# RUNTIME-ASSISTED COHERENT CACHING

---


**Paul Caheny**

Barcelona, 2020

Advisors: **Miquel Moretó Planas,**
**Lluc Álvarez Martí**


A thesis submitted in fulfillment of the requirements for the degree of
Doctor of Philosophy


in the Departament d'Arquitectura de Computadors

Universitat Politècnica de Catalunya

# Abstract

In the early years of the 2000s a fundamental change of course occurred in the pursuit of performance in computer architecture. Avenues of improvement such as frequency scaling and instruction level parallelism that had been successfully exploited for decades were now providing rapidly diminishing returns. Since then, scaling up the thread-level parallelism provided by multicore processors has become a strong driver of performance gains.

Increasing core counts have exacerbated the pre-existing problem of the Memory Wall. In response to this, both cache and main memory architecture have become more complex in organisation and implementation, while still maintaining a shared view of memory to software. Trends of increasing parallelism, heterogeneity and resource distribution continue apace in computer architecture. As such trends continue, the contribution of the memory hierarchy as a proportion of the overall system performance profile will continue to grow.

As thread-level parallelism has increased in markets ranging from SoCs for wearable devices, to smartphones, up to the largest shared memory servers available, the problem of programmability has been brought into sharper focus. One of the most promising developments in programming models and their associated runtime systems in the past decade and a half has been task-based programming models. Such programming models provide ease of programmability to the user, at a level which is abstract enough to allow the runtime system layer to expertly optimise application execution for the underlying hardware. Thus, the runtime system layer, enabled by the programming model, can help tackle the increasing diversity and complexity of modern hardware. The main goal of this thesis is to exploit existing information already available in today's task-based programming models to drive optimisations and efficiencies in the operation of the memory hierarchy, through a hardware/software co-design approach.

Data movement becomes the primary factor affecting power and performance as shared memory system architectures continue to scale up in core count and therefore network diameter. The first contribution of the thesis studies the ability of a task-based programming model to constrain off-chip data movement in a real, very large shared memory system with a hierarchical coherence directory. It characterises directly and in detail the effectiveness of the programming

model's runtime system at minimising data traffic in the hardware. The analysis demonstrates that the runtime system of a task-based programming model can maximise locality between tasks and the data they use, thus minimising the traffic in the cache coherent interconnect.

The second and third contributions of the thesis investigate hardware/software co-design proposals to increase efficiency within the on-chip memory hierarchy. These two contributions exploit information already captured within existing task-based programming models. They communicate this information from the runtime system to the hardware and use it there to drive power, performance and area improvements in the memory hierarchy. A simulator based approach is used to model and analyse both the second and third contributions. It employs gem5, a detailed cycle-level computer architecture simulator coupled with the ruby memory hierarchy simulator to provide a detailed evaluation of both contributions.

The second contribution of the thesis focuses on the scalability of cache coherence. Scaling cache coherence to a growing number of caches is a crucial issue in computer architecture as core counts continue to increase. The second contribution demonstrates the ability of a runtime system and hardware co-design to dramatically reduce demand for entries in the coherence directory, the size of which is a fundamental issue in the scalability of cache coherence.

In addition to the number of private caches increasing alongside core count, non-uniform cache access (NUCA) shared caches are also increasing in size and share of on-chip resources as the last line of defence against costly off-chip memory accesses. The third contribution of the thesis is concerned with optimising the operation and utilisation of such NUCA caches to increase their effectiveness at dealing with the bottleneck between computation and memory. It shows a runtime system and hardware co-design approach can successfully reduce the network distance costs in a shared NUCA cache by placing data nearer to where it will be used, and not placing data in the cache at all, if it will not be re-used.

Together, the three contributions of this thesis demonstrate the potential for task-based programming models to address key elements of scalability in the memory hierarchy of future systems at the level of private caches, shared caches and main memory.

# Table of contents

**Abstract** i

**Contents** v

**1 Introduction** 1

1.1 Thesis Objectives and Contributions . . . . . 3

1.1.1 Reducing Coherence Traffic with a NUMA-Aware Runtime Approach 3

1.1.2 Runtime-Assisted Coherence Deactivation in Task Parallel Programs 5

1.1.3 Runtime-Directed Last Level Cache . . . . . 6

1.2 Thesis Outline . . . . . 7

**2 Background** 9

2.1 Memory Hierarchies in Multiprocessors . . . . . 9

2.1.1 ccNUMA architectures . . . . . 11

2.1.2 Cache hierarchies . . . . . 12

2.1.3 Maintaining Cache Coherence . . . . . 14

2.2 Scaling Cache and Memory . . . . . 17

2.2.1 Scaling ccNUMA . . . . . 17

2.2.2 Scaling Directory based Cache Coherence . . . . . 19

2.2.3 Scaling NUCA Last Level Cache . . . . . 23

2.3 Programming Modern Multiprocessors . . . . . 30

2.3.1 Low Level Shared Memory Programming . . . . . 31

2.3.2 High Level Shared Memory Programming Models . . . . . 31

2.3.3 Task-Based Programming Models . . . . . 32

2.4 Runtime-aware architectures . . . . . 34

**3 Experimental Methodology** **37**
3.1 Very Large Scale NUMA System . . . . . 37
3.2 Simulation Infrastructure . . . . . 39
3.2.1 Simulators . . . . . 40
3.2.2 Baseline Simulated System Architecture . . . . . 40
3.2.3 Simulated System Environment . . . . . 41
3.3 Benchmarks . . . . . 41

**4 Reducing Cache Coherence Traffic with a NUMA-aware Runtime Approach** **45**
4.1 Introduction . . . . . 45
4.2 Cache Coherence in a Very Large Scale NUMA System . . . . . 48
4.2.1 Categories of Coherence Traffic . . . . . 48
4.2.2 ccNUMA in the bullion S System . . . . . 49
4.3 Memory Allocation and Scheduling for ccNUMA . . . . . 51
4.3.1 Programming Model and Runtime System NUMA-awareness . . . . . 51
4.3.2 Scheduling and Memory Allocation Regimes . . . . . 52
4.4 Results and Analysis . . . . . 53
4.4.1 Introduction . . . . . 53
4.4.2 DWB, DTC and Control Coherence Traffic . . . . . 55
4.4.3 Control Coherence Traffic Decomposition . . . . . 62
4.4.4 Traffic Symmetry Over Modules . . . . . 64
4.5 Summary . . . . . 66

**5 Runtime-Driven Cache Coherence Deactivation** **69**
5.1 Introduction . . . . . 69
5.2 Opportunity to Deactivate Coherence . . . . . 71
5.3 RaCCD: Runtime-assisted Cache Coherence Deactivation . . . . . 72
5.3.1 Runtime System - Architecture Interface . . . . . 73
5.3.2 Runtime System Extensions . . . . . 73
5.3.3 Architectural Support . . . . . 75
5.3.4 Adaptive Directory Reduction . . . . . 78
5.3.5 Additional Considerations . . . . . 79
5.4 Evaluation . . . . . 80
5.4.1 Static Directory Reduction . . . . . 80
5.4.2 Adaptive Directory Reduction . . . . . 85

5.4.3 RaCCD Overheads . . . . . 87
5.5 Summary . . . . . 87

**6 Runtime-Driven LLC Management** **89**
6.1 Introduction . . . . . 90
6.2 RDLLCM Design . . . . . 91
6.2.1 Overview . . . . . 91
6.2.2 Opportunity for Runtime Driven LLC Management . . . . . 93
6.2.3 Software - Hardware Interface . . . . . 93
6.2.4 RDLLCM Hardware Extensions . . . . . 94
6.2.5 RDLLCM Runtime System Extensions . . . . . 97
6.2.6 Additional Considerations . . . . . 100
6.3 Evaluation . . . . . 101
6.3.1 Metrics and Comparators . . . . . 101
6.3.2 Results and Analysis . . . . . 102
6.4 Summary . . . . . 109

**7 Conclusions** **111**
7.1 Goals, Contributions and Main Conclusions . . . . . 111
7.2 Future Work . . . . . 114
7.3 Publications . . . . . 115
7.4 Financial and Technical Support . . . . . 116

**Bibliography** **117**

**List of Figures** **135**

**List of Tables** **135**

**Glossary** **135**

# Chapter 1
# Introduction

Computing has been dealing with the consequences of a fundamental shift in its foundations for over a decade and a half now. Suddenly, in 2004, several related fundamental trends in computer architecture changed course [45] due to the stagnation of Dennard Scaling [50] and the continuation of Moore's Law [96]. Instruction level parallelism and frequency scaling were rich seams of progress which had been mined profitably for the previous four decades; however, they were now providing rapidly diminishing returns. Instead of these long-pursued avenues of improving single-threaded performance, progress would now be achieved by moving to multicore processors and increasing the number of cores per processor. At the same time as the move towards multicore processors occurred, problems such as the Power [99] and Memory [129] Walls heralded a shift towards increased heterogeneity and specialisation in processing units, alongside deeper, more complex memory hierarchies.

The proliferation of cores within a single processor has spurred many changes in the memory hierarchy. Multicore processors rely on the memory hierarchy to keep their multiple processing cores supplied with instructions and data. However, over recent decades improvements in the performance of memory technologies have not kept pace with the improvements in processing power, making the memory hierarchy a growing focus for computer architects. In order to ease the transition to the multicore era from a software perspective, cache coherent shared memory architectures have dominated the memory hierarchy landscape. These architectures render caches largely invisible to the programmer and give the software a global view of a shared main memory space from all cores. At the same time, under the hood, they allow hardware designers to vary the number, size, cost and organisation of the different types of physical cache and memory technologies available within the memory hierarchy.

Typically today we see multicores with deep and complex memory hierarchies in form factors ranging from smartphones up to the largest shared memory server architectures. Alongside multiple cores per processor, designs use between 2 and 4 layers of private and shared

caches plus a shared main system memory. In order to scale capacity and throughput both the shared caches and shared main memory are typically constituted of discrete and physically distributed cache and memory banks, networked together so they are all accessible from any core. From a functional perspective such hardware maintains a logically shared, flat view of the memory hierarchy. Despite this, from a performance perspective, there are very significant non-uniformities as a consequence of the underlying physical arrangement of the hardware. The name for this non-uniformity of performance in relation to memory is Non-Uniform Memory Access (NUMA), whereas for shared caches the respective term is Non-Uniform Cache Access (NUCA). From a functional and correctness perspective software need not concern itself with NUMA and NUCA effects. However, in order to achieve maximum performance, both NUMA and NUCA effects continue to become more and more important as per-processor core counts continue to increase.

Cache coherence means ensuring the software only ever sees consistent values for a given memory location, regardless of how many copies of the memory location may exist and potentially experience concurrent writes within the cache hierarchy. In tandem with complex memory hierarchies, approaches to maintaining cache coherence within the hierarchy continue to grow more sophisticated in order to address the challenges of scaling performance. In the early days of cache coherent NUMA (ccNUMA) systems, with relatively modest core counts, simple broadcast message based *snooping* protocols sufficed to provide coherence. Increasing core counts quickly necessitated the move to more scalable *directory* based protocols, which continue to dominate in all but the smallest and most simple systems today [67][119]. Looking to the future, as core counts continue to increase, even directory based cache coherence will require innovative and sophisticated optimisations in order to continue to scale.

It was already clear at the beginning of the multicore era that the profound and rapidly unfolding changes underway in the computer architecture landscape would cause fundamental challenges in how computers are programmed. In order to realise gains in performance with such complex underlying hardware, concerns such as parallelism, heterogeneity and complex memory hierarchies cannot be ignored by software. However, explicitly managing the scheduling and synchronisation of work across increasingly parallel hardware is too great a burden to place on the programmer in the majority of software application domains.

One of the most popular approaches to overcoming this burden has become annotation based parallel programming models, the most popular of which is OpenMP [102]. Such programming models, with the support of their associated runtime system, allow programmers to take advantage of parallel hardware by simply adding annotations to existing sequential

code. The compiler then turns these annotations into calls to the programming model's runtime system library to handle the synchronisation and correct execution of the work on the parallel resources available on any given hardware. Thus, most of the complexity of dealing with sophisticated parallel hardware can be encapsulated within the runtime system library, improving programmability for the end user.

As the parallelism and complexity of hardware has increased, task-based dataflow programming models have gained in popularity. Such programming models provide more scope for the runtime system to organise the execution than in traditional fork/join based parallel programming models. This allows task-based dataflow programming models to better bridge the gap between the explosion in complexity at the hardware level and the need for ease of programmability for the user. The case for task-based dataflow programming models is already very strong given the benefits they can provide (detailed in Section 2.4) in dealing with the complexities of currently available computer hardware.

In this thesis, following a hardware/software co-design approach, we harness the information already available within today's task-based dataflow programming models to drive optimisations within the hardware architecture. Specifically, we consider how the programming model may be exploited to address power, performance and area challenges in the memory hierarchy in novel ways. Next we present the objectives and contributions of this thesis, followed by an outline of the subsequent chapters.

## 1.1 Thesis Objectives and Contributions

The goal of the thesis is to harness information which is already present in today's task-based programming models and, from within the associated runtime system, use it to direct a more efficient implementation of the memory hierarchy in hardware. We target a range of high performance computing and data-centric workloads with these proposals. The approach provides the potential for a range of significant benefits within the memory hierarchy, without imposing any new burden, or impact whatsoever, from the programmer's perspective.

### 1.1.1 Reducing Coherence Traffic with a NUMA-Aware Runtime Approach

The first contribution of this thesis investigates performance issues involved in modern very large ccNUMA architectures, including techniques to scale the cache and memory hierarchy

on both the hardware and software side. This contribution directly, and in detail, characterises cache coherence and memory traffic in a real system with workloads relevant to high performance and data-centric computing. It relates this cache coherence and memory traffic to performance and assesses the effectiveness of combining runtime managed scheduling and data allocation techniques with hardware approaches designed to minimise such traffic. We use a very large ccNUMA architecture, the Bull bullion S server platform, to make our analysis. The bullion S platform utilises a sophisticated ccNUMA architecture including a Bull proprietary Application-Specific Integrated Circuit (ASIC) processor called the Bull Coherence Switch (BCS) which implements a hierarchical coherence directory to enable scaling cache coherence to a larger system size than supported by Intel Xeon CPUs alone.

This contribution uses the measurement capabilities provided by the BCS to perform a direct, fine grain analysis of the memory and coherence traffic within the system. This analysis distinguishes between the effects of NUMA-aware data allocation and NUMA-aware work scheduling to give a complete picture of how each affect both performance and data movement within the system.

Specifically, this contribution comprises a complete performance analysis of a very large ccNUMA architecture comprised of 16 sockets of Intel Xeon CPUs 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, we perform a detailed measurement of the coherence and memory traffic within the 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 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 contribution also shows that the 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.

### 1.1.2 Runtime-Assisted Coherence Deactivation in Task Parallel Programs

The second contribution of this thesis presents *Runtime-assisted Cache Coherence Deactivation* (RaCCD), a hardware/software co-designed approach that leverages information present in the runtime system of task-based programming models to drive a more efficient hardware cache coherence design. RaCCD relies on the implicit guarantees of the memory model of task-based dataflow programming models, which ensure that, during the execution of a task, its inputs will not be written by any other task, and its outputs will neither be read nor written by any other task. As a result, coherence is not required for input and output data of a task during its execution. In RaCCD, the runtime system is in charge of identifying data that does not need coherence by inspecting the inputs and outputs of the tasks. Before a task is executed, the runtime system notifies the hardware about the precise address ranges of the task inputs and outputs. Using simple and efficient hardware support, RaCCD deactivates coherence for these cache lines during the execution of the task. When the task finishes, the runtime system triggers a lightweight recovery mechanism that invalidates the non-coherent cache lines from the private caches of the core that executed the task. As a consequence, RaCCD reduces capacity pressure on the directory, allowing for smaller directory sizes. We extensively explore reduced directory sizes with RaCCD and propose a mechanism to dynamically calibrate the directory size to reduce energy consumption without harming performance.

This contribution makes the following advances beyond the state-of-the-art: (i) A mechanism to deactivate cache coherence driven by runtime system meta-data regarding task inputs and outputs. This mechanism requires simple and efficient architectural support, and avoids the complexity, scalability and accuracy problems of other solutions. (ii) An extensive evaluation that demonstrates the potential of RaCCD to reduce capacity pressure on the directory. RaCCD reduces directory accesses to just 26% of those in the baseline system. Moreover, it allows reducing the directory size by a factor of $64\times$ with only a 2.8% performance impact on average. This results in 93% and 94% savings in dynamic energy consumption and area of the direc-

tory, respectively. (iii) An Adaptive Directory Reduction (ADR) mechanism to calibrate the directory size during execution to save energy consumption in the directory without harming performance, achieving an 86% saving in dynamic energy consumption in the directory.

### 1.1.3 Runtime-Directed Last Level Cache

The third contribution of the thesis is *Runtime-Driven Last Level Cache (LLC) Management* (RDLLCM). RDLLCM is a hardware/software co-design proposal, building on the same principles of task-based programming models which were utilised in the second contribution. While in the second contribution the goal was to improve the scalability of maintaining coherence among multiple private caches by deactivating coherence for data which did not require it, the third contribution has a wider scope within the on-chip memory hierarchy. LLCs typically now make up the largest share of area of any on-die component, reflecting their importance in mitigating the high cost of off chip memory accesses. As the effects of the memory wall persist and the computational resources within a single processor die continue to increase through thread parallelism, vector width and compute specialisation, the demand on LLC as an enabler for increasing compute intensity becomes a crucial factor in the design. In this contribution, we seek to use semantic information about the use of data already captured in existing task-based programming models, in order to drive efficiency in the handling of data within the memory hierarchy. RDLLCM operates on the inputs and outputs of tasks in a task-based programming model. These task inputs and outputs are known as task *dependencies*.

This contribution pursues three broad optimisation strategies in the memory hierarchy, based on this information. First, data which the runtime system predicts will not be reused in the execution is not allocated in the LLC at all, instead it is fetched into the private L1 caches directly from the memory controller, bypassing the LLC entirely. This removes the latency of a fruitless allocation in the LLC, but more importantly, it reduces demand on LLC capacity, leaving more of the LLC resource available to data which will benefit from being allocated there.

Secondly, as per chip core counts continue to increase, so does the diameter of the Network-on-Chip (NoC) that connects the cores, private caches and NUCA LLC slices to each other. All else being equal, this results in wider variability and slower average and worst case NUCA costs when accessing the LLC. To mitigate this problem, for task dependencies which are shared read-write or non-shared read-only, RDLLCM allocates the data in the local LLC slice of the core using the data for the duration of the task. This is enabled by the pre-existing operational

model of the runtime system, which guarantees the synchronisation of tasks so that no races occur on the tasks dependencies during its execution.

Thirdly, and also with the goal of reducing NUCA costs, for shared read-only task dependencies RDLLCM performs replication of the dependency's cache lines in the LLC. Replicating shared data in the LLC allows the cores sharing the data to each access a copy of the data in an LLC slice closer to the executing core than otherwise possible. Data replication in the LLC is not an attractive choice typically because introducing multiple copies of a cache line in a shared LLC would necessitate infrastructure to maintain coherence among the copies, and such infrastructure itself would suffer from scalability issues as all coherence mechanisms do. RDLLCM can safely replicate cache lines for task dependencies which are shared-read only without any hardware infrastructure for maintaining coherence among the replicated copies, due again to the guarantees provided by the task-based programming model. RDLLCM delivers a $1.14\times$ speedup over the baseline system on average, whereas an enhanced variant of a state-of-the-art proposal from the literature for reducing NUCA costs in the LLC achieves only $1.11\times$.

## 1.2 Thesis Outline

The contents of this thesis are organised as follows:

- Chapter 2 presents the background and state-of-the-art in the hardware and software topics upon which this thesis builds.
- Chapter 3 presents the experimental methodology used within the three contributions of this thesis.
- Chapter 4 presents the first contribution of this thesis, which analyses how a runtime system can be used to reduce coherence traffic in a very large real ccNUMA system.
- Chapter 5 presents the second contribution of this thesis, it optimises directory based cache coherence using information from the runtime system.
- Chapter 6 present the third contribution of the thesis, a proposal for Runtime-Driven LLC Management.
- Finally, Chapter 7 concludes by summarising the contributions of this thesis, listing the publications resulting from it and considering what future potential research directions it suggests.

# Chapter 2
# Background

This chapter presents the background and outlines the related work in the topics this thesis will address. Section 2.1 gives an introduction to the relevant foundational concepts in the memory hierarchy. Section 2.2 discusses more advanced work which builds upon the concepts introduced in Section 2.1, including the state-of-the-art in the specific areas this thesis will contribute to. Section 2.3 provides context for the software side of this thesis' hardware/software co-design approach. Finally, Section 2.4 details other recent work which, like this thesis, pursues a closer collaboration between runtime systems and hardware architecture.

## 2.1 Memory Hierarchies in Multiprocessors

Across application domains and in market segments ranging from smartphones to supercomputers, computer programs have a voracious appetite for memory. In addition to large capacity, the other primary concern for computer memory is that it should be fast. There are many different technologies available today with which to implement computer memory. Unfortunately, choosing among these options means dealing with an inverse relationship between speed and capacity, due to several concerns such as cost, power, wire delay and microarchitectural implementation. Therefore, the faster the memory technology, the less of it available in memory design.

In response to this inverse relationship between speed and capacity in memory design, almost all computers today rely on a hierarchy of different types of memory technologies working together to achieve a blend of speed and capacity at the right cost. Typically, these different memory technologies within a single system are organised into a hierarchy. As you move closer to the cores, each level of the hierarchy provides a smaller and faster type of storage as can be seen in Figure 2.1.

The most fundamental distinction to make between the different levels of the hierarchy is that between main memory and caches. The level of the hierarchy furthest from the cores is

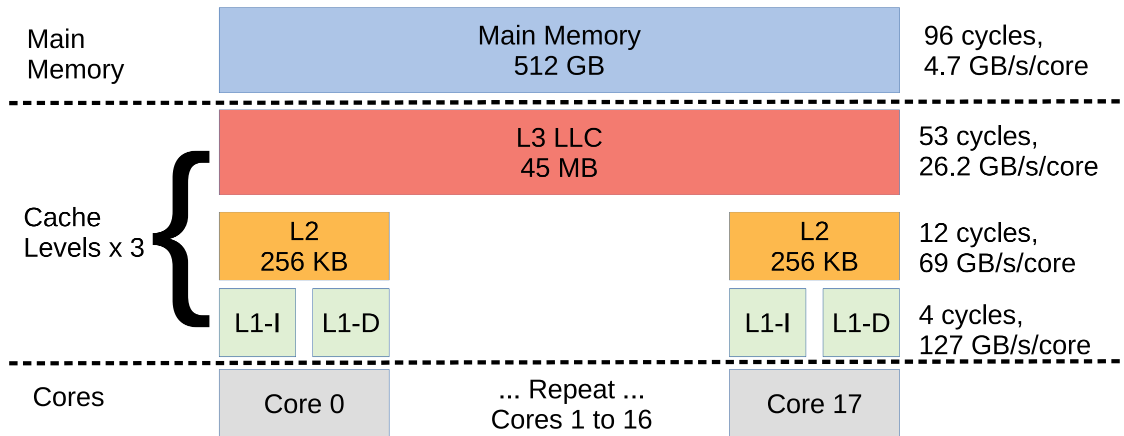



Figure 2.1: Logical view of Intel Haswell Generation Xeon-EX Memory Hierarchy with Bandwidth and Latency per level

main memory, while all the layers between main memory and the cores are different levels of cache memory. Main memory is explicitly exposed to software as a resource for the software's working data set, whereas the cache layers are not explicitly exposed to the software. The caches are managed and operated by the hardware and can only hold copies of a subset of the data that is already stored in main memory. The caches can provide the data they contain to the cores much faster, and with higher bandwidth, than main memory can. In recent decades, in the search for increased performance, memory hierarchies have grown deeper (i.e. more levels) and become more sophisticated, both within each level, and in the interface between the various levels.

All commercially available processors ranging from those found in wearable devices up to the largest data-centres and HPC systems employ *virtual memory* managed by the Operating System (OS). Virtual memory enables flexibility and security in the OS's management of memory. Furthermore, virtual memory allows the OS to provide the illusion of a larger memory capacity than the amount of physical memory capacity in the system. It achieves this by using a backing store such as a hard disk to swap chunks of data in and out of memory as they are used or idle during execution. Under virtual memory, the OS provides each application with its own private virtual address space. The OS maps each application's private virtual address space to the available physical memory in the system in chunks called memory pages (typically 4KB

to 2GB in size). The mapping from virtual to physical page addresses is stored in a software structure within the OS called the page table. In the remainder of this section we introduce the other main foundational aspects of the memory hierarchy which this thesis builds upon.

### 2.1.1 ccNUMA architectures

The first computer systems with multiple processors sharing memory appeared in mainframe computers in the late 1960s, such as the IBM 360 Model 65 MP [70]. In those early days of parallel computing it was challenging to take advantage of parallel processing due to poor support from operating systems and programming tools. It was not until the 1980s that multiprocessor systems became more successful, as they became easier to use [67]. In these early systems which popularised shared memory multiprocessors, the processors were typically connected by a bus to a single monolithic shared memory resource [17], exemplified by popular machines like the Synapse Multiprocessor System [63] and the Silicon Graphics 4D-MP [13]. Under this approach, retrospectively known as UMA (uniform memory access), each core experiences the same bandwidth and latency when accessing the shared memory resource. However, UMA systems were bottlenecked by multiple cores contending for the single shared memory resource and could only accommodate a small number of processors sharing memory.

In response to this, the natural first step was to start to parallelise the memory subsystem, as well as the cores themselves. Early examples of such systems were the Stanford DASH computer [85] and SGI Origin [83], which physically distributed the shared main memory in addition to the processors. They connected each processor plus its local main memory together via a network. This overcame the contention problem of UMA designs, but as a side effect the uniformity of access to shared main memory was broken. A processor accessing its local portion of the main memory would see lower latency and higher bandwidth than a processor accessing a portion of main memory via the network. The scaling advantages enabled by such NUMA designs far outweighed the drawbacks however, and NUMA designs quickly came to dominate.

While NUMA designs originated where multiple socketed processors were used in a single shared memory space, today we now also see NUMA designs within a single socket. One example is the recent multi-chip module packaging solutions employed by AMD [131] within a single socket. Some products, such as Intel's Xeon CPUs even employ NUMA designs within a single silicon die [77], in a feature called *cluster-on-die*. This cluster-on-die feature was prompted by the elaborate NoC implemented to connect increasing numbers of cores, caches and memory controllers present on a single die. As computer architects seek to further scale the

number of cores sharing main memory into the foreseeable future, the importance of NUMA hardware, and its implications for software performance, will increase.

### 2.1.2 Cache hierarchies

Caches store copies of data in memory units called cache *lines*. Cache lines are typically 8 to 16 times the word size of the computer. When a core requests a word from memory, first the L1 cache level is looked up. If the cache line is found, it is returned to the requesting core, otherwise each of the next levels in the memory hierarchy is looked up in turn until the cache line is found, which in the worst case will be at the main memory. When a requested memory location is not found in any cache level and is retrieved from memory, the entire cache line containing the memory location and its neighbouring locations in the same cache line are copied into the caches before being furnished to the requesting core.

Caches exploit the fact that computer programs exhibit locality of reference. Locality of reference describes two phenomena observed throughout the history of computer programming: computer programs do not access the data within their working set in a random pattern. They typically re-reference the same data which they recently referenced (temporal locality) and they typically reference data in nearby memory locations to those recently referenced (spatial locality).

The caches at each level of the memory hierarchy are usually a small fraction of the capacity of the cache at the next level, and the largest cache level's capacity is usually a small fraction of the memory allocated during execution and stored in main memory. Despite this fact, due to locality of reference, the caches can serve a much larger proportion of the memory references requested by a program than suggested by the caches' size relative to the total working set size in main memory.

In order to mitigate the effects of the Memory Wall and provide for the demands of increasing core counts, cache hierarchies have continued to become deeper and more sophisticated over recent years. Typically, each core is supplied data by one or two levels of fast private caches with small capacity. At the next level of the hierarchy the private caches are in turn supplied by a much larger LLC which consumes a high proportion of the die area.

Keeping the one or two levels of small and fast caches nearest the core private means they can keep the core supplied with data at a high rate, free from interference from other cores. Smaller caches are faster than larger caches (for any given technology used to implement the caches) due to the fact that all caches are associative structures which contain <address, data> pairs and are looked up by some form of search by address. Therefore the smaller the cache,

the quicker a lookup can be completed. This has prompted the adoption of 2 levels of private caches in many multiprocessors, with a faster, smaller private L1 backed by a larger, slower private L2.

Almost ubiquitously in modern architectures there is an LLC which is much larger in capacity than the caches nearer the cores. Given the large size of the LLC it is important to make it as flexible a resource as possible so that it may be of benefit in as many use cases as possible. In pursuit of this goal the LLC is typically a shared resource among all cores. This means when demand for LLC capacity is unbalanced among the cores sharing it, cores with high demand may benefit from capacity unused by cores with low or no demand. Secondly, it allows inter-core communication at the LLC level, if there were no shared cache within the memory hierarchy, all inter-core communication would occur via off-chip accesses to main memory. The drawback of this sharing of the LLC is that any given core can see a large variance in the performance quality of the LLC depending on how other cores are utilising it.

Accesses to a cache are either cache *hits* or cache *misses*. If a copy of the memory address accessed is present in the cache, the access is a cache hit. If a copy of the memory address accessed is not present in the cache, the access is a cache miss. On a cache miss the access must proceed to the next level of the memory hierarchy, and thus incurs a latency penalty compared to a cache hit. Cache misses can be categorised into three broad groups which are useful for the design and analysis of caches. These groups are known as the *three Cs* [68] [67]: Compulsory, Capacity and Conflict misses. Compulsory misses occur on a program's first access to a cache line. As a cache line must be accessed before it is moved into a cache, this first access is a compulsory miss which cannot be avoided. Capacity misses occur when the program accesses more cache lines than the cache can hold. In this case, even if the cache is fully associative (any cache line may be placed in any position in the cache), it must evict some cache lines in order to make room for others. If such a fully associative cache later re-accesses a cache line it previously evicted, this is a capacity miss. Conflict misses are any extra misses introduced by reducing the associativity of a cache from fully associative to *n-way associative*. In an n-way associative cache, any given cache line can occupy up to n ways (or positions), which limits the time needed to find it during a lookup in the cache. N-way associative caches are constructed as multiple *sets*, each containing n ways, and implementations today typically range from 2-way up to 16-way, depending on the requirements and location of the cache in the memory hierarchy. In an n-way associative cache, conflict misses misses can occur if too many cache lines map to the same set, even though there may be spare capacity in other sets in the cache.

In order to minimise misses (which cause expensive off chip accesses to main memory) LLCs use more sophisticated designs than the private caches. For example, the associativity of LLCs must scale with the number of cores in the system to avoid increased conflict misses in the cache. This is expensive to implement from a latency perspective as increased associativity means increased lookup time. Also, LLCs typically implement sophisticated cache replacement policies which try to minimise evictions of data that will be reused in the future. Ultimately, unlike in smaller private caches, in large shared LLC design, some increased latency from such sophisticated design elements is acceptable in order to reduce the number of misses that must be serviced by main memory.

### 2.1.3 Maintaining Cache Coherence

As already noted in Section 2.1.2, the purpose of cache memories is to provide faster to access *copies* of data that is already present in main memory. In typical modern cache hierarchies, there may be multiple cached copies of a particular cache line across the multiple vertical levels of the cache hierarchy. In addition there may also be multiple cached copies of a particular cache line within a single level of the memory hierarchy. LLCs are almost universally implemented as a shared cache, available to all cores. Such a shared cache usually permits only a single copy of a cache line to exist in it. However the smaller faster caches typically used to implement the one or two cache levels closest to the cores are typically private caches, accessed only by their local core. In these private cache levels, each private cache within the level may concurrently store a copy of a particular cache line.

When a core performs a write, caches enable that write to be stored, at the lowest latency possible, in the copy of the cache line in the private cache nearest the core. This copy of the cache line is updated with the write before it is reflected anywhere else in the caches or memory. This is perfectly legal from the software's perspective, as long as the software can never see any out of date copy of the data, if it exists somewhere in the caches or memory. The design of cache hierarchies follows the principle to *make the common case fast* [67]. Usually, when one core writes to a memory address, the written value is not required immediately at another core and the private cache allows the write to proceed without incurring the latency penalty of immediately updating other remote copies of the cache line in other caches or at main memory.

This leaves the issue of how to ensure that copies of a cache line which contain out of date values can never be seen by the software. This problem is known as *cache coherence*. A practical and useful characterisation of cache coherence is provided by [119] in the form of two invariant properties it mandates within the memory hierarchy:

1 Single Writer Multiple Reader (SWMR) invariant: For any memory location A, at any given time, there exists only a single core that may write to A (and can also read it), or, some number of cores that may only read A.

2 Data-Value invariant: The value of the memory location at the start of an epoch is the same as the value of the memory location at the end of its last read–write epoch.

The SWMR invariant is quite straightforward and intuitive. It defines the only two allowable conceptual states a cache line may be in. The Data-Value invariant is a little more subtle. It uses the term *epoch* to define the period of time during which the cache line is in one of the two allowable states, without changing state. Therefore, as seen in Figure 2.2, an epoch is delimited at its start and its end by a transition from one of the two allowable SWMR states to the other SWMR state. Or, stated another way, every transition between the two allowable SWMR states represents the start of a new epoch.

The Data-Value invariant explicitly mandates a property which might at first be taken for granted: the last write to a cache line during a read-write epoch must be propagated to the subsequent epochs. This requirement is not as trivial as it may first seem, because in parallel processing a read-write epoch may occur on one core, and the next epoch for that cache line may take place on a different core. Therefore, to implement cache coherence where we have multiple cores sharing memory, there must be a system whereby all the cores in the system can collaborate in order to collectively maintain the SWMR and Data-Value invariants as any core in the system can access any memory address.

In order to maintain the two invariants, all commercially available cache coherence implementations today follow an invalidation based approach to handle writes. When a particular core, *P*, wishes to write to a memory location, if its private cache does not already possesses a read-write copy of the cache line, core *P* must start a new read-write epoch for the cache line containing the memory location to be written to. In order to do this and satisfy the SWMR and Data-Value invariants, core *P* must invalidate any copies of the cache line currently possessed by other cores' private caches, whether they are read-only or read-write copies. Once this is complete, no other core has access to a copy of the cache line, and core *P* may begin its read-write epoch for the cache line.

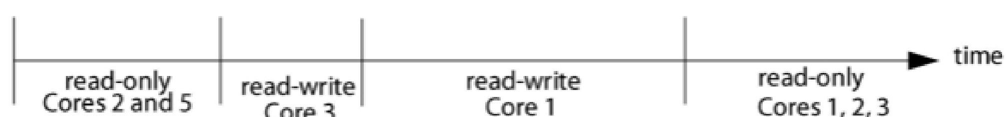



Figure 2.2: The division of a single cache line's lifetime into coherence epochs

Cache coherence is implemented in hardware by a system called a cache coherence protocol. The earliest distinction made in the design of cache coherence protocols is between *snooping based* and *directory based* cache coherence protocols. Both approaches can achieve the same goal, but they differ significantly in their implementation and scalability. Snooping based coherence protocols [7] are the most simple and straightforward to implement, and dominated early cache coherent systems. Snooping based protocols are so named because they involve all the cores in the system watching or *snooping* all the private cache misses of all the other cores in the system, all the time. Snooping protocols were a natural fit for early cache coherent systems, which employed a shared bus implementation to connect all the private caches together with main memory, or a shared LLC. While a shared bus is simple and cheap to implement, it forces a *broadcast* based communication pattern, where each cache connected to the bus sees (or *snoops*) all the accesses made to it by all the remote caches. More formally, a broadcast message means the message is received by all the entities connected to interconnect.

In a broadcast based snooping coherence protocol, when a private cache miss occurs, the private cache must broadcast its request for the memory location to all the other caches in the system, and all those caches must check whether the request impacts them or not. Purely broadcast based snooping protocols like those used on early bus based designs suffer an inherent and fundamental lack of scalability caused by their communication pattern, and are therefore not used outside of small numbers of parallel cores sharing memory.

Directory based protocols [1] were popularised during the move away from shared bus interconnects to point-to-point NoC interconnects. NoCs can support broadcast based communication patterns but they additionally allow unicast (one sender to one receiver) and multicast (one sender to multiple receivers, but less receivers than a broadcast) messages between any entities connected to the NoC. Directory based protocols remove the need for broadcast communication by introducing a *directory* which tracks metadata on the location and status of cache lines among the different caches in the system. When a cache wishes to request a cache line it doesn't currently have, it contacts this directory with a unicast message. Based on the information stored in the directory for the requested cache line, the request can be satisfied with a limited number of further unicast messages, depending on the current state and locations of any copies of the cache line already present elsewhere in the memory hierarchy. The drawback of directory based protocols compared to snooping based protocols is that they require additional area to store metadata and additional latency to lookup that metadata, in comparison to snooping based protocols.

As almost all modern architectures beyond the smallest system sizes implement point-to-point NoCs rather than shared bus interconnects, there is no longer a clear dichotomy between archetypal snooping and directory based coherence protocols. In fact, there now exists a continuous spectrum of design points between these two extremes. For example, to save area, coherence directories can store inexact [65] [1], rather than exact, metadata about cache lines. This results in some level of communication (multicast) greater than exact metadata would require (unicast), but less than archetypal snooping (broadcast). From the opposite end of the spectrum, snooping based protocols can implement small and simple *snoop filters* [97] which store metadata that can be used to filter what would otherwise be a broadcast message down to a multicast or unicast message. All these design points along the spectrum of coherence protocols between archetypal snooping and directory protocols trade off requirements for network bandwidth with the area and latency required to store and lookup metadata about cache lines. Solutions to the challenges [92] of scaling directory based cache coherence is an important area of current research and motivates the second contribution of this thesis.

## 2.2 Scaling Cache and Memory

This section presents the relevant state of the art techniques applied within the memory hierarchies of modern systems, upon which the contributions of this thesis advance.

### 2.2.1 Scaling ccNUMA

NUMA architectures were introduced to alleviate the bottleneck experienced by early shared memory systems which comprised a single monolithic main memory. NUMA sought to decompose the monolithic main memory of such architectures and physically distribute it in the system. Physically distributing the main memory allowed multiple parallel streams of memory accesses and improved scalability. However, it also caused side-effects such as non-uniform performance depending on which physical memory location was being accessed and the path from the accessing core to it.

All parallel shared memory systems available today that use NUMA main memory designs also use coherent caches. The performance effects encountered in such ccNUMA designs do not depend solely on the non-uniform network distances between the cores and the physically distributed main memories in the system. Performance of memory accesses also depend on the increasingly deep and sophisticated cache hierarchies and their coherence protocols. For example, when a core needs to access a particular memory location, in the first instance it will

contact the physical home location for that memory in the physically distributed NUMA main memory. However, in the presence of caching the currently valid copy of the data may not be in the main memory. The valid value for the memory location may in fact reside in a copy of the memory location in a cache which is physically remote to both the requesting core and home location in main memory. In this case the coherence protocol needs to communicate from the cache holding the valid copy to the copy's home in main memory before the valid value can be forwarded to the requesting core. Therefore, the latency and bandwidth experienced by the requesting core does not solely depend on the route between the requesting core and the home location of the data in main memory. It also depends on the structure and implementation of the cache hierarchy and its coherence protocol.

As the complex performance effects arising from ccNUMA hardware depend both on the architecture of the hardware and how the software uses it, scaling performance of ccNUMA designs is a problem which concerns both hardware and software design. Software based approaches to scaling memory performance in ccNUMA systems are rooted in the concept of *locality*. Locality refers to the distance between where the computation takes place in a core, and where the data required for the computation is stored. In order to achieve optimal data locality, there must be co-ordination between where a particular computation takes place and where the data it requires is allocated so that the distance between the two is as small as possible. For example, in many multiprocessor ccNUMA architectures, every processor socket forms a local group with attached local main memory and is connected to the other remote groups of processors plus main memory via a system interconnect. The optimal data locality in this case means computation accessing data in its own local portion of the shared main memory.

The potential to achieve data locality depends in the fist instance on the application and the pattern of data accesses it requires. The vast majority of parallelised software applications do not access the shared global memory in a random fashion, and therefore exhibit patterns in their memory accesses which may be exploited to increase locality during their execution.

There has been extensive research on locality-aware software. Work on fostering locality is as widespread throughout the software ecosystem as are techniques for managing parallelism. This include approaches at the Operating System, Runtime System, Compiler and Application Levels of the software stack. At the operating system level approaches include [11] modifying the system page allocator to make it NUMA-aware. This is achieved through an Adaptive First Touch allocation policy which chooses which NUMA node to place a physical page in based on minimising a cost function which considers NUMA distance from the core first touching the page and contention within the memory subsystem. To handle changes in access patterns

during runtime a dynamic page migration technique is used where a physical page is moved to a different NUMA node when a threshold for poor locality of accesses to the pages is crossed.

Other Operating System based approaches propose modifying the Linux scheduler to mitigate NUMA penalties [20] caused by the scheduler trying to load balance to minimise memory contention. Also modifying the Linux scheduler [51] propose a technique to migrate and map threads to cores dynamically for locality, based on logging the source core of memory accesses at the Page Table.

At the runtime system level there have been proposals such as ForestGOMP [23] for fork/join parallelism in OpenMP. ForestGOMP seeks to group threads according to their affinity and migrate data in order to mitigate NUMA effects. It uses information provided within the OpenMP annotations of the application to drive decisions about how best to deliver data locality.

On the hardware side there are also approaches which seek to optimise the data locality through data allocation and affinity. The Translation Lookaside Buffer (TLB) is a hardware structure which is used to cache entries from the operating system's page table, and thus provide fast access for the cores to virtual to physical address mapping. Cruz et al. [46] track patterns between threads and the pages they access at the TLB and uses this information to assign threads to cores with maximum data locality. Tikir and Hollingsworth [123] proposes a hardware counter based profiling technique to detect the pattern of accesses in the memory hierarchy, and make decisions about migrating memory pages.

### 2.2.2 Scaling Directory based Cache Coherence

This section describes state of the art proposals for scaling directory based cache coherence. Three broad categories of techniques are detailed: cache coherence deactivation, microarchitectural cache coherence optimisations and finally, hardware/software cache coherence optimisations.

#### 2.2.2.1 Cache Coherence Deactivation

In recent years, significant work has been done in reducing the complexity and cost of hardware based cache coherence. A fundamental observation of many approaches in this area is that coherence is only required to correctly handle data races. Such data races only happen to *shared* data that is concurrently accessed by multiple cores, at least one of which will write to it. In contrast, when data races do not take place, coherence can be relaxed. Such data

race-free situations exist in the case of *private* data that is only accessed by a single core during the execution, *shared read-only* data that is never modified during the execution, and *temporarily private* data which is accessed from only a single core during a definable period of the execution, before being accessed by another core. In the rest of this thesis, we refer to the private, shared read-only and temporarily private data which we may deactivate coherence for as *non-coherent* data.

The categorisation of data as coherent and non-coherent can be used for *coherence deactivation*. Coherence deactivation is a technique that avoids the tracking of non-coherent lines in the directory, in order to address the scalability challenges [62] [92] posed by directory based cache coherence. Consequently, directories exploit their capacity more efficiently and, as long as the data classification mechanism is accurate, the number of entries required in the directory can be drastically reduced. This can be used to reduce expensive directory cache conflicts [94], save area and energy consumption in the directory in addition to reducing directory conflict induced cache misses [47], or re-purpose directory cache area for other purposes such as increased data cache capacity [43]. To deactivate coherence, cache misses for non-coherent lines in the private caches request data from the LLC or main memory without accessing the directory. Therefore, no coherence actions take place for this data. This requires triggering a recovery operation to avoid inconsistencies when non-coherent data becomes shared or when temporarily private data migrates from one core to another. A more aggressive solution is to self-invalidate the non-coherent lines of the private caches [84], so they never need to be tracked in the directory.

State-of-the-art techniques to identify non-coherent data use TLB and Operating System (OS) page table support [66, 80, 47, 113, 111]. These approaches require changes in the page table and the TLBs to monitor TLB misses and categorise data as shared, private, or shared read-only. When a page is accessed for the first time, it is categorised as private, and the OS sets a private bit in the page table and in the TLB of the accessing core. When another core accesses the page, the OS marks the page as shared and triggers a flush of the cache lines and the TLB entries of the page in the first core. Subsequent accesses by any core to the shared page trigger the coherence actions defined by the cache coherence protocol. A limitation of this approach is that, once a page is categorised as shared, it never transitions back to private, so temporarily private pages are categorised as shared. This problem is particularly important in the presence of dynamic schedulers, where private data often migrates between cores. In addition, this data classification method works at a page granularity, which can lead to misclassified lines.

Extensive and costly modifications in the system are required to identify temporarily private data in TLB based approaches [60, 59, 58]. The idea is to, upon a TLB miss, check if the page

is present in some other TLB and categorise the page as private if the page is not present in any other TLB. Doing this requires TLB-L1 inclusivity, very complex hardware support to perform TLB-to-TLB miss resolution, and costly TLB invalidations during page re-classifications. In addition, this technique is still not accurate because TLB entries may suffer from *dead time*, that is, the elapsed time since the page is last accessed until it is evicted from the TLB. This can be solved by adding a *decay* mechanism that predicts if the page is going to be accessed again and invalidates decayed TLB entries during the TLB-to-TLB miss resolutions. This solution introduces performance overheads due to extra TLB misses, and it also requires additional hardware support in the TLBs and modifications to the TLB coherence protocol.

#### 2.2.2.2 Microarchitectural Cache Coherence Optimisations

In response to the scalability challenges [62] [92] of directory based coherence, many works propose to re-think the directory organisation to reduce its size. These techniques are less aggressive than coherence deactivation because the data categorisation is done in the directory itself, so private lines are still tracked in the directory. Gupta et al. [65] introduce vectorised directory entries to track coherence at a coarse grain. Choi et al. [40] propose a similar solution that uses segments instead of vectors. SPACE [135] stores sharing patterns in a separate table and each directory entry stores a pointer to this table. SCD [117] uses a hierarchical approach where root entries contain pointers to potential sharers, reducing the space required for large sharing vectors. Tagless coherence directories [133] use bloom filters to track private lines, which reduces the storage requirements but has a high cost in power. Cuckoo directories [62] propose a hashing technique that reduces the conflict misses in sparse directories. RegionScout [98] and Cantin et al. [27] track coherence at a coarse granularity in a separate hardware structure and filter broadcasts in snoop based cache coherence protocols. Alisafaee [3] proposes modifying the directory to track shared, private and temporarily private data and compacting consecutive private lines in one single region entry. Zebchuk et al. [132] refine the previous idea by presenting an improved hardware design that fixes some race conditions and reduces the NoC traffic and energy consumption.

The ARMv8 architecture [8] allows the specification of shareable domains, i.e., memory regions private to one core or shared among cores. The way Arm processors exploit shareable domains is implementation dependent. To the best of our knowledge, they are mostly used to define coherence domains in clustered multicores and in heterogeneous processors with integrated GPUs. In addition, Arm processors typically specify such domains at boot time, so they are not intended to dynamically deactivate coherence in parallel programs.

#### 2.2.2.3 Hardware/Software Cache Coherence Optimisations

As explained in Section 2.2.2.1, TLB and OS support can be added to identify non-coherent data and to deactivate coherence for private pages [47], shared read-only pages [48], and temporarily private pages [60, 59, 58], although the latter requires extensive hardware support in the TLBs. VIPS [113] uses the TLB based data categorisation to apply a write-back policy to the private data and a write-through policy for the shared data, which allows to simplify the coherence protocol to only two states (valid/invalid). TLB based classification is also used to filter snoop requests [80], to optimise the data placement in NUCA caches [66], and to partition the cache hierarchy according to the requirements of the applications [124].

Some works optimise cache coherence by exploiting the semantics of Data-Race-Free (DRF) programming models such as Deterministic Parallel Java [22]. DRF programming models add well-defined synchronisation points and ensure no data races will happen to any data between two synchronisation points. However, unlike OpenMP, DRF programming models do not offer means to differentiate shared and private data.

VIPS-M [113] classifies shared and private data in the TLBs and exploits the DRF properties to eliminate the directory. To do so, VIPS-M uses a write-through policy for shared lines while, for private lines, it deactivates coherence and self-invalidates them from the private caches at synchronisation points. Due to the lack of directory, VIPS-M needs to implement synchronisation in the LLC, which has a very high cost in large-scale multicores. VIPS-H [111] extends VIP-M to hierarchical coherence, tracking shared and private data at the different levels of the hierarchy to cut-off the forwarding of self-invalidations.

DeNovo [39] exploits the DRF semantics to eliminate the transient states of the cache coherence protocol. This reduces the number of states of the protocol, makes it easier to verify, and reduces the size of the directory entries. However, instead of categorising shared and private data, DeNovo applies this to all the cache lines. This breaks the implementation of atomic operations, that rely on the transient states to correctly handle data races. This issue can be addressed by adding hardware support to implement synchronisation primitives [120].

Another recent approach from Ros and Jimborean [112] to improving the scalability of directory coherence is to use compile time analysis of OpenMP code to identify *data race free* (DRF) regions of the code. Where such regions can be identified, the memory consistency model can be safely relaxed from the default Sequential Consistency (SC) model during the DRF section, and SC consistency re-established on transition out of the DRF region, so that overall, the whole execution appears to obey SC. In the DRF regions, where SC can be relaxed, an optimised and simpler version of the coherence protocol can be employed safely, which

reduces demand on the coherence directory and cache misses caused by directory conflicts. The authors demonstrate that this approach can show performance improvements of 24% and reductions in energy consumption of 32%, on average, for a 64-core chip multiprocessor. The authors further extended this work in [59] to add dynamic runtime information on page management from the OS to help improve the classification of accesses. More recently the authors have also published work on extending this technique to the more general programming model of Pthreads [75] [74].

Compile-time techniques [87, 88] have also been proposed to classify shared and private data. This approach introduces a new form of `malloc` for the compiler to indicate the category of the data, and the operating system passes the categorisation to the microarchitecture during the memory allocation. The main problem with this approach is that it relies on alias analysis to determine which variables will be accessed in different regions of the code; an extremely challenging problem in non-trivial codes and, particularly, in the presence of pointers.

### 2.2.3 Scaling NUCA Last Level Cache

This section first introduces the foundational background to NUCA caches. Then it presents state-of-the-art proposals for Dynamic NUCA caches which are invisible to software and managed entirely within hardware. Finally, state-of-the-art proposals for hardware-software co-designed NUCA caches are discussed.

#### 2.2.3.1 Foundations of NUCA Caches

The foundational work on NUCA Caches was presented in 2002 by Kim et al. [79]. In it they describe a NUCA cache as a cache comprised of multiple separate physical banks, connected together by a packet switched NoC to create a single logical cache. They demonstrate the advantages of such a NUCA design over the UCA alternative which is less scalable. They successfully motivate a NUCA design based on the scalability of the cache capacity alone, in the context of a single core system. They show that as the cache size grows UCA caches become prohibitively costly to implement and suffer very low scalability. Even when serving only a single core, NUCA caches comprised of separate physical cache banks provide the benefits of reduced average access latency, reduced contention per bank and increased overall bandwidth. For NUCA Caches, Kim et al. describe two variants: *Static* NUCA (**S-NUCA**) and *Dynamic* NUCA (**D-NUCA**). These two variants of NUCA caches differ in three crucial design points which apply to all NUCA designs:

- ***NUCA-Mapping***: How to decide which physical cache bank a particular cache line should be placed in on allocation in the shared cache.
- ***NUCA-Search***: When referenced, how to locate a particular cache line within the multiple physical banks which make up the shared cache.
- ***NUCA-Movement***: The process required to change a block's location, if desired, to minimise network distance to the physical cache bank containing it.

S-NUCA is the most simple and straightforward NUCA variant possible, both in conception and implementation. It distributes the sets of the cache across the physical banks it is made up of, placing all the ways of each set within a single bank. This means the block address and set index bits from the address of a memory reference are sufficient to precisely locate a cache line within its physical bank in the cache, providing a trivial NUCA-Mapping function. As the address uniquely maps a cache line to a particular physical cache bank, no NUCA-Search logic is required and NUCA-Movement is not supported by the S-NUCA design. D-NUCA caches introduce flexibility into the placement of cache lines among the physical cache banks, at the expense of added complexity in NUCA-Mapping, NUCA-Search and NUCA-Movement.

#### 2.2.3.2 D-NUCA Caches

Since the advent of NUCA cache design there has been a considerable body of work done on D-NUCA designs. Beckmann and Wood [15] was the most important early proposal for a D-NUCA cache serving multiple cores. In it, they organise a relatively large number of NUCA banks (256) into 16 bankclusters which are shared among 8 cores. The 16 bankclusters are geographically divided into 8 local bankclusters (1 per core), 4 intermediate bankclusters (1 per two cores) and 4 shared bankclusters.

Their proposal first maps a cache line randomly within the banks based on some bits within the address. It then migrates (via inter-bank copies) the cache line among the bankclusters in a defined order as blocks are accessed, with the goal of minimising the access distance (and hence the NUCA latency). The main drawback of their approach is that while NUCA-Mapping and NUCA-Movement is relatively simple, NUCA-Search is not. When any given core wishes to access a memory reference in the NUCA cache, it does not know which, if any, bank holds a copy of the cache line. Therefore a search is required in the NUCA cache for the specific cache line. The proposal performs a two-step multicast search, first multicasting lookups to the requesting core's unique local and intermediate bankclusters plus the 4 shared bankclusters. If

this search fails, a lookup request is multicasted to the remaining 10 bankclusters. This two step multicast lookup process is an attempt to balance the latency of the search process with the load imposed on the NoC and banks. A broadcast lookup to all 16 bankclusters would reduce latency per search to a minimum, but raise load on the NoC and banks to a maximum and vice versa for a sequential unicast lookup of all 16 bankclusters.

Since this seminal D-NUCA proposal there have been several important contributions to the topic. In addition to the previously mentioned NUCA design points of NUCA-Mapping, NUCA-Search and NUCA-Movement, a fourth NUCA design point was introduced: ***NUCA-Replication***. NUCA-Replication seeks to achieve a similar goal to NUCA-Movement, namely to minimise the NUCA distance and therefore latency cost of NUCA accesses. However, instead of moving a cache line nearer to one particular core at any given moment, NUCA-Replication allows the cache line to be replicated multiple times across the NUCA banks. This is beneficial where a cache line is widely shared and does not have an affinity with any particular core in the system. Some of the early work supporting NUCA-Replication is from Chishti, Powell, and Vijaykumar [36] who proposed NuRAPID in the single-core context, and later, CMP-NuRAPID [37] for multicores. NuRAPID and CMPNuRAPID decouple tag storage from data storage in the cache. The tags are organised in per-core private storage and point to their associated data block in the distributed and shared NUCA banks. Their proposal requires forwards and backwards pointers from the tag entries to the cache line entries. Zhang and Asanovic [134] make a proposal called Victim Replication which is built on an S-NUCA baseline. Their central idea is that when an L1 cache evicts a block due to capacity pressure, instead of notifying the coherence directory as usual, it places the block in the local NUCA bank of the L1 which evicted the cache line. Since the directory still thinks the L1 is caching the line, if any other core were to try to write to the cache line, the directory would need to send an invalidation to the L1 which has already evicted the line to its local L2. If the L1 receives such an invalidation from the directory and is no longer caching the line, it must also check its local L2 and invalidate the line there if present. The victim replication in the local LLC slice is only carried out if there is spare capacity in the local LLC slice. This provides a limited and tightly constrained level of replication at low cost in added complexity. Beckmann, Marty, and Wood [14] propose Adaptive Selective Replication, a technique for replication in NUCA caches based on a probabilistic cost-benefit measure of replications. Their proposal is built from hardware counters which enable on the fly decision making in hardware on whether to replicate cache lines in a local LLC slice or not.

Huh et al. [69] proposes two techniques, one for a D-NUCA cache and one for an S-NUCA cache with replication. For a D-NUCA cache the authors propose a novel solution for NUCA-Search: storing partial cache line tags per column of NUCA LLC banks in a tiled architecture. Looking up the partial tags indicates whether that column of LLC banks can possibly contain the referenced block, without the need to lookup all the LLC banks. The added partial tag stores must be updated on each NUCA-Mapping and NUCA-Movement. The partial tag store updates and lookups cost almost half the benefit afforded by the reduced NUCA distances achieved by the D-NUCA organisation. For an S-NUCA cache the authors propose using an S-NUCA baseline but adding a configurable degree of replication (called *Sharing* Degree or SD) within the S-NUCA LLC. This enables reducing the NUCA latency through replication without the NUCA-Search or NUCA-Movement costs of a D-NUCA cache. This requires the support of a directory managing coherence among the replicants within the LLC. The authors demonstrate that different applications benefit from different degrees of replication within the LLC and therefore a dynamic configuration of the SD is desirable. As a first step they propose a simple static choice of SD as 2 or 4 to balance performance and complexity.

Merino et al. [95] introduce SP-NUCA a shared/private NUCA cache built on an S-NUCA baseline. In this proposal when a core looks up the NUCA LLC for a particular cache line, it first looks up the local NUCA bank. This represents a guess that if the line is already present in the LLC it has only been accessed by this core in the past. If the line is not already present anywhere in the LLC it is allocated in the local NUCA bank of the accessing core in the hope that it will be accessed exclusively by that core, or at least repeatedly before another core accesses it. Upon lookup in the local NUCA bank, if the line is not present there, the core secondarily looks up the NUCA bank which the line would have been stored at in the baseline S-NUCA implementation. This is where the cache line will be permanently moved to upon its first transition from being accessed by a single core to more than one core. If the cache line is not present in the local NUCA bank of the accessing core, nor in the bank it would be in under the baseline S-NUCA placement, all the other cache banks are looked up to find if the cache line is currently in the local NUCA bank of another core. If found in such a bank, it transitions from private to shared and is migrated from local NUCA bank it was found in to the NUCA bank it would be in under the baseline S-NUCA placement. If these three phases of LLC lookup (local NUCA bank, home NUCA bank, all other NUCA banks) do not result in a hit only then is an off chip request to memory sent. The drawback here is that multiple sequential lookups of the LLC are required for accesses to data that is not core private.

Das, Aamodt, and Dally [49] introduce Sub-Level Insertion Policies (SLIP). They make the observation that D-NUCA techniques such as NUCA-Migration and NUCA-Replication introduce significant energy costs as migrating and replicating lines within the cache has an energy cost which may be wasteful if the cache line is not reused, or reused very little. Their proposal focuses on reducing energy use in LLC caches by segmenting a single cache level into sub-levels of NUCA banks based on energy efficiency. They use additional hardware based cache line reuse counters with associated per-cache line metadata storage to determine the most energy efficient sub-level to insert a cache line at, based on its reuse characteristics.

Despite the large body of work devoted to hardware managed D-NUCA caches, none of the proposals to date managed to compellingly overcome the inherent problems of NUCA-Search or NUCA-Migration. All the proposals introduce significant complexity and cost in either area or latency. Furthermore, since the middle of the 2000s, a lot of the attention on optimising the organisation of NUCA caches has shifted away from the hardware-only D-NUCA proposals. Instead, focus has moved to proposals which pursue a hardware-software co-designed approach to solving the problems of S-NUCA caches, which are outlined next.

#### 2.2.3.3 Hardware-Software Co-Designed NUCA Caches

**Page Based**

Despite the large body of work on D-NUCA improvements over the simple S-NUCA cache, S-NUCA caches were still attractive propositions due to several attributes. S-NUCA caches are simple to implement, they provide fast block look up due to the absence of NUCA-Migration and they provide optimal hit rates in the LLC due to their lack of NUCA-Replication. Conceptually, the central problem with S-NUCA caches which the work on D-NUCA caches seeks to address is their sub-optimal average wire delay per access. This is due to the fact that S-NUCA caches statically place blocks in NUCA banks based on some bits in the physical block address. This will lead to the blocks being distributed evenly across the NUCA banks which make up the cache *if* the bits of the physical addresses of the accessed blocks are random. If the blocks are uniformly distributed among the NUCA banks, then the average block access distance will be half the network diameter of the NUCA network.

Rather than seek to address this issue at the hardware level as previous D-NUCA approaches had done, Cho and Jin [38] approached the problem from the Operating System, observing that block physical addresses are under the control of the Operating System's Page Table and thus need not be random. They built on previous work on improving NUMA locality via OS-based page colouring [82] [44]. This technique can orchestrate the virtual to physical page mappings

in order to place virtual pages in particular banks of memory, or LLC and offers a new way to balance the mutually exclusive benefits of S-NUCA and D-NUCA caches. In follow on work, Jin and Cho [76] propose using compiler hints to optimise OS page colouring choice using compile time analysis. The main drawbacks to the OS page colouring approach is that pages are coloured on first touch of the page by the operating system, and thus are most useful in single-threaded workloads, or multi-threaded workloads where the data is mostly private to each accessing core. Secondly, the page based colouring maps an entire page to a single NUCA bank, which can result in capacity pressure and necessitates a spilling mechanism to allow capacity from neighbouring banks be utilised to avoid performance drawbacks. Chaudhuri [35] and Awasthi et al. [10] seek to address these issues by allowing page migration (which requires copying pages in physical main memory) and formalising the cost/benefit decision making for page spreading.

The most promising page based technique in the literature is Reactive-NUCA (R-NUCA) [66]. In this proposal, rather than orchestrating the virtual to physical mapping of pages within the Operating System to control physical page placement, they classify pages at the page table based on their use and communicate this classification to the hardware via the TLB. Pages are classified into 3 types: (i) Pages containing only instructions, such pages are read-only and typically widely shared across cores. (ii) Data pages which have been accessed only by a single core (core-private). (iii) Data pages which have been accessed by more than a single core (core-shared). This classification can be performed at the operating system page table each time the page table is accessed on a TLB miss and the classification cached in the TLB with the page's associated TLB entry. The proposal details a simple and effective system to handle the three types of page classification in the LLC. Instruction pages which are typically widely shared among all cores can benefit from being replicated in the LLC to minimise the average NUCA access cost. The potential drawback of replicating data in the LLC is that it reduces the effective capacity of the cache. To balance these competing concerns of effective capacity and NUCA distance with replication, the authors propose a rotational interleaving technique, which allows for a statically configurable degree of replication within the cache. The paper studies a 16 core chip with a replication factor of 4 allowed for replicated cache lines and performs replication only for the pages it classifies as instruction pages. The classification splits data pages into those that have been accessed only by a single core (private), and those accessed by more than one core (shared). Private pages are placed in the local LLC slice of the core accessing them. Upon first access, a page will always by classified as private to the core making the first access. It will remain classified as private to that core until another core accesses it.

When this happens, during the second core's page fault handling the Operating System detects the page is transitioning from private to shared R-NUCA classification. This triggers a flush of the relevant TLB entry from the core which had accessed the page as private and also flushes the page from the LLC slice of that core. From then on the page is treated as a shared page and is placed in the cache as it would be in the S-NUCA baseline (i.e. randomly address interleaved among all the slices of the NUCA cache). The authors observe from benchmark behaviour that there is little to be gained from a more sophisticated NUCA-Mapping for such shared pages. R-NUCA improves upon the previous OS Page Colouring based techniques in that it does not interfere with the virtual to physical page mapping process, and therefore does not have any impacts requiring copies of pages in main memory when they change classification or placement within the NUCA LLC.

### Software Defined Cache

Some recent work has focused on adding increased flexibility into the organisation and operation of the static resources which make up typical cache hierarchies and can broadly be termed Software Defined Caches. These Software Defined Caches propose a more fundamental and sophisticated reconfigurability of the memory hierarchy than previous hardware only proposals which sought to decide how best to utilise multi-level cache hierarchies [118]. In Jigsaw [16], software can divide a distributed NUCA cache into constituent virtual caches and map pages to individual virtual caches. Jenga [124] seeks to increase the scope of the software defined aspect to impact the organisation of the cache hierarchy, rather than just the operation of a single level within the hierarchy targeted by Jigsaw. It uses an OS runtime layer and hardware modifications to allow the software layer to dynamically configure the resources of the cache hierarchy across multiple levels. Ni et al. [100] propose Software Defined Cache (SDC). This proposal seeks to conceive of a cache as a key-value data store, allowing data to be cached at a granularity smaller than a cache line and providing software with an interface which allows it to explicitly insert, retrieve or invalidate entries in the cache.

## 2.3 Programming Modern Multiprocessors

Most software developers begin by learning to design and implement software in a non-parallel fashion for a single thread of execution. Only once they have gained some skill in this regard can they successfully move to exploit the performance benefits offered by parallel hardware. When that time comes, the move to shared memory parallel programming is both conceptually, and in its implementation, a much smaller leap than moving to parallel programming for disjoint memory spaces. A shared view of memory frees the parallel software developer from the need to explicitly decompose both their algorithm and the data it operates on into physically separate sub-units which require explicit communication between them to achieve their collective goal.

This is why shared memory is the overwhelmingly dominant paradigm for programming systems ranging in size from multicore SoCs in wearables (such as smartwatches and wireless earbuds) up to very large scale servers containing multiple multicore sockets and terabytes of DRAM in a single operating system image. The enduring and widespread popularity of the shared memory programming paradigm in multicore and multisocket environments has prevailed now for more than four decades, despite the rapidly increasing thread level parallelism available in such systems. While the central principle of providing programmers with a shared

view of memory has been remarkably enduring, it would be a mistake to think it has not left room for innovation and development in the shared memory parallel programming interface.

### 2.3.1 Low Level Shared Memory Programming

Since the dawn of parallel programming, and to this day, Operating Systems have provided a low level interface allowing user programs to take advantage of the parallel hardware threads provided by multiple cores sharing memory. These low level OS interfaces such as POSIX threads in Linux and the Win32 threads in Windows offer the primitive operations needed to express efficient and correct multi-threaded programs.

While such OS provided interfaces allow very fine grained control, they require expert knowledge to use correctly and are not suitable for general purpose, large scale application development. In the early days of parallel programming, these interfaces were the only option available for writing parallel software. While such interfaces persist to this day, they are generally used in a very small proportion of software development, by expert programmers and for the purpose of producing software libraries or other system software, rather than end user applications.

### 2.3.2 High Level Shared Memory Programming Models

As shared memory parallel computers started to move into the mainstream in the 90s, first for government, academic and business applications, and later for consumer products in the 2000s, there was a need for a higher level, easier to use interface for general purpose application development targeting parallel shared memory hardware.

This was a problem both academia and commercial system vendors took a keen interest in. They began to use the low level interfaces to shared memory programming provided by Operating Systems (described above) to build higher level, more abstract interfaces to shared memory parallel programming. Such higher level programming interfaces could be more easily, productively and widely adopted for application development by end users.

To this end, the OpenMP standard was founded in the late 1990s, gathering together disparate efforts by vendors and academia up until then. The OpenMP standard codified a generic API for shared-memory parallel programming in high level languages such as FORTRAN and C/C++. The creators of OpenMP followed a twin track approach. First, they introduced a lightweight, easy to use method of annotating source code with directives for

parallelism. Second, via the compiler, they linked these directives to a highly optimised runtime system library created by expert programmers and performance analysts.

The OpenMP source code directives indicate how the user wishes to exploit parallelism on their source code. An OpenMP program starts off in a sequential single-threaded execution and as OpenMP directives are then encountered in the source code, the OpenMP runtime system introduces and controls a multi-threaded execution of a particular portion (i.e. code block) of the source code. This led to OpenMP being described as a *fork/join* parallel programming model. *Fork* means to start new parallel threads of execution, for example at the start of a parallel region, while join describes all the threads waiting for each other to complete their work at the end of a parallel region. After the join the parallelism is collapsed and execution proceeds again on a single thread, until the next parallel region is encountered.

For example, the most common and simple directive used in OpenMP is the `#pragma omp parallel for` directive which can preface a traditional `for` loop. This simple one line directive instructs the runtime system to divide the iteration space of the loop into *chunks*, and execute those chunks in parallel on multiple threads. Each thread, upon completion of its local chunks of work, then waits until all other threads also complete their work, at which point the threads have joined. Only then does execution again proceed, on a single thread, to the next program statement following the end of the for loop, processing statements as usual until the next parallel region in the source code is encountered.

The parallelism requested by the directives is achieved by the compiler transforming the original source code according to the OpenMP directive, and adding calls to the OpenMP runtime system library to carry out the necessary logistical operations such as thread creation, synchronisation and termination and data privatization or sharing among threads where required. Thus the OpenMP programming model provides an unobtrusive and intuitive interface to its user while the OpenMP runtime system library can hide all the low level complexity of the highly optimised parallelism from the user.

### 2.3.3 Task-Based Programming Models

Task-based programming models structure the execution of a parallel program as a set of *tasks* with dependences between them. Typically, the programmer writes sequential code and adds annotations to define the tasks and the data they access. The annotations specify the range of data accessed by each task using array sections and whether the data is read, written, or both (labelled *input*, *output* and *inout*, respectively). An example of these annotations can be seen at the colour coded points in the Cholesky code snippet in Figure 2.3. The runtime

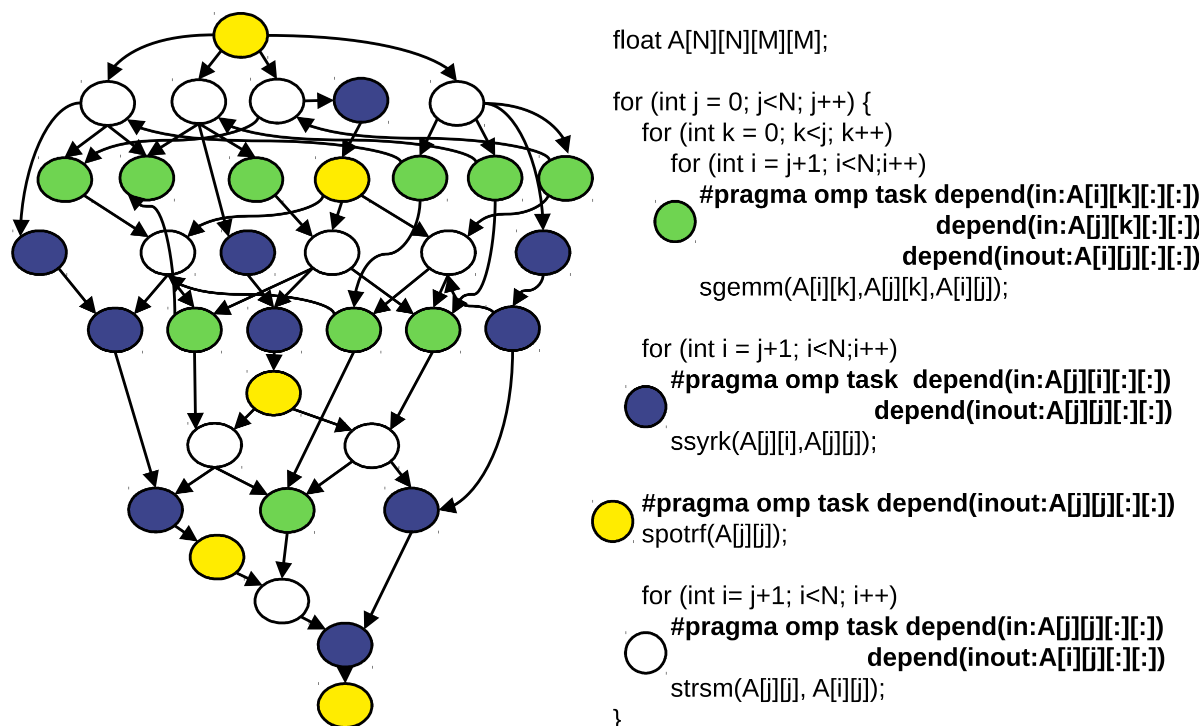



Figure 2.3: Cholesky task-based code (right) and task dependence graph (left).

system dynamically executes tasks by means of a *Task Dependence Graph (TDG)*. The TDG is a directed acyclic graph where the nodes represent tasks and the edges are data dependences between tasks. The TDG in Figure 2.3 has the tasks colour coded to match the colour coded pragmas they have been created from in the code snippet in Figure 2.3.

Following an execution model which decouples the static specification of the code from its dynamic execution, threads first execute the application code (creating all the tasks they encounter) until they reach a global synchronisation point. Then, they execute tasks asynchronously. When tasks are created they are inserted into the TDG based on their data dependencies. Only when all the dependencies of a task have been satisfied does a task move from created, to ready. Ready tasks are stored in a ready queue from which the scheduler distributes tasks among all threads for asynchronous execution. This model is similar to how an out-of-order core schedules instructions to hardware functional units. The runtime system dynamically schedules tasks to cores when all their inputs are ready and, when the execution of a task finishes, its outputs become ready for the next tasks. This decoupling of the specification of the program from its dynamic execution eases programmability and enables many optimisations at the runtime system level in a generic and application-agnostic way [104, 30, 125, 91].

Following the development and success of task-based programming models such as OmpSs [55], TBB [110] and Cilk [21], the OpenMP 4.0 standard [102] released in 2013 added comprehensive support for task-based programming. Since then, interest in task-based programming has continued to increase, as motivating trends for it such as increased parallelism and heterogeneity in hardware continue to grow at pace. In this thesis, we use information already present in existing task-based programming models to make our contributions in Chapters 4 to 6.

## 2.4 Runtime-aware architectures

During the decades when single-threaded performance was increasing steadily due to Moore's Law, Dennard Scaling and microarchitectural innovations, the improvement and optimisation of hardware and software proceeded largely separate from each other. While there were continuing improvements to be gained by innovating independently within the hardware and software realms, it was an advantage for both to share a well defined and stable interface, namely the ISA. During this time, ISAs of course improved, but mainly by extending their existing functionality in predictable and evolutionary fashion, rather than changing the roles and responsibilities on each side of the interface. Most of the disruptive innovation during these decades took place away from the ISA interface; in the microarchitecture and memory hierarchy on the hardware side, and in languages, optimising compilers, programming models and runtime systems on the software side.

Since the end of Dennard Scaling and the exhaustion of ILP improvements we have witnessed the move to multicore, which has for the first time in decades forced significant downstream disruption onto the software side of the ISA interface. In this new era, software performance improvements became dependent, to an unprecedented degree, on software taking account of the architectural details of the hardware it runs on. Software optimisation today must pay attention to factors such as parallelism and heterogeneity of execution resources, cache hierarchy dimensions and organisation and main memory NUMA effects, all of which vary widely depending on the hardware platform the software will run on. While these hardware characteristics do not form part of the functional interface between the hardware and software encapsulated in the ISA, they increasingly form an implicit performance interface which software must be designed against.

On the software side, many of the significant developments in programming models and their associated runtime systems in recent years such as the increasing popularity of task-based

programming have been prompted by the need for the software to adapt to the increasing complexity and variety of hardware architectures it is expected to encounter. Task-based programming models, by decoupling the specification of the software program from its dynamic execution, allow the runtime system to build introspective knowledge about what the program intends to do in the near future, via the TDG. This knowledge in the TDG about the software's future execution, combined with the runtime system's knowledge of the hardware resources available to it on a given system, is what enables the runtime system to make real time intelligent decisions about how best to conduct the execution.

This move to a more abstract programming model structured around tasks and the data they use (rather than the more low level thread-based fork/join model of parallelism exploited by earlier versions of OpenMP) has been pursued in the first instance as a reaction to the increasingly complex and parallel hardware available and the programmability wall it entails. However, such a programming model and its associated runtime system layer also represents a rich opportunity for closer hardware-software co-design, through an appropriate interface.

To this end, Valero et al. [125] outline the idea of *runtime-aware architectures* in which the hardware and software are co-designed to take full advantage of the information on both sides of their interface. Such a proposal enables a whole raft of optimisations and improvements that are not possible with today's hardware - software interfaces. Casas et al. [29] expand on this concept and establish the potential for hardware-software optimisations through the use of information from the runtime system. Since these papers outlined the potential of runtime-aware architectures, a significant body of work has looked to exploit that potential. Chronaki et al. [41, 42] introduce proposals for task scheduling on multicore and heterogeneous architectures. Proposals from Chasapis et al. [33, 34] consider runtime guided power optimisation in the presence of manufacturing variability in multiprocessor systems. A runtime managed DVFS optimisation scheme based on task criticality is proposed by Castillo et al. [32] and achieves 18% and 30% improvements in execution time and energy delay product respectively over a non-criticality aware scheduler. Brumar et al. [24] demonstrate accuracy preserving prediction of task outputs through runtime assisted memoisation of task dependencies. Sánchez Barrera et al. [116, 115] propose analyses of the task dependency graph in the runtime system with the goal of improving data locality in NUMA systems. Garcia et al. [64] and Papaefstathiou et al. [105] bring prefetching to bear at the level of the runtime system for task-based parallel programming models to optimise data accesses for tasks in advance of their execution, achieving performance improvements of 10% and 17% on average over hardware based prefetching only. RADAR [90] performs dead cache line prediction utilising details from the programming

model and runtime system regarding tasks to evict data which is least likely to be used in the future. Work by Alvarez et al. [6, 5] describes schemes to manage scratchpad memories in both fork/join and task-based parallel paradigms. Information from the runtime system of task-based programming models has also been shown to be valuable in managing recent novel additions to memory hierarchies like stacked DRAM [4] and hybrid DRAM/NVM memory [89]. Jaulmes et al. [71, 72, 73] demonstrate the potential for runtime assisted reliability and resilience. Castillo et al. [31], Etsion et al. [61], Kumar, Hughes, and Nguyen [81], and Tan et al. [121, 122] consider the benefits of implementing performance critical elements of the runtime system for task-based programming models in hardware, instead of in software. These proposals achieve performance improvements for task-based benchmarks ranging from 12% to 109%. Finally, Dimic et al. [52] describes a runtime-assisted cache insertion policy based on re-reference intervals.

Runtime-aware architectures is still a young (all the works cited above were published since 2015) but promising area of research. The goal of the work in this thesis is to follow this runtime-aware architecture approach to drive efficiency and optimisations within the cache and memory hierarchy of modern hardware architectures.

# Chapter 3

# Experimental Methodology

This chapter describes the experimental methodology used in this thesis. The first section outlines the very large ccNUMA system we use in the first contribution of the thesis in Chapter 4 to investigate the interplay between hardware and software techniques for scaling ccNUMA. In the second section we describe the simulation based infrastructure used in both the second and third contributions of the thesis. It details the baseline system model used in our work, which our proposals in Chapters 5 and 6 build upon. Finally, the third section describes all the benchmarks used in our work.

## 3.1 Very Large Scale NUMA System

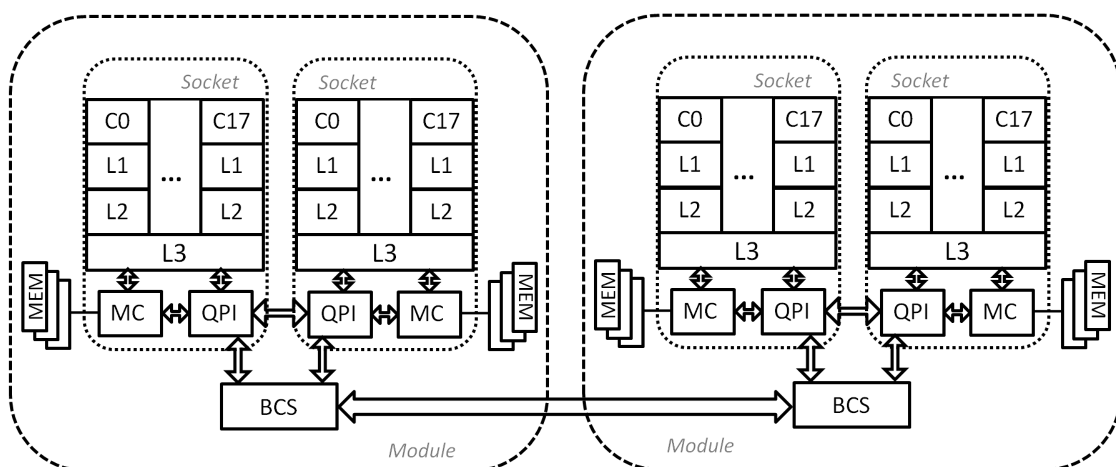



Figure 3.1: Logical view of a dual module bullion S system [9]

In order to investigate ccNUMA performance we utilise a specialised very large scale NUMA platform from the vendor Bull Atos. This platform is called the bullion S and employs a

proprietary Bull ASIC chip in a modular topology to enable scaling beyond the normal scaling limit for Intel's QPI ccNUMA interconnect. Figure 3.1 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.

Each module (see Figure 3.1) 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. The system runs RHEL 6.5 with a Linux kernel version of 2.6.32.

Table 3.1: NUMA distances and their average Bandwidths and Latencies

| Type | Local | Near Remote | Far Remote |
|---|---|---|---|
| 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 |

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 3.1, 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 3.1 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) [128] to measure the latencies and the STREAM benchmark [93] to measure the memory bandwidths.

The average latencies in the system (Table 3.1) follow the same pattern and similar ratios to the NUMA distances in Table 3.1. On average there is a 42% latency penalty in accessing

Table 3.2: Configuration of gem5 full-system simulations.

| | |
|---|---|
| Cores | 16 Out-of-order cores, 4 inst. wide, 2.0GHz |
| Branch predictor | Tournament: 2K local pred., 8K global and choice pred., 4-way BTB 4K entries, RAS 16 entries |
| Execution | ROB 128 entries. IQ 64 entries, 4 INT ALU, 2 FP ALU, 2 LD/ST units, 256/256 INT/FP RegFile. |
| L1I / L1D cache | Each 32KB, 8-way, 64B/line (2 cycles) |
| ITLB / DTLB | Each 256 entries fully-associative (1 cycle) |
| LLC | Shared unified 32MB, banked 2MB/core 64B/line, 15 cycles, 8-way, pseudoLRU |
| Coherence Protocol | MESI with blocking states, silent evictions |
| Directory | Total 524288 entries, banked 32768 entries/core 15 cycles, 8-way, pseudoLRU |
| NoC | 4x4 mesh, link 1 cycle, router 1 cycle |

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 3.1 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 remote NUMA region incurs an 82% lower available bandwidth than accessing the near NUMA region. Comparing local and far accesses the bandwidth penalty is 89%.

## 3.2 Simulation Infrastructure

The infrastructure we use for our simulation based work in this thesis (Chapters 5 and 6) is composed of several components. We simulate the entire software environment as it would be on a real system, encompassing the full Operating System, Runtime System and Benchmark Applications. On the hardware side we use the gem5 architectural simulator and the Ruby memory hierarchy simulator to provide the most detailed and accurate evaluation possible for our proposals.

### 3.2.1 Simulators

We use the detailed cycle-level gem5 simulator to model a baseline architecture and the hardware extensions involved in our proposals in Chapters 5 and 6. gem5 supports a variety of ISAs and for each ISA provides a choice of different levels of detail in the modelling. Both in the CPUs and the memory hierarchy there are models which vary from less detailed with faster simulation time to more detailed with slower simulation time. In our simulations we use the most detailed out of order x86 CPU model, called the O3 CPU in gem5, and we use the most detailed option for the memory hierarchy which is the Ruby memory hierarchy simulator. The Ruby memory hierarchy model simulates each cache level, the main memory controllers and the cache coherence protocol in detail. These choices allows us to most accurately model the effects of our proposals, which have impacts from the CPU cores throughout every level of the memory hierarchy and the cache coherence protocol, from a performance perspective. We use gem5 in *full-system mode*, meaning the simulated system runs the entire software stack, just as a real system would. This includes a full Linux Operating System in addition to our runtime system and benchmark applications.

Additionally, we use McPAT [86] to model the area and power impacts of our proposals. McPAT is a power, area and timing simulator which models all the components of a modern CPU such as cores, caches, NoC and Memory Controllers. We improve the accuracy of McPAT's inbuilt component models by applying the changes recommended by [130]. Our McPAT simulation is based upon a 22nm process technology node and McPAT's default clock gating scheme.

### 3.2.2 Baseline Simulated System Architecture

We simulate a 16-core processor using gem5's detailed O3CPU and Ruby memory hierarchy model. Table 3.2 summarises the main parameters of the baseline simulated architecture.. We use 16 cores in a 4 by 4 mesh topology. The NoC latency is 1 cycle per link and per router traversed. The 32KB L1I / L1D caches per core are each 32KB and 8 way set associative, storing 64 byte cache lines. They take 2 cycles to access. The LLC is banked with 2MBs of capacity per core and is shared among all cores as a NUCA cache. It also stores 64 byte cache lines and is 8 way set associative, uses a pseudoLRU replacement policy and has an access latency of 15 cycles.

Cache coherence is maintained using a directory based approach and employs the widely used MESI [67] coherence protocol. The LLC is inclusive of the L1 caches, meaning all cache

lines present in the L1s are also present in the LLC. The cache coherence directory is embedded in the inclusive LLC. The extensions we make to this baseline architecture for the contributions in this thesis are explained in detail in the corresponding Chapters, 5 and 6.

### 3.2.3 Simulated System Environment

Our simulated system runs Ubuntu 14.04 as the operating system with version 4.3 of the Linux kernel. We use the Nanos++ 0.10a [56] runtime system, which supports the OpenMP 4.0 [103] task-based programming model and additional tasking constructs provided by the OmpSs task-based programming model. All benchmarks run in the simulated system are compiled with the associated source-to-source compiler for Nanos++, Mercurium version 1.99 [12]. Mercurium translates the tasking directives annotating the source code into calls to the Nanos++ runtime system. The post-Mercurium sources are compiled using GCC 4.8.2 with the -O3 optimisation flag.

## 3.3 Benchmarks

Our evaluations use a set of task-based parallel benchmarks programmed with OpenMP 4.0 annotations and listed in Table 3.3. These benchmarks are representative of a wide range of important problems from a variety of computational domains, with significant diversity in their data access patterns.

**Integral Histogram** [107] is an important algorithm for computing image histograms. Histograms are one of the most common components used in computer vision problems such as object based retrieval, segmentation, detection and tracking. This implementation decomposes the input image into tiles which may be processed in parallel tasks, propagating data between tasks via halos. **TinyJPEG** is a benchmark based on a lightweight JPEG decoder. It takes a JPEG image as input and converts it to an uncompressed bitmap image.

**Jacobi**, **Gauss-Seidel** and **Redblack** are well known and important iterative methods for solving systems of linear equations. In these benchmarks they are employed to solve a heat diffusion problem in a two dimensional grid. **Conjugate Gradient** implements the conjugate gradient iterative algorithm for solving large, sparse systems of linear equations. Such systems of linear equations typically arise when solving partial differential equations. **Cholesky** implements the Cholesky Factorisation, an important problem in linear algebra with applications in least square, partial differential, optimisation and linear programming problems. **Symmetric Matrix Inversion** is a more complex linear algebra problem, with

applications in Medical Imaging and Earth and Aerospace Science. **Matrix Multiplication** is an implementation of matrix multiplication with a block based parallel decomposition over tasks.

The **Kmeans** clustering algorithm is an important problem in statistics and data mining. It groups streaming input data into groups by similarity, based on multiple dimensions of the input data points. Kmeans clustering has applications in image analysis, machine learning, data compression and computer graphics. **KNN** implements the K-nearest neighbours algorithm, a type of instance-based learning. This problem is important in applications such as pattern recognition and machine learning. **MD5** is a well know cryptographic hashing algorithm. This benchmark applies the MD5 hash to random data input buffers. **Streamcluster** targets the online clustering problem, categorising large volumes of streaming data into clusters. It has applications in areas such as network intrusion detection, data mining and pattern recognition.

Tables 3.4, 3.5 and 3.6 shows the input set sizes used for the benchmarks in each of the three contributions of this thesis. The benchmarks listed for Chapter 4 are used on the real, very large (16 socket multiprocessor, 288 core) ccNUMA system we use for the first contribution of this thesis. The benchmarks listed for Chapters 5 and 6 are run on the simulated single multicore processor with 16 cores used for the second and third contributions of this thesis respectively. The benchmark problem set sizes are chosen in line with the resources of the system they are run on in the given contribution. In both the simulated and real environment, the benchmarks are run from start to completion. The results presented in the evaluations are collected from the entire post-initialisation parallel execution phase of the benchmarks. To control the impact of inherent variation in benchmark execution arising from preceding system state, the benchmark executions for every experiment are repeated 5 times, perturbing the simulated system before each benchmark execution. From the resulting 5 repeated executions the fastest and slowest executing repetitions are excluded and the mean of the remaining 3 repetitions is reported.

Table 3.3: Benchmark Descriptions

| **Benchmark** | **Description** | **Source** |
| --- | --- | --- |
| Cholesky | Factorises a hermitian, positive definite matrix into the product of a lower triangular matrix and its conjugate transpose. | BAR |
| Symmetric Matrix Inversion | Computes the inverse of a symmetric matrix. | [101] |
| Streamcluster | Solves the Online Clustering Problem; groups streaming input data points based on multiple dimensions. | PARSEC [19] |
| Conjugate Gradient | An iterative algorithm for solving sparse systems of linear equations. | [72] |
| Gauss-Seidel | Implements the Gauss-Seidel method for solving a system of linear equations. | BAR |
| Integral Histogram | Computes a cumulative histogram over the pixels of an image using a crossweave scan of the image | [18] |
| Jacobi | Implements the Jacobi method for solving a system of linear equations. | BAR |
| TinyJPEG | Decodes a JPEG image with fixed encoding of 2x2 MCU size and YUV color. | [114] |
| Kmeans | Implements the Kmeans clustering algorithm. | HCA Apps |
| KNN | Implements the K-nearest neighbour algorithm. | HCA Apps |
| MD5 | Cryptographically hashes input data buffers. | HCA Apps |
| Redblack | Implements the Red-Black method for solving a system of linear equations. | BAR |
| Matrix Multiplication | Implements a task-based blocked matrix multiplication $A \times B = C$ | BAR |

Table 3.4: Benchmark configurations for Chapter 4

| Benchmark | Problem Set |
|---|---|
| Cholesky | 2D Matrix $N^2$ = 3497066496 |
| Symmetric Matrix Inversion | 2D Matrix $N^2$ = 3497066496 |
| Streamcluster | Centres: 10 - 20, Dims: 128, Data Points: 7.2 million |
| Jacobi Solver | 6400 Blocks, 800 x 20000 blocksize |
| Integral Histogram | 65536 x 65536 pixels, 32 bins, 512 x 512 blocksize |

Table 3.5: Benchmark configurations for Chapter 5

| Benchmark | Problem Set |
|---|---|
| Conjugate Gradient | 3D Matrix $N^3$ = 884736, 3 iters. |
| Gauss-Siedel | 2D Matrix $N^2$ = 2359296, 10 iters. |
| Integral Histogram | 1000x1000 pixels, 50 bins, 50 x 50 blocksize |
| Jacobi Heat Diffusion | 2D Matrix $N^2$ = 2359296, 10 iters. |
| TinyJPEG | 2992 x 2000 pixel JPEG image |
| Kmeans | 150000 pts., 30 dims, 6 clusters, 3 iters. |
| KNN | 16384 training pts, 8192 pts to classify, 4 dims, 4 classes |
| MD5 | 128 buffers of 512KB to hash |
| Redblack | 2D Matrix $N^2$ = 2359296, 10 iters. |

Table 3.6: Benchmark configurations for Chapter 6

| Benchmark | Problem Set |
|---|---|
| Gauss-Siedel | 2D Matrix $N^2$ = 28901376, 2 iters. |
| Jacobi Heat Diffusion | 2D Matrix $N^2$ = 16777216, 5 iters. |
| Integral Histogram | 1000x1000 pixels, 50 bins, 50 x 50 blocksize |
| Matrix Multiplication | 128 x 128 blocksize, 8 x 8 blocks per matrix |
| Kmeans | 450000 pts., 45 dims, 6 clusters, 1 iter. |
| KNN | 512 training pts, 229376 pts to classify, 12 dims, 8 classes |
| MD5 | 128 x 2 MB buffers |
| Redblack | $N^2$ = 28901376, 5 iters. |

Chapter 4

# Reducing Cache Coherence Traffic with a NUMA-aware Runtime Approach

## 4.1 Introduction

The ccNUMA approach to memory hierarchy organisation has become near ubiquitous in the design of shared memory 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. In large ccNUMA machines it is common for multiple processors, each with their own LLC, to share main memory. In such systems where multiple LLCs are coupled with a shared yet distributed NUMA main memory, coherence is required in the memory hierarchy at the interface between the multiple LLCs and the main memory. Then, every access to main memory is a transaction in the coherence protocol at this interface in the memory hierarchy. Depending on the state of the accessed memory location, 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 shared memory architectures is responsible for a significant proportion of the system energy consumption [106]. Judicious management of data locality (either performed by the runtime system, or the application itself) is therefore crucial for both energy efficiency and performance.

Historically, the most common way to program shared memory systems are fork/join and thread-based programming models like OpenMP [102] or Pthreads [26], which provide primitive mechanisms to handle NUMA architectures. OpenMP introduced support for tasking and data dependencies in version 4.0 of the standard. These features provide the opportunity to write task-based programs in OpenMP 4.0 and for the runtime system to automatically handle

data locality in a NUMA-aware fashion for such programs. 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 [53, 127]. 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 cache coherent shared memory 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 a ccNUMA system there are two requirements: **(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 requirement (i) or (ii). In this work we distinguish between the effects of requirement (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. Furthermore, we 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 ccNUMA 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 and is described in 3.1. 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 ccNUMA 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 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, this work presents the first study on how a hierarchical directory approach [92] to scaling cache

coherence interacts with a runtime managed strategy to promote data locality in a ccNUMA platform.

Specifically, these are the principle advances made in this contribution:

- A complete performance analysis of a large ccNUMA 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 cc-NUMA 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 rest of this chapter is organised as follows: Section 4.2 details the classes of coherence traffic we use in our analysis and explains how they operate within the bullion S system. Section 4.3 details the programming model and runtime system features which enable three different regimes of work scheduling and data allocation that we use in our study. Section 4.4 presents the results of our analysis of the performance and coherence traffic. Lastly, we summarise this contribution in Section 4.5.

## 4.2 Cache Coherence in a Very Large Scale NUMA System

In this section we first describe a useful categorisation of coherence traffic. Secondly, we describe how these categories of traffic are used in the very large ccNUMA system we study, the bullion S system.

### 4.2.1 Categories of Coherence Traffic

While the invisible caches and shared main memory view a ccNUMA system offers the user considerably eases the programming burden, it requires a sophisticated hardware 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 ccNUMA architecture 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 to what extent the NUMA-aware software techniques and the hierarchical directory implemented by 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*

Table 4.1: 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 |

*(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 4.1), cover all possible traffic types at the LLC to main memory interface within the memory hierarchy. They also provide an insightful basis upon which to analyse the effectiveness of the three work scheduling and data allocation regimes we study and the BCS in constraining the amount of coherence traffic in the system.

### 4.2.2 ccNUMA in the bullion S System

Figure 3.1 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.

The BCS is the glue for connecting multiple modules into a single ccNUMA 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 3.1) 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 3.1) of a processor unless otherwise indicated and the coherence traffic observed represents only coherence transactions at the interface of the LLC and the system memory (via a BCS). The coherence agents for the system memory in Figure 3.1 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 caches module level directory information about cache lines which have been exported from a memory in its local NUMA regions to LLCs in other modules. The directory cache implemented in the BCS is inclusive of all cache lines exported from its local module to remote module LLC. This enables the BCS to respond in place of a remote LLC 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 remote module LLC. 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 4.1 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.

## 4.3 Memory Allocation and Scheduling for ccNUMA

This section explains the specific features of the programming model and runtime system we use to define three different regimes of scheduling and memory allocation for our analysis.

### 4.3.1 Programming Model and Runtime System NUMA-awareness

In order to minimise the amount of coherence traffic required in the system for a given problem, we use a task-based dataflow programming model. Such a programming model is supported in OpenMP since version 4.0 of the standard. In task-based dataflow programming models the execution of a parallel program is structured as a set of tasks and an execution ordering among them based on dataflow constraints. The programmer identifies tasks by annotating serial code with directives. Dataflow is represented by clauses in the directives which specify whether data is read by a task (input dependencies), written by a task (output dependencies), or both (inout dependencies). The runtime system manages parallel execution of the tasks based on these directives, relieving the programmer from the need to explicitly synchronise and schedule tasks and thus promoting programmability.

We use the OmpSs [54] task-based dataflow 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 a superset of the OpenMP 4.0 task-based constructs, however we do not use any programming model features of OmpSs outside those supported by OpenMP 4.0. The task-based dataflow 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 [25] in the release we use, version 0.10.3.

The default (NUMA-oblivious) Nanos++ scheduler maintains one global queue of ready tasks for the entire shared memory system. Tasks are scheduled among cores without considering where their data dependencies reside in main memory. In contrast, the NUMA-aware Nanos++ scheduler maintains one ready queue per NUMA region within the shared memory 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[101]. 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 4.2.1.

### 4.3.2 Scheduling and Memory Allocation Regimes

The OmpSs and Nanos++ 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 cores in the system, ensuring computational load is balanced. Data is allocated in the NUMA region of the core which first accesses the data. This is governed by the memory allocation policy of the operating system, which in the case of Linux is a first touch allocation policy by default. How the memory allocation is balanced among the NUMA regions depends on how the data is initialised by the application. For example, if the data is initialised on the master thread, before the application instantiates any tasks, all the data will be allocated in the NUMA region which is running the master thread until that NUMA region is full. This can lead to a very unbalanced utilisation of memory from a NUMA perspective. If the data is initialised by tasks, it will be allocated in whichever NUMA region the initialising tasks execute in, leading to a randomly distributed allocation among the NUMA regions.

***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 [101] recognises tasks which first touch data (the first tasks specifying the data as 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). Subsequent 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 with the OmpSs NUMA-aware scheduling feature, as the OmpSs NUMA-aware scheduling feature depends on First Touch allocation in order to allow the runtime system to distribute the data allocation in a NUMA-aware layout, before it schedules the work based on this.

## 4.4 Results and Analysis

### 4.4.1 Introduction

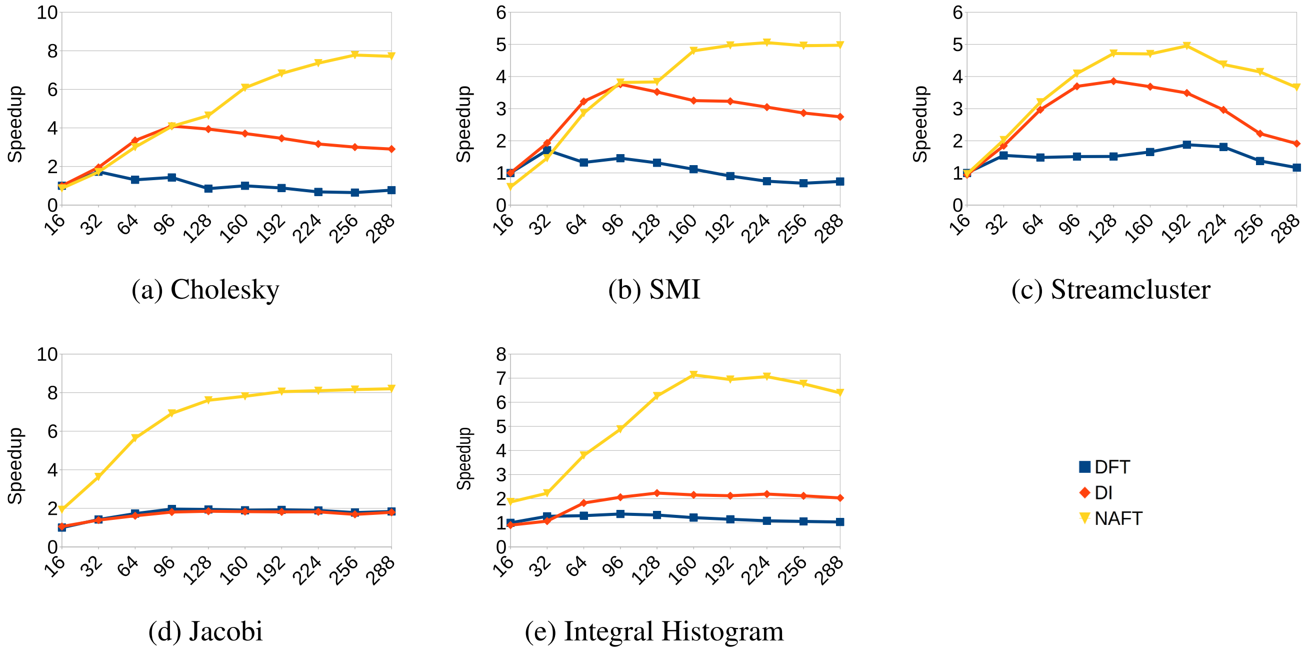
In this section we present a detailed analysis of the coherence traffic generated by running the benchmarks listed alongside their problem sizes in Table 3.4. These benchmarks are described in Section 3.3. We present results for the benchmarks under the three regimes detailed in Section 4.3.2 : DFT, DI and NAFT. First, in Section 4.4.2 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 4.4.3 we present a detailed breakdown of the Control traffic type into its individual message classes. Thirdly,

in Section 4.4.4 we focus on how uniformly the system-wide 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: **(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 4.2 to 4.6 show the system-wide (i.e. all 8 modules aggregated) coherence traffic. In Figures 4.2 to 4.6, the traffic's directionality split (see Section 4.2.2) 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 [93] is 275 GB/s incoming and outgoing simultaneously.

To place the coherence traffic analysis that follows in a performance context Figure 4.1 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 4.1) and are related to the coherence traffic measurements throughout Section 4.4.2. Section 4.4.3 presents a detailed breakdown of the Control coherence traffic. Finally Section 4.4.4 discusses the symmetry of the execution over the large 8 module system, which provides important context for the performance scalability results in Figure 4.1.

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 system-wide 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 of bandwidth to memory will be $K/R$ times the system-wide 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




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

(d) Jacobi (e) Integral Histogram

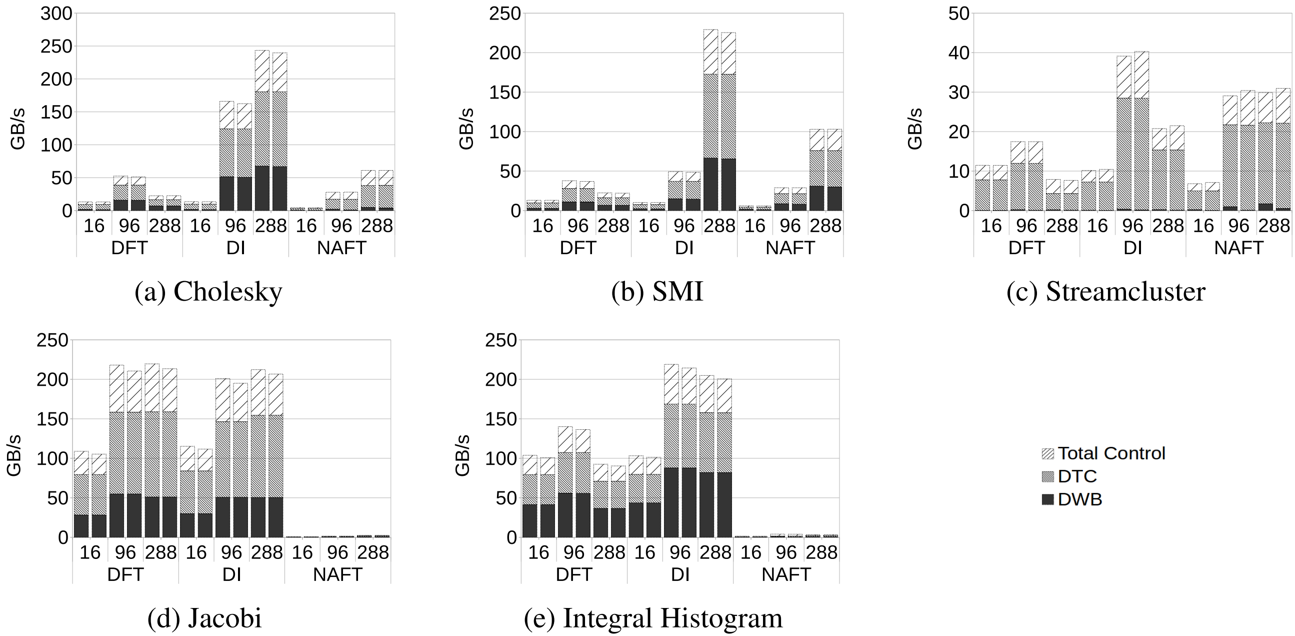
Figure 4.1: Speedup DFT, DI, NAFT regimes under increasing thread count. Threads distributed uniformly among all available NUMA regions. Figures normalised to DFT with 16 threads.

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 system-wide BCS capacity resources are utilised, which may negatively impact the system in both performance and energy efficiency.

### 4.4.2 DWB, DTC and Control Coherence Traffic

Figures 4.2 and 4.3 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 system-wide level of Data traffic is symmetric in directionality. This symmetry arises because messages belonging to these classes originate at either a CPU cache or local memory within one module and always 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. This reflects the fact that these two coherence messages which carry data payloads always originate and terminate




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

(d) Jacobi (e) Integral Histogram

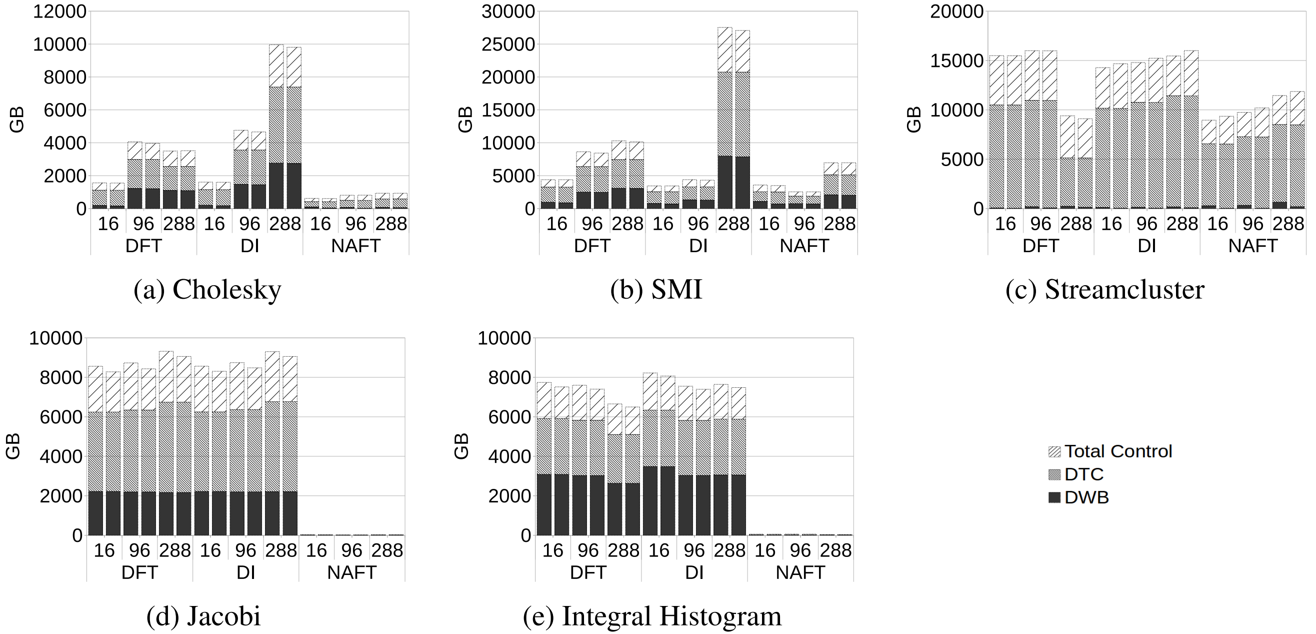
Figure 4.2: Coherence Bandwidth: DWB, DTC and Control traffic

at a CPU or a memory controller, and (unlike Control coherence messages) never at a BSC. 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, as can be seen in Figures 4.2 and 4.3.

#### 4.4.2.1 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. As noted earlier, all traffic measured in our evaluation is inter-module traffic, thus 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 4.1a) 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




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

(d) Jacobi (e) Integral Histogram

Figure 4.3: Coherence Movement: DWB, DTC and Control traffic

from utilising up to 256 threads. As can be seen in Figure 4.1a 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 4.2a) and coherence movement (Figure 4.3a) under the NAFT regime versus the DI regime.

At 96 threads both regimes are performing similarly (Figure 4.1a), 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.

#### 4.4.2.2 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 4.1b). 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 4.2b and 4.3b respectively), at 96 threads both the NAFT and DI regimes are performing similarly (Figure 4.1b 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.

#### 4.4.2.3 Streamcluster

We can see from the traffic profile of the Streamcluster benchmark (Figures 4.2c and 4.3c) 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 4.1c 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 4.2c) and less coherence movement (Figure 4.3c), 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.

#### 4.4.2.4 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 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.

#### 4.4.2.5 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.

#### 4.4.2.6 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 under-utilisation of resources such as bandwidth to memory and the BCS directory capacity (as explained earlier in Section 4.4.1), 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, and is investigated further in Section 4.4.4.

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 4.1a, 4.1b and 4.1c respectively) the NAFT regime performs similarly to the DI regime until we reach 96 threads. As can be seen from Figures 4.2a, 4.2b and 4.2c, 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

Table 4.2: 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% |

quickly under the DI regime than the other three benchmarks. As can be seen from Figures 4.2d and 4.2e, 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 traffic is presented in Tables 4.2a and 4.2b. Table 4.2a shows the results for the NAFT regime in comparison to the DFT regime, while Table 4.2b shows the results for the NAFT regime in comparison to the DI regime.

In all cases in Tables 4.2a and 4.2b 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 4.4.4. Comparing the two best performing regimes (Table 4.2b), 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.

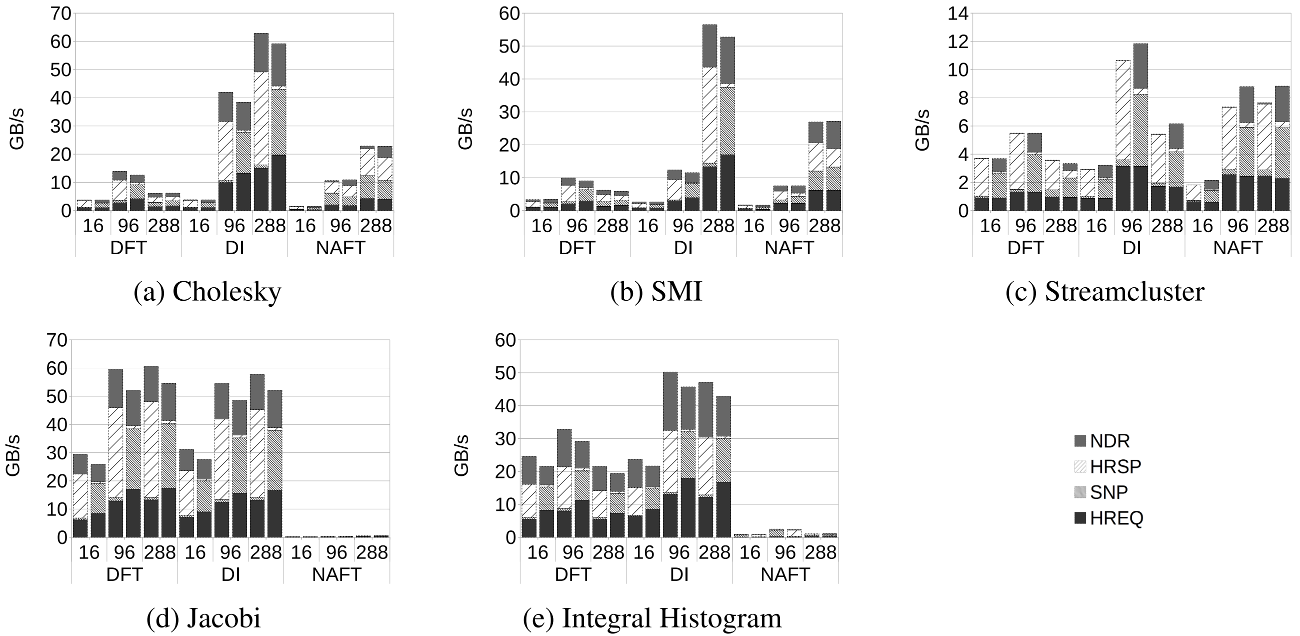
The large reductions in coherence traffic achieved by the NAFT regime and summarised in Tables 4.2a and 4.2b have important implications for energy efficiency which is a crucial factor in the way future computing architectures and software stacks are designed [78, 28, 57, 125].

### 4.4.3 Control Coherence Traffic Decomposition

Figures 4.4 and 4.5 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 4.2 and 4.3 (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 3.1. 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.

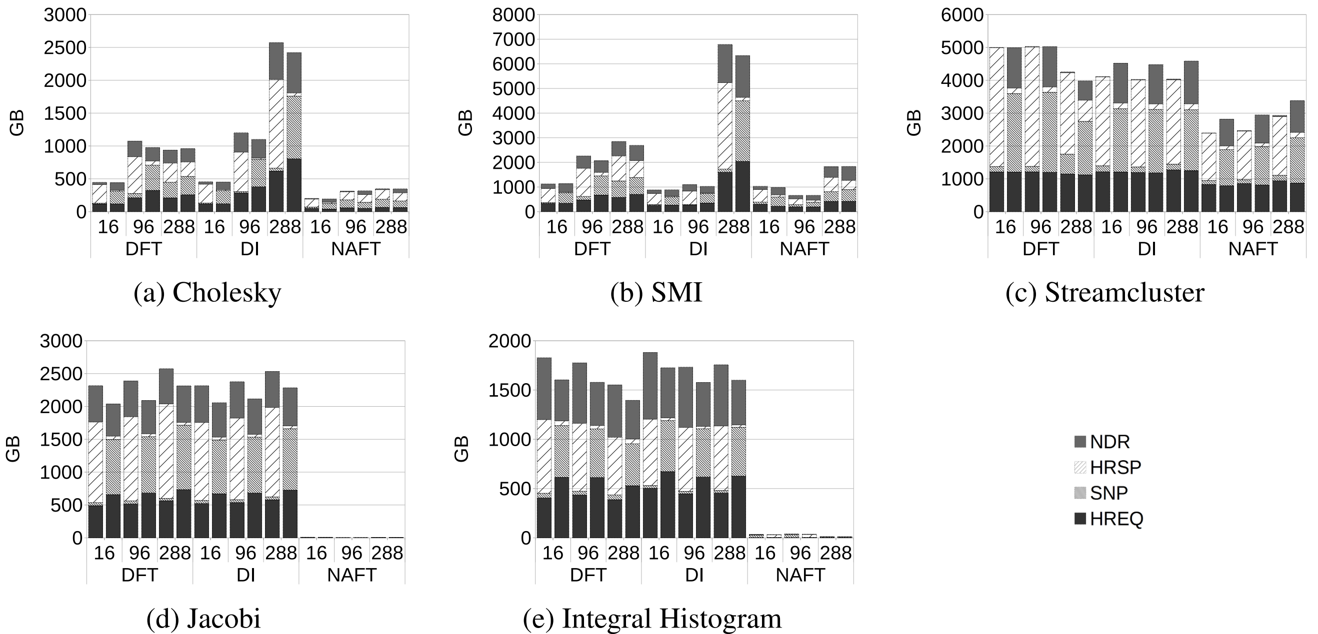
These asymmetries in the SNP and HRSP traffic represent cases where the action of the BCS in the coherence protocol is having a significant impact on the SNP and reciprocal HRSP traffic. The BCS is often able to filter SNP messages it receives from its local CPUs from the system and answer the SNP message 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




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

(d) Jacobi (e) Integral Histogram

Figure 4.4: Coherence Bandwidth: Control traffic by message class




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

(d) Jacobi (e) Integral Histogram

Figure 4.5: Coherence Movement: Control traffic by message class

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.

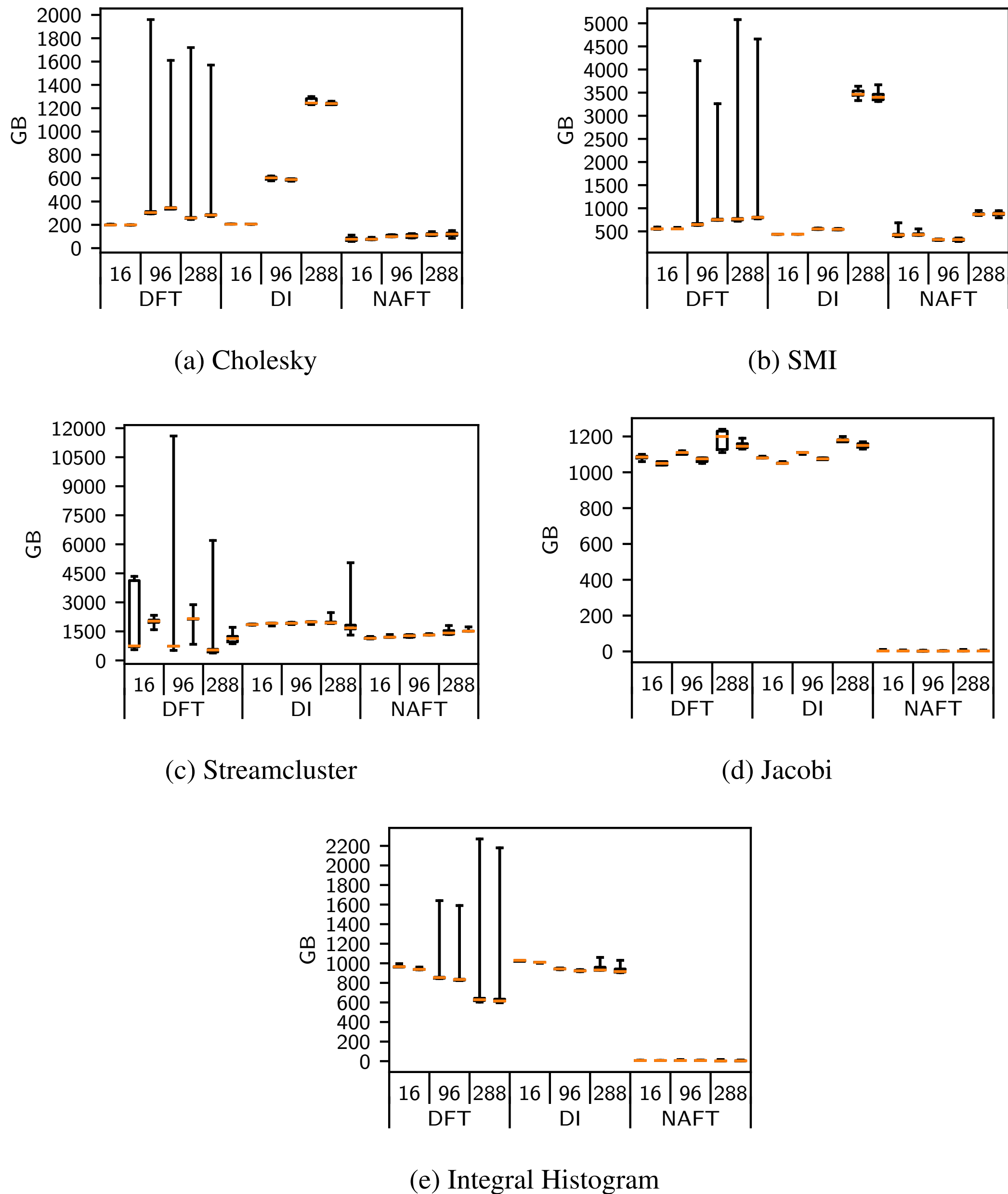
In Figures 4.4 and 4.5 we also see asymmetry between the incoming and outgoing traffic for the NDR message class. This is particularly pronounced across all thread counts and regimes in the Streamcluster benchmark but is also apparent in the results for the other benchmarks to some degree. This is for the reason explained in Section 3.1, 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.

### 4.4.4 Traffic Symmetry Over Modules

Figure 4.6 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 4.3.2. 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




(a) Cholesky

(b) SMI

(c) Streamcluster

(d) Jacobi

(e) Integral Histogram

Figure 4.6: Coherence Movement distribution by module

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 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 4.4.1 if there is an uneven allocation of data among the NUMA regions in the system, this results in an under-utilisation 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 system-wide resource is distributed 1/8th in each of the 8 modules of the system. Therefore, as Figure 4.6 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 system-wide 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.

## 4.5 Summary

NUMA architectures continue to dominate the shared memory 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 ccNUMA 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 present evidence for the ability of the BCS to mitigate the cost of coherence traffic with increasing system scale.

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.

# Chapter 5

# Runtime-Driven Cache Coherence Deactivation

With increasing core counts, the scalability of directory based cache coherence has become a challenging problem. To reduce the area and power needs of the directory, recent proposals reduce its size by classifying data as private or shared, and disable coherence for private data. However, existing classification methods suffer from inaccuracies and require complex hardware support with limited scalability.

This contribution proposes a hardware/software co-designed approach: the runtime system identifies data that is guaranteed by the programming model semantics to not require coherence and notifies the microarchitecture. The microarchitecture deactivates coherence for this private data and powers off unused directory capacity. Our proposal reduces directory accesses to just 26% of the baseline system, and supports a 64× smaller directory with only 2.8% performance degradation. By dynamically calibrating the directory size our proposal saves 86% of dynamic energy consumption in the directory without harming performance.

## 5.1 Introduction

Since the end of Dennard scaling, multicore architectures have proliferated. Among the different paradigms, shared-memory multiprocessors have been dominant due to their advantages in programmability. The ease of programmability of shared-memory architectures is granted by hardware cache coherence, which manages the cache hierarchy transparently to the software. Modern architectures typically use directory based coherence protocols, since they are the most scalable approach. However, directory based protocols incur significant area and power overheads, as the number of directory entries scales up with the size of the caches and the size of each directory entry scales up with the number of cores [119].

In recent years, much emphasis has been placed on reducing the costs of directory based coherence protocols. Many of these studies realise that, fundamentally, coherence is only

needed for data that is shared between multiple logical cores, at least one of which will write to it. Otherwise data races do not happen and coherence is not required. Based on this observation, many works propose to identify data that does not require coherence and to exploit this information to optimise the coherence protocol. State-of-the-art techniques aimed at identifying shared and private data rely on Operating System (OS) page table and Translation Lookaside Buffer (TLB) support [66, 80, 47, 113, 111], which can only operate at page granularity and require extensive changes in the TLBs.

The popularisation of large-scale multicores in HPC has also driven the evolution of parallel programming paradigms. Traditional fork-join programming models are not well suited for large-scale multicore architectures, as they include many elements (heterogeneous cores, dynamic voltage and frequency scaling, simultaneous multi-threading, non-uniform cache and memory access latencies, varying network distances, etc.) that compromise uniform execution across threads and, thus, significantly increase synchronisation costs. For this reason, task-based programming models such as OpenMP 4.0 [103] have received a lot of attention in recent years. Task-based parallelism requires the programmer to divide the code into tasks and to specify what data they read (inputs) and write (outputs). Using this information the runtime system manages the parallel execution, discovering dependences between tasks, dynamically scheduling tasks to cores, and taking care of synchronisation between tasks.

This chapter presents *Runtime-assisted Cache Coherence Deactivation* (RaCCD), a hardware/software co-designed approach that leverages the information present in the runtime system of task-based dataflow programming models to drive a more efficient hardware cache coherence design. RaCCD relies on the implicit guarantees of the memory model of task-based dataflow programming models, which ensure that, during the execution of a task, its inputs will not be written by any other task, and its outputs will neither be read nor written by any other task. As a result, coherence is not required for input and output data of a task during its execution. In RaCCD, the runtime system is in charge of identifying data that does not need coherence by inspecting the inputs and outputs of the tasks. Before a task is executed, the runtime system notifies the hardware about the precise address ranges of the task inputs and outputs. Using simple and efficient hardware support, RaCCD deactivates coherence for these cache lines during the execution of the task. When the task finishes, the runtime system triggers a lightweight recovery mechanism that invalidates the non-coherent cache lines from the private caches of the core that executed the task. As a consequence, RaCCD reduces capacity pressure on the directory, allowing for smaller directory sizes. We extensively explore reduced directory sizes with RaCCD and propose a mechanism to dynamically calibrate the directory size to

reduce energy consumption without harming performance. This chapter makes the following advancements beyond the state-of-the-art:

- RaCCD, a mechanism to deactivate cache coherence driven by runtime system metadata regarding task inputs and outputs. This mechanism requires simple and efficient architectural support, and avoids the complexity, scalability and accuracy problems of other solutions.
- An extensive evaluation that demonstrates the potential of RaCCD to reduce capacity pressure on the directory. RaCCD reduces directory accesses to just 26% of those in the baseline system. Moreover, it allows reducing the directory size by a factor of 64× with only a 2.8% performance impact on average. This results in 93% and 94% savings in dynamic energy consumption and area of the directory, respectively.
- An Adaptive Directory Reduction (ADR) mechanism to calibrate the directory size during execution to save energy consumption in the directory without harming performance, achieving an 86% saving in dynamic energy consumption in the directory.

## 5.2 Opportunity to Deactivate Coherence

The execution model of task-based programming models guarantees that, during the execution of a task, its inputs will not be modified by another task and its outputs will not be accessed by any other task. This precludes data races occurring on the task inputs and outputs, making coherence redundant. Thus, the runtime system can precisely identify non-coherent data without the hardware complexity nor the accuracy problems of other approaches. To exploit this, the runtime system can direct the hardware cache coherence substrate to deactivate coherence for the inputs and outputs of tasks during their execution, and self-invalidate the non-coherent data when tasks finish.

Figure 5.1 shows the percentage of non-coherent blocks for a set of representative benchmarks under the OS page table (PT) and RaCCD approaches. PT identifies blocks as non-coherent by default, and they become coherent if they are accessed by more than one core. PT does not employ the extra complexity of TLB based approaches and thus does not identify temporarily private data as non-coherent. RaCCD identifies all task inputs and outputs as non-coherent. In Figure 5.1 a block is marked as coherent if it is ever accessed as coherent during the execution. On average RaCCD identifies 78.6% of the blocks as non-coherent, 2.9× more than identified by PT (26.9%). RaCCD significantly outperforms PT in CG, Gauss,

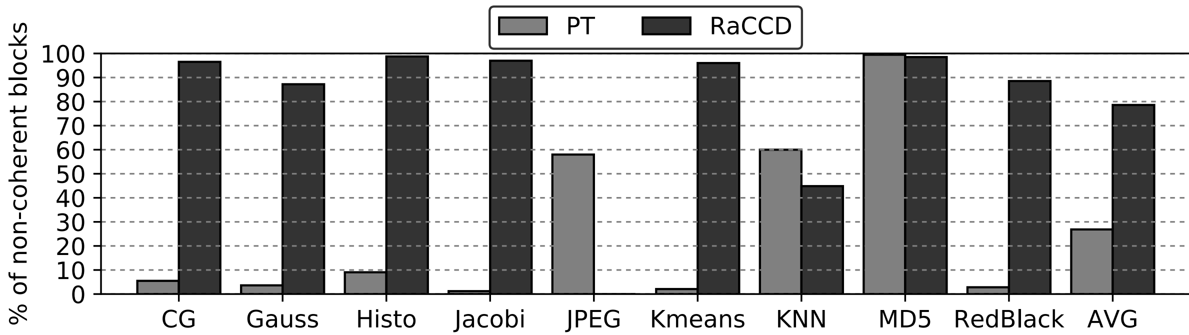



Figure 5.1: Percentage of non-coherent cache lines

Histo, Jacobi, Kmeans and RedBlack because, in these benchmarks, the data often migrates from one core to another in different application phases, so identifying temporarily private data is very important. RaCCD and PT perform similarly well on MD5 due to its streaming read behaviour with low data reuse, while in KNN, PT slightly improves over RaCCD. In JPEG the tasks have no input or output annotations, which is the worst-case scenario for our approach, so RaCCD is unable to identify any non-coherent blocks.

It is clear that task-based programming models and coherence deactivation are a natural fit. Most of the data accessed in task parallel programs does not require coherence, and the runtime system can easily and precisely identify this data to guide coherence deactivation. As a consequence, most of the dataset of the application does not need to be tracked in the directory, and its size can be reduced to save area and power consumption.

## 5.3 RaCCD: Runtime-assisted Cache Coherence Deactivation

This chapter presents RaCCD (Runtime-assisted Cache Coherence Deactivation), a hardware/-software co-designed approach that leverages information present in the runtime system of task-based dataflow programming models to direct a more efficient hardware cache coherence design. The goal of RaCCD is to identify data that is guaranteed to be data race-free at the runtime system level, and to deactivate coherence for it at the microarchitecture level. Thus, the size of the directory may be drastically scaled down, reducing its storage requirements and power consumption without any performance impact.

RaCCD takes advantage of the implicit guarantee present in task-based dataflow programming models that, during the execution of a task, no data races will occur on its inputs and outputs. This guarantee provides scope to deactivate coherence for task inputs and outputs

while they execute. In order to achieve this co-operation between the programming model and the hardware cache coherence substrate, the runtime system communicates the addresses of task inputs and outputs just before task execution. On the microarchitecture side, minimal hardware support is introduced to store this information and allow non-coherent memory accesses during task execution. When tasks finish, the runtime system manages the invalidation of the input and output data from the private caches, ensuring no incoherent data is present in the cache hierarchy.

### 5.3.1 Runtime System - Architecture Interface

RaCCD offers an interface between the runtime system and the architecture so that they cooperate in the management of non-coherent memory regions. The interface consists of two new ISA instructions that are issued by the runtime system at the beginning and at the end of the execution of each task to allow the deactivation of coherence at the microarchitecture level.

- *raccd_register(initial_virtual_address, size)*: Before executing a task, the runtime system uses this instruction to inform the microarchitecture of the initial address and size of an input or output of the task. An instruction is issued to specify each input and output of the task. The microarchitecture then deactivates coherence for these address ranges.
- *raccd_invalidate()*: When a task finishes, the runtime system uses this instruction to invalidate all non-coherent data from the private cache of the core which executed the task.

An alternative implementation could manage non-coherent memory regions through a memory mapped schema. We have selected the ISA extension approach due to its simplicity. As only a few instructions are issued per executed task, both solutions are expected to behave similarly.

### 5.3.2 Runtime System Extensions

RaCCD extends the operational model of task-based programming models to assist coherence deactivation at the microarchitecture level. Figure 5.2 shows the behaviour of two threads, including the three main phases of a task parallel program: scheduling, task execution and wake-up. The two additional operations RaCCD introduces are also shown: deactivate coherence and invalidate non-coherent data.

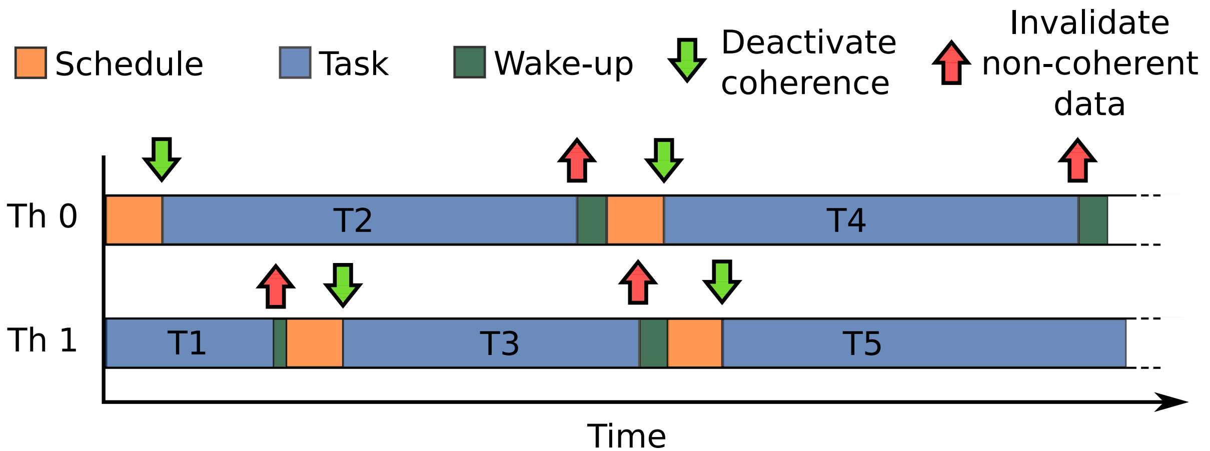



Figure 5.2: Runtime system support with additional operations to deactivate coherence and invalidate non-coherent data.

In the scheduling phase, the thread requests a ready task from the scheduler, which selects a ready task-based on a scheduling policy. Then, in the task execution phase, the scheduled task is executed. Finally, in the wake-up phase, tasks that depend on the executed task are analysed. If all the dependencies are satisfied, the dependent task is marked as ready and placed in the ready queue, from where the scheduler may allocate it to an executing thread in the scheduling phase.

RaCCD enables deactivating coherence for the inputs and outputs of the tasks. To do so, just before a task is executed, the thread iterates over the inputs and outputs of the task. For each input and output of the task the thread executes a *raccd_register* instruction, communicating to the hardware the start address and the size of the memory region. Note that the specification of the address ranges of task inputs and outputs is already required by task-based parallel programming models. This information allows such programming models to correctly execute tasks dynamically at runtime. Therefore, RaCCD does not place any additional requirements on existing task parallel programming models. An example of these address range specifications is shown in the annotated code in Figure 2.3. The array notation in the address range specification identifies the start address of the task dependency, as the base address of the array, and the size of the dependency, given by the explicit array subscripts included.

Based on the information communicated from the runtime system to the hardware, RaCCD deactivates coherence for the inputs and outputs of the executing task. This action is performed without any further involvement of the runtime system or the programmer.

To ensure that no incoherent data is present in the cache hierarchy, RaCCD makes use of a simple mechanism. When a task finishes executing, the thread that ran the task executes a *raccd_invalidate* blocking instruction. This instruction triggers the invalidation (and write

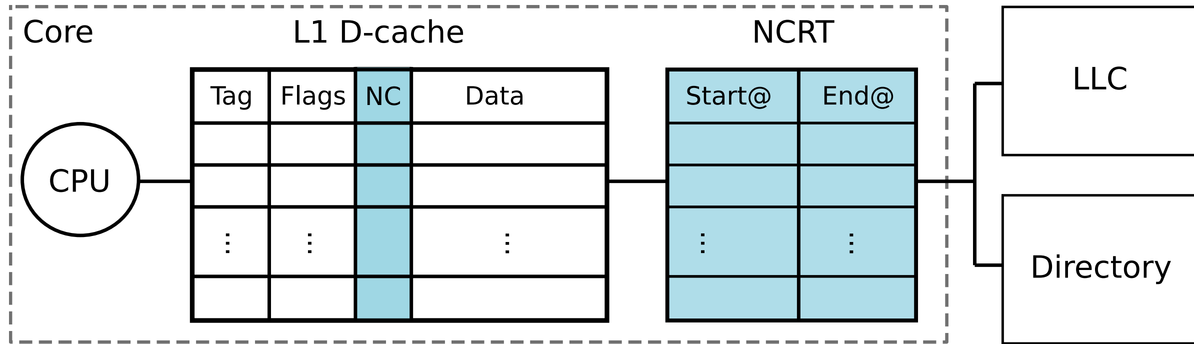



Figure 5.3: Architectural support for RaCCD: Non-Coherent Region Table (NCRT), and a Non-Coherent (NC) bit per cache line in the private cache.

back, if dirty) of any non-coherent data in the private caches of the core that executed the task. Section 5.3.3 describes the hardware support required to perform this operation. When the *raccd_invalidate* instruction is completed, any modifications made to the outputs of the finished task are either cached in the LLC only or stored in memory. As a result, the valid output data is visible to all subsequent tasks that may consume it. Thus, the thread can proceed to the wake-up phase.

### 5.3.3 Architectural Support

#### 5.3.3.1 Hardware Extensions

RaCCD introduces simple and efficient hardware support to manage non-coherent memory regions. The hardware additions, shown shaded in Figure 5.3, consist of a new per-core structure called the *Non-Coherent Region Table* (NCRT), and a *Non-Coherent* (NC) bit per cache line in the private data caches.

The NCRT holds the start and end addresses of the non-coherent memory regions specified as inputs and outputs of a task while it executes on a core. The entries of the NCRT consist of two fields to store the start and end physical address of a non-coherent memory region. Our experimental setup uses 42-bit physical addresses, but the design is open to any physical address size. The runtime system is responsible for managing the contents of the NCRT by executing the *raccd_register* and *raccd_invalidate* instructions before and after tasks execute. Private cache misses look up the NCRT to determine if the request to the LLC is coherent or non-coherent.

RaCCD also introduces a NC bit in the tag array of the private data caches to distinguish coherent from non-coherent cache lines. The NC bit is also introduced in the request and

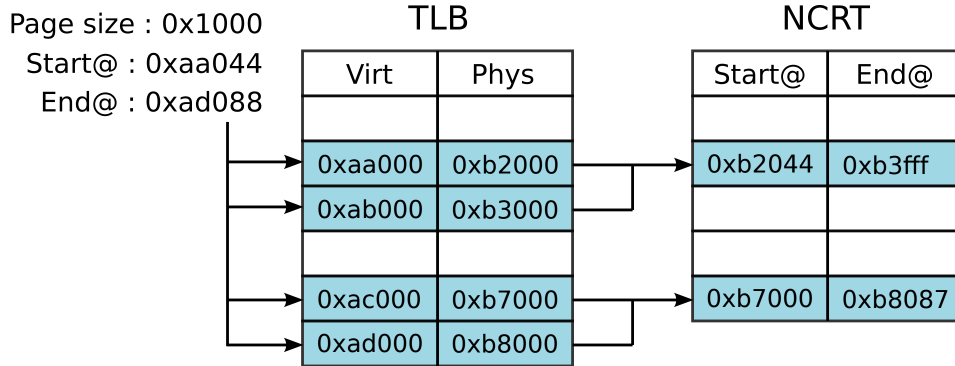



Figure 5.4: Address translation for non-coherent memory regions in the NCRT.

response messages between the private caches and the LLC, and between the LLC and the memory controllers.

#### 5.3.3.2 Registering Non-Coherent Memory Regions

Non-coherent memory regions are registered in the NCRT when the runtime system executes the *raccd_register* instruction for task inputs and outputs. The start and end addresses passed by the runtime system are virtual addresses, so they must be translated to physical addresses before registering them in the NCRT. Figure 5.4 shows a synthetic example of this process.

To translate the virtual address range to a physical address range the execution of the *raccd_register* follows an iterative process. The start address is iteratively incremented by the page size to generate a list of virtual pages that belong to the virtual address range. At every iteration, the virtual page is looked up in the TLB to retrieve the corresponding physical page. Contiguous physical pages retrieved in consecutive iterations are collapsed in the same physical address range in the NCRT. When a non-contiguous physical page is retrieved or the whole virtual address range is traversed, the physical address range is registered in the NCRT. Depending on the size of the memory region to translate, this iterative process can take multiple cycles to register a non-coherent memory region in the NCRT. The example in Figure 5.4 requires 4 TLB accesses and registers 2 collapsed regions in the NCRT.

Note that, due to the virtual-to-physical address mappings of the non-coherent memory regions, a virtual address range specified as input or output in the task annotations can require more than one entry in the NCRT. However, in the full system simulations in our evaluation, we observe that the unmodified Linux kernel allocates the contiguous virtual memory pages of the data sets of the benchmarks to contiguous physical pages, so this situation has minimal impact on our experiments. Finally, if no space is available in the NCRT, the non-coherent

memory region is not registered and accesses to this region happen as in the baseline coherent architecture.

#### 5.3.3.3 Non-Coherent Memory Accesses

Non-coherent memory requests correspond to memory references within the dependencies of the task a core is currently executing. During task execution, the core first attempts to resolve accesses in its private cache, as in the baseline architecture. In RaCCD, if the access misses in the private cache, the core then consults the NCRT to determine whether the memory reference is within a non-coherent memory region. Note that this operation adds a delay to the private cache misses. If the memory address hits in the NCRT, RaCCD triggers a non-coherent variant of the coherence transaction to the next level of the memory hierarchy. If the memory address misses in the NCRT, the access proceeds to the next level of the memory hierarchy as in the baseline architecture, i.e. as a coherent transaction.

Non-coherent requests are resolved without communicating with, or creating an entry in, the directory. To do so, non-coherent requests are sent to the LLC. If the request misses in the LLC, the LLC issues a non-coherent request to memory. Then the LLC miss is resolved when the requested data arrives to the LLC via a non-coherent response. Once the data has been filled in the LLC, the private cache miss is resolved when the LLC forwards the data to the private cache via a non-coherent response. This response sets the NC bit in the non-coherent cache line delivered to the private cache. Note that the NCRT is not accessed during this process, as the non-coherent information is carried in the request and response messages.

In the case of write-through private caches, evictions of non-coherent cache lines are silent. Writes to non-coherent cache lines use a non-coherent variant of the respective coherence transaction. This variant simply writes the data in the LLC without communicating with the directory. In the case of write-back private caches, evictions of clean non-coherent cache lines are silent. Write-backs of dirty non-coherent cache lines also use a non-coherent variant of the respective coherence transaction.

Thus, RaCCD allows cache lines specified as task inputs or outputs to flow through the memory hierarchy with coherence deactivated, lowering capacity pressure in the directory.

#### 5.3.3.4 Coherence Recovery

When a task finishes, the runtime system executes a *raccd_invalidate* instruction to invalidate the non-coherent data from the private caches. When the core executes this instruction, it sequentially traverses the cache lines of its private cache and flushes all cache lines that have

the NC bit set. Clean non-coherent cache lines are silently evicted, while dirty non-coherent cache lines are written back to the LLC via a non-coherent write-back transaction. Once the flushing is complete, the *raccd_invalidate* instruction commits and the runtime system proceeds into the wake-up phase to notify any dependent tasks of the just completed task execution.

### 5.3.4 Adaptive Directory Reduction

One of the main benefits of RaCCD is that it decreases the storage requirements of the directory without impacting performance. However, reducing the size of the directory at design time can significantly hurt the performance of non-task parallel programs. For this reason, we propose Adaptive Directory Reduction (ADR), a hardware mechanism to dynamically reconfigure the size of the directory. The goals of this technique are to capitalise on the benefits of RaCCD without impacting non-task programs, and to adapt the directory size to the exact requirements of any task-parallel program.

To dynamically resize the directory, RaCCD powers off parts of the directory, similarly to how it is done in other pipeline structures [2] and caches [109, 126]. To power off parts of the directory we use a Gated-Vdd technique [108] that drastically reduces the leakage power. When resizing the directory, we only change its number of sets while keeping the associativity constant. To support multiple directory sizes, the tag has to work for the smallest possible directory size, leaving some tag bits unused as the directory size increases. To support a maximum of 256$\times$ directory size reduction, an additional 8 bits per tag are required in the directory.

To drive reconfigurations, RaCCD adds a monitor that tracks the occupancy of the directory. When a new entry is allocated to the directory, the monitor is increased and, when an entry is evicted, the monitor is decreased. When the occupancy monitors reach certain thresholds $\theta_{inc}$ and $\theta_{dec}$, the size of the directory is increased or decreased, respectively. In our experiments, we decide to halve or double the size of directory to simplify the indexing function. Finer grain reconfigurations could be done, but would be more complex to handle. In this case, using $\theta_{inc} = 80\% \cdot current_size$ and $\theta_{dec} = 20\% \cdot current_size$ provides a hysteresis loop with good reaction time with a reduced number of reconfigurations.

When a directory reconfiguration happens, the tag bit selection and the indexing function are updated, and the contents of the directory are moved to the appropriate entries according to the new tag bit selection and indexing function. This operation is time consuming, adds power overheads and blocks directory accesses during reconfigurations. However, if reconfigurations

happen occasionally, these overheads are largely compensated by the benefits of the adaptive mechanism, similarly to set-partitioned caches [109, 126].

### 5.3.5 Additional Considerations

Task-based dataflow programming models guarantee no data races will occur for data that is only accessed from within tasks which specify the data as a dependency. OpenMP allows the programmer to step outside this guarantee by accessing such data from code outside a task specification. However, when doing so, OpenMP puts the responsibility on the programmer to avoid inconsistencies by explicitly adding code annotations (*#pragma omp flush*) to flush the data from the private caches. This means OpenMP already guarantees that data accessed as both non-coherent and coherent will only exist in the LLC or memory at the time of a transition between coherent and non-coherent, so RaCCD simply needs to allocate or deallocate a directory entry as required when a cache line transitions between coherent and non-coherent.

Under RaCCD, cache coherence is deactivated at the granularity of a cache line. To ensure correct operation, all the data in a given cache line must be eligible for coherence deactivation in order to deactivate coherence for that cache line. To comply with this requirement, the runtime system does not apply coherence deactivation for task dependencies which are smaller in size than a cache line. For the same reason, in the case of dependencies which are larger than a single cache line, if the dependency's start or end address is not aligned to a cache line boundary, the runtime system will not apply coherence deactivation to this first or last cache line in the dependency. This ensures only cache lines which are entirely contained within the bounds of the task dependency start and end address have coherence deactivated.

RaCCD allows deactivating cache coherence in task parallel programs without affecting the execution of legacy code and non task-based programs. For these programs the hardware support for RaCCD can be powered down, so the only overhead introduced is the small area of the NCRTs. Moreover, RaCCD simply restricts the data to which the cache coherence protocol applies, so it is compatible with any coherence protocol without modification or extra verification cost.

The proposed hardware support for RaCCD can be extended to support context switches and multiprogrammed workloads. A simple and effective solution is to tag the NCRTs with the OS thread ID. As a result, different processes and threads can use the NCRTs concurrently and the NCRTs do not need to be saved and restored at a context switch. When the OS migrates a thread from a source core to a destination core, the NCRT entries belonging to the thread must

also be migrated. Also, all the non-coherent data in the private cache of the source core must be invalidated with a *raccd_invalidate* instruction.

The proposed coherence recovery mechanism for RaCCD flushes all the non-coherent cache lines from the private cache of the core. To prevent performance penalties in SMT cores the non-coherent bit per cache line added to the private caches can be extended to store the thread ID of the cache line. This requires 1/2/3 extra bits for 2/4/8-way SMT cores and allows to selectively invalidate the non-coherent data of only one thread.

The *raccd_register* and *raccd_invalidate* instructions are restartable instructions which place NCRT entries into the NCRT at commit. Should a hardware interrupt occur during the virtual to physical translation required by the instructions, the execution of the instruction is abandoned and restarted from scratch after the interrupt handler returns.

## 5.4 Evaluation

This section evaluates the behaviour of RaCCD versus the baseline fully coherent system (FullCoh), and the Page Table approach (PT) [47]. The FullCoh system tracks coherence for all memory accesses, while PT and RaCCD deactivate coherence for the non-coherent data they identify. All the experiments with RaCCD consider a NCRT of 32 entries per core with a latency of 1 cycle. To implement PT we add a private/shared bit per TLB entry and intercept page faults in the simulator to record which cores access each page. When a page fault is resolved and the translation is stored in the TLB, we set the TLB entry to private if only one core has ever accessed the page, otherwise we set it to shared.

### 5.4.1 Static Directory Reduction

We study the impact of RaCCD on execution time, directory accesses, LLC hit rate, NoC traffic and energy consumption. For each metric we compare the three system types over a wide range of directory sizes to show the trend as directory capacity is reduced. Our experiments consider a range of seven directory sizes, labelled 1:*N*, meaning that the directory has *N* times less entries than the LLC. In the 1:1 configuration, both the LLC and the directory have the same number of entries, 32768 per core, while in the 1:256 configuration, the directory is reduced to just 128 entries per core. Figures 5.5, 5.6, 5.7, 5.8, and 5.9 present results for the 9 benchmarks as the capacity of the directory is reduced. Figures 5.6, 5.7, 5.8, and 5.9 show the results in pairs of benchmarks. One benchmark of every pair is shaded dark and the other shaded light,

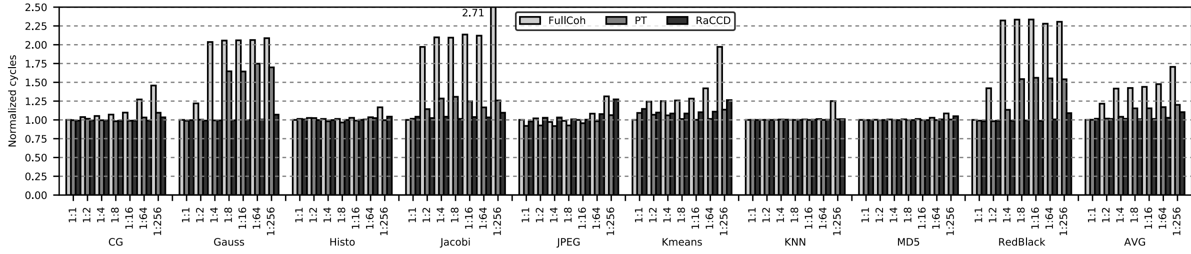



Figure 5.5: Normalised cycles by directory size. Configuration 1:N has N times less directory entries than the baseline.

and each is split in three line styles, long dash for FullCoh, fine dash for PT and solid line for RaCCD.

#### 5.4.1.1 Performance

Figure 5.5 reports the number of execution cycles for the 7 directory sizes normalised to the 1:1 configuration of FullCoh. In RaCCD and PT, accesses to non-coherent data are not tracked in the directory. On average there is a negligible performance difference between the three systems ($< 2\%$) in the 1:1 configuration. Kmeans is the only benchmark that suffers performance degradations, 9.2% in PT and 14.6% in RaCCD in the 1:1 configuration. This is due to the coherence recovery mechanism, which flushes non-coherent cache lines at the end of a task execution, harming the hit rate in the private caches and causing an increased number of writebacks from the private caches to the LLC (up by 2.3% in PT and 4.7% in RaCCD). Among the other 8 benchmarks the largest increase in writebacks seen under RaCCD or PT is 0.2%. This demonstrates that RaCCD and PT achieve competitive performance with FullCoh when the directory size is not constrained.

As the directory size is reduced, Figure 5.5 shows a significant performance impact in FullCoh. Just halving the directory size already degrades performance by 22% on average. In the 1:256 configuration we see degradations ranging from 8.6% (MD5) to 171% (Jacobi) with an average performance penalty of 71%. A dramatic drop in LLC hit ratio (from 56% to 24% on average) is the cause of such performance degradation.

In contrast to FullCoh, PT and RaCCD show significantly less performance degradation as the directory size is reduced. RaCCD tolerates directory reductions much better than PT. At 1:8 on average PT shows a 15% penalty whereas RaCCD suffers only 0.9%. At the most extreme directory reduction of 1:256, the average performance penalty for RaCCD is 10%, still less than the 15% degradation PT suffers at 1:8.

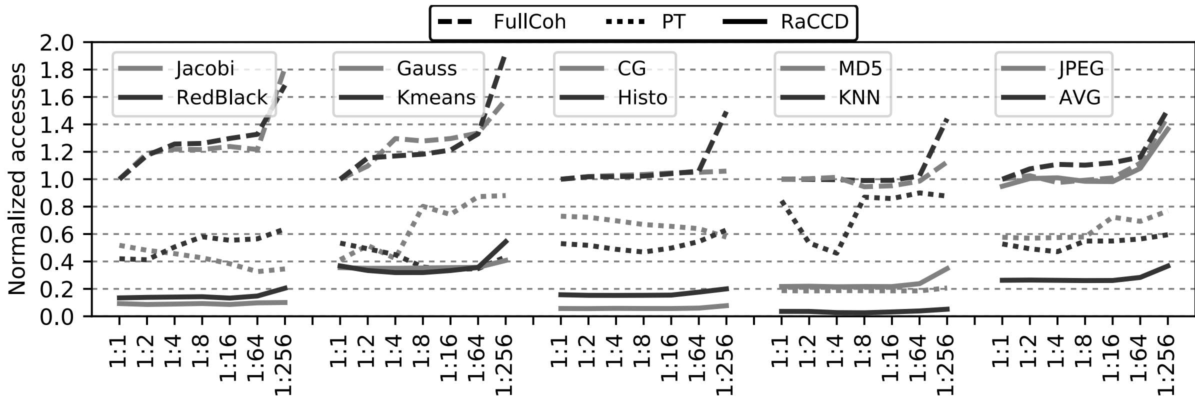



Figure 5.6: Directory accesses by directory size. Configuration 1:N has N times less directory entries than the baseline.

#### 5.4.1.2 Directory Accesses

Figure 5.6 shows the number of accesses to the directory. Results are normalised to FullCoh 1:1. The coherence deactivation in PT and RaCCD drastically reduces the number of accesses to the directory. This is because data with coherence deactivated under PT or RaCCD will have an L1 miss resolved at the LLC or main memory without reference to, or allocation in, the directory.

Figure 5.6 shows that at 1:1 RaCCD requires only between 6% and 37% of the directory accesses incurred by FullCoh across all the applications except JPEG. For JPEG, RaCCD reduces directory accesses by only 5.2% versus FullCoh. This is due to the low opportunity for RaCCD to deactivate coherence in JPEG as shown in Figure 5.1. JPEG is the only application where PT is clearly better than RaCCD. On average all approaches incur an increasing number of directory accesses as the directory size is reduced. This is the result of capacity pressure causing thrashing in the directory. On average across the directory sizes (1:1 to 1:256), RaCCD maintains an advantage ranging from 74% to 77% over FullCoh and 38% (1:256) to 53% (1:16) over PT.

#### 5.4.1.3 LLC Hit Rate

When an entry is evicted from the directory to free space for a new allocation, the corresponding cache line must also be invalidated from the LLC. As the directory comes under increasing capacity pressure with reduced size, evictions to free space for new allocations in the directory cause invalidations in the LLC, as the directory is inclusive of the LLC. This negatively impacts LLC hit rate as cache lines that will be reused get invalidated. The coherence deactivation approaches mitigate this by reducing capacity pressure on the directory. The result of Directory-L1 inclusivity is clearly apparent in Figure 5.7. We see a rapid deterioration in the LLC hit

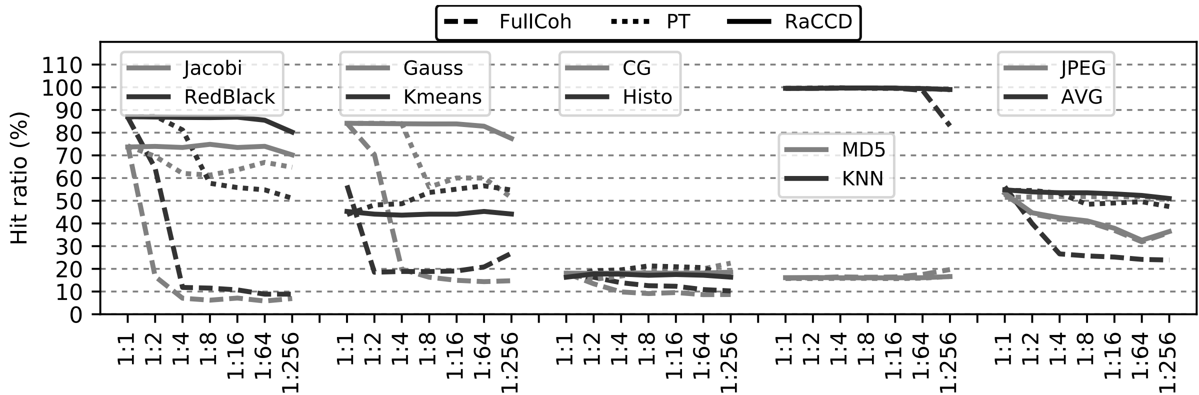



Figure 5.7: LLC hit ratio by directory size. Configuration 1:N has N times less directory entries than the baseline.

rate for FullCoh as the directory size is reduced. Moving from the 1:1 to the 1:4 configuration, the average LLC hit rate drops from 56% to 27%. In the 1:256 configuration, the LLC hit rate decreases to 24% on average.

In the case of the coherence deactivation approaches, we see the largest benefits in Gauss, Jacobi, Kmeans and Redblack. At 1:256 the hit rate under RaCCD is 3.7× FullCoh and 23% higher than PT for these four applications on average. MD5 does not suffer reduced LLC hit rate with reduced directory capacity. Its predominant memory access pattern is streaming read with very little data reuse, and thus its LLC accesses are dominated by compulsory misses, which neither directory capacity nor coherence deactivation have an impact on. Therefore, the LLC hit rate is in the low, narrow range of 16% to 20% regardless of directory size and system type. On average across all benchmarks, at the extreme directory reduction of 1:256, the hit rate of PT has deteriorated to 47% down from 55% at 1:1, which RaCCD improves on with a hit rate of 51% at 1:256 also down from 55% at 1:1

#### 5.4.1.4 Network-on-Chip Traffic

Figure 5.8 shows the NoC traffic as the directory capacity is reduced. Reduced directory capacity impacts NoC traffic negatively due to capacity pressure requiring replacement of entries which will be reused in future. KNN has a small working set size. Therefore, even under FullCoh, it only suffers a significant increase in NoC traffic at the most extreme directory reduction of 1:256, where it incurs 39% more traffic than the baseline. Under both PT and RaCCD KNN sees a negligible increase in NoC traffic of less than 1.5% at 1:256 It is clear from Figure 5.8 that NoC traffic is well constrained under both coherence deactivation approaches with RaCCD showing a slight advantage over PT on average as the directory size is reduced.

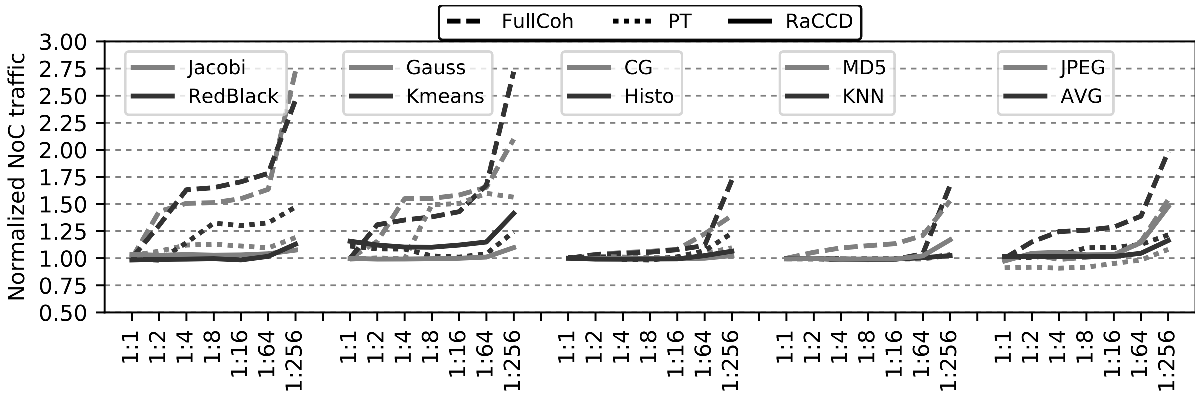



Figure 5.8: NoC traffic by directory size. Configuration 1:N has N times less directory entries than the baseline.

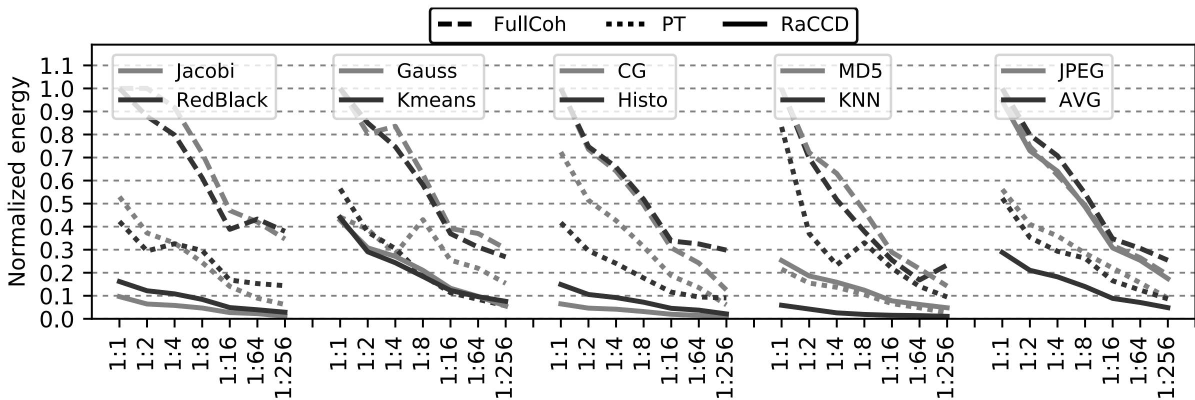



Figure 5.9: Energy consumption (bottom) by directory size. Configuration 1:N has N times less directory entries than the baseline.

At the most extreme directory size reduction of 1:256, compared with the respective 1:1, NoC traffic has grown by 19% for PT and 15% for RaCCD, whereas under FullCoh it has increased by 91%.

#### 5.4.1.5 Energy Consumption

Figure 5.9 shows the impact of reducing the directory size on energy consumption. It reports the normalised dynamic energy consumed in the directory, which represents 1.55% of the total processor energy. Comparing dynamic energy consumption for both coherence deactivation approaches against FullCoh, we see substantial energy reductions among all benchmarks except JPEG under RaCCD.

Reducing the directory size from the 1:1 configuration down to the 1:256 configuration reduces the dynamic energy in all cases. This is due to lower energy consumption per access as the directory size is reduced. Comparing the two coherence deactivation techniques, RaCCD

Table 5.1: Directory size and area

| | 1 : 1 | 1 : 2 | 1 : 4 | 1 : 8 | 1 : 16 | 1 : 64 | 1 : 256 |
|---|---|---|---|---|---|---|---|
| KB | 4224 | 2112 | 1056 | 528 | 264 | 66 | 16.5 |
| Area ($mm^2$) | 106.08 | 53.92 | 34.08 | 21.28 | 14.88 | 6.18 | 2.64 |

wins over PT on average across all directory sizes except in JPEG, where PT has a significant advantage over RaCCD. At the 1:1 configuration, RaCCD consumes 71% less dynamic energy than FullCoh and 45% less than PT and maintains a margin of at least 38% benefit over PT for all the directory sizes down to 1:256, where it consumes 80% less than FullCoh and 43% less than PT.

RaCCD also impacts the energy consumed in the NoC and LLC, which respectively make up 15% and 26% of total energy consumed. Comparing FullCoh and RaCCD for the 1:256 directory size, RaCCD saves 35% and 19% of the dynamic energy consumption in the NoC and LLC, respectively.

Table 5.1 shows the storage requirements of the directory size configurations considered in this chapter. Each directory entry is made up of 42 bits of tag and 3 bytes to store the state of the cache line and the bit-vector of sharer cores. It can be observed that the storage requirements of the directory linearly decrease along the proposed configurations, reaching up to a 97.5% reduction of the directory area for 1:256.

### 5.4.2 Adaptive Directory Reduction

The Adaptive Directory Reduction (ADR) mechanism dynamically reduces the directory size during the execution while still providing just enough capacity based on its occupancy. Figure 5.10 shows the average occupancy of the directory during the execution of the benchmarks. In FullCoh the occupancy of the directory only monotonically increases (up to capacity) during execution. In PT and RaCCD, the occupancy of the directory may increase or decrease because capacity pressure in the LLC may force the replacement of coherent lines (and their associated directory entries) with non-coherent lines (which do not have an associated directory entry). By deactivating coherence for non-coherent cache lines, PT and RaCCD achieve lower directory occupancy than FullCoh across all benchmarks. On average, FullCoh presents an occupancy of 65.7%, PT has 20.3%, and RaCCD reduces it to only 10.8%.

Figure 5.11 shows the performance of RaCCD+ADR versus the FullCoh, PT and RaCCD 1:1 configurations. All results are normalised to the FullCoh 1:1 configuration per benchmark.

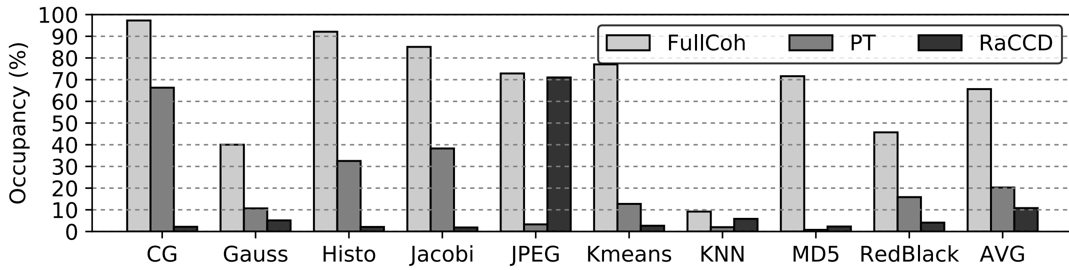



Figure 5.10: Average occupancy of the directory.

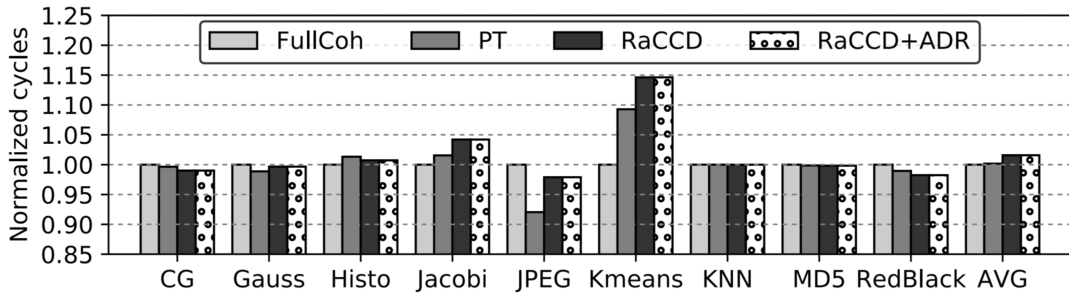



Figure 5.11: Normalised performance with adaptive directory reduction.

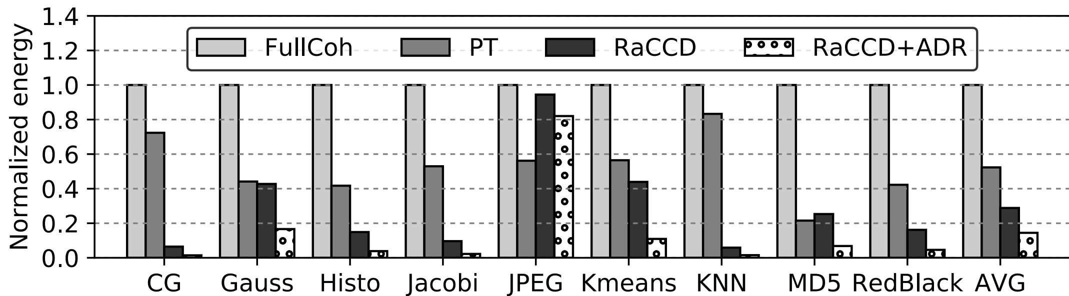



Figure 5.12: Normalised energy consumption with adaptive directory reduction.

Results show that RaCCD achieves competitive performance with FullCoh ($< 2\%$ difference on average) when comparing both in the 1:1 case. The only exception is Kmeans, where the flushing of non-coherent cache lines at the end of the task execution affects L1 cache hit rate. Figure 5.11 also shows that combining the ADR mechanism with RaCCD does not harm performance, since the overheads of resizing the directory are negligible due to the low number of reconfigurations. Of the 9 benchmarks in our analysis, the most reconfigurations occur for CG, at 70 reconfigurations. Integral Histogram experiences the second most reconfigurations with only 14 occurring. The remaining 8 benchmarks all experience less than 9 reconfigurations. The average number of reconfigurations across the 9 benchmarks is 11.8.

Reducing the capacity demand in the directory allows ADR to use smaller directory sizes and thus reduce dynamic energy consumption. Figure 5.12 shows the dynamic energy consumed

by the directory under RaCCD+ADR versus the FullCoh, PT and RaCCD 1:1 configurations. We can see that RaCCD+ADR reduces the directory energy consumption compared to RaCCD 1:1 across all benchmarks, achieving energy savings that range from 13% (JPEG) to 78% (CG). On average the reduction by RaCCD+ADR of dynamic energy consumption in the directory is 50% versus RaCCD 1:1 and 72% against PT 1:1. Note that ADR is a generic mechanism that could also be employed alongside FullCoh and PT. However, the energy savings of ADR coupled with these two approaches would be less than with RaCCD because they are not able to reduce the occupancy of the directory as much as RaCCD, as shown in Figure 5.10.

### 5.4.3 RaCCD Overheads

RaCCD introduces minimal overheads to reduce the capacity pressure on the directory. Performance-wise, the NCRT adds a delay of 1 cycle to the private cache misses, but this causes a negligible overhead of 0.1% compared to an ideal NCRT design with zero latency. In addition, augmenting the NCRT latency to 2, 3, 5 and 10 cycles only adds average overheads of 0.5%, 0.7%, 1.2% and 3.5%, respectively.

In terms of storage requirements, RaCCD only requires 5.25 KB for all the NCRTs and 1KB for the NC bits in the caches. The overheads in energy consumption are also negligible, as the NCRTs consume less than 0.1% of the total energy.

## 5.5 Summary

This chapter proposes a hardware/software co-design approach which strikingly mitigates the challenges of scaling directory-based cache coherence protocols. Our approach harnesses information present in the runtime system of task parallel programming models, which includes the precise specification of data that is going to be accessed by the tasks. With this information the runtime system directs cache coherence deactivation by communicating the addresses of non-coherent memory regions to the hardware cache coherence substrate. The microarchitecture maintains this information in cost-effective hardware structures and uses it to generate non-coherent requests for the specified memory regions. As a consequence, the data specified in the task annotations is never tracked in the directory, so our proposal dramatically reduces its capacity requirements. Our approach allows a 64× smaller directory with only a 2.8% performance degradation.

# Chapter 6

# Runtime-Driven LLC Management

With increasing core counts, heterogeneity and the ever progressing gulf in performance between processors and memory, the scalability of the memory hierarchy is becoming an increasingly crucial aspect of computer architecture. LLCs in particular have increased in importance as an enabler of scalability. Despite the increasing proportion of die area devoted to LLC in today's tiled architectures, LLC capacity remains a very limited resource. When the LLC cannot satisfy a request due to lack of capacity, the power and performance penalty is very high, requiring an off-chip rather than on-chip access.

Even when the LLC can satisfy a request, the latency and power cost of that request is becoming increasingly variable as NoC diameters connecting NUCA LLCs increase in line with core counts. To mitigate off-chip LLC misses and NUCA LLC hit costs, there has been much recent work on how best to organise and utilise LLCs, as described in Chapter 2.1. However, to date, existing methods have not taken advantage of the valuable information captured by task-based programming models in order to optimise the management of LLCs.

Leveraging task-based programming models, this chapter proposes a hardware software co-designed approach to optimising the management of LLCs. In this approach, the runtime system uses its knowledge of the application's current and future use of data to decide whether to place data in the LLC or not, and if so, where and for how long. The runtime system then communicates this information to the microarchitecture which, with minimal hardware support, can use it to perform optimisations within the on-chip memory hierarchy. The proposal achieves a 1.14× speedup over the baseline system on average, whereas an enhanced variant of the state-of-the-art Reactive-NUCA proposal [66] from the literature achieves only 1.11×.

## 6.1 Introduction

For some decades now, caches have been the primary instrument available to architects in the battle to satisfy the data demands from increasingly parallel and specialised compute units. The accelerating trends towards parallelism and heterogeneity in compute units has prompted a commensurate growth in the sophistication and complexity of the entire memory hierarchy, particularly in the caches. LLCs have increased rapidly in importance and size as they are the last line of defence against expensive off-chip accesses to main memory. At the same time, their best and worst case performance metrics have diverged ever further as the diameter of the NoC connecting them scales up with increasing core counts. Many recent works (detailed in Section 2.2.3) have focused on how to achieve the best access latency from LLCs given the increasing variability of their access characteristics. Other work has harnessed information from runtime systems in order to optimise victim selection [91] in the LLC. Despite this large body of existing work the topic of how best to utilise widely distributed NUCA caches continues to be an area attracting much interest, due to the increasing importance of the LLC.

The changing trends in computer architecture in the past decade and a half have also spurred much work in programming model research and design, seeking new ways to program hardware in the era of parallelism and specialisation. Today's hardware has many new features in addition to increased parallelism, such as heterogeneous cores, accelerators, dynamic frequency scaling, simultaneous multi-threading and increased diameter of on and off-chip networks with larger NUCA and NUMA effects. These features make once popular approaches to shared memory parallel programming such as fork-join programming models too inflexible to take advantage of the potential performance available on modern hardware.

Task-based programming models (described in Section 2.3.3) have emerged as one of the most important antidotes to the problem of programming on the increasingly varied and complex hardware architectures now available. Such programming models structure a program as a series of tasks with defined data inputs and outputs. They therefore allow the runtime system to take on responsibility for dynamically scheduling and synchronising the work on the hardware resources available at runtime while at the same time simplifying the requirements on the programmer.

This contribution proposes Runtime-Driven LLC Management (RDLLCM), a hardware software co-designed approach to improving the utilisation and operation of existing cache hierarchies. RDLLCM requires only small and straightforward changes in hardware and utilises information already captured in existing task-based programming models. In task-based

programming models, the programmer defines the data dependencies of each task, and whether they are read or written by the task. In existing task-based programming models, the runtime system uses this information to create a task dependency graph and dynamically schedule and synchronise the application's execution. This proposal communicates this information from the runtime system to the hardware, where it is used to drive optimisation of the placement of data within the shared NUCA LLC.

## 6.2 RDLLCM Design

This section describes the RDLLCM proposal. First, it gives an overview of the design and the opportunity for a runtime driven LLC management. Then it presents the detail of the software - hardware interface required, followed by the additions in the hardware microarchitecture and the runtime system that enable the proposal.

### 6.2.1 Overview

RDLLCM is a hardware/software co-designed approach to optimising the utilisation of NUCA caches by leveraging information present in the runtime system of task-based programming models. The goal of RDLLCM is to use knowledge about the application's use of data, present at the runtime system level, to optimise how that data is managed in the memory hierarchy. At execution time, the runtime system of a task-based programming model knows what data a task will use before the task starts to execute, and whether that data will be read or written by the task. It also has knowledge about the future data needs of the application, via the task dependence graph it maintains. RDLLCM leverages this knowledge to optimise utilisation of the limited resources in the on-chip memory hierarchy.

The three new data placement choices RDLLCM introduces within the memory hierarchy are listed and labelled in Table 6.1. They all take advantage of the implicit guarantee in task-based programming models that, during the execution of a task, no data races will occur on the task's inputs and outputs, due to the synchronisation between tasks enforced by the runtime system. The first two data placements in Table 6.1 are conceptually straightforward. In LLC Bypass, if the data will not be reused by future tasks in the TDG, it is better not to allocate it in the LLC at all and place it directly in the L1 cache of the core executing the task. This will reduce demand on the LLC, thus providing a larger share of the LLC to data that will make use of it. In LLC Local Slice, the task's dependencies are mapped to the local LLC slice where the task executes, and then flushed from the LLC slice on the completion of the

Table 6.1: RDLLCM Memory Hierarchy Data Placements

| Label | Description |
|---|---|
| Bypass | Bypass the LLC if it would be of no benefit to cache data there, e.g. when the data is unlikely to be reused from the LLC. |
| Local Slice | Caching data in the LLC slice nearest to where it is used and invalidating the cached data when it is finished being used. |
| Cluster Repl | Clustered Replication of data in the LLC while it is read only. |

task. This minimises the NUCA cost associated with any LLC accesses the task makes to its dependencies.

To understand the motivation for LLC Clustered Replication, consider the following scenario in a typical LLC implementation. In the LLC, only one copy of each cache line may be present in the LLC at a given time. If this cache line is shared among all the cores in the system, the core on the same slice as the LLC bank storing the cache line will experience the optimal NUCA distance to access the cache line in the LLC. All other cores that access it will experience varying degrees of NUCA distance and hence latency in their LLC accesses to the cache line. The final RDLLCM data placement choice, LLC Clustered Replication, seeks to mitigate this problem. In LLC Clustered Replication, the runtime system conceptually divides the tiled architecture into clusters of slices. For this contribution we have evaluated dividing a 16 slice tiled architecture into 4 clusters, each containing 4 cores. LLC Clustered Replication allows replication of read-only data in the LLC, in a controlled manner. Each cluster may maintain its own copy of the read-only cache line, thus reducing the worst case NUCA distance from the chip-wide NoC diameter to the cluster-wide NoC diameter, and reducing contention for the cache line in the LLC by up to a factor of $4\times$.

In order to achieve the necessary co-operation between the runtime system and the hardware cache coherence substrate for RDLLCM, the runtime system communicates the address ranges of task dependencies to the hardware, alongside a bitmask which governs how the dependency will be handled in the memory hierarchy. On the hardware side, minimal microarchitectural support is introduced to store the necessary information provided by the runtime system and enable optimisations in the memory hierarchy based on it. The next sections detail RDLLCM's software - hardware interface, hardware support and runtime system extensions.

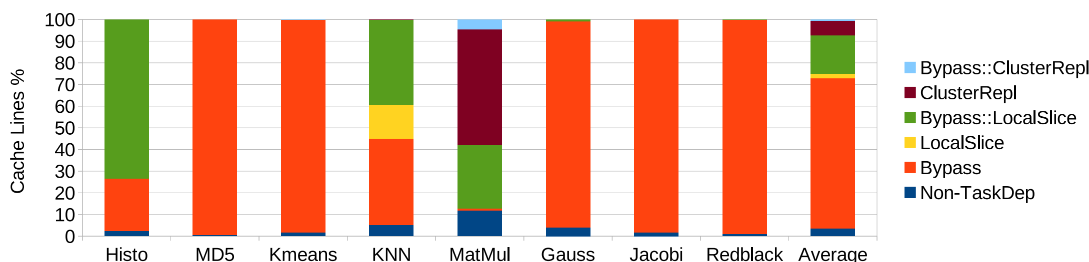



Figure 6.1: Cache Lines by RDLLCM category.

### 6.2.2 Opportunity for Runtime Driven LLC Management

As can be seen in Figure 6.1, the task dependencies (i.e. all the data except the Non-TaskDep category) on average make up over 96% of the total data accessed. This provides the runtime system with knowledge about a large proportion of the applications' data and its use, and therefore wide scope to optimise the behaviour of the applications' data in the memory hierarchy. For the task dependencies, Figure 6.1 shows which RDLLCM data placement category, or combination of categories, RDLLCM places data in during execution. Section 6.2.5 details how the runtime system decides which of the data placement categories to apply for a given task dependency.

### 6.2.3 Software - Hardware Interface

RDLLCM defines an interface between the runtime system and the hardware architecture so they can co-operate in the management of the memory regions that hold task dependencies. The interface consists of three new ISA instructions that are issued by the runtime system as required at the start and end of task executions. The instructions use the concept of *map_mask* and *core_mask*, which are bit vectors that specify to which cores the operation is applied. In our 16-core experimental setup the masks contains 16 bits, one per core.

- *rdllc_register(initial_virtual_address, size, map_mask)*: Before executing a task, the runtime system uses this instruction to create an entry in a *Runtime Region Table* (RRT), a hardware storage table in the core executing the task. This entry informs the microarchitecture about the memory region corresponding to a data dependency within the programming model and how it should be handled in the cache hierarchy. To this end, the map_mask specifies the set of LLC slices to which the private cache has to request and write back the cache lines belonging to the memory region.

- *rdllc_flush(initial_virtual_address, cache_level, core_mask)*: The runtime system uses this instruction to flush cache lines belonging to a particular RDLLCM entry from the private caches or from the LLC slices of the cores specified in the core_mask.
- *rdllc_invalidate(initial_virtual_address, size, core_mask)*: This instruction invalidates an entry in the RRT of the cores specified in the core_mask. Without any entry in the RRT, memory is treated as in the baseline system from a caching perspective.

In addition there is a single memory mapped register with 1 bit per core which the hardware uses to signal completion of a cache flush to the runtime system. An alternative implementation could communicate from the runtime system to the hardware through a memory mapped schema, instead of via the ISA additions. We have selected the ISA extension approach for runtime system to hardware communication due to its simplicity. As only a small number of instructions are issued per executed task, both solutions are expected to behave similarly.

### 6.2.4 RDLLCM Hardware Extensions

RDLLCM introduces simple, small and efficient hardware support to manage RDLLCM memory regions. The hardware storage structures and the operations based on them are described next.

#### 6.2.4.1 Microarchitectural Storage Structures

The hardware additions, shown shaded in Figure 6.2, consist of a new per-core structure called the RRT. The RRT holds the information of the memory regions RDLLCM communicates from the runtime system via the RDLLCM software - hardware interface. The entries of the RRT consist of three fields to store the start and end physical address of a RDLLCM memory region along with its associated RDLLCM map_mask, which specifies the LLC slices to be used to store the cache lines belonging to the memory region. Our experimental setup uses 42-bit physical addresses, but the design is open to any physical address size.

#### 6.2.4.2 Microarchitectural Operation

*(A) - Registering and De-registering RRT Entries*

The runtime system is responsible for managing the contents of the RRT by executing the *rdllc_register* and *rdllc_invalidate* instructions. When the runtime system executes the *rdllc_register* instruction, a new entry is registered in the core's RRT. The start address passed

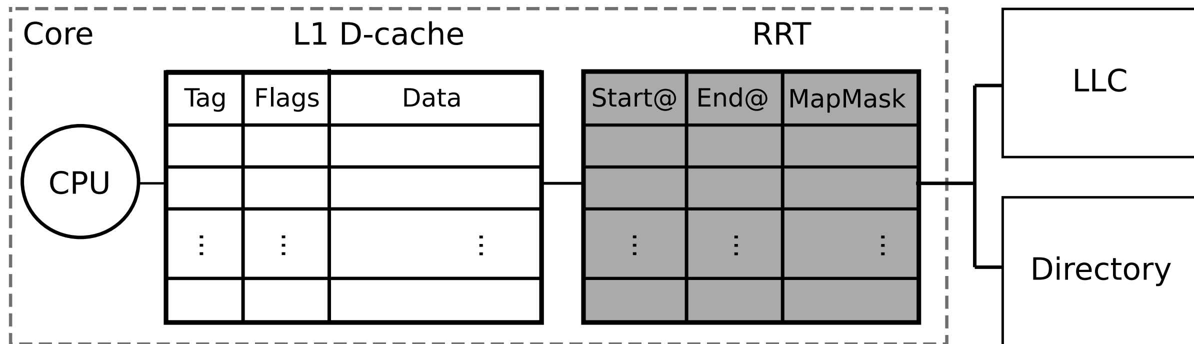



Figure 6.2: Architectural support for RDLLCM: Runtime Region Table (RRT).

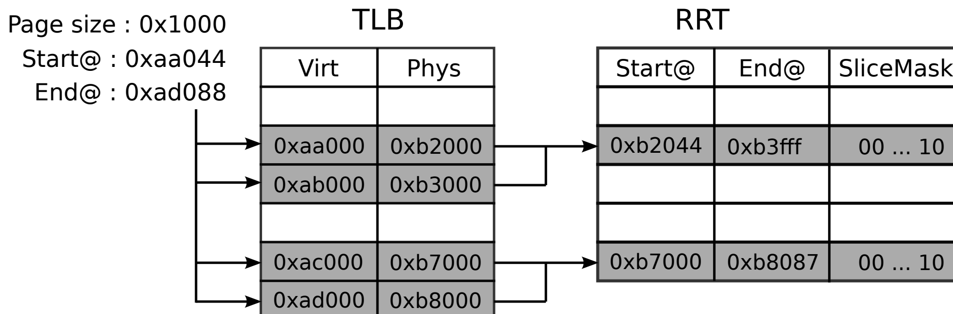



Figure 6.3: Address translation for memory regions in the RRT.

by the runtime system is a virtual address, so it must be translated to a physical address before registering the entry in the RRT. Figure 6.3 shows a synthetic example of this process.

To translate the virtual address range to a physical address range the execution of the *rdllc_register* follows an iterative process. The start address is iteratively incremented by the page size to generate a list of virtual pages that belong to the virtual address range. At every iteration, the virtual page is looked up in the TLB to retrieve the corresponding physical page. Contiguous physical pages retrieved in consecutive iterations are collapsed in the same physical address range in the RRT. When a non-contiguous physical page is retrieved or the whole virtual address range is traversed, the physical address range is registered in the RRT. Depending on the size of the memory region to translate, this iterative process can take multiple cycles to register a memory region in the RRT. The example in Figure 6.3 requires 4 TLB accesses and registers 2 collapsed regions in the RRT.

An RRT entry is de-registered in the RRT each time the runtime system executes the *rdllc_invalidate* instruction. The execution of the rdllc_invalidate follows the same iterative process of virtual to physical address translation as just outlined for the rdllc_register instruction. By iterating over the virtual address range specified in its parameters it removes the respective physical address entries in the RRT.

*(B) - Memory References under RDLLCM*

Under RDLLCM, memory referencing instructions executed by a core first attempt to resolve

the references in the core's private cache, as in the baseline system. If the access misses in the private cache, the core then consults the RRT to determine whether the memory reference is within the address range of an RRT entry. Note that this operation adds a delay to the private cache misses. If the address of the memory reference does not hit in the RRT, the memory reference proceeds as it would in the baseline system. If the address of the memory reference does hit in the RRT, the map_mask of the RRT entry is checked to identify the LLC slice where the request has to be sent. The runtime system guarantees that the contents of the map_mask field match one of the following conditions:

(i) All bits in SliceMask set to zero (LLC Bypass): If zero bits are set in the entry's map_mask this indicates it should bypass the LLC. RDLLCM triggers an LLC Bypass variant of the coherence transaction directly to the memory controller. The memory controller replies to such a request by forwarding the cache line directly to the requesting private cache, bypassing the LLC. Upon receiving the reply from the memory controller, the private cache allocates an entry for the cache line. When the cache line has to be written back, the RRT is checked and the writeback is sent directly to main memory, also bypassing the LLC.

(ii) Exactly 1 bit set (LLC Local Slice): If exactly one bit of the entry's map_mask is set, this indicates the dependency is LLC Local Slice mapped. In order to communicate with the LLC the private cache sets the destination slice for its coherence message to the position of the single set bit in the map_mask and then sends the coherence messages to that LLC slice when requesting and writing back a cache line.

(iii) Exactly 4 bits set (LLC Cluster Replication): The 4 set bits match the identities of the cores of the local cluster of the core executing the task. In LLC Cluster Replication, cache lines which comprise the memory region described by the RRT entry are address interleaved among the 4 slices which make up the cluster. The hardware calculates which of the 4 slices in the local cluster to set as the destination for its coherence message based on the address of the cache line that is requested or written back.

*(C) - Cache Flushing*

The *rdllc_flush* instruction flushes the cache lines of the desired cache level of the cores specified in the map_mask. The instruction's cache_level argument indicates whether the flush is applied to an LLC Slice or to a private cache. To flush the private caches all the cache lines are checked and, if they belong to the memory region identified by the initial_virtual_address argument, they are self invalidated. During this procedure, dirty cache lines are written back to the LLC slices specified in the map_mask of the RRT entry, or sent directly to the memory controller if no bit is set. If the cache_level argument indicates this is an LLC flush, the LLC

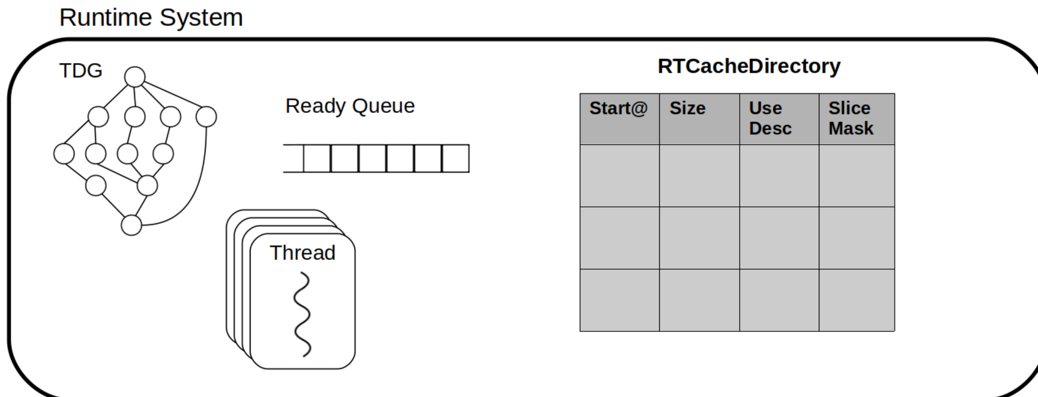


Figure 6.4: Runtime system structures with additions for RDLLCM shaded.

slice issues a flush transaction for all cache lines in the address range of the RRT entry. If any of these lines are present within the LLC slice, they are evicted from the LLC slice (which triggers invalidation and writebacks if necessary in any private caches holding copies of the line, due to LLC - private cache inclusivity). In both cases, the runtime system waits for completion of the flush event by testing a memory mapped flag that the hardware sets once the flush is complete.

### 6.2.5 RDLLCM Runtime System Extensions

The typical behaviour of a thread in a task parallel program is a process consisting of three consecutive phases, repeated for each task the thread executes. In the scheduling phase, the thread requests a ready task from the scheduler, which selects a ready task-based on a scheduling policy. Then, in the task execution phase, the scheduled task is executed. Finally, in the wake-up phase, yet to be executed tasks in the task dependency graph that depend on the task just completed are analysed. If all the dependencies of any such task are satisfied, it is marked as ready and placed in the ready queue, from where the scheduler may allocate it to a thread for execution in the scheduling phase.

In addition to this baseline behaviour of the runtime system, RDLLCM may execute the *rdllc_register*, the *rdllc_flush* and the *rdllc_invalidate* instructions at the beginning or at the end of any task. Note these RDLLCM instructions are executed as needed by the operational model of RDLLCM, explained in Section 6.2.5.2, and not necessarily on every task start or finish.

#### 6.2.5.1 Runtime System Data Structures

Figure 6.4 shows the existing components of the runtime system which are relevant to RDLLCM as non-shaded. In order to enable the three RDLLCM data placement choices listed in Table 6.1, the runtime system is augmented with the shaded RTCacheDirectory structure shown in Fig-

ure 6.4. The RTCacheDirectory contains a unique entry for each task dependency encountered by the runtime system. RTCacheDirectory entries are created as the runtime system creates tasks, and they are updated when a task using that dependency starts or finishes execution.

The RTCacheDirectory is indexed by the unique start address of each dependency, and each entry stores the fields described in Table 6.2. The information stored in the RTCacheDirectory is used by the runtime system as outlined in Section 6.2.5.2 to make decisions about which RDLLCM data placement to apply to each task dependency. In particular, the UseDesc field of a dependency is incremented every time a task that uses that dependency is created and it is decremented every time a task that uses the dependency is executed. The SliceMask field is a bitvector that encodes the LLC slices where the dependency is mapped to at any point in time.

Table 6.2: RDLLCM runtime system RTCacheDirectory fields

| Label | Description |
|---|---|
| Size | Size in bytes of the dependency. |
| UseDesc | A *use descriptor* that counts how many times the dependency will be used in future. |
| SliceMask | Tracks which, if any, LLC slices the dependency is currently mapped to. |

#### 6.2.5.2 Runtime System Operational Model

RDLLCM extends the operational model of task-based programming models to decide which RDLLCM data placement to apply to task dependencies and communicate the necessary information to the hardware at task start and task end points, through the RDLLCM software - hardware interface described in Section 6.2.3.

After the runtime system has scheduled a task to a particular thread, but before it allows the task to start executing, it iterates over the data dependencies of the task. For each data dependency of the task, it decides which of the three RDLLCM data placement choices to perform. It then communicates three characteristics of the dependency to the hardware through the *rdllc_register* instruction: dependency start address, dependency size and dependency map_mask. The runtime system uses the information in the RTCacheDirectory to make a decision on which data placement to apply as outlined in the flowchart in Figure 6.5.

*(A) - Bypass LLC:* If the use descriptor in the dependency's entry in the RTCacheDirectory shows that there are no outstanding tasks in the task dependency graph which use the dependency, then the runtime system chooses the Bypass LLC data placement for the dependency. Zero bits are set in the mask_map communicated by the *rdllc_register* instruction to indicate

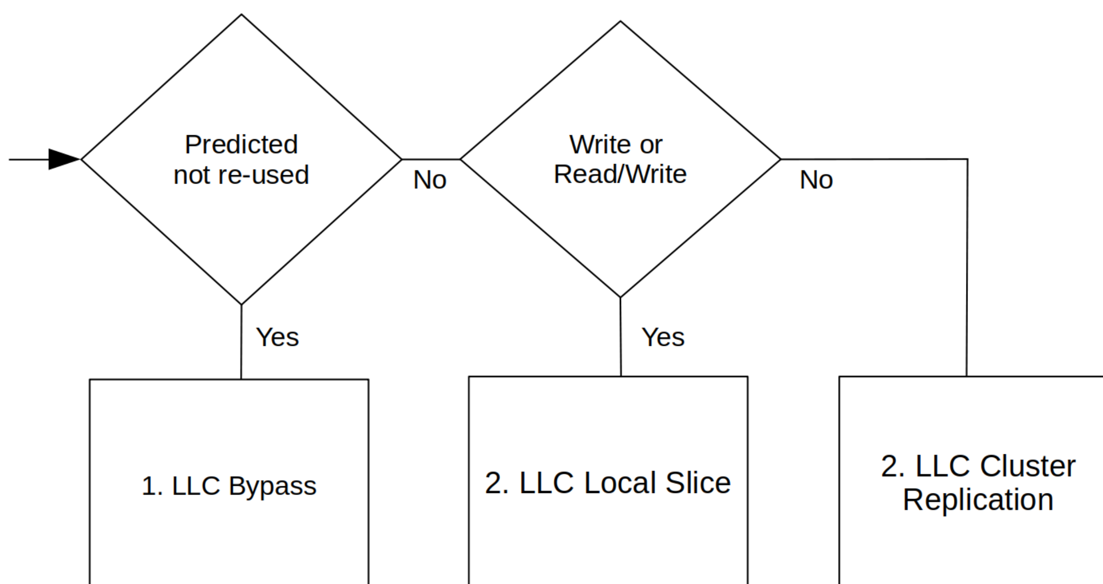



Figure 6.5: Runtime decision structure for task dependencies at task start.

Bypass LLC. On task end, the runtime system executes the *rdllc_flush* instruction to flush the dependency from the L1 cache of the core specified in the SliceMask field of the RTCacheDirectory entry corresponding to the dependency, and then it executes the *rdllc_invalidate* instruction to clear the entry from the RRT of that core.

*(B) - Local Slice LLC Mapping:* If the dependency is an Output (write-only) or Input/Output (read-write) task dependency the runtime system chooses the Local Slice LLC data placement for the task dependency. Before the task starts the *rdllc_register* instruction is issued with exactly 1 bit set in the map_mask, specifically the bit in the mask position corresponding to the Local Slice LLC the dependency is mapped to, and the SliceMask field of the RTCacheDirectory is updated accordingly. On task end, the runtime system executes the *rdllc_flush* instruction to flush the dependency from the LLC slice and the private caches of the cores the data is mapped to (specified in the SliceMask of the RTCacheDirectory field). Then, it executes a *rdllc_invalidate* instruction to clear the corresponding RRT entries.

*(C) - Cluster Replicated LLC Mapping:* If the dependency has not been assigned Bypass or Local Slice data placement via the previous two tests (*A* and *B*), then the dependency is: (i) reused in the future TDG and (ii) an Input (read-only) dependency. In this case, the runtime system chooses the Cluster Replicated data placement. Before the task execution the runtime system issues the *rdllc_register* instruction with exactly 4 bits set in the map_mask, specifically the 4 bits in the mask positions corresponding to the 4 slices which make up the

local cluster of the core executing the task. These bits are also set to 1 in the SliceMask field of the RTCacheDirectory entry for that dependency. At the end of the task, the mapping of the dependency remains in the hardware to be used by future tasks which may reuse the dependency and any of its cache lines still present in the LLC.

Note that this operational model, combined with the pre-existing task synchronisation enforced by the runtime system, ensures that no coherence issues will arise during the execution. Task dependencies that are bypassed or written to are always eagerly invalidated at the end of the task, so neither the cache hierarchy nor the RRT will contain any stale data when a subsequent task which uses the same dependency starts.

### 6.2.6 Additional Considerations

Task-based dataflow programming models guarantee no data races will occur for data that is only accessed from within tasks which specify the data as a dependency. OpenMP allows the programmer to step outside this guarantee by accessing such data from code outside a task specification. However, when doing so, OpenMP puts the responsibility on the programmer to avoid inconsistencies by explicitly adding code annotations (*#pragma omp flush*) to flush the data from the private caches. This means OpenMP already guarantees that data accessed as both a task dependency and non-task dependency will only exist in the LLC or memory at the time of a transition between task dependent and non-task dependent, and thus presents no obstacle to the correct operation of RDLLCM

In the hardware, RDLLCM operates at cache line granularity. To ensure correct operation, any given cache line must be entirely contained within the start and end address of a task dependency in order for RDLLCM to alter its behaviour in the memory hierarchy. To comply with this requirement, the runtime system treats task dependencies which are smaller than a cache line in size as non-task dependencies, i.e. it does not modify behaviour for them compared to the baseline system. For the same reason, in the case of dependencies which are larger than a single cache line, if the dependency's start or end address is not aligned to a cache line boundary, the runtime system will treat the first and last cache line in the dependency as a non-task dependency. This ensures only cache lines which are entirely contained within the bounds of the task dependency start and end address have their behaviour in the memory hierarchy modified.

RDLLCM allows optimising the handling of task dependencies of task parallel programs in the memory hierarchy without affecting the execution of legacy code and non task-based

programs. For these programs the hardware support for RDLLCM can be powered down, so the only overhead introduced is the small area of the RRTs.

The proposed hardware support for RDLLCM can be extended to support context switches and multiprogrammed workloads. A simple and effective solution is to tag the RRTs with the OS thread ID. As a result, different processes and threads can use the RRTs concurrently and the RRTs do not need to be saved and restored at a context switch. When the OS migrates a thread from a source core to a destination core, the RRT entries belonging to the thread must also be migrated. Also, all the data in the private cache of the source core must be invalidated with a *rdllc_invalidate* instruction.

Both the *rdllc_register* and *rdllc_invalidate* instructions are restartable instructions which place RRT entries into the RRT at commit. If a hardware interrupt occurs during the execution of these instructions, the virtual to physical translations are aborted and the instruction is restarted from scratch after the interrupt handler returns.

## 6.3 Evaluation

This section presents the results and analysis from the evaluation of RDLLCM. In all the experiments, RDLLCM uses a 32-entry RRT per core with a latency of 1 cycle. RDLLCM is compared against both the baseline system and an enhanced variant of R-NUCA [66].

### 6.3.1 Metrics and Comparators

Section 6.3.2 presents a number of results that help provide insight into the operation and benefits of the RDLLCM proposal. It compares RDLLCM against the baseline system (see Section 3.2) and an enhanced variant of R-NUCA (eR-NUCA), described below.

R-NUCA [66] allows replication of data within the LLC in order to reduce NUCA distance and therefore access latency. Under R-NUCA the only cache lines which are replicated in the LLC are the cache lines of memory pages which exclusively contain instructions. Such pages are read-only and typically shared among all cores.

The comparative results and analysis presented in Section 6.3.2 employ eR-NUCA. To improve R-NUCA, eR-NUCA increases R-NUCA's scope to replicate data within the LLC. In addition to memory pages containing instructions, eR-NUCA allows replication for pages which are read-only data and shared by more than one core. The operation of eR-NUCA's LLC replication policy for shared read-only pages is identical to its existing behaviour with instruction pages, while the page remains shared read-only. An important difference between

pages containing only instructions and pages containing read-only data is that the former stay read-only for the duration of a benchmark's execution, while the latter may not. If and when cache lines containing shared read-only data experience a write access, they transition to become read-write. In eR-NUCA all copies of cache lines belonging to a page are flushed from the LLC upon such a shared read-only to read-write page transition. This mechanism is consistent with the approach used in the R-NUCA proposal for flushing the LLC on private to shared page transitions. After a shared read-only to read-write page transition (and for the remainder of the execution) read-write pages are treated as read-write pages are in the R-NUCA proposal. This enhancement to R-NUCA provides it with more opportunity to optimise data placement for reduced NUCA latency within the LLC.

In the rest of this Chapter the three systems used in the comparative evaluation are labelled as follows:

- **BL-NUCA**: The baseline NUCA cache, an S-NUCA cache as described in Section 3.2. Cache lines are address interleaved among the 16 slices which make up the LLC.
- **eR-NUCA**: The enhanced R-NUCA proposal, described in Section 6.3.1.
- **RDLLCM**: The RDLLCM proposal, described in Section 6.2.

The metrics analysed in this Chapter are presented in Figures 6.6 to 6.15. Each of these figures contains one cluster of bars per benchmark. Each cluster is made up of three bars, one for each of the three systems tested: BL-NUCA, eR-NUCA and RDLLCM.

The performance of the three systems is presented first in Section 6.3.2.1. Then, the remaining metrics which provide the basis for a comparative analysis of the performance are presented. These include behaviour within the LLC in Section 6.3.2.2 and L1 hit ratio and NUCA NoC distance in Section 6.3.2.3. Section 6.3.2.4 analyses the costs introduced by the cache flushing required in eR-NUCA and RDLLCM. Finally, Section 6.3.2.5 considers the energy impacts and Section 6.3.2.6 the overheads of RDLLCM.

### 6.3.2 Results and Analysis

#### 6.3.2.1 Performance Speedup

Firstly, Figure 6.6 reports the speedup in execution time for both eR-NUCA and RDLLCM in comparison to BL-NUCA. RDLLCM achieves large speedups over both BL-NUCA and eR-NUCA in 3 of the 7 benchmarks, namely Gauss, Jacobi and RedBlack. For these three

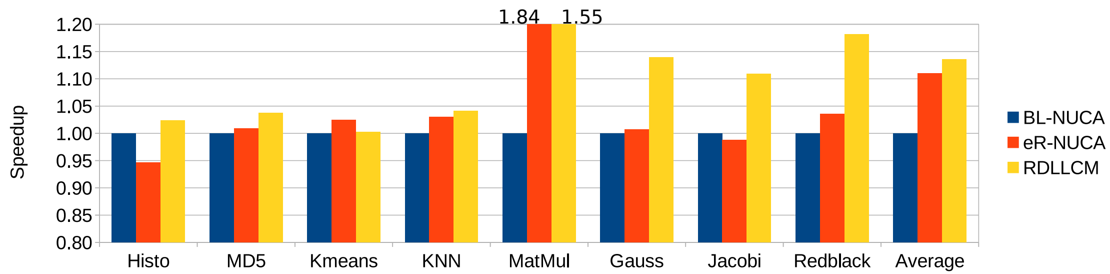



Figure 6.6: Speedup of eR-NUCA and RDLLCM versus BL-NUCA.

benchmarks, RDLLCM achieves speedups of 1.14×, 1.11× and 1.18× respectively, compared to BL-NUCA. The corresponding speedups for eR-NUCA's versus BL-NUCA are all worse at 1.01×, 0.99× and 1.04× respectively. In 3 of the remaining 5 benchmarks, Histo, MD5 and KNN, RDLLCM achieves speedups over both BL-NUCA and eR-NUCA. In Kmeans, RDLLCM performs similarly to BL-NUCA, while eR-NUCA achieves a modest speedup of 1.02× over BL-NUCA. In MatMul, both eR-NUCA (1.84×) and RDLLCM (1.55×) produce large speedups over BL-NUCA. Across all benchmarks on average, eR-NUCA provides a speedup of 1.11×, whereas RDLLCM achieves a 1.14× speedup over BL-NUCA.

#### 6.3.2.2 LLC Accesses and Hit Ratio

Figure 6.7 demonstrates how eR-NUCA and RDLLCM impact the number of Load (LD) and Store (ST) accesses made to the LLC. It shows the number of LD/ST accesses made to the LLC by both proposals, normalised to the number of LD/ST accesses made by BL-NUCA. Figure 6.8 shows the LLC hit ratio achieved.

Figure 6.7 shows that in Gauss, Jacobi and Redblack, eR-NUCA requires significantly increased LD/ST accesses to the LLC compared to BL-NUCA. This is caused by an increased rate of conflicts in the LLC (as can be seen in Figure 6.11) as a result of eR-NUCA assigning private data to the local slice of the accessing core and thus experiencing more contention for space in the LLC than is the case in BL-NUCA. Due to LLC - L1 inclusivity, the extra conflicts in the LLC cause some extra misses in the L1 caches, negatively impacting the L1 hit ratio by around 2%for these benchmarks, as can be seen in Figure 6.9.

RDLLCM reduces the number of LLC LD/ST accesses required versus both BL-NUCA and eR-NUCA in all benchmarks, ranging from 0.99× as many LD/ST accesses as BL-NUCA in KNN, down to 0.17× as many as BL-NUCA in RedBlack. This is due to the LLC bypass feature of RDLLCM which entirely bypasses the LLC for dependencies that the runtime system predicts are not reused. On average over all the benchmarks, RDLLCM performs 0.55× the

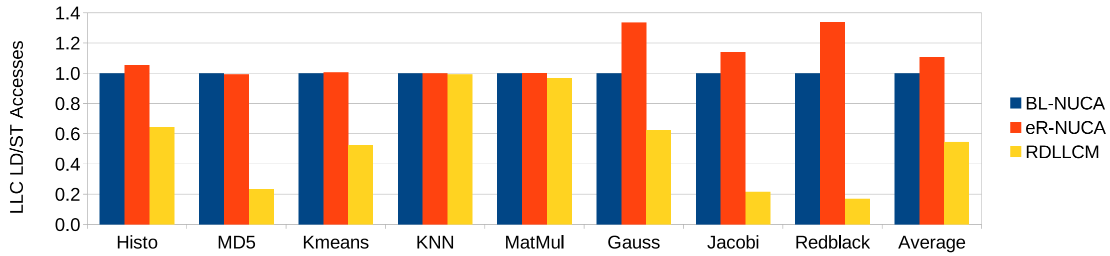



Figure 6.7: Total number of LD/ST accesses to the LLC by eR-NUCA and RDLLCM normalised to the number performed by BL-NUCA.

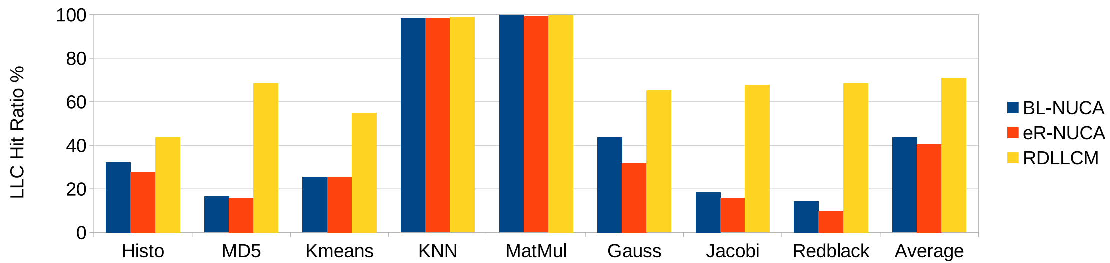



Figure 6.8: The LLC hit ratio experienced by BL-NUCA, eR-NUCA and RDLLCM.

LD/ST accesses to the LLC compared to BL-NUCA, whereas eR-NUCA makes 1.11× as many LD/ST accesses to the LLC versus BL-NUCA.

In terms of LLC hit ratio, shown in Figure 6.8, there is a small difference of 3% between BL-NUCA and eR-NUCA on average. RDLLCM however shows significantly higher hit ratios in the LLC for all the benchmarks except KNN and MatMul, which are both close to 100% LLC hit ratio in all three of BL-NUCA, eR-NUCA and RDLLCM. In all the other benchmarks, the LLC hit ratio is substantially increased for RDLLCM versus BL-NUCA and eR-NUCA. This is due to RDLLCM's lower demand for LLC capacity due to LLC Bypass. This reduced capacity pressure on the LLC means that the remaining non-bypassed data which is allocated in the LLC experiences less LLC conflicts and therefore cache lines can remain in the LLC longer before being evicted, thus increasing the hit ratio.

#### 6.3.2.3 L1 Hit Ratio and NoC distance to NUCA LLC

Figure 6.9 presents the L1 hit ratio results. It shows that RDLLCM does not negatively impact L1 hit ratios (there is less than 0.001% difference between BL-NUCA and RDLLCM L1 hit ratio on average). It shows a slight reduction in L1 hit ratio for Jacobi, Redblack and Gauss under eR-NUCA (around 2% each) compared to BL-NUCA. This is caused by increased

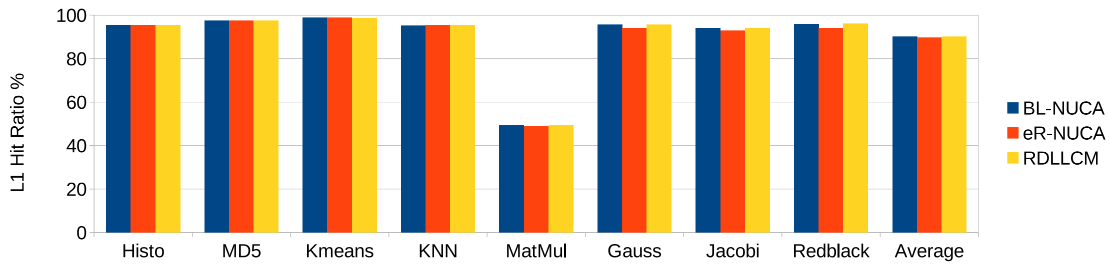



Figure 6.9: The L1 hit ratio experienced by BL-NUCA, eR-NUCA and RDLLCM.

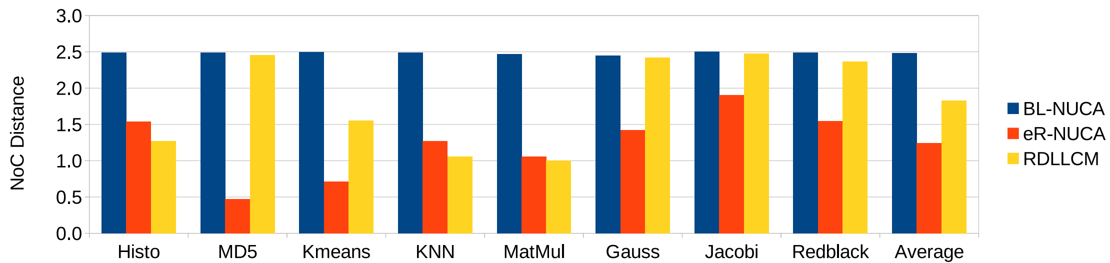



Figure 6.10: The average NoC distance for L1 accesses to the LLC (local LLC slice = 0, other slices = number of NoC links traversed).

invalidations in the L1 caused by additional conflicts in the LLC when eR-NUCA maps private data to the local slice of the accessing core. Due to LLC - L1 inclusivity these LLC conflicts trigger eviction of any copies of the replaced LLC cache line from any private L1s they are present in.

Figure 6.10 shows the distance travelled in the NoC for messages from the L1 caches to the NUCA LLC. In this NoC distance metric, an L1 cache accessing its local LLC slice counts as distance 0, an L1 accessing a next-door neighbour slice's LLC (i.e. directly adjacent to its north, south, east, or west) is counted as distance 1, and so on for greater distances. This metric shows that eR-NUCA succeeds in its goal of reducing the NoC distance to LLC data. eR-NUCA achieves this by mapping private data pages into the local LLC slice of the accessing core instead of being address interleaved among the 16 cores of the processor, and by replicating read-only shared data in multiple LLC slices. eR-NUCA reduces the NoC distance from the 2.49 average for BL-NUCA down to 1.24. The NoC distance for RDLLCM (1.83) shows less improvement over BL-NUCA than eR-NUCA does, however this is to be expected, given that under RDLLCM the LLC only contains data which was not LLC bypassed.

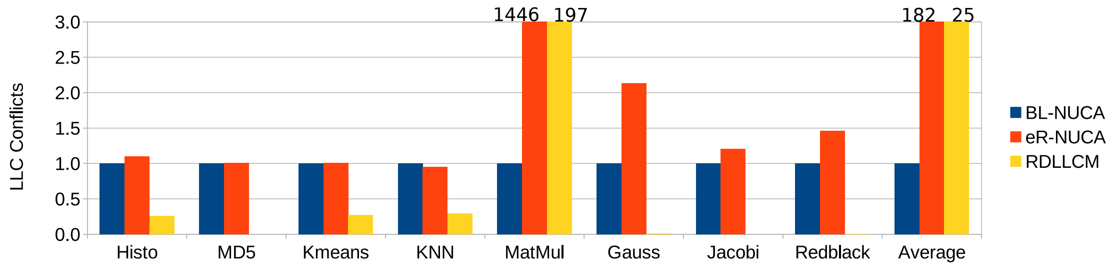



Figure 6.11: Total number of cache line conflicts experienced in the LLC by eR-NUCA and RDLLCM normalised to the number experienced by BL-NUCA.

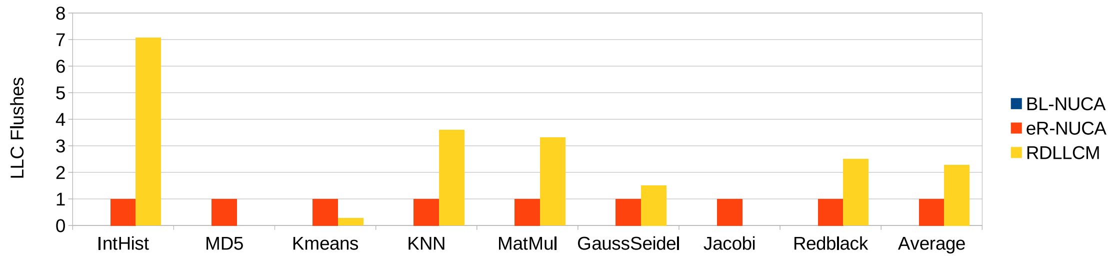



Figure 6.12: Total number of cache line flushes experienced in the LLC by RDLLCM normalised to the number experienced by eR-NUCA.

#### 6.3.2.4 LLC Conflicts and Cache Line Flushing

In the baseline system (BL-NUCA) cache lines are only evicted from the LLC due to cache line conflicts. A cache line conflicts occurs when placing a new cache line into the cache requires evicting an existing cache line to make room. Both eR-NUCA and RDLLCM introduce a second situation where cache lines get evicted from the LLC. As noted in the design of both eR-NUCA and RDLLCM (see Section 6.2), flushing of cache lines is required at certain points in the operation of both mechanisms. Flushing means evicting cache lines from the LLC on request. The impact of eR-NUCA and RDLLCM on both cache line conflicts and cache line flushes in the LLC, compared to BL-NUCA is examined next.

Figure 6.11 demonstrates how eR-NUCA and RDLLCM affect the number of cache line conflicts in the LLC, compared to BL-NUCA. It shows that RDLLCM significantly reduces the number of LLC conflicts in comparison to BL-NUCA and eR-NUCA in all benchmarks except MatMul. Reduced LLC conflicts can be attributed to two features of RDLLCM. Firstly, for task dependencies which RDLLCM placed in the local slice of the LLC to the core executing the task, RDLLCM flushes the dependency from the local LLC slice at the end of the task. Cache lines flushed in this manner are therefore removed from the LLC earlier than they would

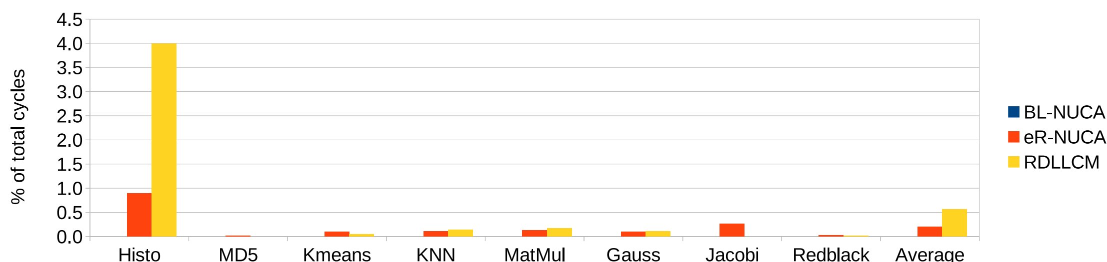



Figure 6.13: The percentage of total cycles spent flushing LLC cache lines by eR-NUCA and RDLLCM.

be under BL-NUCA, leaving a free space in the cache for a new cache line to be allocated in, without causing a conflict. Secondly, due to the bypassing feature of RDLLCM, task dependencies which the runtime system predicts will not be reused are not allocated in the LLC at all, reducing the demand for capacity and therefore the number of conflicts in the LLC.

Under eR-NUCA, there is an increased number of cache conflicts in the LLC for four different benchmarks compared to BL-NUCA: MatMul, Gauss, Jacobi and Redblack. RDLLCM increases the number of cache conflicts versus BL-NUCA in only one benchmark, MatMul. These increases in conflicts in the LLC versus BL-NUCA can be attributed to two possible factors in both eR-NUCA and RDLLCM: (i) placing data in the local LLC slice of the core using it, and (ii) replicating data in the LLC. Placing data in the local LLC slice of the core using it limits the available capacity for that data in the LLC to the size of the local LLC slice. Replicating data in the LLC reduces the effective capacity of the LLC compared to BL-NUCA, which allows only a single copy of a cache line in the LLC. As can be seen from Figure 6.1, RDLLCM performs both the LocalSlice and ClusterReplication data placements for significant portions of the benchmark's working set, leading to the increase in LLC conflicts. Importantly, the increased LLC conflicts caused by RDLLCM in MatMul do not negatively impact the hit ratios in the LLC or the L1 caches, and therefore do not negatively affect performance.

Figure 6.12 shows the number of cache lines flushed by RDLLCM normalised to the number flushed by eR-NUCA. Figure 6.13 show the amount of LLC flushing introduced by both the eR-NUCA and RDLLCM policies has negligible impact on the performance of the benchmarks. The percentage of the total execution time of the benchmarks spent in flushing the LLC is 0.2% and 0.6% of the execution time on average for eR-NUCA and RDLLCM respectively.

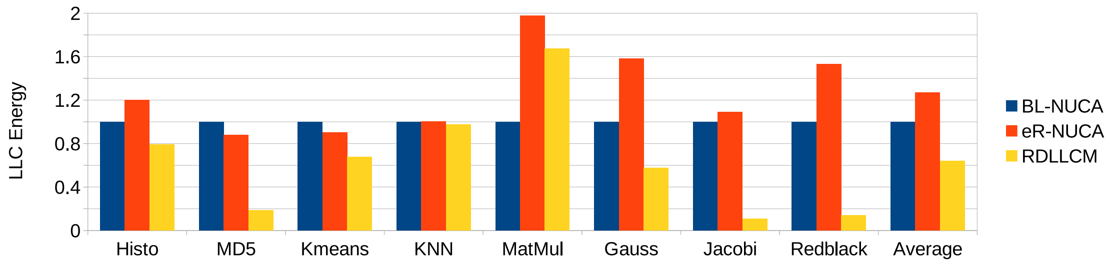



Figure 6.14: Energy Consumption in the LLC by eR-NUCA and RDLLCM, normalised to BL-NUCA.

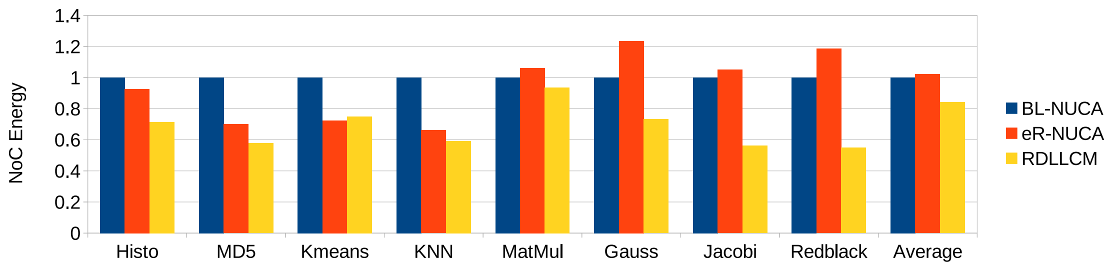



Figure 6.15: Energy Consumption in the NoC by eR-NUCA and RDLLCM, normalised to BL-NUCA.

#### 6.3.2.5 Energy Consumption

RDLLCM and eR-NUCA modify the operation of the LLC. This impacts energy consumption within both the LLC and the NoC which connects the core-private L1 caches to the shared and distributed LLC. Figures 6.14 and 6.15 present the impact of RDLLCM and eR-NUCA on the energy consumption in the LLC and the NoC respectively.

In Figure 6.14 we see that eR-NUCA incurs significantly more energy consumption for the Gauss, Jacobi and Redblack benchmarks at 1.58×, 1.09× and 1.53× more than BL-NUCA respectively. The increased energy consumption experienced under eR-NUCA by these benchmarks is due to an increased number of conflicts in the LLC caused by eR-NUCA. MatMul incurs increased energy consumption in the LLC under both eR-NUCA and RDLLCM for the same reason.

RDLLCM significantly reduces the energy consumed in the LLC for several benchmarks. The largest energy saving in the LLC under RDLLCM is achieved in Jacobi which consumes only 0.11× the energy in the LLC consumed by BL-NUCA. Redblack, Gauss, MD5 and Kmeans also see significant reductions in energy consumption. This is due to the LLC Bypass feature of RDLLCM which bypasses the LLC entirely for task data dependencies which the

runtime system predicts will not be re-used in future. On average across all the benchmarks eR-NUCA incurs 1.27× the energy consumption of BL-NUCA in the LLC, while RDLLCM consumes 0.64× the energy of BL-NUCA.

Figure 6.15 shows the energy consumption in the NoC. eR-NUCA consumes between 0.66× (KNN) and 1.24× (Gauss) the energy in the NoC compared to BL-NUCA. RDLLCM consumes between 0.55× and 0.94× the energy in the NoC compared to BL-NUCA. On average across all the benchmarks eR-NUCA consumes 1.02× and RDLLCM consumes 0.84× the energy in the NoC compared to BL-NUCA.

#### 6.3.2.6 RDLLCM Overheads

RDLLCM introduces minimal hardware overheads to manage the LLC. In terms of storage requirements, the RDLLCM introduces a 32-entry RRT per core, which requires a total storage of 6.25 KB. The RRTs also add negligible overheads in energy consumption, as they consume less than 0.1% of the total energy consumed by the simulated architecture.

Performance-wise, the RRTs add a delay of 1 cycle to the private cache misses, but this has a negligible impact on the overall performance. Compared to an ideal RRT design with zero latency the average performance overhead is 0.1% and augmenting the latency of the RRTs up to 4 cycles maintains the average performance overhead below 1%.

## 6.4 Summary

This contribution details RDLLCM, a proposal for Runtime-Driven LLC Management. Across a wide range of benchmarks the average amount of the benchmark's working set captured by task dependency specifications of the programming model is 96%. The runtime system therefore has valuable knowledge about a large proportion of the working set of the benchmarks. This knowledge, which the runtime system already uses to correctly synchronise and schedule the execution, is exploited to drive a more efficient handling of the data in the hardware cache coherence substrate.

RDLLCM communicates the relevant information from the runtime system to the hardware architecture via small and simple additions to the software - hardware interface. The proposal shows the hardware can store this information with small microarchitectural additions and act upon it to enable optimisations within the memory hierarchy. As a consequence, on average across 8 important benchmarks, RDLLCM achieves an average speedup of 1.14× over the

baseline system while an enhanced variant of a state-of-the-art proposal from the literature produces a 1.11× speedup.

# Chapter 7
# Conclusions

This thesis has presented a range of techniques leveraging information already captured in modern task-based programming models. These techniques tackle the challenges of scaling the memory hierarchy to service ever larger core counts. This chapter details the main conclusions from the contributions of this thesis and then outlines the possibilities for future research they suggest. Finally, this chapter lists the publications resulting from this thesis and acknowledges the financial and technical support that made it possible.

## 7.1 Goals, Contributions and Main Conclusions

Since the start of the multicore era 15 years ago, achieving performance improvements has depended on efficiently utilising increasing parallelism in the number of cores sharing memory. This trend towards increasing numbers of cores sharing memory has necessitated the introduction of deeper, more complex cache hierarchies and an increasingly distributed physical architecture of the memory hierarchy. With these trends in hardware architecture come many potential performance pitfalls for software.

In main memory, the move to increasingly distributed shared memory has resulted in ever more stark NUMA performance penalties for NUMA-oblivious software. In the large ccNUMA system used in Chapter 4 for example, there is an 8.9× bandwidth and 3.3× latency penalty when accessing the most distant NUMA regions versus the local NUMA region. This issue is even more problematic for NUMA-oblivious software given that only a small percentage (6.25% in the case of the system used in Chapter 4) of the shared main memory available to any core is of the fast, NUMA-local type. The majority of the shared memory is the slower, NUMA distant type.

The trend of increasing core counts has also brought to the fore fundamental challenges to the scalability of cache hierarchies and cache coherence. This is now true even for cache

hierarchies built upon directory based coherence protocols, the more scalable of the two approaches to providing cache coherence. Storage and power costs for the coherence directory increase as core counts rise. This has prompted work both on how to limit the storage required per directory entry and how to limit demand for entries in the directory. Mirroring the case with main memory, as the diameter of physical distribution increases for shared caches, techniques are required to avoid worst case access paths to the shared cache.

In the interests of programmability, these characteristics such as NUMA, NUCA and the dimensions of the cache levels in the memory hierarchy are not exposed to the programmer from a functional perspective. Therefore, the performance of software is very significantly limited unless the software layer *chooses* to take responsibility for managing these issues in the memory hierarchy. In response to the increasing need for software to leverage performance based on knowledge of the implementation details of the underlying hardware, task-based programming models have emerged as an important approach to parallel software development. Such programming models help in bridging the gap between the need for ease of programmability and the reality of dealing with the wide diversity and complexity of the underlying hardware which software is expected to take advantage of.

Noting these recent developments on both sides of the hardware-software divide, the goal of this thesis is to harness untapped potential present in existing task-based programming models to tackle the scalability problems in the memory hierarchy outlined above.

The first contribution of this thesis demonstrates in detail state-of-the-art techniques on both the software and the hardware side to scale ccNUMA to a very large system size, comprising 16 sockets, 288 cores and 2 TBs of main memory in a single operating system instance. On the hardware side this system uses a two level hierarchical directory in order to provide coherence at the LLC to main memory interface. This contribution characterises in detail the coherence and memory traffic profiles at the interface between the multiple LLCs and main memory. It shows the mutual benefit of the hierarchical directory approach in hardware combined with runtime guided NUMA optimisations in software. The runtime system can uniformly distribute the data among the NUMA regions of the system and then maximise data locality by scheduling tasks near where their data is located. Together, these hardware and software techniques combine to provide significantly improved performance and energy costs versus NUMA-oblivious executions.

The second contribution tackles the problem of scalability of the cache coherence directory that maintains coherence among the per-core private caches of a multicore processor. Using information present in existing task-based programming models it precisely identifies, in

advance of access, which data requires coherence and which data does not. The contribution demonstrates that using this information in the cache hierarchy enables deactivating coherence for a significant proportion of the working set of a range of benchmarks from a varied number of computational domains. The proposal improves performance where the directory size is static and under capacity pressure, or, allows drastically reducing the directory size and energy cost with negligible effect on average performance of the benchmarks. Furthermore, it describes a dynamic scheme for fitting the directory capacity to demand during execution which can cut energy consumption in the directory with no negative impact on performance.

The final contribution of this thesis proposes a runtime-driven management of the LLC in a multicore processor. It targets a more efficient use of NUCA LLC hardware by exploiting semantic information captured in existing task-based programming models about data use. The contribution observes that a large proportion of the working set of a range of task-based benchmarks are captured as task dependencies by the programming model. Based on the guarantees enforced by the programming model to correctly schedule and synchronise the tasks and their data dependencies several changes in the utilisation are allowable and investigated. The goal of the proposed changes is to reduce capacity pressure on the LLC. This is achieved by not allocating data in the LLC when it will not be reused and minimising the NUCA costs by mapping data to the local LLC slice and replicating shared read-only data in multiple LLC slices. The changes made to the operation of the memory hierarchy are safe and allowable based on programming model guarantees enforced by the runtime system, augmented with some limited flushing of the cache at certain points determined by the runtime system. The proposal achieves a speedup of $1.14\times$ over the baseline S-NUCA system, while an enhanced variant of the state-of-the-art Reactive-NUCA proposal from the literature can only achieve a $1.11\times$ speedup.

The first contribution of this thesis utilises a real and very large ccNUMA system, while the second and third contributions pursue a simulation based approach to evaluation. In the first contribution, the access provided to us by the system vendor (to otherwise non-user accessible hardware counters in the Bull Coherence Switch ASIC) enabled us to carry out a novel analysis of the ability of a NUMA-aware runtime scheduler to constrain the NUMA data sharing and coherence traffic for a range of benchmarks. Conversely, in the second and third contributions, due to the novel microarchitectural support required for both contributions, they could only be investigated in a simulation based approach. Due to the very large execution time overhead imposed by detailed microarchitectural simulation, the system and problem sizes employed in simulation based studies cannot be as large as might be used in a real system. Despite this,

simulation has been a valuable and important tool in computer architecture research for decades, as evidenced by the number and importance of works cited in this thesis which use it.

In the first contribution, given the large benchmark problem sizes required to stress the capacity of the off-chip NUMA interconnect and the hierarchical coherence directory cache layer implemented by the BSC, such a characterisation study would not be feasible through a simulation based approach. The second and the third contributions of the thesis focus on proposals within the on-chip memory hierarchy, and thus can be explored with smaller problem and system sizes than in the first contribution, making them feasible to study via simulation. While the third contribution is quite a focused proposal for a single layer of the memory hierarchy (LLC cache), the second contribution could have applications in settings other than the intra-socket cache coherence studied in the second contribution. The challenges of providing cache coherence also exist at lower levels of the memory hierarchy, for example in the inter-socket coherence among discrete LLC caches, as studied in the first contribution. The proposal for coherence deactivation in the second contribution of this thesis is amenable to use at any level of the coherence directory, and can be used orthogonally to other techniques for improving the scalability of coherence directories, such as hierarchical coherence directories and techniques that reduce the storage per directory entry. Indeed as the memory hierarchy grows deeper and more complex with features such as multiple LLCs in a single socket in multi-chip modules, and very large off-chip L4 LLCs, all avenues to scaling coherence will need to be pursued. The evaluation of the second proposal demonstrates the accuracy and specificity with which the runtime system can direct coherence deactivation, which is applicable to any level of the memory hierarchy where cache coherence is required and it becomes more useful as system sizes (and thus the costs of directory coherence) scale up.

## 7.2 Future Work

The work presented in this thesis suggests many possible avenues for future work. Detailed below are three which stand out as of particular potential.

- *NUCA aware runtime-driven scheduling.* In the work presented in this thesis we consider optimisations to the microarchitectural implementation of cache coherence and the operational model of the cache hierarchy. These optimisations are based on the runtime system's knowledge about what data tasks use or will use and optimise their handling in the caches. Future work could go further and alter the scheduling within the runtime system based on the optimisations already proposed. For example, the third

contribution of the thesis proposes the runtime system decides and directs placement of data dependencies within the distributed NUCA LLC. An interesting future work could investigate how the runtime scheduler could exploit the LLC NUCA placement knowledge maintained by the runtime system to optimise the scheduling of task to cores, fostering data locality and reuse within a single NUCA bank of a shared LLC.

- *Runtime-driven management of coherence islands.* The contributions in this thesis all build upon a fully cache coherent baseline. As compute unit parallelisation and specialisation proceeds, memory hierarchies are likely to add features which break the coherent paradigm, such as scratchpad memories, disjoint memory spaces and coherence islands within multicore processors. The contributions developed in this thesis are interesting in such non-coherent hardware also. The second contribution for scaling cache coherence among private caches in ccNUMA is also applicable to settings where shared memory abstractions are provided across disjoint memory spaces attached to specialised heterogeneous compute units, or where islands of coherence are implemented among groups of cores to enhance scalability in homogeneous architectures.

- *Runtime defined memory hierarchy.* One of the main strengths of the contributions of this thesis is that they provide significant benefits without imposing any new requirements on the programming model and requiring very small additions to the hardware software interface and within the hardware microarchitecture. With a view to far future scalability, the techniques developed in this thesis on the runtime side could be extended to take advantage of more extensive hardware impacts which would allow the runtime system to direct the dynamic configuration of the memory hierarchy resources rather than simply optimising the operation and utilisation of a static configuration of them.

## 7.3 Publications

This section lists below the publications that resulted from the work on this thesis.

- **Paul Caheny**, Marc Casas, Miquel Moretó, Hervé Gloaguen, Maxime Saintes, Eduard Ayguadé, Jesús Labarta, Mateo Valero: Reducing Cache Coherence Traffic with Hierarchical Directory Cache and NUMA-Aware Runtime Scheduling. PACT 2016: 275-286

- **Paul Caheny**, Lluc Alvarez, Said Derradji, Mateo Valero, Miquel Moretó, Marc Casas: Reducing Cache Coherence Traffic with a NUMA-Aware Runtime Approach. IEEE Trans. Parallel Distrib. Syst. 29(5): 1174-1187 (2018)
- **Paul Caheny**, Lluc Alvarez, Mateo Valero, Miquel Moretó, Marc Casas: Runtime-assisted cache coherence deactivation in task parallel programs. SC 2018: 35:1-35:12
- Runtime-Driven Non-Uniform Cache Architecture (NUCA) Management. Under preparation.

## 7.4 Financial and Technical Support

This thesis 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 2017-SGR-1414 and 2017-SGR-1328), by the RoMoL ERC Advanced Grant (grant agreement 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° 671697 and 779877.

# Bibliography


[1] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. "An Evaluation of Directory Schemes for Cache Coherence". In: *International Symposium on Computer Architecture (ISCA)*. 1988, pp. 280–298.

[2] D. H. Albonesi, R. Balasubramonian, S. G. Dropsbo, S. Dwarkadas, E. G. Friedman, M. C. Huang, V. Kursun, G. Magklis, M. L. Scott, G. Semeraro, P. Bose, A. Buyuktosunoglu, P. W. Cook, and S. E. Schuster. "Dynamically tuning processor resources with adaptive processing". In: *IEEE Computer* 36.12 (Dec. 2003), pp. 49–58. DOI: 10.1109/MC.2003.1250883.

[3] Mohammad Alisafaee. "Spatiotemporal Coherence Tracking". In: *International Symposium on Microarchitecture (MICRO)*. 2012, pp. 341–350. DOI: 10.1109/MICRO.2012.39.

[4] Lluc Alvarez, Marc Casas, Jesús Labarta, Eduard Ayguadé, Mateo Valero, and Miquel Moreto. "Runtime-Guided Management of Stacked DRAM Memories in Task Parallel Programs". In: *Proceedings of the 32nd International Conference on Supercomputing*. ICS '18. ACM, 2018, pp. 218–228. ISBN: 978-1-4503-5783-8. DOI: 10.1145/3205289.3205312. URL: http://doi.acm.org/10.1145/3205289.3205312.

[5] Lluc Alvarez, Miquel Moretó, Marc Casas, Emilio Castillo, Xavier Martorell, Jesús Labarta, Eduard Ayguadé, and Mateo Valero. "Runtime-Guided Management of Scratchpad Memories in Multicore Architectures". In: *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques*. PACT '15. 2015, pp. 379–391. ISBN: 978-1-4673-9524-3. DOI: 10.1109/PACT.2015.26.

[6] Lluc Alvarez, Lluís Vilanova, Miquel Moreto, Marc Casas, Marc Gonzàlez, Xavier Martorell, Nacho Navarro, Eduard Ayguadé, and Mateo Valero. "Coherence Protocol for Transparent Management of Scratchpad Memories in Shared Memory Manycore Architectures". In: *Proceedings of the 42nd Annual International Symposium on Com-*

*puter Architecture*. ISCA '15. ACM, 2015, pp. 720–732. ISBN: 978-1-4503-3402-0. DOI: 10.1145/2749469.2750411. URL: http://doi.acm.org/10.1145/2749469.2750411.

[7] James Archibald and Jean-Loup Baer. "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model". In: *ACM Trans. Comput. Syst.* 4.4 (Sept. 1986), pp. 273–298. ISSN: 0734-2071. DOI: 10.1145/6513.6514. URL: https://doi.org/10.1145/6513.6514.

[8] *Programmer's Guide for ARMv8-A. Version 1.0. 2015.*

[9] Bull Atos. *bullion S2 Factsheet*. 2015. URL: https://atos.net/wp-content/uploads/2017/06/bullion_s2_e7v3.pdf (visited on 08/21/2017).

[10] M. Awasthi, K. Sudan, R. Balasubramonian, and J. Carter. "Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches". In: *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. 2009, pp. 250–261.

[11] Manu Awasthi, David W. Nellans, Kshitij Sudan, Rajeev Balasubramonian, and Al Davis. "Handling the Problems and Opportunities Posed by Multiple On-Chip Memory Controllers". In: *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. PACT '10. Vienna, Austria: Association for Computing Machinery, 2010, pp. 319–330. ISBN: 9781450301787. DOI: 10.1145/1854273.1854314. URL: https://doi.org/10.1145/1854273.1854314.

[12] J. Balart, A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé, and J. Labarta. "Nanos Mercurium: a Research Compiler for OpenMP". In: *European Workshop on OpenMP (EWOMP)*. 2004, pp. 103–109.

[13] F. Baskett, T. Jermoluk, and D. Solomon. "The 4D-MP graphics superworkstation: computing+graphics=40 MIPS+MFLOPS and 100000 lighted polygons per second". In: *Digest of Papers. COMPCON Spring 88 Thirty-Third IEEE Computer Society International Conference*. 1988, pp. 468–471.

[14] B. M. Beckmann, M. R. Marty, and D. A. Wood. "ASR: Adaptive Selective Replication for CMP Caches". In: *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. 2006, pp. 443–454.

[15] B. M. Beckmann and D. A. Wood. "Managing Wire Delay in Large Chip-Multiprocessor Caches". In: *37th International Symposium on Microarchitecture (MICRO-37'04)*. 2004, pp. 319–330.

[16] N. Beckmann and D. Sanchez. "Jigsaw: Scalable software-defined caches". In: *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*. 2013, pp. 213–224.

[17] Chester Bell. "Multis: A New Class of Multiprocessor Computers". In: *Science (New York, N.Y.)* 228 (May 1985), pp. 462–7. DOI: 10.1126/science.228.4698.462.

[18] Pieter Bellens, Kannappan Palaniappan, Rosa M. Badia, Guna Seetharaman, and Jesus Labarta. "Parallel Implementation of the Integral Histogram". In: *Advanced Concepts for Intelligent Vision Systems*. Ed. by Jacques Blanc-Talon, Richard Kleihorst, Wilfried Philips, Dan Popescu, and Paul Scheunders. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 586–598. ISBN: 978-3-642-23687-7.

[19] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. "The PARSEC Benchmark Suite: Characterization and Architectural Implications". In: *International Conference on Parallel Architectures and Compilation (PACT)*. 2008, pp. 72–81. DOI: 10.1145/1454115.1454128.

[20] S. Blagodurov, A. Fedorova, S. Zhuravlev, and A. Kamali. "A case for NUMA-aware contention management on multicore systems". In: *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2010, pp. 557–558.

[21] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. "Cilk: An Efficient Multithreaded Runtime System". In: *Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 1995, pp. 207–216. DOI: 10.1145/209936.209958.

[22] Robert L. Bocchino Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. "A Type and Effect System for Deterministic Parallel Java". In: *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2009, pp. 97–116.

[23] François Broquedis, Nathalie Furmento, Brice Goglin, Pierre-André Wacrenier, and Raymond Namyst. "ForestGOMP: an efficient OpenMP environment for NUMA architectures". In: *International Journal of Parallel Programming* 38 (Oct. 2010). DOI: 10.1007/s10766-010-0136-3.

[24] Iulian Brumar, Marc Casas, Miquel Moretó, Mateo Valero, and Gurindar S. Sohi. "ATM: Approximate Task Memoization in the Runtime System". In: *Proceedings of the IEEE 31st International Parallel and Distributed Processing Symposium*. IPDPS. 2017, pp. 1140–1150. DOI: 10.1109/IPDPS.2017.49.

[25] 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.

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

[27] Jason F. Cantin, Mikko H. Lipasti, and James E. Smith. "Improving Multiprocessor Performance with Coarse-Grain Coherence Tracking". In: *International Symposium on Computer Architecture (ISCA)*. 2005, pp. 246–257. DOI: 10.1109/ISCA.2005.31.

[28] 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.

[29] Marc Casas, Miquel Moretó, Lluc Alvarez, Emilio Castillo, Dimitrios Chasapis, Timothy Hayes, Luc Jaulmes, Oscar Palomar, Osman S. Unsal, Adrián Cristal, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. "Runtime-Aware Architectures". In: *Euro-Par 2015: Parallel Processing - 21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015, Proceedings*. 2015, pp. 16–27.

[30] Marc Casas, Miquel Moreto, Lluc Alvarez, Emilio Castillo, Dimitrios Chasapis, Timothy Hayes, Luc Jaulmes, Oscar Palomar, Osman Unsal, Adrian Cristal, et al. "Runtime-Aware Architectures". In: *International Conference on Parallel and Distributed Computing (Euro-Par)*. 2015, pp. 16–27.

[31] Emilio Castillo, Lluc Alvarez, Miquel Moretó, Marc Casas, Enrique Vallejo, Jose Luis Bosque, Ramon Beivide, and Mateo Valero. "Architectural Support for Task Dependence Management with Flexible Software Scheduling". In: *IEEE 24th International Symposium on High Performance Computer Architecture*. HPCA. 2018, pp. 283–295. DOI: 10.1109/HPCA.2018.00033.

[32] Emilio Castillo, Miquel Moreto, Marc Casas, Lluc Alvarez, Enrique Vallejo, Kallia Chronaki, Rosa Badia, Jose Luis Bosque, Ramon Beivide, Eduard Ayguadé, Jesus Labarta, and Mateo Valero. "CATA: Criticality Aware Task Acceleration for Multicore Processors". In: *Proceedings of the IEEE 30th International Parallel and Distributed Processing Symposium*. IPDPS. 2016, pp. 413–422. DOI: 10.1109/IPDPS.2016.49.

[33] Dimitrios Chasapis, Marc Casas, Miquel Moretó, Martin Schulz, Eduard Ayguadé, Jesus Labarta, and Mateo Valero. "Runtime-Guided Mitigation of Manufacturing Variability in Power-Constrained Multi-Socket NUMA Nodes". In: *Proceedings of the 2016 International Conference on Supercomputing*. ICS '16. Istanbul, Turkey: Association for Computing Machinery, 2016. ISBN: 9781450343619. DOI: 10.1145/2925426.2926279.

[34] Dimitrios Chasapis, Miquel Moretó, Martin Schulz, Barry Rountree, Mateo Valero, and Marc Casas. "Power Efficient Job Scheduling by Predicting the Impact of Processor Manufacturing Variability". In: *Proceedings of the ACM International Conference on Supercomputing*. ICS '19. Phoenix, Arizona: Association for Computing Machinery, 2019, pp. 296–307. ISBN: 9781450360791. DOI: 10.1145/3330345.3330372.

[35] M. Chaudhuri. "PageNUCA: Selected policies for page-grain locality management in large shared chip-multiprocessor caches". In: *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. 2009, pp. 227–238.

[36] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. "Distance associativity for high-performance energy-efficient non-uniform cache architectures". In: *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* 2003, pp. 55–66.

[37] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. "Optimizing replication, communication, and capacity allocation in CMPs". In: *32nd International Symposium on Computer Architecture (ISCA'05)*. 2005, pp. 357–368.

[38] S. Cho and L. Jin. "Managing Distributed, Shared L2 Caches through OS-Level Page Allocation". In: *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. 2006, pp. 455–468.

[39] Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Smolinski, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Nicholas P. Carter, and Ching-Tsun Chou. "DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism". In: *International*

*Conference on Parallel Architectures and Compilation (PACT)*. 2011, pp. 155–166. DOI: 10.1109/PACT.2011.21.

[40] Jong Hyuk Choi and Kyu Ho Park. "Segment directory enhancing the limited directory cache coherence schemes". In: *International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (IPPS/SPDP)*. 1999, pp. 258–267. DOI: 10.1109/IPPS.1999.760473.

[41] Kallia Chronaki, Alejandro Rico, Rosa M. Badia, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. "Criticality-Aware Dynamic Task Scheduling for Heterogeneous Architectures". In: *Proceedings of the 29th International Conference on Supercomputing*. ICS '15. ACM, 2015, pp. 329–338. ISBN: 978-1-4503-3559-1. DOI: 10.1145/2751205.2751235. URL: http://doi.acm.org/10.1145/2751205.2751235.

[42] Kallia Chronaki, Alejandro Rico, Marc Casas, Miquel Moretó, Rosa M. Badia, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. "Task Scheduling Techniques for Asymmetric Multi-Core Systems". In: *IEEE Transactions on Parallel and Distributed Systems* 28.7 (July 2017), pp. 2074–2087. ISSN: 1045-9219. DOI: 10.1109/TPDS.2016.2633347.

[43] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. "Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor". In: *IEEE Micro* 30.2 (2010), pp. 16–29.

[44] Julita Corbalan, Xavier Martorell, and Jesús Labarta. "Page Migration with Dynamic Space-Sharing Scheduling Policies: The Case of the SGI O2000". In: *International Journal of Parallel Programming* 32 (Aug. 2004), pp. 263–288. DOI: 10.1023/B:IJPP.0000035815.13969.ec.

[45] National Research Council. *The Future of Computing Performance: Game Over or Next Level?* Ed. by Samuel H. Fuller and Lynette I. Millett. Washington, DC: The National Academies Press, 2011. ISBN: 978-0-309-15951-7. DOI: 10.17226/12980. URL: https://www.nap.edu/catalog/12980/the-future-of-computing-performance-game-over-or-next-level.

[46] E. H. M. Cruz, M. Diener, and P. O. A. Navaux. "Using the Translation Lookaside Buffer to Map Threads in Parallel Applications Based on Shared Memory". In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. 2012, pp. 532–543.

[47] Blas A. Cuesta, Alberto Ros, Marıéa E. Gómez, Antonio Robles, and José F. Duato. "Increasing the Effectiveness of Directory Caches by Deactivating Coherence for Private Memory Blocks". In: *International Symposium on Computer Architecture (ISCA)*. 2011, pp. 93–104. DOI: 10.1145/2000064.2000076.

[48] Blas Cuesta, Alberto Ros, Maria E. Gomez, Antonio Robles, and Jose Duato. "Increasing the Effectiveness of Directory Caches by Avoiding the Tracking of Noncoherent Memory Blocks". In: *IEEE Transactions on Computers* 62.3 (Mar. 2013), pp. 482–495. ISSN: 0018-9340. DOI: 10.1109/TC.2011.241.

[49] S. Das, T. M. Aamodt, and W. J. Dally. "SLIP: Reducing wire energy in the memory hierarchy". In: *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 2015, pp. 349–361.

[50] Robert H. Dennard, Fritz H. Gaensslen, Hwa-nien Yu, V. Leo Rideout, Ernest Bassous, Andre, and R. Leblanc. "Design of Ion-implanted MOSFETs with Very Small Physical Dimensions". In: *IEEE Journal of Solid-State Circuits* 9.5 (Oct. 1974), pp. 256–268.

[51] M. Diener, E. H. M. Cruz, and P. O. A. Navaux. "Communication-Based Mapping Using Shared Pages". In: *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. 2013, pp. 700–711.

[52] Vladimir Dimic, Miquel Moretó, Marc Casas, and Mateo Valero. "Runtime-Assisted Shared Cache Insertion Policies Based on Re-reference Intervals". In: Aug. 2017, pp. 247–259. ISBN: 978-3-319-64202-4. DOI: 10.1007/978-3-319-64203-1_18.

[53] A Drebes, K Heydemann, N Drach, A Pop, and A Cohen. "Topology-Aware and Dependence-Aware Scheduling and Memory Allocation for Task-Parallel Languages". In: *ACM Trans. Archit. Code Optim.* 11.3 (2014), 30:1–30:25.

[54] A Duran, E Ayguadé, R Badia, J Labarta, L Martinell, X Martorell, and J Planas. "OmpSs: A PROPOSAL FOR PROGRAMMING HETEROGENEOUS MULTI-CORE ARCHITECTURES". In: *Parallel Process. Lett.* 21 (2011), pp. 173–193.

[55] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. "OmpSs: A Proposal For Programming Heterogeneous Multi-Core Architectures". In: *Parallel Processing Letters* 21.02 (2011), pp. 173–193.

[56] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. “OmpSs: A Proposal for Programming Heterogeneous Multi-core Architectures”. In: *Parallel Processing Letters* 21.2 (2011), pp. 173–193.

[57] 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.

[58] Albert Esteve, Alberto Ros, María Engracia Gómez, Antonio Robles, and José Duato. “TLB-Based Temporality-Aware Classification in CMPs with Multilevel TLBs”. In: *IEEE Transactions on Parallel and Distributed Systems* 28.8 (Jan. 2017), pp. 2401–2413. DOI: 10.1109/TPDS.2017.2658576.

[59] Albert Esteve, Alberto Ros, Marıéa Engracia Gómez, Antonio Robles, and José Duato. “Efficient TLB-Based Detection of Private Pages in Chip Multiprocessors”. In: *IEEE Transactions on Parallel and Distributed Systems* 27.3 (Mar. 2016), pp. 748–761. DOI: 10.1109/TPDS.2015.2412139.

[60] Albert Esteve, Alberto Ros, Antonio Robles, Marıéa Engracia Gómez, and José Duato. “TokenTLB: A Token-Based Page Classification Approach”. In: *International Conference on Supercomputing (ICS)*. 2016, 26:1–26:13. DOI: 10.1145/2925426.2926280.

[61] Yoav Etsion, Felipe Cabarcas, Alejandro Rico, Alex Ramirez, Rosa M. Badia, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. “Task Superscalar: An Out-of-Order Task Pipeline”. In: *Proceedings of the 43rd Annual ACM/IEEE International Symposium on Microarchitecture*. MICRO 43. 2010, pp. 89–100. DOI: 10.1109/MICRO.2010.13.

[62] Michael Ferdman, Pejman Lotfi-Kamran, Ken Balet, and Babak Falsafi. “Cuckoo Directory: A Scalable Directory for Many-core Systems”. In: *International Symposium on High Performance Computer Architecture (HPCA)*. 2011, pp. 169–180.

[63] Steve Frank and Armond Inselberg. “Synapse Tightly Coupled Multiprocessors: A New Approach to Solve Old Problems”. In: *Proceedings of the July 9-12, 1984, National Computer Conference and Exposition*. AFIPS ’84. Las Vegas, Nevada: Association for Computing Machinery, 1984, pp. 41–50. ISBN: 0882830430. DOI: 10.1145/1499310.1499317. URL: https://doi.org/10.1145/1499310.1499317.

[64] Victor Garcia, Alejandro Rico, Carlos Villavieja, Paul Carpenter, Nacho Navarro, and Alex Ramirez. "Adaptive Runtime-Assisted Block Prefetching on Chip-Multiprocessors". In: *International Journal of Parallel Programming* (2016), pp. 1–21. ISSN: 1573-7640. DOI: 10.1007/s10766-016-0431-8.

[65] Anoop Gupta, Wolf-Dietrich Weber, and Todd Mowry. "Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes". In: *International Conference on Parallel Processing (ICPP)*. 1990, pp. 312–321.

[66] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. "Reactive NUCA: Near-optimal Block Placement and Replication in Distributed Caches". In: *International Symposium on Computer Architecture (ISCA)*. 2009, pp. 184–195. DOI: 10.1145/1555754.1555779.

[67] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN: 012383872X.

[68] Mark Donald Hill. "Aspects of Cache Memory and Instruction Buffer Performance". PhD thesis. EECS Department, University of California, Berkeley, Nov. 1987. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/1987/5701.html.

[69] Jaehyuk Huh, Changkyu Kim, Hazim Shafi, Lixin Zhang, Doug Burger, and Stephen W. Keckler. "A NUCA Substrate for Flexible CMP Cache Sharing". In: *Proceedings of the 19th Annual International Conference on Supercomputing*. ICS '05. Cambridge, Massachusetts: Association for Computing Machinery, 2005, pp. 31–40. ISBN: 1595931678. DOI: 10.1145/1088149.1088154. URL: https://doi.org/10.1145/1088149.1088154.

[70] *IBM System/360 System Summary*. 1974. URL: http://bitsavers.org/pdf/ibm/360/systemSummary/GA22-6810-12_360sysSumJan74.pdf (visited on 06/03/2020).

[71] Luc Jaulmes, Marc Casas, Miquel Moretó, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. "Exploiting Asynchrony from Exact Forward Recovery for DUE in Iterative Solvers". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '15. Austin, Texas: Association for Computing Machinery, 2015. ISBN: 9781450337236. DOI: 10.1145/2807591.2807599. URL: https://doi.org/10.1145/2807591.2807599.

[72] Luc Jaulmes, Miquel Moreto, Eduard Ayguade, Jesús Labarta, Mateo Valero, and Marc Casas. "Asynchronous and Exact Forward Recovery for Detected Errors in Iterative Solvers". In: *IEEE Transactions on Parallel and Distributed Systems* PP (Mar. 2018), pp. 1–1. DOI: 10.1109/TPDS.2018.2817524.

[73] Luc Jaulmes, Miquel Moreto, Mateo Valero, and Marc Casas. "A Vulnerability Factor for ECC-protected Memory". In: July 2019, pp. 176–181. DOI: 10.1109/IOLTS.2019.8854397.

[74] A. Jimborean, P. Ekemark, J. Waern, S. Kaxiras, and A. Ros. "Automatic Detection of Large Extended Data-Race-Free Regions with Conflict Isolation". In: *IEEE Transactions on Parallel and Distributed Systems* 29.3 (2018), pp. 527–541.

[75] A. Jimborean, J. Waern, P. Ekemark, S. Kaxiras, and A. Ros. "Automatic detection of extended data-race-free regions". In: *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2017, pp. 14–26.

[76] L. Jin and S. Cho. "SOS: A Software-Oriented Distributed Shared Cache Management Approach for Chip Multiprocessors". In: *2009 18th International Conference on Parallel Architectures and Compilation Techniques*. 2009, pp. 361–371.

[77] Rama Karedla. *Intel Xeon E5-2600 v3 (Haswell) Architecture & Features*. 2014. URL: http://repnop.org/pd/slides/PD_Haswell_Architecture.pdf.

[78] John H. Kelm, Daniel R. Johnson, Matthew R. Johnson, Neal C. Crago, William Tuohy, Aqeel Mahesri, Steven S. Lumetta, Matthew I. Frank, and Sanjay 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.

[79] Changkyu Kim, Doug Burger, and Stephen W. Keckler. "An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated on-Chip Caches". In: *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS X. San Jose, California: Association for Computing Machinery, 2002, pp. 211–222. ISBN: 1581135742. DOI: 10.1145/605397.605420. URL: https://doi.org/10.1145/605397.605420.

[80] Daehoon Kim, Jeongseob Ahn, Jaehong Kim, and Jaehyuk Huh. "Subspace Snooping: Filtering Snoops with Operating System Support". In: *International Conference on Parallel Architectures and Compilation (PACT)*. 2010, pp. 111–122. DOI: 10.1145/1854273.1854292.

[81] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen. “Carbon: Architectural Support for Fine-grained Parallelism on Chip Multiprocessors”. In: *Proceedings of the 34th Annual International Symposium on Computer Architecture*. ISCA ’07. ACM, 2007, pp. 162–173. ISBN: 978-1-59593-706-3. DOI: 10.1145/1250662.1250683. URL: http://doi.acm.org/10.1145/1250662.1250683.

[82] Richard P. LaRowe and Carla Schlatter Ellis. “Page placement policies for NUMA multiprocessors”. In: *Journal of Parallel and Distributed Computing* 11.2 (1991), pp. 112–129. ISSN: 0743-7315. DOI: https://doi.org/10.1016/0743-7315(91)90117-R. URL: http://www.sciencedirect.com/science/article/pii/074373159190117R.

[83] James Laudon and Daniel Lenoski. “The SGI Origin: A ccNUMA Highly Scalable Server”. In: *International Symposium on Computer Architecture (ISCA)*. 1997, pp. 241–251. DOI: 10.1145/264107.264206.

[84] Alvin R. Lebeck and David A. Wood. “Dynamic Self-invalidation: Reducing Coherence Overhead in Shared-memory Multiprocessors”. In: *International Symposium on Computer Architecture (ISCA)*. 1995, pp. 48–59. DOI: 10.1145/223982.223995.

[85] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. “The Stanford Dash multiprocessor”. In: *Computer* 25.3 (1992), pp. 63–79.

[86] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. “McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures”. In: *International Symposium on Microarchitecture (MICRO)*. 2009, pp. 469–480. DOI: 10.1145/1669112.1669172.

[87] Yong Li, Ahmed Abousamra, Rami Melhem, and Alex K. Jones. “Compiler-assisted Data Distribution for Chip Multiprocessors”. In: *International Conference on Parallel Architectures and Compilation (PACT)*. 2010, pp. 501–512. DOI: 10.1145/1854273.1854335.

[88] Yong Li, Rami Melhem, and Alex K. Jones. “Practically Private: Enabling High Performance CMPs Through Compiler-assisted Data Classification”. In: *International Conference on Parallel Architectures and Compilation (PACT)*. 2012, pp. 231–240. DOI: 10.1145/2370816.2370852.

[89] Haikun Liu, Yujie Chen, Xiaofei Liao, Hai Jin, Bingsheng He, Long Zheng, and Rentong Guo. "Hardware/Software Cooperative Caching for Hybrid DRAM/NVM Memory Architectures". In: *Proceedings of the International Conference on Supercomputing*. ICS '17. 2017, 26:1–26:10. DOI: 10.1145/3079079.3079089. URL: http://doi.acm.org/10.1145/3079079.3079089.

[90] M. Manivannan, V. Papaefstathiou, M. Pericas, and P. Stenstrom. "RADAR: Runtime-assisted dead region management for last-level caches". In: *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2016, pp. 644–656.

[91] Madhavan Manivannan, Vassilis Papaefstathiou, Miquel Pericàs, and Per Stenström. "RADAR: Runtime-assisted dead region management for last-level caches". In: *International Symposium on High Performance Computer Architecture (HPCA)*. 2016, pp. 644–656. DOI: 10.1109/HPCA.2016.7446101.

[92] Milo M. K. Martin, Mark D. Hill, and Daniel J. Sorin. "Why On-chip Cache Coherence is Here to Stay". In: *Commun. ACM* 55.7 (2012), pp. 78–89.

[93] J McCalpin. "Memory Bandwidth and Machine Balance in Current High Performance Computers". In: *IEEE TCCA Newsletter* (1995), pp. 19–25.

[94] J. D. McCalpin. "HPL and DGEMM Performance Variability on the Xeon Platinum 8160 Processor". In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 2018, pp. 225–237.

[95] Javier Merino, Valentıén Puente, Pablo Prieto, and José Ángel Gregorio. "SP-NUCA: A Cost Effective Dynamic Non-Uniform Cache Architecture". In: *SIGARCH Comput. Archit. News* 36.2 (May 2008), pp. 64–71. ISSN: 0163-5964. DOI: 10.1145/1399972.1399973. URL: https://doi.org/10.1145/1399972.1399973.

[96] Gordon E. Moore. "Cramming More Components onto Integrated Circuits". In: *Electronics* 38.8 (Apr. 1965), pp. 114–117.

[97] A. Moshovos, G. Memik, B. Falsafi, and A. Choudhary. "JETTY: filtering snoops for reduced energy consumption in SMP servers". In: *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*. 2001, pp. 85–96.

[98] Andreas Moshovos. "RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence". In: *International Symposium on Computer Architecture (ISCA)*. 2005, pp. 234–245. DOI: 10.1109/ISCA.2005.42.

[99] T. Mudge. “Power: a first-class architectural design constraint”. In: *Computer* 34.4 (2001), pp. 52–58.

[100] Fan Ni, Song Jiang, Hong Jiang, Jian Huang, and Xingbo Wu. “SDC: A Software Defined Cache for Efficient Data Indexing”. In: *Proceedings of the ACM International Conference on Supercomputing*. ICS ’19. Phoenix, Arizona: Association for Computing Machinery, 2019, pp. 82–93. ISBN: 9781450360791. DOI: 10.1145/3330345.3330353. URL: https://doi.org/10.1145/3330345.3330353.

[101] 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”. In: *Supercomputing Frontiers and Innovations* 2.1 (2015).

[102] “OpenMP: Application Program Interface, Version 4.0”. In: (2013). URL: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

[103] *OpenMP Application Program Interface. Version 4.0. July 2013*.

[104] V Papaefstathiou, M Katevenis, D Nikolopoulos, and D Pnevmatikatos. “Prefetching and Cache Management Using Task Lifetimes”. In: *Proc. 27th Int. Conf. Supercomputers (ICS’13)*. 2013, pp. 325–334.

[105] Vassilis Papaefstathiou, Manolis G.H. Katevenis, Dimitrios S. Nikolopoulos, and Dionisios Pnevmatikatos. “Prefetching and Cache Management Using Task Lifetimes”. In: *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*. ICS ’13. Eugene, Oregon, USA, 2013, pp. 325–334. ISBN: 978-1-4503-2130-3. DOI: 10.1145/2464996.2465443. URL: http://doi.acm.org/10.1145/2464996.2465443.

[106] 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.

[107] Fatih Porikli. “Integral Histogram: A Fast Way to Extract Histograms in Cartesian Spaces”. In: *Computer Society Conference on Computer Vision and Pattern Recognition Workshops (CVPR)*. 2005, pp. 829–836. ISBN: 0-7695-2372-2. DOI: 10.1109/CVPR.2005.188.

[108] Michael Powell, Se-Hyun Yang, Babak Falsafi, Kaushik Roy, and T. N. Vijaykumar. "Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-submicron Cache Memories". In: *International Symposium on Low Power Electronics and Design (ISLPED)*. 2000, pp. 90–95.

[109] Parthasarathy Ranganathan, Sarita V. Adve, and Norman P. Jouppi. "Reconfigurable caches and their application to media processing". In: *International Symposium on Computer Architecture (ISCA)*. 2000, pp. 214–224.

[110] James Reinders. *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. O'Reilly Media, 2007, pp. I–XXV, 1–303.

[111] A. Ros, M. Davari, and S. Kaxiras. "Hierarchical private/shared classification: The key to simple and efficient coherence for clustered cache hierarchies". In: *International Symposium on High Performance Computer Architecture (HPCA)*. 2015, pp. 186–197. DOI: 10.1109/HPCA.2015.7056032.

[112] A. Ros and A. Jimborean. "A Dual-Consistency Cache Coherence Protocol". In: *2015 IEEE International Parallel and Distributed Processing Symposium*. 2015, pp. 1119–1128.

[113] Alberto Ros and Stefanos Kaxiras. "Complexity-effective Multicore Coherence". In: *International Conference on Parallel Architectures and Compilation (PACT)*. 2012, pp. 241–252. DOI: 10.1145/2370816.2370853.

[114] Luc Saillard. *TinyJPEG Decoder*. 2009. URL: https://www.saillard.org/programs/tinyjpegdecoder/ (visited on 05/29/2020).

[115] Isaac Sánchez Barrera, Marc Casas, Miquel Moretó, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. "Graph Partitioning Applied to DAG Scheduling to Reduce NUMA Effects". In: *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '18. ACM, 2018, pp. 419–420. ISBN: 978-1-4503-4982-6. DOI: 10.1145/3178487.3178535. URL: http://doi.acm.org/10.1145/3178487.3178535.

[116] Isaac Sánchez Barrera, Miquel Moretó, Eduard Ayguadé, Jesús Labarta, Mateo Valero, and Marc Casas. "Reducing Data Movement on Large Shared Memory Systems by Exploiting Computation Dependencies". In: *Proceedings of the 2018 International Conference on Supercomputing*. ICS '18. ACM, 2018, pp. 207–217. ISBN: 978-1-4503-

5783-8. DOI: 10.1145/3205289.3205310. URL: http://doi.acm.org/10.1145/3205289.3205310.

[117] Daniel Sanchez and Christos Kozyrakis. “SCD: A Scalable Coherence Directory with Flexible Sharer Set Encoding”. In: *International Symposium on High Performance Computer Architecture (HPCA)*. 2012, pp. 1–12. DOI: 10.1109/HPCA.2012.6168950.

[118] A. Sembrant, E. Hagersten, and D. Black-Schaffer. “Data placement across the cache hierarchy: Minimizing data movement with reuse-aware placement”. In: *2016 IEEE 34th International Conference on Computer Design (ICCD)*. 2016, pp. 117–124.

[119] Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. 1st. Morgan & Claypool Publishers, 2011. ISBN: 1608455645, 9781608455645.

[120] Hyojin Sung, Rakesh Komuravelli, and Sarita V. Adve. “DeNovoND: efficient hardware support for disciplined non-determinism”. In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2013, pp. 138–148.

[121] Xubin Tan, Jaume Bosch, Daniel Jiménez-González, Carlos Álvarez-Martínez, Eduard Ayguadé, and Mateo Valero. “Performance analysis of a hardware accelerator of dependence management for task-based dataflow programming models”. In: *International Symposium on Performance Analysis of Systems and Software*. ISPASS. 2016, pp. 225–234. DOI: 10.1109/ISPASS.2016.7482097.

[122] Xubin Tan, Jaume Bosch, Miquel Vidal, Carlos Álvarez, Daniel Jiménez-González, Eduard Ayguadé, and Mateo Valero. “General Purpose Task-Dependence Management Hardware for Task-Based Dataflow Programming Models”. In: *Proceedings of the IEEE 31st International Parallel and Distributed Processing Symposium*. IPDPS. 2017, pp. 244–253. DOI: 10.1109/IPDPS.2017.48.

[123] Mustafa M. Tikir and Jeffrey K. Hollingsworth. “Using Hardware Counters to Automatically Improve Memory Performance”. In: *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*. SC ’04. USA: IEEE Computer Society, 2004, p. 46. ISBN: 0769521533. DOI: 10.1109/SC.2004.64. URL: https://doi.org/10.1109/SC.2004.64.

[124] Po-An Tsai, Nathan Beckmann, and Daniel Sanchez. “Jenga: Software-Defined Cache Hierarchies”. In: *International Symposium on Computer Architecture (ISCA)*. 2017, pp. 652–665. DOI: 10.1145/3079856.3080214.

[125] Mateo Valero, Miquel Moreto, Marc Casas, Eduard Ayguade, and Jesus Labarta. "Runtime-Aware Architectures: A First Approach". In: *International Journal on Supercomputing Frontiers and Innovations* 1.1 (June 2014), pp. 29–44. ISSN: 2313-8734.

[126] Keshavan Varadarajan, S. K. Nandy, Vishal Sharda, Bharadwaj Amrutur, Ravi R. Iyer, Srihari Makineni, and Donald Newell. "Molecular Caches: A caching structure for dynamic creation of application-specific Heterogeneous cache regions". In: *International Symposium on Microarchitecture (MICRO)*. 2006, pp. 433–442.

[127] 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.

[128] V Viswanathan, K Kumar, and T Willhalm. *Intel Memory Latency Checker v2*. 2013. URL: https://software.intel.com/en-us/articles/intelr-memory-latency-checker (visited on 10/01/2015).

[129] Wm. A. Wulf and Sally A. McKee. "Hitting the memory wall: implications of the obvious". In: *ACM SIGARCH Computer Architecture News* 23.1 (Mar. 1995), pp. 20–24. ISSN: 0163-5964. DOI: 10.1145/216585.216588.

[130] Sam Xi, Hans Jacobson, Pradip Bose, Gu-Yeon Wei, and David Brooks. "Quantifying sources of error in McPAT and potential impacts on architectural studies". In: *International Symposium on High Performance Computer Architecture (HPCA)*. 2015, pp. 577–589.

[131] J. Yin, Z. Lin, O. Kayiran, M. Poremba, M. Shoaib Bin Altaf, N. Enright Jerger, and G. H. Loh. "Modular Routing Design for Chiplet-Based Systems". In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 2018, pp. 726–738.

[132] Jason Zebchuk, Babak Falsafi, and Andreas Moshovos. "Multi-grain Coherence Directories". In: *International Symposium on Microarchitecture (MICRO)*. 2013, pp. 359–370. DOI: 10.1145/2540708.2540739.

[133] Jason Zebchuk, Vijayalakshmi Srinivasan, Moinuddin K. Qureshi, and Andreas Moshovos. "A Tagless Coherence Directory". In: *International Symposium on Microarchitecture (MICRO)*. 2009, pp. 423–434. DOI: 10.1145/1669112.1669166.

[134] M. Zhang and K. Asanovic. “Victim replication: maximizing capacity while hiding wire delay in tiled chip multiprocessors”. In: *32nd International Symposium on Computer Architecture (ISCA’05)*. 2005, pp. 336–345.

[135] Hongzhou Zhao, Arrvindh Shriraman, and Sandhya Dwarkadas. “SPACE: Sharing Pattern-based Directory Coherence for Multicore Scalability”. In: *International Conference on Parallel Architectures and Compilation (PACT)*. 2010, pp. 135–146. DOI: 10.1145/1854273.1854294.