Runtime-Assisted Cache Coherence Deactivation in Task Parallel Programs

Paul Caheny*†, Lluc Alvarez*, Mateo Valero*†, Miquel Moretó*†, Marc Casas*
*Barcelona Supercomputing Center (BSC)
†Universitat Politècnica de Catalunya (UPC)
{firstname}.{lastname}@bsc.es

***Abstract*—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 paper 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.

***Index Terms*—Cache memory, Memory architecture, Parallel programming, Runtime environment**

I. Introduction

Since the end of Dennard scaling [1], multicore architectures have proliferated in the High-Performance Computing (HPC) domain. 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 [2].

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 [3]–[7], 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 multithreading, 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 [8] 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 (RTS) manages the parallel execution, discovering dependences between tasks, dynamically scheduling tasks to cores, and taking care of synchronisation between tasks.

This paper presents *Runtime-assisted Cache Coherence Deactivation* (RaCCD), a hardware/software co-designed approach that leverages the information present in the RTS of task-based data-flow programming models to drive a more efficient hardware cache coherence design. RaCCD relies on the implicit guarantees of the memory model of task-based data-flow 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 RTS 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 RTS 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 blocks during the execution of the task. When the task finishes, the RTS triggers a lightweight recovery mechanism that invalidates the non-coherent blocks 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 paper makes the following contributions beyond the state-of-the-art:

SC18, November 11-16, 2018, Dallas, Texas, USA
978-1-5386-8384-2/18/$31.00 ©2018 IEEE

- RaCCD, 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.
- 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 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.

II. Background and Motivation

A. Cache Coherence Deactivation

In recent years, significant work has been done in reducing the complexity and cost of hardware-based cache coherence techniques. 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 that is accessed from different cores at different points in time, but never concurrently. In the rest of this paper, 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*, which is a technique that avoids the tracking of non-coherent blocks in the directory. 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 to save area and energy consumption. To deactivate coherence, cache misses for non-coherent blocks 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 blocks of the private caches [9], so they never need to be tracked in the directory.

B. Identification of Non-Coherent Data

State-of-the-art techniques to identify non-coherent data use TLB and OS page table support [3]–[7]. 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 blocks 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 blocks.

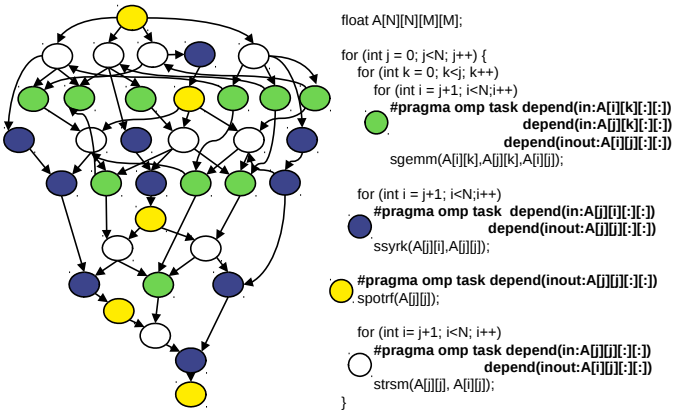


Fig. 1: Cholesky task-based code (right) and task dependence graph (left).

Extensive and costly modifications in the system are required to identify temporarily private data in TLB-based approaches [10]–[12]. 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.

C. Task-Based Programming Models

Task-based data-flow programming models such as OpenMP 4.0 [8] conceive the execution of a parallel program as a set of *tasks* with data 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). The RTS 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. Figure 1 shows the task-based implementation of a Cholesky factorisation algorithm and its corresponding TDG. The code uses OpenMP 4.0 clauses to specify tasks and their data dependences (*#pragma omp task depend(in/out/inout)*).

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 dependences. Only when all the dependences 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 decoupling of the specification of the program from its dynamic execution eases programmability and enables many optimisations at the RTS level in a generic and application-agnostic way [13]–[16].

D. Opportunities 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 RTS can precisely identify non-coherent data without the hardware complexity nor the accuracy problems of other approaches. To exploit this, the RTS 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 2 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 2 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, $2.9\times$ more than identified by PT (26.9%). RaCCD significantly outperforms PT in CG, Gauss, 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.

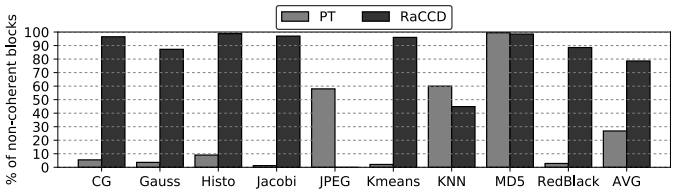


Fig. 2: Percentage of non-coherent cache blocks

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 RTS 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.

III. RaCCD: Runtime-assisted Cache Coherence Deactivation

This paper presents RaCCD (Runtime-assisted Cache Coherence Deactivation), a hardware/software co-designed approach that leverages information present in the runtime system (RTS) of task-based data-flow 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 RTS 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 data-flow 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 RTS 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 RTS manages the invalidation of the input and output data from the private caches, ensuring no incoherent data is present in the cache hierarchy.

A. Runtime System - Architecture Interface

RaCCD offers an interface between the RTS 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 RTS 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_address, size)*: Before executing a task, the RTS 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.

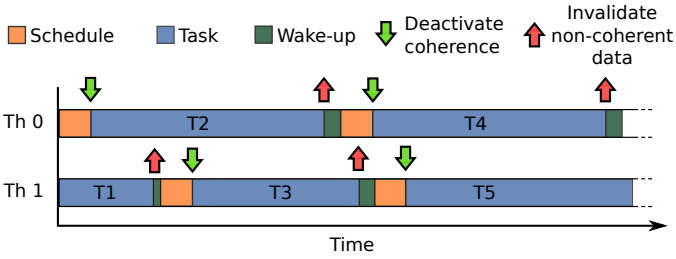


Fig. 3: Runtime system support with additional operations to deactivate coherence and invalidate non-coherent data.

- *raccd_invalidate()*: When a task finishes, the RTS 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.

B. Runtime System Extensions

RaCCD extends the operational model of task-based programming models to assist coherence deactivation at the microarchitecture level. Figure 3 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.

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 dependences 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 1.

Based on the information communicated from the RTS 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 RTS 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

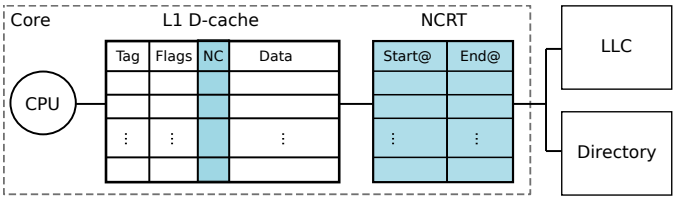


Fig. 4: Architectural support for RaCCD: Non-Coherent Region Table (NCRT), and a Non-Coherent (NC) bit per cache block in the private cache.

the invalidation of any non-coherent data in the private caches of the core that executed the task. Section III-C 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.

C. Architectural Support

1) Hardware Extensions: RaCCD introduces simple and efficient hardware support to manage non-coherent memory regions. The hardware additions, shown shaded in Figure 4, consist of a new per-core structure called the *Non-Coherent Region Table* (NCRT), and a *Non-Coherent* (NC) bit per cache block 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 RTS 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 blocks. The NC bit is also introduced in the request and response messages between the private caches and the LLC, and between the LLC and the memory controllers.

2) Registering Non-Coherent Memory Regions: Non-coherent memory regions are registered in the NCRT when the RTS executes the *raccd_register* instruction for task inputs and outputs. The start and end addresses passed by the RTS are virtual addresses, so they must be translated to physical addresses before registering them in the NCRT. Figure 5 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

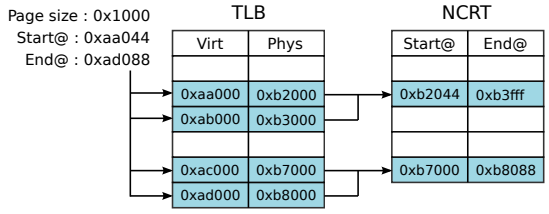


Fig. 5: Address translation for non-coherent memory regions in the NCRT.

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 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.

3) Non-Coherent Memory Accesses: Non-coherent memory requests correspond to memory references within the dependences 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 block 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 blocks are silent. Writes to non-coherent blocks use a non-coherent variant of the respective coherent 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 blocks are silent. Write-backs of dirty non-coherent blocks also use a non-coherent variant of the respective coherent transaction.

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

4) Coherence Recovery: When a task finishes, the RTS 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 blocks of its private cache and flushes all cache blocks that have the NC bit set. Clean non-coherent blocks are silently evicted, while dirty non-coherent blocks 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 RTS proceeds into the wake-up phase to notify any dependent tasks of the just completed task execution.

D. 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 [17] and in set-partitioned caches [18], [19]. To power off parts of the directory we use a Gated-Vdd technique [20] that drastically reduces the leakage power. When resizing the directory, we only change its number of sets while keeping the associativity constant.

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 θ_{inc} and θ_{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 [18], [19]. To support multiple directory sizes, the tag has to work for the smallest possible directory size, leaving some bits unused as the directory size increases.

E. Additional Considerations

Task-based data-flow programming models guarantee no data races will occur for data that is only accessed from within tasks which specify the data as a dependence. 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 block transitions between coherent and non-coherent.

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 blocks from the private cache of the core. To prevent performance penalties in SMT cores the non-coherent bit per block added to the private caches can be extended to store the thread ID of the block. 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.

IV. Experimental Framework

A. Full-System Simulation Infrastructure

We employ gem5 [21] to simulate an x86 full-system environment that models the application, the RTS, the OS and the microarchitecture in detail. We simulate a 16-core processor using the detailed out-of-order CPU and ruby memory model. Table I summarises the main parameters of the simulated architecture. We extend gem5 with the proposed architectural support for RaCCD presented in Section III-C, which is also summarised in the RaCCD section of Table I.

TABLE I: Configuration of gem5 full-system simulations.

Cores	16 Out-of-order cores, 4 inst. wide, 1.0GHz
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, 2-way, 64B/line (2 cycles)
ITLB / DTLB	Each 256 entries fully-associative (1 cycle)
L2 cache	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
RaCCD	
NCRT	32 entries/core, 1 cycle access time
NC bit	1 bit per cache block in the private L1 data caches

Our simulations use Ubuntu 14.04 with kernel version 4.3. We use the Nanos++ 0.10a [22] RTS, which supports OpenMP 4.0 [8]. The runtime system is extended to communicate with RaCCD using the instructions described in Section III. The ISA is extended to support the new instructions and their execution is modelled in the microarchitecture. The latency of the *raccd_register* instruction depends on the iterative process for the virtual-to-physical address translation of the non-coherent regions. Similarly, the latency of the *raccd_invalidate* instruction depends on the number of non-coherent blocks that must be invalidated from the private caches. Both mechanisms are simulated in cycle-by-cycle detail in gem5.

Power consumption is evaluated with McPAT [23] using a process technology of 22 nm, voltage of 0.6V and the default clock gating scheme. We add the changes suggested by Xi *et al.* [24] to improve the accuracy of the models. The hardware structures of RaCCD are modeled using CACTI 6.0 [25].

B. Benchmarks

The evaluation uses a set of representative parallel benchmarks programmed with OpenMP 4.0 task annotations. CG is the conjugate gradient algorithm for solving large sparse systems of linear equations. Gauss solves the stationary heat diffusion problem using the iterative Gauss-Seidel method with a 4-element stencil. Histogram computes a cumulative histogram for all pixels of an image using a cross-weave scan. Jacobi solves the stationary heat diffusion problem using the iterative Jacobi method with a 5-element stencil. JPEG performs the decoding of JPEG images with fixed encoding of 2x2 MCU size and YUV color. Kmeans implements the Kmeans clustering algorithm which has important applications in statistics and data mining. KNN implements the K-nearest neighbours algorithm which is an important algorithm in pattern recognition and machine learning. MD5 cryptographically hashes random input buffers. RedBlack solves the stationary heat diffusion problem with a 4-element stencil. Table II shows the input set sizes used for each benchmark. The benchmarks are compiled with Mercurium 1.99 source-to-source compiler [26], using gcc 4.6.4 as a backend.

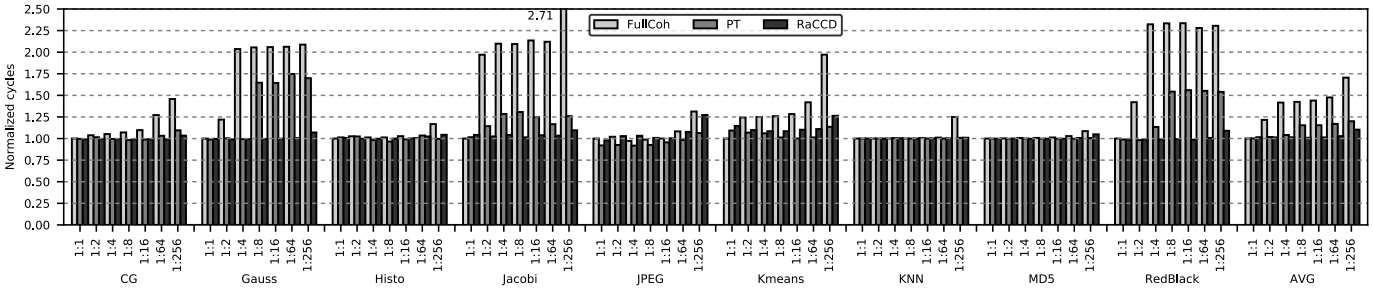


Fig. 6: Normalised cycles by directory size. Configuration 1:N has N times less directory entries than the baseline.

TABLE II: Application problem sizes

Application	Problem Set
CG	3D Matrix N^3 = 884736, 3 iters.
Gauss	2D Matrix N^2 = 2359296, 10 iters.
Histo	1000x1000 pixel image, 50 bins
Jacobi	2D Matrix N^2 = 2359296, 10 iters.
JPEG	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.

V. Evaluation

A. Static Directory Reduction

This section evaluates the behaviour of RaCCD versus the baseline fully coherent system (FullCoh), and the Page Table approach (PT) [5]. The FullCoh system tracks coherence for all memory accesses, while PT and RaCCD deactivate coherence for the non-coherent data they identify. 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.

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 6 and 7 present results for the 9 benchmarks as the capacity of the directory is reduced. Figure 7 shows the results in pairs of benchmarks, one shaded dark and one shaded light, and each benchmark is split in three lines for FullCoh, PT and RaCCD using three line styles: long dash, fine dash and solid line, respectively.

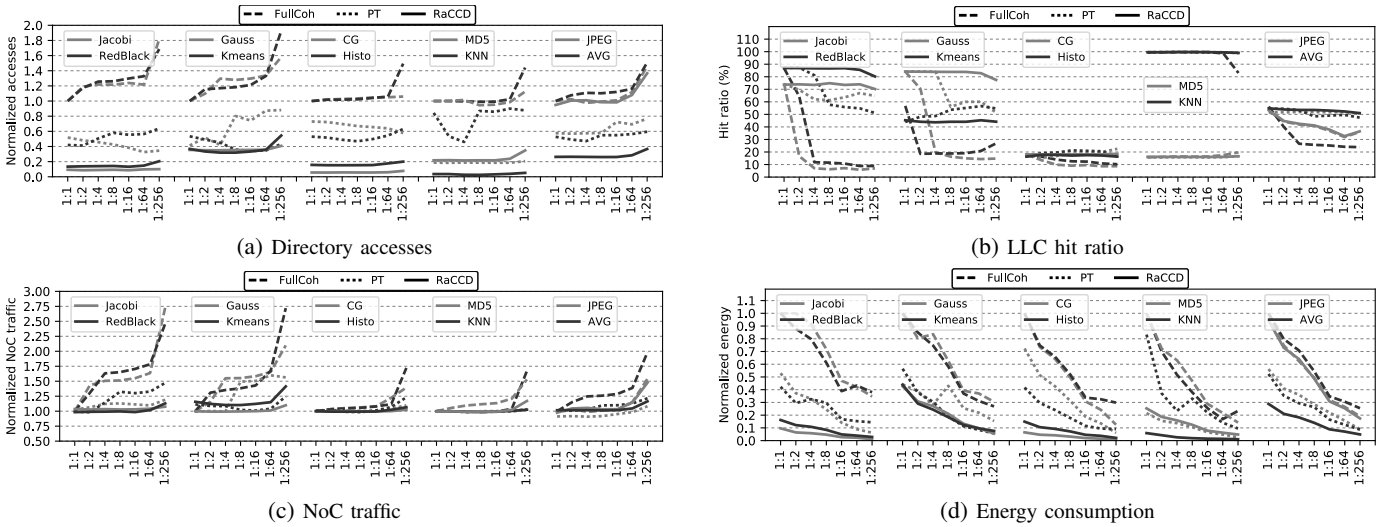
1) Performance: Figure 6 reports the number of execution cycles for the 7 directory sizes normalised to the 1:1 configuration of FullCoh. In RaCCD and PT, accesses to non-coherent data are not tracked in the directory. On average there is a negligible performance difference between the three systems ($<$ 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 blocks 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 6 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.

2) Directory Accesses: Figure 7a 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 7a 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 2. 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.



(a) Directory accesses

(b) LLC hit ratio

(c) NoC traffic

(d) Energy consumption

Fig. 7: Metrics by directory size. Configuration 1:N has N times less directory entries than the baseline.

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 7b. We see a rapid deterioration in the LLC hit 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

4) Network-on-Chip Traffic: Figure 7c 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 7c 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. 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) Energy Consumption: Figure 7d 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 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 III shows the storage requirements of the directory size configurations considered in this paper. Each directory entry is made up of 42 bits of tag and 3 bytes to store the state of the cache block 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.

TABLE III: 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

B. 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 8 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 blocks (and their associated directory entries) with non-coherent blocks (which do not have an associated directory entry). By deactivating coherence for non-coherent blocks, 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%.

Figures 9 and 10 show the performance and dynamic energy consumption in the directory under RaCCD+ADR versus the FullCoh, PT and RaCCD 1:1 configurations. All results are normalised to the FullCoh 1:1 configuration per benchmark. Figure 9 shows 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 blocks at the end of the task execution affects L1 cache hit rate. Figure 9 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.

Reducing the capacity demand in the directory allows ADR to use smaller directory sizes and thus reduce dynamic energy consumption. We can see in Figure 10 that RaCCD+ADR reduces energy consumption compared to RaCCD 1:1 across all benchmarks, without negatively impacting performance. RaCCD+ADR reduces dynamic energy consumption in the directory versus RaCCD 1:1 in a 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.

C. 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.

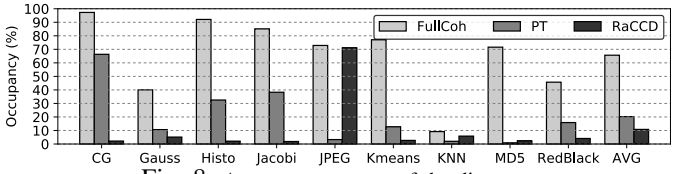


Fig. 8: Average occupancy of the directory.

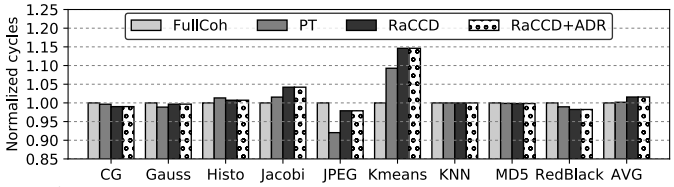


Fig. 9: Normalised performance with adaptive directory reduction.

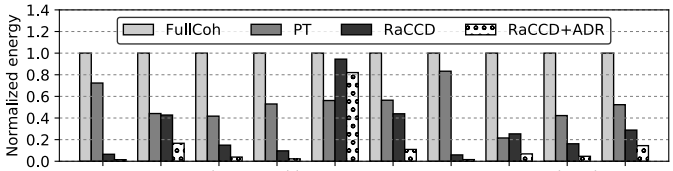


Fig. 10: Normalised energy consumption with adaptive directory reduction.

VI. Related Work

A. Microarchitectural Cache Coherence Optimisations

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 blocks are still tracked in the directory. Gupta et al. [27] introduce vectorised directory entries to track coherence at a coarse grain. Choi et al. [28] propose a similar solution that uses segments instead of vectors. SPACE [29] stores sharing patterns in a separate table and each directory entry stores a pointer to this table. SCD [30] uses a hierarchical approach where root entries contain pointers to potential sharers, reducing the space required for large sharing vectors. Tagless coherence directories [31] use bloom filters to track private blocks, which reduces the storage requirements but has a high cost in power. Cuckoo directories [32] propose a hashing technique that reduces the conflict misses in sparse directories. RegionScount [33] and Cantin et al. [34] track coherence at a coarse granularity in a separate hardware structure and filter broadcasts in snoop-based cache coherence protocols. Alisafaee [35] proposes modifying the directory to track shared, private and temporarily private data and compacting consecutive private blocks in one single region entry. Zebchuk et al. [36] 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 [37] 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.

B. Hardware/Software Cache Coherence Optimisations

As explained in Section II-A, TLB and OS support can be added to identify non-coherent data and to deactivate coherence for private pages [5], shared read-only pages [38], and temporarily private pages [10]–[12], although the latter requires extensive hardware support in the TLBs. VIPS [6] 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 [4], to optimise the data placement in NUCA caches [3], and to partition the cache hierarchy according to the requirements of the applications [39].

Some works optimise cache coherence by exploiting the semantics of Data-Race-Free (DRF) programming models such as Deterministic Parallel Java [40]. 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 [6] 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 blocks while, for private blocks, 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 [7] 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. Compared to these approaches, RaCCD deactivates coherence of private blocks without affecting the memory accesses to shared blocks nor the atomic operations.

DeNovo [41] 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 blocks. 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 [42]. In contrast to DeNovo, RaCCD deactivates coherence of private blocks to reduce the capacity pressure and the number of entries of the directory, not the size of the entries, and does it without affecting the synchronisation mechanisms.

Compile-time techniques [43], [44] 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 of this approach is that it relies on alias analyses to determine the variables that are going to be accessed in the different parts of the code, which is extremely challenging in non-trivial codes and, specially, in the presence of pointers.

C. Task-Based Programming Models

RaCCD targets OpenMP 4.0 [8]. However, it can be adapted to any runtime-managed task-based programming model that specifies data dependences between tasks, either using the real data addresses or some abstraction from which the runtime system can extract the addresses, such as OmpSs [22], Codelets [45], Charm++ [46], StarPU [47], Legion [48], Sequoia [49] and Habanero [50]. Similar properties are also present in streaming programming models such as Fastflow [51], StreamIt [52] or RaftLib [53], as well as in offload programming models like OpenACC [54] and OpenHMPP [55], that could also benefit from RaCCD. However, in task-based programming models that do not specify data dependences, like Cilk [56] or Intel TBB [57], the runtime system does not know what data is going to be accessed by the tasks, so the proposed ideas are not applicable.

Similar to RaCCD, other works exploit the task annotations to perform optimisations [14], [15]. The input and output information allows the runtime system to transparently manage GPUs [47], [58], stacked DRAM memories [59], multi-node clusters [60], and scratchpad memories [61]. With some additional hardware support, the runtime system can do value approximation [62], software-guided prefetching [13], dead block prediction [16], accelerate critical tasks [63], reduce coherence traffic in CC-NUMA systems [64], [65], and optimise communications in producer-consumer task relationships [66].

VII. Conclusions

This paper 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 $64\times$ smaller directory with only a 2.8% performance degradation.

Acknowledgements

This work has been supported by the RoMoL ERC Advanced Grant (GA 321253), by the European HiPEAC Network of Excellence, by the Spanish Ministry of Economy and Competitiveness (contract TIN2015-65316-P), by the Generalitat de Catalunya (contracts 2014-SGR-1051 and 2014-SGR-1272) and by the European Unions Horizon 2020 research and innovation programme (grant agreements 671697 and 779877). M. Moreto has been partially supported by the Spanish Ministry of Economy, Industry and Competitiveness under Ramon y Cajal fellowship number RYC-2016-21104.

REFERENCES

[1] R. H. Dennard, F. H. Gaensslen, H. nien Yu, V. L. Rideout, E. Bassous, Andre, and R. Leblanc, "Design of ion-implanted MOSFETs with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, Oct. 1974.

[2] D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence*, 1st ed. Morgan & Claypool Publishers, 2011.

[3] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive nuca: Near-optimal block placement and replication in distributed caches," in *International Symposium on Computer Architecture (ISCA)*, 2009, pp. 184–195.

[4] D. Kim, J. Ahn, J. Kim, and J. Huh, "Subspace snooping: Filtering snoops with operating system support," in *International Conference on Parallel Architectures and Compilation (PACT)*, 2010, pp. 111–122.

[5] B. A. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. 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.

[6] A. Ros and S. Kaxiras, "Complexity-effective multicore coherence," in *International Conference on Parallel Architectures and Compilation (PACT)*, 2012, pp. 241–252.

[7] 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.

[8] "OpenMP Application Program Interface. Version 4.0. July 2013."

[9] A. R. Lebeck and D. A. Wood, "Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors," in *International Symposium on Computer Architecture (ISCA)*, 1995, pp. 48–59.

[10] A. Esteve, A. Ros, A. Robles, M. E. Gómez, and J. Duato, "Tokentlb: A token-based page classification approach," in *International Conference on Supercomputing (ICS)*, 2016, pp. 26:1–26:13.

[11] A. Esteve, A. Ros, M. E. Gómez, A. Robles, and J. Duato, "Efficient tlb-based detection of private pages in chip multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 3, pp. 748–761, Mar. 2016.

[12] A. Esteve, A. Ros, M. E. Gómez, A. Robles, and J. Duato, "Tlb-based temporality-aware classification in cmps with multilevel tlbs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 8, pp. 2401–2413, Jan. 2017.

[13] V. Papaefstathiou, M. G. Katevenis, D. S. Nikolopoulos, and D. Pnevmatikatos, "Prefetching and cache management using task lifetimes," in *International Conference on Supercomputing (ICS)*, 2013, pp. 325–334.

[14] M. Casas, M. Moreto, L. Alvarez, E. Castillo, D. Chasapis, T. Hayes, L. Jaulmes, O. Palomar, O. Unsal, A. Cristal *et al.*, "Runtime-aware architectures," in *International Conference on Parallel and Distributed Computing (Euro-Par)*, 2015, pp. 16–27.

[15] M. Valero, M. Moreto, M. Casas, E. Ayguade, and J. Labarta, "Runtime-aware architectures: A first approach," *International Journal on Supercomputing Frontiers and Innovations*, vol. 1, no. 1, pp. 29–44, Jun. 2014.

[16] M. Manivannan, V. Papaefstathiou, M. Pericàs, and P. Stenström, "RADAR: runtime-assisted dead region management for last-level caches," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 644–656.

[17] 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," *IEEE Computer*, vol. 36, no. 12, pp. 49–58, Dec 2003.

[18] P. Ranganathan, S. V. Adve, and N. P. Jouppi, "Reconfigurable caches and their application to media processing," in *International Symposium on Computer Architecture (ISCA)*, 2000, pp. 214–224.

[19] K. Varadarajan, S. K. Nandy, V. Sharda, B. Amrutur, R. R. Iyer, S. Makineni, and D. 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.

[20] M. Powell, S.-H. Yang, B. Falsafi, K. 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.

[21] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Computer Architure News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[22] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "OmpSs: A proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, no. 2, pp. 173–193, 2011.

[23] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. 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.

[24] S. Xi, H. Jacobson, P. Bose, G.-Y. Wei, and D. 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.

[25] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A tool to model large caches," 2009.

[26] 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.

[27] A. Gupta, W.-D. Weber, and T. Mowry, "Reducing memory and traffic requirements for scalable directory-based cache coherence schemes," in *International Conference on Parallel Processing (ICPP)*, 1990, pp. 312–321.

[28] J. H. Choi and K. H. 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.

[29] H. Zhao, A. Shriraman, and S. Dwarkadas, "Space: Sharing pattern-based directory coherence for multicore scalability," in *International Conference on Parallel Architectures and Compilation (PACT)*, 2010, pp. 135–146.

[30] D. Sanchez and C. Kozyrakis, "Scd: A scalable coherence directory with flexible sharer set encoding," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2012, pp. 1–12.

[31] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos, "A tagless coherence directory," in *International Symposium on Microarchitecture (MICRO)*, 2009, pp. 423–434.

[32] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi, "Cuckoo directory: A scalable directory for many-core systems," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2011, pp. 169–180.

[33] A. Moshovos, "Regionscout: Exploiting coarse grain sharing in snoop-based coherence," in *International Symposium on Computer Architecture (ISCA)*, 2005, pp. 234–245.

[34] J. F. Cantin, M. H. Lipasti, and J. E. Smith, "Improving multiprocessor performance with coarse-grain coherence tracking," in *International Symposium on Computer Architecture (ISCA)*, 2005, pp. 246–257.

[35] M. Alisafaee, "Spatiotemporal coherence tracking," in *International Symposium on Microarchitecture (MICRO)*, 2012, pp. 341–350.

[36] J. Zebchuk, B. Falsafi, and A. Moshovos, "Multi-grain coherence directories," in *International Symposium on Microarchitecture (MICRO)*, 2013, pp. 359–370.

[37] "Programmer's Guide for ARMv8-A. Version 1.0. 2015."

[38] B. Cuesta, A. Ros, M. E. Gomez, A. Robles, and J. Duato, "Increasing the effectiveness of directory caches by avoiding the tracking of non-coherent memory blocks," *IEEE Transactions on Computers*, vol. 62, no. 3, pp. 482–495, Mar. 2013.

[39] P.-A. Tsai, N. Beckmann, and D. Sanchez, "Jenga: Software-defined cache hierarchies," in *International Symposium on Computer Architecture (ISCA)*, 2017, pp. 652–665.

[40] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. 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.

[41] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C. Chou, "Denovo: Rethinking the memory hierarchy for disciplined parallelism," in *International Conference on Parallel Architectures and Compilation (PACT)*, 2011, pp. 155–166.

[42] H. Sung, R. Komuravelli, and S. 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.

[43] Y. Li, A. Abousamra, R. Melhem, and A. K. Jones, "Compiler-assisted data distribution for chip multiprocessors," in *International Conference on Parallel Architectures and Compilation (PACT)*, 2010, pp. 501–512.

[44] Y. Li, R. Melhem, and A. 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.

[45] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao, "Using a "Codelet" program execution model for exascale machines: Position paper," in *Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era (EXADAPT)*, 2011, pp. 64–69.

[46] L. V. Kale and S. Krishnan, "CHARM++: A portable concurrent object oriented system based on C++," in *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1993, pp. 91–108.

[47] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A unified platform for task scheduling on heterogeneous multicore architectures," in *International Conference on Parallel and Distributed Computing (Euro-Par)*, 2009, pp. 863–874.

[48] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012, pp. 66:1–66:11.

[49] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: Programming the memory hierarchy," in *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2006, pp. 83:1–83:11.

[50] J. Shirako, J. M. Zhao, V. K. Nandivada, and V. N. Sarkar, "Chunking parallel loops in the presence of synchronization," in *International Conference on Supercomputing (ICS)*, 2009, pp. 181–192.

[51] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati, "Accelerating code on multi-cores with fastflow," in *International Conference on Parallel and Distributed Computing (Euro-Par)*, 2011, pp. 170–181.

[52] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "Streamit: A language for streaming applications," in *International Conference on Compiler Construction (CC)*, 2002, pp. 179–196.

[53] J. C. Beard, P. Li, and R. D. Chamberlain, "Raftlib: A c++ template library for high performance stream parallel processing," in *International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM)*, 2015, pp. 96–105.

[54] "The OpenACC Application Programming Interface. Version 2.5. October 2015."

[55] R. Dolbeau, S. Bihan, and F. Bodin, "Hmpp: A hybrid multi-core parallel programming environment," in *Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU)*, 2007.

[56] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 1995, pp. 207–216.

[57] J. Reinders, *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. O'Reilly Media, 2007.

[58] J. Planas, R. M. Badia, E. Ayguade, and J. Labarta, "Self-adaptive OmpSs tasks in heterogeneous environments," in *International Parallel and Distributed Processing Symposium (IPDPS)*, 2013, pp. 138–149.

[59] L. Alvarez, M. Casas, J. Labarta, E. Ayguade, M. Valero, and M. Moreto, "Runtime-guided management of stacked dram memories in task parallel programs," in *International Conference on Supercomputing (ICS)*, 2018, pp. 379–391.

[60] J. Bueno, X. Martorell, R. M. Badia, E. Ayguadé, and J. Labarta, "Implementing OmpSs support for regions of data in architectures with multiple address spaces," in *International Conference on Supercomputing (ICS)*, 2013, pp. 359–368.

[61] L. Alvarez, M. Moreto, M. Casas, E. Castillo, X. Martorell, J. Labarta, E. Ayguade, and M. Valero, "Runtime-guided management of scratchpad memories in multicore architectures," in *International Conference on Parallel Architectures and Compilation (PACT)*, 2015, pp. 379–391.

[62] I. Brumar, M. Casas, M. Moretó, M. Valero, and G. S. Sohi, "ATM: approximate task memoization in the runtime system," in *International Parallel and Distributed Processing Symposium (IPDPS)*, 2017, pp. 1140–1150.

[63] E. Castillo, M. Moretó, M. Casas, L. Alvarez, E. Vallejo, K. Chronaki, R. M. Badia, J. L. Bosque, R. Beivide, E. Ayguadé, J. Labarta, and M. Valero, "CATA: criticality aware task acceleration for multicore processors," in *International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 413–422.

[64] P. Caheny, L. Alvarez, S. Derradji, M. Valero, M. Moretó, and M. Casas, "Reducing cache coherence traffic with a numa-aware runtime approach," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 5, pp. 1174–1187, May 2018.

[65] P. Caheny, M. Casas, M. Moretó, H. Gloaguen, M. Saintes, E. Ayguadé, J. Labarta, and M. Valero, "Reducing cache coherence traffic with hierarchical directory cache and numa-aware runtime scheduling," in *International Conference on Parallel Architectures and Compilation (PACT)*, 2016, pp. 275–286.

[66] M. Manivannan, A. Negi, and P. Stenström, "Efficient forwarding of producer-consumer data in task-based programs," in *International Conference on Parallel Processing (ICPP)*, 2013, pp. 517–522.