

- We show that extra layer of the coherence hierarchy implemented by the BCS works optimally when combined with runtime managed NUMA-aware scheduling and data allocation. Such a regime of scheduling and data allocation reduces capacity pressure on the directory cache in the BCS enabling it to make a net reduction in the amount of systemwide snoop related traffic of up to 35%. In contrast, we demonstrate that NUMA-oblivious policies or uneven memory allocations may overwhelm the capacity of the BCS, forcing it to inject extra snoop traffic into the system to maintain its directory state.

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 which supports the three different regimes of work scheduling and data allocation. Section 5 details the three 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 is convenient and considerably eases the programming burden, it does require a sophisticated mechanism to enforce coherence between the multiple physically distributed caches and memories 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 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 would be 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 or another cache. If a memory 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 belong to the *HREQ (Home Request)* class. For example such a request could be the cache asking the memory 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*

Table 1: Coherence protocol message classes

Type	Class	Description
Data	DWB	Data Write Back, message from cache to memory with cache line payload
Data	DTC	Data To Cache, message towards cache with cache line payload
Control	HREQ	Home Request, cache requesting a cache line in a certain state from memory
Control	SNP	Snoop, memory requesting state of cache line or invalidating cache line from cache
Control	HRSP	Home Response, cache providing state of cache line to memory
Control	NDR	Non-Data Response, memory signals transaction completion to cache, data not required

(Snoop) messages are requests from a memory to a cache asking the cache 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 which is a confirmation sent from a cache to a memory that the action requested by the SNP is completed. An *NDR (Non-Data Response)* message is sent from a memory to a cache to signal the completion of a coherence transaction, where the memory 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.

All these classes, conveniently classified as Data or Control traffic and summarised in Table 1, comprise a total decomposition of the cache coherence traffic at the LLC (Last Level Cache) to memory 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.

3. BULLION S ARCHITECTURE

Figure 1 shows a dual module bullion S system. The bullion S platform has the capability to scale to eight modules in its maximum configuration. Each module comprises two Intel Xeon sockets and their local memory connected to a single Bull proprietary ASIC, the BCS.

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 directly between the two CPUs 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). In Figure 1, for the purposes of cache coherence transactions, the system memory is represented by the units labelled MC (Memory Controller).

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 caches 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 act as a proxy for an L3 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 other messages classes as a bandwidth optimisation implemented by the CPUs. To optimise for latency, the BCS does not piggyback NDR messages on other message classes. For these reasons there may be an asymmetry between the incoming and outgoing traffic levels for the SNP, HRSP and NDR message types.

For the HREQ, DTC and DWB message classes, messages recorded as incoming to the BCS in one module will always pass through the inter-module link and appear as outgoing from the BCS in another module and vice versa. This means the amount of traffic for these message classes in the entire system is symmetric in directionality.

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 experimental dual module installation of the bullion S platform, running RHEL 6.5 with a Linux kernel version of 2.6.32. Each module is composed of two 15 core Intel *Ivy Bridge EX* E7 8890 v2 processors. Each Intel Xeon socket has a local NUMA region containing 8 GB of system memory made up of two 4 GB DIMMs. This means that of the eight memory channels connected to each socket, two are occupied by DIMMs.

The main memory subsystem may be operated in one of two modes, selectable as a systemwide parameter in the firmware. *Lockstep Mode* provides higher RAS (Reliability, Availability, Service) features than *Performance Mode*. Typically Lockstep Mode can provide around 60% of the memory bandwidth available in performance mode if both modes are configured optimally.

The memory subsystem in the system we use is configured in Lockstep Mode. Therefore the complete system has four sockets and NUMA regions, 32GB of system memory, and a total physical core count of 60. The memory configuration results in each socket having a theoretical maximum memory bandwidth within its local NUMA region of 10.6 GB/s

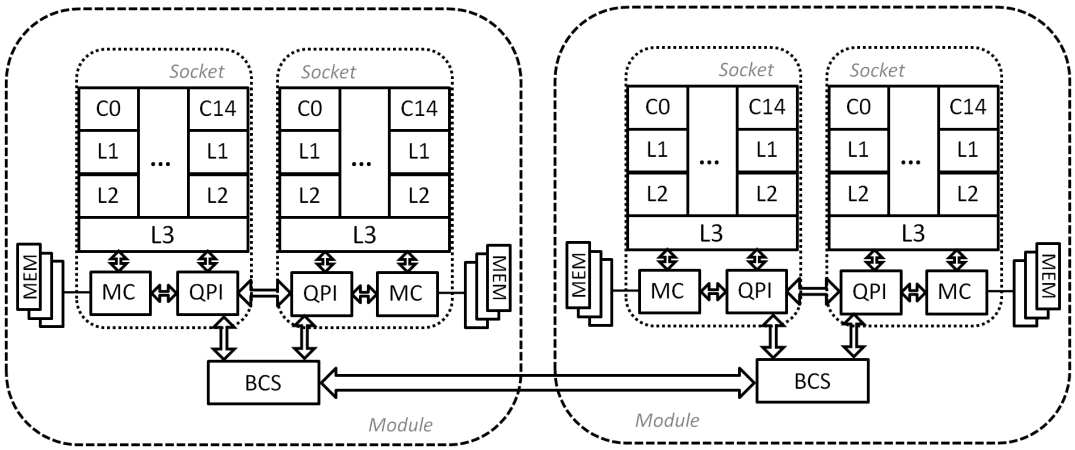


Figure 1: Logical view of a dual module bullion S system

Table 2

(a) NUMA distances

Region	0	1	2	3
0	10	15	40	40
1	15	10	40	40
2	40	40	10	15
3	40	40	15	10

(b) Latencies (ns)

0	1	2	3
115.4	178.5	407.5	404.5
183.2	109.9	400	398
419.6	401.9	109.9	173.4
415.5	401.8	173.7	110.6

Table 3: Bandwidth for the different NUMA distances in the system with the STREAM Triad benchmark (GB/s)

Region	0	1	2	3
0	5.39	4.15	4.12	**4.92**
1	4.14	5.40	4.12	**4.92**
2	4.11	4.12	5.40	**4.96**
3	4.09	4.12	4.15	**6.86**

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 Tables 2 and 3 by the three different colours, the system has three levels of NUMA distance. A coherence message may travel within the local NUMA region (green), to the single 'near' remote NUMA region, i.e. the other NUMA region in the local module (yellow), or to one of two 'far' remote NUMA regions, i.e. the NUMA regions in the remote module (red). Table 2a 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 latencies in the system (Table 2b) follow the same pattern and similar ratios to the NUMA distances in Table 2a. On average there is a 58% 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 129% to access data in either of the NUMA regions in the other module (or a 264% penalty for inter-module access compared to local NUMA region access).

Table 3 shows the memory bandwidths measured in the system for the STREAM Triad benchmark. These results use all the threads available on a single socket (15) to saturate the bandwidth to the memory of a given NUMA region. The memory bandwidth values measured follow the same pattern seen in Tables 2a and 2b, with a small aberration, the values in bold (column 3 of Table 3) are higher than the figures for the other NUMA distances in the same class (i.e. of the same colour in the table). This is due to dual rank DIMMs being installed in NUMA region 3 and single rank DIMMs elsewhere in the system.

4. MEMORY ALLOCATION AND SCHEDULING

In this work we use a task-based data-flow programming model, an approach supported by OpenMP 4.0. In this model the execution of a parallel program is structured as a set of tasks with dependences among them. 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 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 OmpSs runtime system, called Nanos++ [3], already supports NUMA-aware scheduling in the release we use, version 0.9a.

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 accomplishes this through bookkeeping of data location [5]. When data is first tied to a physical memory location (at the time of the data's initialisation) within an OmpSs task the runtime system records which NUMA region the data is physically located in. 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.

We use three regimes of task scheduling and memory allocation 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. In all cases where we use less than the maximum number of cores in the system the cores in use are uniformly distributed among the four NUMA regions of the system.

***D**efault scheduling & **F**irst **T**ouch allocation (**DFT**):* Tasks are scheduled in a NUMA oblivious fashion among all available cores in the system, ensuring load is balanced. Data is allocated in the NUMA region of the core where the allocation happens to execute, via the Linux first touch memory allocation policy. In a worst case scenario, all data may happen to be allocated in a single NUMA region, leaving others entirely unused. Tasks using the allocated data are scheduled among the available cores without any consideration for where their requisite 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, all data allocations are 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 the allocation runs on. 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.

***N**UMA-**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 recognises tasks which allocate data and schedules them in a uniformly distributed arrangement among the NUMA regions. Each task which allocates data allocates all its data locally in the NUMA region it runs in via 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-task allocation 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. Uniquely under NAFT, it is possible load imbalance may occur due to the NUMA-aware scheduling. In this case the runtime system handles it via work-stealing. Load imbalance was negligible in all our experiments.

A regime utilising NUMA-aware scheduling & interleaved allocation is not a valid combination. This is because, in order to create data locality, the NUMA-aware scheduler depends on each task allocating data to do so in its local NUMA region only, via first touch.

5. BENCHMARKS

We chose three benchmarks for use in our evaluations, representative of important problems in both high performance and data centric computing, with significantly different data access patterns. One is from the PARSEC benchmark suite [4] and two are based on linear algebra problems [2].

PARSEC Streamcluster This benchmark from the PARSEC suite is based on the online clustering problem. 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. This benchmark runs using the native input data set as specified by the PARSEC documentation and is executed on a varying number of cores (16 to 32) distributed evenly across the four sockets in the system.

Cholesky This benchmark uses a modern 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.

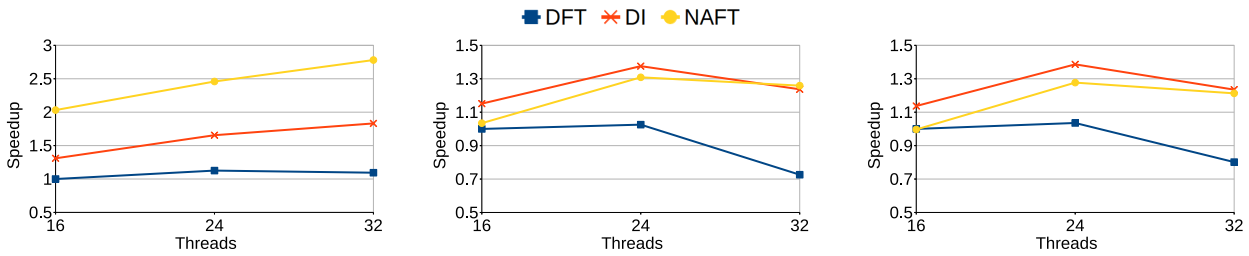
Symmetric Matrix Inversion (SMI) This benchmark is a larger linear algebra problem which inverts a symmetric matrix in three stages. It also follows a tile-based algorithm thus exposing fine grained parallelism. It is implemented in the task-based OmpSs programming model.

Both linear algebra benchmarks exhibit a complex data access pattern comprising a mix of reads and writes and are run across different core counts (16 to 32) with threads distributed evenly across the four sockets in the system. In all cases, the size (N) of the NxN matrix we use in both linear algebra benchmarks is 20480. At this value of N, in all cases, the benchmark's performance has reached the maximum attainable (in GFLOP/s) and is no longer increasing with larger values of N. We experimented with different tile sizes for both linear algebra benchmarks and found a tile size of 512 to give the best performance in all cases and this is the tile size used for all the linear algebra results we present. All performance and coherence traffic measurements are repeated three times and the mean value reported.

6. RESULTS AND ANALYSIS

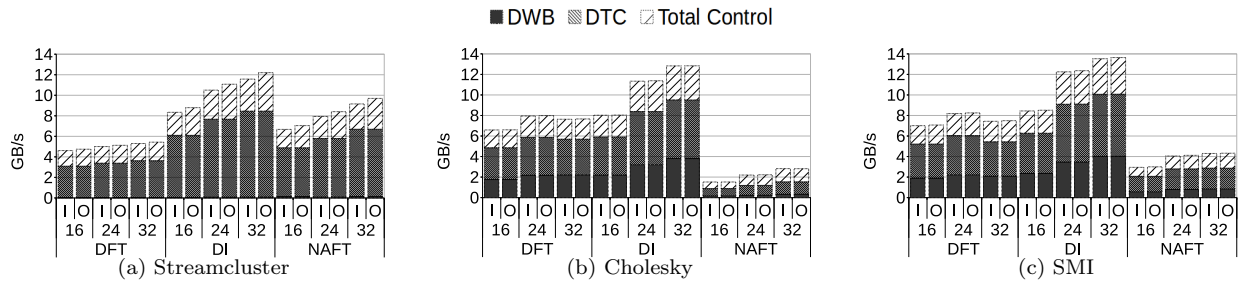
6.1 Introduction

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



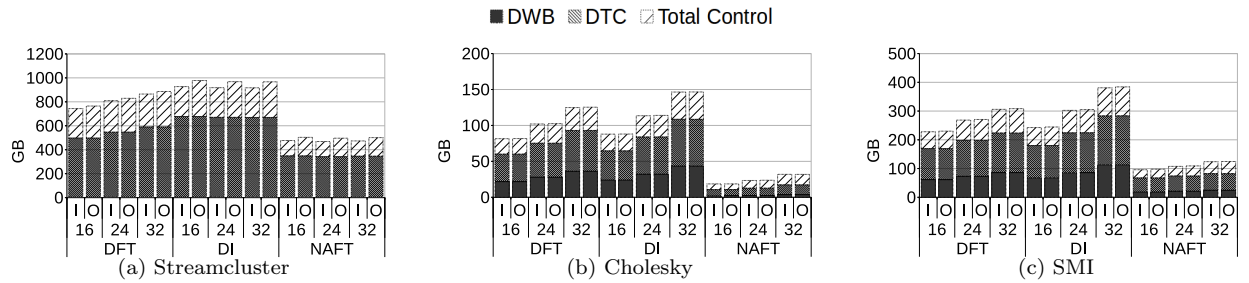
(a) Streamcluster (b) Cholesky (c) SMI

Figure 2: Speedup DFT, DI, NAFT regimes under increasing thread count



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

Figure 3: Coherence Bandwidth: DWB, DTC and Control traffic



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

Figure 4: Coherence Movement: DWB, DTC and Control traffic

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 the interplay between the three software based regimes for task scheduling and memory allocation and the two level directory hierarchy managing coherence among the L3s in the system.

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. All Figures show the systemwide (i.e. both modules aggregated) coherence traffic. The traffic's directionality (see Section 3.1) is labelled as I and O in all figures for incoming and outgoing traffic respectively. In order to clearly distinguish between the two, we always refer to the bandwidth to memory in the system as memory bandwidth and the bandwidth of coherence traffic travelling between modules and measured at the BCS as coherence bandwidth.

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 as the baseline (equal to 1 in Figure 2) and are discussed in Section 6.2.

It is important to note that in a ccNUMA architecture the bandwidth to memory is also 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 of memory bandwidth will be K/R times the systemwide maximum possible. Similarly the system contains two BCS, one in each module. Each BCS implements a module level directory which caches directory information for cache lines exported from that module to L3s in a remote module. If a workload allocates memory in only 1 module, only half of the systemwide BCS resources are utilised.

Table 4: Speedup and reduction in coherence movement at best performing thread count: Streamcluster (32 threads), Cholesky (24 threads) and SMI (24 threads).

(a) NAFT regime in comparison to DFT regime

Benchmark	Speedup	DWB	DTC	Control	Total
Streamcluster	2.54x	12%	42%	50%	44%
Cholesky	1.28x	91%	78%	60%	77%
SMI	1.23x	71%	58%	52%	60%

(b) NAFT regime in comparison to DI regime

Benchmark	Speedup	DWB	DTC	Control	Total
Streamcluster	1.52x	12%	49%	48%	48%
Cholesky	0.95x	92%	80%	63%	79%
SMI	0.92x	75%	62%	57%	64%

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. In all cases the Data message classes are 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 the other module. Therefore these messages always appear as both incoming to the BCS in one module and outgoing from the BCS in the other module.

6.2.1 Streamcluster

The Streamcluster results show markedly different performance (Figure 2a) among the three scheduling and memory allocation regimes as the number of threads employed increases. The NAFT regime performs best at all thread counts. Using the DFT regime at 16 threads as a baseline, the best performance measured is with the NAFT regime at 32 threads, where it performs at 2.78x the performance baseline compared to 1.09x and 1.83x the performance baseline for the DFT and DI regimes respectively at 32 threads.

Figure 3a shows that for Streamcluster both the DI and NAFT regimes utilise more coherence bandwidth than the DFT regime at all thread counts. This happens because, as trace analysis shows, under the DFT regime all the data allocated by the benchmark is located in NUMA region 0 and the other NUMA regions remain unused. In contrast, under the DI and NAFT regimes data is allocated uniformly (see Section 4.1) among all four NUMA regions. Thus under the DFT regime the memory bandwidth is limited to that of the single NUMA region the benchmark's data is allocated in (i.e. 1/4 of the systemwide memory bandwidth).

The coherence bandwidth utilised by Streamcluster increases with thread count at varying rates under the DI and NAFT regimes (Figure 3a). The increase is not very significant in the case of the DFT regime due to the previously mentioned bottleneck caused by all memory being allocated in a single NUMA region. Despite the NAFT scheduler utilising more coherence bandwidth than the DFT scheduler, Figure 4a shows that it causes significantly less coherence movement, and therefore consumes less energy, than the DFT regime over the course of the benchmark.

The traffic results of the three regimes allow us to decouple the performance limiting factors of the Streamcluster benchmark when run under the DFT regime.

First, since the DI regime uniformly spreads the memory allocation among all four NUMA regions, the single NUMA region utilisation bottleneck of the DFT regime is eliminated, allowing the Streamcluster benchmark to use the full systemwide memory bandwidth. At 32 threads the DFT and DI regimes have a similar level of aggregate (Incoming + Outgoing) coherence movement in the system at 1751 GB and 1884 GB respectively. However, the DI regime moves its coherence data at a substantially higher aggregate (Incoming + Outgoing) coherence bandwidth through the two BCS in the system, reaching 23.8 GB/s of aggregate coherence bandwidth. In comparison, the DFT regime utilises only 10.7 GB/s of aggregate coherence bandwidth.

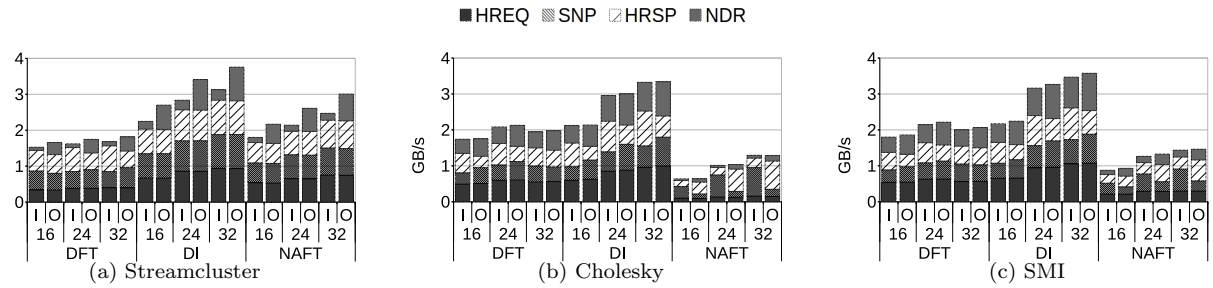
Under the NAFT regime, as well as utilising the full systemwide memory bandwidth due to a uniform allocation of memory among the NUMA regions, the coherence movement (and thus energy use) is significantly reduced in comparison to both DI and DFT regimes due to the higher data locality achieved by the NUMA-aware task scheduling.

We can also see from Figure 4a that the Streamcluster Data coherence movement is almost entirely made up of the DTC message class with a negligible amount of DWB traffic present. DWB traffic makes up less than 3% of the Data coherence movement across all regimes and thread counts. This reflects the characteristics of the Streamcluster benchmark which has a streaming read pattern of memory access with little data being written. The large proportion of the traffic made up of DTC messages represent cache lines being delivered from the memory of one NUMA region to the LLC cache of a socket in the other module for reading, via the inter-module link. The Control message type makes up between 26% and 35% of total coherence movement and is analysed in more detail in Sections 6.3 and 6.4.

6.2.2 Linear Algebra benchmarks

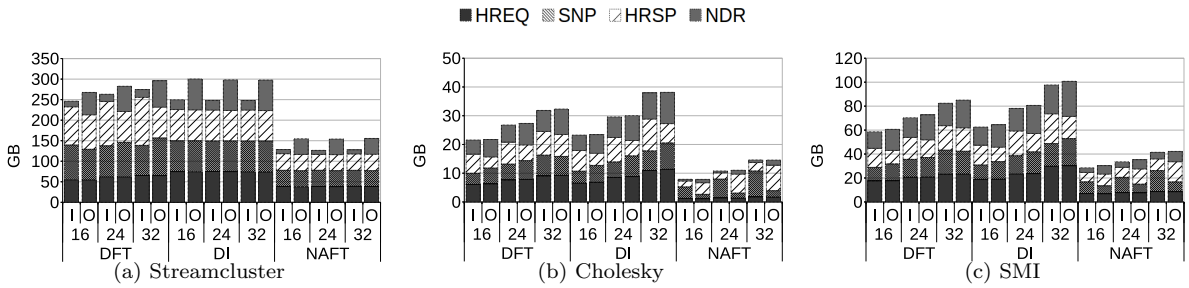
The mix of both DWB and DTC message classes in the Data coherence movement of both linear algebra benchmarks demonstrates the more complex memory access patterns involved in these benchmarks compared to the Streamcluster benchmark. The DWB message class represents modified cache lines being written back from the LLC of one socket to memory in the cache line's home NUMA region in a remote module. Under the DFT and DI regimes, DWB traffic makes up between 26% and 30% of Data coherence movement for both Cholesky and the more complex SMI benchmark. Under the NAFT regime, DWB traffic as a percentage of total Data coherence traffic falls to 11% and 19% respectively for Cholesky and SMI. This demonstrates that the NAFT regime is particularly effective in reducing the amount of inter-module DWB traffic. The Control message type makes up between 25% and 27% of total coherence movement under the DFT and DI regimes for both linear algebra benchmarks. This figure rise to between 30% and 46% of the reduced total coherence movement under the NAFT regime. The Control message type is analysed in greater detail in Sections 6.3 and 6.4.

For both Cholesky (Figures 3b and 4b) and SMI (Figures 3c and 4c) we can see that the NAFT regime is utilising less



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

Figure 5: Coherence Bandwidth: Control traffic by message class



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

Figure 6: Coherence Movement: Control traffic by message class

coherence bandwidth and causes significantly less coherence movement than either the DFT or DI regimes. This shows that the NUMA-aware scheduling employed by the NAFT regime achieves a strong co-location of tasks and their requisite data within each of the individual NUMA regions of the system.

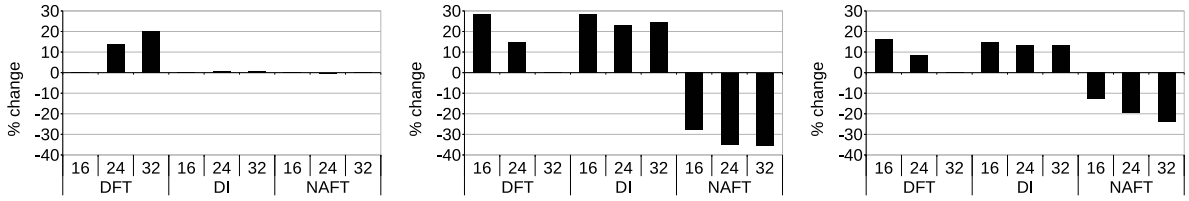
In terms of performance, in both Cholesky (Figure 2b) and SMI (Figure 2c) the DFT regime shows almost no scaling moving from 16 to 24 threads. This is due to a large majority of the required memory being allocated in only the first and second NUMA regions (both in the same module), with very little of the required memory allocated in the third and fourth NUMA regions of the system. So, in the case of the DFT regime only close to 2/4 of the systemwide memory bandwidth is used. Both the DI and NAFT regimes utilise the full systemwide memory resources available and scale better than the DFT regime. However, performance deteriorates moving from 24 to 32 threads for both the DI and NAFT regimes, due to contention among the increasing number of threads for the same limited (detailed in Section 3.2) memory bandwidth in the system.

Figures 3b and 3c show that under the DI regime Cholesky reaches 22.7 GB/s at 24 threads and 25.7 GB/s at 32 threads of aggregate coherence bandwidth while the corresponding figures for SMI are 24.6 GB/s at 24 threads and 27.2 GB/s at 32 threads. Under the NAFT regime, coherence bandwidth does not reach above 8.6 GB/s at any thread count for Cholesky or SMI. In this case hardware counter analysis shows that IPC per thread is decreasing as thread count is increasing. As can be seen from the low levels of inter-module coherence traffic under the NAFT regime in Figures 3b and 3c most coherence traffic is localised within each of the individual NUMA regions. The performance bottleneck for all regimes exists within the memory subsystem (memory controller, bandwidth to DRAM) within each separate NUMA region and not in the inter-module link routed via the BCS. Were the inter-module link via the BCS to become the performance bottleneck (if, for example, greater memory bandwidth were available or there were more modules in the system) then the NAFT regime would, in addition to reducing coherence traffic and energy cost, enable better scaling of performance than DFT or DI due to its much lower coherence bandwidth and coherence movement requirements.

6.2.3 *Summary*

In the coherence movement view (Figure 4) for all benchmarks, at all thread counts, the DI regime requires the largest coherence movement closely followed by the DFT regime. The NAFT regime enables a significant reduction in coherence movement in comparison to the other two regimes, the magnitude of which depends on the benchmark and thread count. The reductions in coherence movement by the NAFT regime in comparison to the DFT regime at the best performing thread count for each benchmark are listed in Table 4a. The NAFT regime reduces the total coherence movement by 44%, 77% and 60% respectively for the Streamcluster, Cholesky and SMI benchmarks in comparison to the the DFT regime. Table 4b summarises the differences between the DI and NAFT regimes.

It is important to note for both linear algebra benchmarks in Table 4b that even though the NAFT regime does not outperform the DI regime in the system we use, it does significantly reduce coherence movement (by 48%, 79% and 64% for Streamcluster, Cholesky and SMI respectively) which has substantial implications for energy efficiency. In both linear algebra benchmarks under the DI and NAFT regimes the computation is limited by the memory resources in the experimental bullion S system installation we used rather than by the inter-module coherence traffic. In an installation of this system with a larger memory configuration or in a lager



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

Figure 7: SNP/HRSP coherence movement only, net change due to BCS

(> 2 modules) bullion S system the coherence bandwidth generated would become a more important factor in performance where the substantial reduction in coherence movement realised by the NAFT regime would enable greater scaling than both the DFT and DI regimes.

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 only, now decomposed into the individual Control message classes HREQ, SNP, HRSP and NDR.

In the Control message classes we see a symmetry between the incoming and outgoing traffic for the HREQ message class, as was also evident for the Data message classes (DWB and DTC). Each HREQ message originates at a cache within one module and travels through the inter-module interconnect (via first the local and then the remote BCS) to its respective memory destination in the other module. Therefore, in the two module system we use, HREQ messages always appear as incoming traffic to the BCS in their source module and as outgoing traffic from the BCS in their destination module.

For the remaining Control type messages (SNP, HRSP and NDR) this symmetry between incoming and outgoing traffic does not necessarily hold, as explained in Section 3. Figures 6b and 6c show the asymmetry in directionality of the SNP and HRSP traffic is most pronounced under the NAFT regime for both linear algebra benchmarks. These asymmetries represent cases where the action of the BCS in the coherence protocol is having a significant impact on the SNP and reciprocal HRSP coherence movement and we investigate this behaviour further in Section 6.4. The best performing thread count for both linear algebra benchmarks is 24 threads. With Cholesky at 24 threads the total outgoing SNP traffic in the system is only 26% of the total incoming SNP traffic. For the more complex SMI linear algebra problem, the total outgoing SNP traffic in the system is 56% of the total incoming SNP traffic. The asymmetry visible across the benchmarks for the NDR traffic is due to the CPU (but not the BCS) in some cases piggybacking the NDR message on top of unused bits in other message classes as a bandwidth optimisation.

We also note a significant difference in the ratio of the count (not coherence bandwidth or coherence movement) of correlated HREQ and DTC messages between the Streamcluster benchmark and both linear algebra benchmarks. In the Streamcluster benchmark across all thread counts and regimes, there is never more than a 4% deviation from a 1:1 ratio for the message counts of HREQ:DTC. This is evidence of the streaming read nature of Streamcluster where HREQ messages ask for a cache line in a read state and do not ask to further change the state of the cache line before evicting it. This results in a 1:1 call and response between HREQ and DTC. For the Cholesky and SMI benchmarks, which have a more complex memory access pattern including write accesses and shared data among threads, the same HREQ:DTC ratio varies from 1:0.63 up to 1:0.91 depending on the regime and thread count. This is explained by the observation that in some cases cache lines will be first requested read-only (1:1 HREQ:DTC ratio) and subsequently upgraded to modified for writing. In the upgrade case, a HREQ is required to ask to upgrade the state of the cache line. However, as the cache already has a valid copy of the data, no DTC is required.

6.4 Impact of NUMA-Aware Scheduling on Effectiveness of BCS

The BCS may affect the amount of SNP/HRSP traffic in two ways: **(1)** Filter SNP/HRSP transactions out of the system where it can participate in a transaction in place of a CPU, reducing SNP/HRSP traffic. **(2)** Add SNP/HRSP transactions in the process of maintaining its own directory state. The directory cache maintained by the BCS is inclusive of all cache lines exported from its local module to L3 caches in any remote module, and L3 caches may silently evict cache lines. In order to maintain inclusivity, if the directory cache in the BCS is under capacity pressure due to a high volume of cache lines being exported from the module (poor data locality) and needs to evict an existing directory entry in order to allocate a new one, this eviction from the directory cache in the BCS requires the cache line be evicted from the remote L3 caching it, requiring a BCS initiated evicting SNP/HRSP transaction.

The actual SNP and HRSP coherence movement measured in the system (which includes the impact of the BCS in the protocol) is made up of the sum of the incoming SNP, outgoing SNP, incoming HRSP and outgoing HRSP traffic featured in Figure 6.

We measure how effectively the BCS affects the number of SNP/HRSP transactions in the system by comparing the SNP/HRSP transactions initiated by the CPUs with the total system SNP/HRSP transactions in the system. CPU initiated SNP/HRSP transactions include only incoming SNPs (which originate at CPUs) and outgoing HRSPs (response to the incoming SNPs). If the BCS did not exist in the system all the incoming SNPs measured would have appeared as outgoing SNPs in the opposite module and all the outgoing HRSPs would have appeared as incoming HRSPs in the opposite module. Therefore we double the (incoming SNP + outgoing HRSP) traffic to extrapolate what the total traffic

in both modules would have been without the intervention of the BCS in the coherence protocol.

Figure 7 presents the net change in coherence movement for the SNP/HRSP message classes in isolation, due to the actions of the BCS. It shows the BCS can produce a net reduction in SNP/HRSP traffic under the NAFT regime for both linear algebra benchmarks resulting in reductions of between 13% and 35%. Due to the high data locality produced by the NAFT regime, the directory cache in the BCS can filter more SNP/HRSP traffic from the system than any amount it introduces in maintaining its directory. Under the other regimes with poorer data locality (DFT, DI) where we see an increase in SNP/HRSP traffic (up to 29%) the BCS is aiding scalability by creating an extra (less granular) hierarchical level of coherence tracking to the system, at the cost of the added SNP traffic to maintain its directory state.

These figures show the high data locality of the software based NAFT regime exploits the function of the BCS to best effect. Under the NAFT regime the BCS achieves a net reduction of the SNP/HRSP coherence movement in two of the three benchmarks, and in all cases the NAFT regime uses the BCS more effectively than the DFT or DI regimes.

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 coherency 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 coherency system. Manivannan et al. [18] [17] showed how the runtime and coherency system could co-operate to provide performance benefits for 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 coherency system. All these papers utilised a simulation based approach to evaluate the impact of their designs on coherence whereas in our work we characterise directly 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. [2] measured the performance benefits of NUMA-aware scheduling for both the Cholesky and Symmetric Matrix Inversion algorithms versus the best state of the art implementations that are widely used in modern production environments. Muddukrishna et al. also 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 the 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 [27] [8] proposals follow a hardware/software co-design approach to relaxing hardware provided cache coherency and moving 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 in combination with hierarchical directory coherence on cache coherence traffic.

8. CONCLUSIONS

ccNUMA 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 in SMP designs. 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 characterise the coherence traffic within a modern large SMP design at the granularity of individual classes of coherence traffic using three important scientific benchmarks. We show the balance between the different types of coherence traffic, Data and Control traffic, and further break down these types into individual message classes. We present evidence for the ability of the BCS to mitigate the cost of coherence traffic with increasing system scale.

We show that NUMA-aware policies improve the efficacy of the BCS. This combined hardware-software solution achieves significant reductions in coherence movement which in turn has implications for energy savings. We show evidence that the NAFT regime combined with the BCS is able to reduce the coherence movement in all cases, with reductions ranging from 44% to 77%. Other NUMA-oblivious regimes combined with the BCS do not show such synergistic potential, which is indeed an important conclusion to take into account in the way future computing infrastructures and system software stacks are designed [15, 7, 12, 27].

Acknowledgements

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 Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 610402 and from the EU's H2020 Framework Programme (H2020/2014-2020) under grant agreement n° 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).

9. REFERENCES

[1] Data center optimization with bullion. 2015.

[2] 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*, 2(1), 2015.

[3] J. Balart, A. Duran, M. González, X. Martorell, E. Ayguadé, and J. Labarta. Nanos mercurium: a research compiler for openmp. In *European Workshop on OpenMP*, EWOMP '04, pages 103–109, 2004.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81. ACM, 2008.

[5] J. Bueno, X. Martorell, R. M. Badia, E. Ayguadé, and J. Labarta. Implementing ompss support for regions of data in architectures with multiple address spaces. In *Proceedings of the 27th International Conference on Supercomputing*, ICS '13, pages 359–368. ACM, 2013.

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

[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 *Proceedings of the 19th International Symposium on High Performance Computer Architecture*, HPCA '13, pages 198–209. IEEE Computer Society, 2013.

[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 *Proceedings of the 21st International Conference on Parallel and Distributed Computing*, Euro-Par '15, pages 16–27. Springer Berlin Heidelberg, 2015.

[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 *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques*, pages 155–166. IEEE Computer Society, 2011.

[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.*, 11(3):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.*, 21:173–193, 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 *17th Euromicro Conference on Digital System Design*, DSD '14. IEEE Computer Society, 2014.

[13] D. Hackenberg, D. Molka, and W. E. Nagel. Comparing cache architectures and coherency protocols on x86-64 multicore smp systems. In *Proceedings of the International Symposium on Microarchitecture*, MICRO 42, pages 413–422. IEEE Computer Society, 2009.

[14] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st International Symposium on Computer Architecture*, ISCA '04, pages 102–113. IEEE Computer Society, 2004.

[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 *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 140–151. ACM, 2009.

[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. StenstrÃűm. Efficient forwarding of producer-consumer data in task-based programs. In *Proceedings of the 42nd International Conference on Parallel Processing*, ICPP '13, pages 517–522. IEEE Computer Society, 2013.

[18] M. Manivannan and P. Stenstrom. Runtime-guided cache coherence optimizations in multi-core architectures. In *Proceedings of the 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 625–636. IEEE Computer Society, 2014.

[19] M. M. K. Martin, M. D. Hill, and D. J. Sorin. Why on-chip cache coherence is here to stay. *Commun. ACM*, 55(7):78–89, 2012.

[20] J. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[21] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller. Memory performance and cache coherency effects on an intel nehalem multiprocessor system. In *18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 261–270. IEEE Computer Society, 2009.

[22] A. Muddukrishna, P. A. Jonsson, V. Vlassov, and M. Brorsson. Locality-aware task scheduling and data distribution on numa systems. In *Proceedings of the 9th International Workshop on OpenMP*, IWOMP '13, pages 156–170. Springer Berlin Heidelberg, 2013.

[23] OpenMP: Application program interface, version 4.0. 2013.

[24] A. Patel and K. Ghose. Energy-efficient MESI cache coherence with pro-active snoop filtering for multicore microprocessors. In *Proceedings of the International*

Symposium on Low Power Electronics and Design, ISPLED '08, pages 247–252. ACM, 2008.

[25] D. Patterson and J. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 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*, 30(5):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*, 1(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 *Proceedings of the 11th International Workshop on OpenMP*, IWOMP '15, pages 60–72. Springer International Publishing, 2015.

[29] V. Viswanathan, K. Kumar, and T. Willhalm. Intel memory latency checker v2, 2013.