TD-NUCA: Runtime Driven Management of NUCA Caches in Task Dataflow Programming Models

Paul Caheny[†1] *paul.caheny@intel.com*
Lluc Alvarez[‡§] *lluc.alvarez@bsc.es*
Marc Casas[‡§] *marc.casas@bsc.es*
Miquel Moreto[‡§] *miquel.moreto@bsc.es*

[†]Intel Corporation, Leixlip, Ireland
[‡]Barcelona Supercomputing Center (BSC), Barcelona, Spain
[§]Polytechnic University of Catalonia, Barcelona, Spain

***Abstract*—In high performance processors, the design of on-chip memory hierarchies is crucial for performance and energy efficiency. Current processors rely on large shared Non-Uniform Cache Architectures (NUCA) to improve performance and reduce data movement. Multiple solutions exploit information available at the microarchitecture level or in the operating system to optimize NUCA performance. However, existing methods have not taken advantage of the information captured by task dataflow programming models to guide the management of NUCA caches.**

In this paper we propose TD-NUCA, a hardware/software co-designed approach that leverages information present in the runtime system of task dataflow programming models to efficiently manage NUCA caches. TD-NUCA identifies the data access and reuse patterns of parallel applications in the runtime system and guides the operation of the NUCA caches in the hardware. As a result, TD-NUCA achieves a 1.18x average speedup over the baseline S-NUCA while requiring only 0.62x the data movement.

***Index Terms*—cache memory, data flow computing, parallel architectures**

I. Introduction

Since the end of Dennard scaling [36], the increase in the number of cores has caused an expansion in the size of the on-chip Last-Level Caches (LLCs) to supply the cores with data. Due to this trend, physical and manufacturing constraints have demanded that LLCs are organized in multiple banks that are physically distributed across the die area [9]. Such organization gives rise to varying access latencies between the cores and the LLC banks, leading to a Non-Uniform Cache Access (NUCA) organization, where the latency of a cache access depends on the physical distance between the requesting core and the location of the data.

Different solutions have been proposed to manage shared NUCA LLCs. Modern commercial processors implement Static NUCA (S-NUCA), which uses a simple static address interleaving approach to uniquely place cache blocks among the banks. This approach maximizes the capacity of the cache and simplifies cache coherence, but results in a sub-optimal distribution of cache blocks across the banks from an access latency perspective. To overcome this limitation, many works in the literature propose Dynamic NUCA (D-NUCA) designs that dynamically change the bank location of cache blocks to minimize the latency of cache accesses. To do so, D-NUCA designs decide the best allocation strategy for each cache block based on its access pattern and apply well-known strategies such as allocating private cache blocks close to the accessing core, replicating shared read-only cache blocks in multiple banks, and bypassing the LLC for non-reused cache blocks. Although many proposals perform these actions at the microarchitectural level [13], [14], [31], [32], [35], [45], [56], [98], state-of-the-art techniques such as Reactive NUCA (R-NUCA) [48], [49] rely on Operating System (OS) support to identify the data access patterns [34], [48], [49], [59], [79], [80], which has important drawbacks [41]–[43]. OS-based approaches identify shared and private data, but they are unable to identify temporarily private and non-reused data. Moreover, they operate at the page granularity, so they suffer from misclassified cache blocks, and they require extensive changes in the TLBs and costly TLB invalidations.

The era of parallelism and specialization has also motivated important innovations in parallel programming, as traditional fork-join programming models are too inflexible to take advantage of the potential performance of modern hardware. Task dataflow programming models such as OpenMP 4.0 [72] have emerged as a promising solution to the programmability problems of complex architectures. In these programming models, parallelism is expressed by simply dividing the code into tasks with data inputs and outputs. In the task dataflow execution model, the tasks are created in program order, and the runtime system constructs a Task Dependency Graph (TDG) by analyzing the data inputs and outputs of the tasks, dynamically schedules tasks on the available hardware resources, and synchronizes the tasks respecting their dependencies.

This paper proposes TD-NUCA, a hardware/software co-designed approach that leverages the information present in the runtime system of task dataflow programming models to efficiently manage NUCA LLCs. TD-NUCA works transparently to the applications, without any source code modifications. At the runtime system level, TD-NUCA identifies the access and reuse patterns of the data in a parallel application, and drives the management of the NUCA LLC accordingly. To this end, the runtime system monitors the creation, execution and finalization of tasks to classify the data they access as private, shared read-only or shared, as well as predicting which

[1]Paul Caheny contributed to this work prior to joining Intel.

 DOI 10.1109/SC41404.2022.00085

data will not be reused in future tasks. With this information, the runtime system orchestrates the allocation, migration, replication and flushing of the data in the NUCA LLC. The decisions taken by the runtime system are communicated to the hardware, which introduces a lightweight directory per core to map task dependencies to their location in the NUCA LLC. When a private cache requests or writes back a cache block from/to the LLC, it looks up its directory to determine which LLC bank the request/writeback will be sent to. To communicate between the runtime system and the architecture, TD-NUCA introduces three simple ISA instructions that allow the runtime system to manage the directories and to trigger the necessary cache flushes when data is relocated in the NUCA LLC. This paper makes the following contributions:

- We demonstrate that current approaches to manage NUCA LLCs do not cope well with task dataflow programs. The dynamic nature of these applications prevents existing approaches from efficiently managing the LLC.
- We propose low-overhead extensions to the runtime systems of task dataflow programming models to detect data access and reuse patterns to manage the NUCA LLC.
- We propose minimal hardware support to allow the runtime system to communicate NUCA management decisions to the architecture. The hardware modifications consist of a lightweight directory per core and simple ISA instructions to trigger operations in the NUCA cache.
- We model in detail TD-NUCA in gem5 [16] and we demonstrate that, in a 16-core processor with a 4x4 mesh, TD-NUCA outperforms S-NUCA by an average speedup of 1.18x while reducing data movement by 0.62x on average.

II. Background and Motivation

A. Management of NUCA Caches

The foundational work on NUCA caches was presented in 2002 by Kim et al. [58]. NUCA caches are comprised of multiple discrete physical cache banks distributed across a chip and connected together by a NoC. Figure 1 shows an example of a tiled architecture with a NUCA LLC. The main advantages of NUCA designs over monolithic caches are their improved scalability, reduced contention per bank, and increased overall bandwidth. NUCA designs differ in two crucial design choices pertinent to all NUCA caches:

- *NUCA Mapping*: How to decide which cache bank a particular cache block is placed in on allocation.
- *NUCA Search*: When requested by a core, how to locate a particular cache block within the multiple cache banks.

Modern commercial processors employ S-NUCA, which is the simplest NUCA variant both in conception and implementation. This design distributes the sets of the cache across the banks and places all the ways of each set within the same bank. The address of a memory reference uniquely maps a cache block to a particular cache bank via a static function. To this end, an *address interleaving* approach distributes the cache blocks among all the banks based on their physical address.

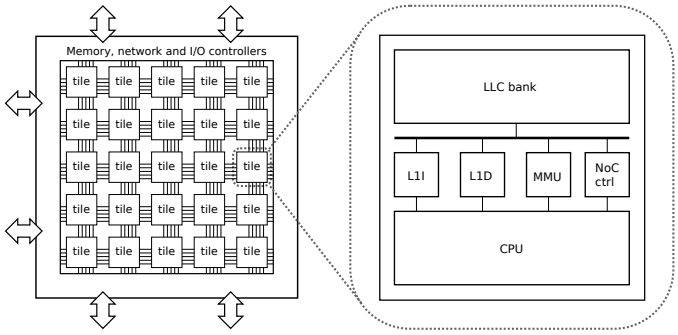

Fig. 1: Tiled architecture with a NUCA LLC.

As a result, NUCA Mapping and Search are trivial, the LLC capacity is maximized, and the utilization of the banks is balanced. However, the main drawback of S-NUCA caches is the sub-optimal distance traversed in the NoC between an accessing core and the required cache block in the LLC, which we call *NUCA distance*. Sub-optimal NUCA distance results in sub-optimal latency for LLC accesses.

Pioneering work on D-NUCA caches for multicore systems by Beckmann et al. [14] and Huh et al. [50] minimize NUCA distance and access latency by dynamically choosing which bank a given cache block is mapped to. As a result, the NUCA Mapping and Search components of the design are more complex. Also, D-NUCA caches pave the way for two additional features to be introduced to the NUCA design:

- *Block Migration*: Dynamically change the bank a cache block is mapped to, based on the cores accessing it.
- *Block Replication*: Replicate a cache block among multiple banks if it is shared read-only among multiple cores.

D-NUCA caches combine dynamic NUCA mapping, block migration and replication to minimize the NUCA distance between cache blocks and their requesting cores. As a result, the NUCA Search function becomes more complex and costly, and additional intelligence is required to decide when and where to map, migrate or replicate cache blocks. This is typically done by monitoring the access and sharing patterns of the cache blocks and using this information to drive such decisions in a sound manner. Ideally, private cache blocks only accessed by one core should be placed in the local bank to minimize access latency; shared read-only cache blocks accessed by many cores could be replicated to reduce access latency; shared cache blocks that are read and written by multiple cores should be uniquely placed in a single bank to simplify coherence and the block search function; and cache blocks without reuse can bypass the LLC to increase its effective capacity without affecting performance [57], [68].

B. OS-based Dynamic NUCA

Despite the large body of work on hardware managed D-NUCA caches, none of the proposals have overcome their fundamental problems without introducing significant complexity and cost in either area or latency. For this reason, state-of-the-art proposals pursue a hardware/software co-designed approach. Among all the solutions, the most well-known and

compelling one is Reactive-NUCA (R-NUCA
biggest breakthrough of R-NUCA with resp
works is to identify data sharing patterns at th
municate them to the hardware via the TLB, a
with simple NUCA Mapping and NUCA Sea

R-NUCA monitors data at a page granul
fies pages into three types: (i) instruction p
read-only and typically shared across cores,
pages, which have been accessed only by a
(iii) shared data pages, which have been acce
cores. With this classification, R-NUCA propo
effective mechanism to manage the NUCA
blocks belonging to instruction pages are r
banks, (ii) cache blocks belonging to private
placed in the local bank of the core acces
(iii) cache blocks belonging to shared data p
interleaved among all the banks of the N
balance cache capacity and NUCA distance
R-NUCA proposes a rotational interleaving
allows for a statically configurable degree o
cache blocks belonging to instruction pages (
copies across the 16 banks used in the R-NU

C. OS-based Data Classification

Identifying data sharing patterns in the OS as in R-NUCA and other works [34], [59], [79], [80] has important drawbacks. This approach categorizes pages as private, shared, or shared read-only. A page is categorized as private when it is accessed for the first time, and the OS sets a private bit in the Page Table and the TLB of the accessing core. When another core accesses the page, the OS marks the page as shared and flushes the cache blocks and the TLB entry of the page in the first core. A page is classified as shared read-only if it is shared and the dirty bit is not set. A limitation of this approach is that, once a page has been categorized as shared, it can never transition back to private. As a result, pages that are temporarily private to different cores in different execution phases are categorized as shared. This problem is particularly important in the presence of dynamic schedulers, where computations and the data they access often migrate across cores. In addition, this technique operates at page granularity, which leads to misclassified blocks if a block in the page is shared or even if two distinct core-private blocks within the same page are accessed by different cores.

Extensive and costly modifications are required to identify temporarily private data in OS-based approaches [41]–[43]. The idea is to, upon a TLB miss, check if the page is present in any other TLB and, if the page is not present in any TLB, categorize the page as private. This approach requires TLB-L1 inclusivity, complex hardware support to perform TLB-to-TLB miss resolution, and costly TLB shootdowns during page re-classifications. In addition, this solution suffers from a lack of temporal precision because TLB entries experience dead time, that is, the time between the last access to a page and its eviction 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, which introduces additional hardware costs and performance overheads due to TLB misses. Note that R-NUCA and other works [34], [59], [79], [80] do not use this approach to identify temporarily private data.

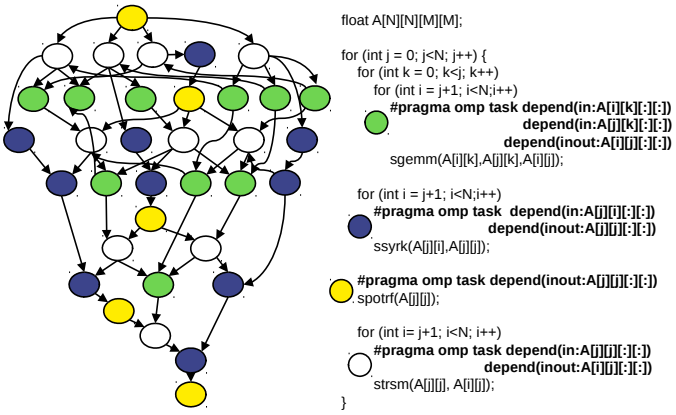

Fig. 2: Cholesky task dependency graph and task-based code.

D. Task Dataflow Programming Models

Task dataflow programming models such as OpenMP 4.0 [72] conceive the execution of a parallel program as a set of *tasks* with data dependencies 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 also whether the data is read, written, or both (labelled *in*, *out* and *inout*, respectively). The runtime system dynamically executes tasks by means of a *Task Dependency Graph (TDG)*, a directed acyclic graph where the nodes represent tasks and the edges are data dependencies between tasks. Figure 2 shows the task dataflow implementation of a Cholesky factorization algorithm and its corresponding TDG. The code uses OpenMP 4.0 clauses to specify tasks and their data dependencies (*#pragma omp task depend(in/out/inout)*).

Following an execution model that 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 synchronization 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 input dependencies of a task have been satisfied, a task moves from created to ready and stored in a ready queue. The dynamic task scheduler distributes ready tasks among all threads for asynchronous execution. This decoupling of the specification of the program from its dynamic execution eases programmability and enables many optimizations at the runtime system level in a generic and application-agnostic way [22], [24], [65], [75], [93].

E. Opportunity for Runtime System Driven NUCA Caches

The code annotations and the execution model of task dataflow programming models provide the runtime system abundant and very valuable information on the data access

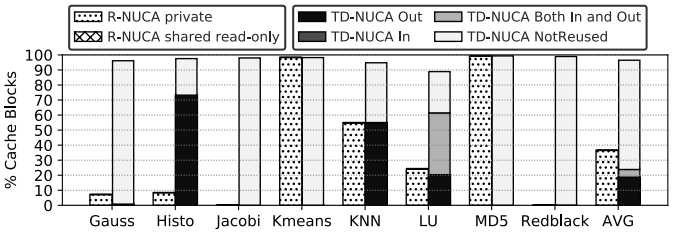

Fig. 3: Categorization of access and reuse patterns of LLC cache blocks in R-NUCA and TD-NUCA.

behavior, which can be exploited to efficiently manage NUCA caches. In particular, the runtime system knows what data is going to be read and written by each task before the task is executed, so it can easily identify private, shared read-only and shared data. In addition, the runtime system is also in charge of scheduling task to cores, so it knows which cores are going to access which data at what time. On top of this, the runtime system also has knowledge of how tasks will access data in the future due to the presence of created, but not yet ready tasks in the TDG. Thus, it can predict if data will not be reused. Thanks to these properties, the runtime system can precisely identify the sharing and reuse patterns of data without the hardware complexity or accuracy problems of other approaches. This information can be exploited to efficiently manage NUCA caches at the runtime system level.

Figure 3 shows the opportunity for both R-NUCA and TD-NUCA to perform NUCA management for a set of task dataflow benchmarks. Section IV describes in detail the experimental methodology. The figure shows a pair of stacked bars per benchmark. The left bar shows the percentage of unique cache blocks that R-NUCA identifies as private and shared read-only. A cache block is recorded as private if it is only accessed by a single core during the execution, and it is recorded as shared read-only if it has been accessed by more than one core during the execution, but never been written. The right bar shows the percentage of unique cache blocks made up of task dependencies in task dataflow programming models, and is broken down into four types: cache blocks belonging to dependencies that are only used as writable (Out), cache blocks belonging to dependencies that are only used as readable (In), cache blocks belonging to dependencies that are used as both readable and writable (Both In and Out), and cache blocks belonging to dependencies that are predicted to be non-reused (NotReused). In both bars, the portion of cache blocks not inside the bars are classified as shared. On average, 96% of the cache blocks belong to task dependencies in TD-NUCA, whereas R-NUCA categorizes as shared 64% of the total cache blocks, so it can only optimize the placement of 36% of the cache blocks. Moreover, R-NUCA categorizes as shared more than 90% of the cache blocks in 4 of the 8 benchmarks. The low coverage of R-NUCA is caused by its inability to identify temporarily private cache blocks, which are abundant in many task dataflow programs due to the dynamic task schedulers used in the runtime system. Note that R-NUCA categorizes less than 1% of cache blocks as shared read-only in all the benchmarks because, after reading a cache block, most often it is never accessed by another core (in which case it is categorized as private) or it is later written by another core (in which case it is categorized as shared). In TD-NUCA, the runtime system is able to predict that a very large amount of cache blocks are not going to be reused, 72% on average, while the OS-based approach is unable to capture data reuse patterns. In conclusion, compared to OS-based approaches, the runtime system of task dataflow programming models is able to capture much more precise information about the sharing and reuse patterns of the data, and thus has significantly more potential to optimize the management of NUCA caches.

III. TD-NUCA: Task Dataflow NUCA

This section presents TD-NUCA, a hardware/software co-designed approach to optimizing the utilization of NUCA caches by leveraging the information present in the runtime system of task dataflow programming models.

At the runtime system level, TD-NUCA monitors the creation, execution and finalization of tasks to identify the access and reuse patterns of the task dependencies. With this information, the runtime system dynamically decides the best data placement strategy for the dependencies of the tasks before they are executed. TD-NUCA makes the following placement decisions: (i) predicted NotReused dependencies bypass the LLC; (ii) Out and InOut dependencies are private to the task, so they are allocated in the local LLC bank of executing core for the duration of the task; (iii) In dependencies are read-only and usually shared by multiple tasks, so they are replicated in multiple LLC banks; and (iv) data not specified in task dependencies is not tracked by TD-NUCA, so it is address interleaved across banks as in an S-NUCA organization.

In the architecture, TD-NUCA introduces a lightweight structure per core called *Runtime Region Table (RRT)* to store the mappings of task dependencies to LLC banks. Three new ISA instructions are added to achieve the necessary cooperation between the runtime system and the RRTs. The first ISA instruction allows the runtime system to register in the RRTs the address ranges of the task dependencies and the LLC banks that will store their cache blocks. The second instruction is used by the runtime system to invalidate RRT entries, and the third one invalidates all the cache blocks belonging to a dependency in the desired LLC bank or private cache.

To control the amount of replication of shared read-only data, we propose an *LLC Cluster Replication* scheme that divides the LLC banks into clusters. In our setup with a tiled architecture with 16 tiles, TD-NUCA divides the chip in quadrants, defining 4 clusters of 4 tiles each. A shared read-only task dependency can be replicated up to 4 times, once in each cluster. This way, the cores sharing a dependency can access the replica in their local cluster, reducing the worst case NUCA distance from the chip-wide NoC diameter to the cluster-wide NoC diameter. When a dependency is mapped to a cluster, its cache blocks are address interleaved across the LLC banks that belong to the cluster, balancing the contention and the storage requirements among the banks in the cluster.

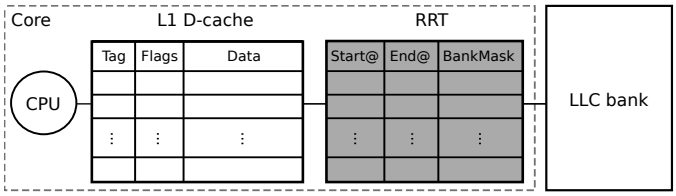

Fig. 4: Architectural support for TD-NUCA.

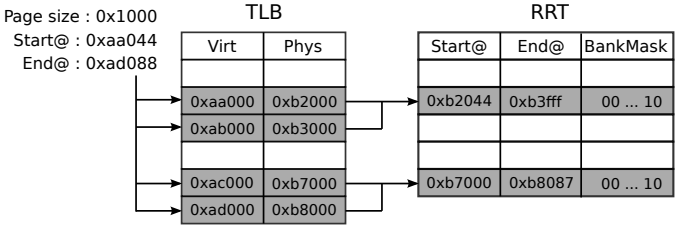

Fig. 5: Address translation for RRT entries.

A. Software - Hardware Interface

TD-NUCA defines an interface between the runtime system and the hardware architecture so they can co-operate in the management of memory regions. The interface consists of three new ISA instructions. The instructions use the concept of *BankMask* and *CoreMask*, which are bit vectors that specify to which tiles the operation is applied. In our 16-core experimental setup the masks contain 16 bits, one per tile.

- `tdnuca_register(initial_address, size, BankMask)` registers an entry in the RRT of the core executing the instruction with the initial virtual address of a task dependency, its size, and a BankMask that specifies the LLC banks the task dependency will be mapped to.
- `tdnuca_invalidate(initial_address, size, CoreMask)` invalidates the entries of a task dependency from the RRTs of the cores specified in the CoreMask.
- `tdnuca_flush(initial_address, size, cache_level, CoreMask)` flushes the cache blocks belonging to a task dependency from the private caches or the LLC banks of the cores specified in the CoreMask.

In addition, TD-NUCA introduces a memory mapped register with 1 bit per core that the hardware uses to signal the completion of a `tdnuca_flush` to the runtime system. The architecture also exposes the number of LLC banks and their placement in the NoC to the runtime system.

B. Hardware Extensions

TD-NUCA introduces simple and efficient hardware support to manage task dependency memory regions. The hardware storage structures and their operation are described next.

1) Microarchitectural Storage Structures: The hardware additions, shown shaded in Figure 4, consist of a new per-core structure called the *Runtime Region Table (RRT)*. The RRT stores the information of the memory regions belonging to task dependencies. The RRT is managed by the runtime system with the instructions detailed in Section III-A. The entries of the RRT have three fields: the start and end physical address of a memory region and its associated BankMask, which specifies the LLC banks the memory region is mapped to. Our experimental setup uses 42-bit physical addresses.

The RRTs perform range lookups to determine if a memory address belongs to a memory region. Current systems typically use Ternary Content-Addressable Memories (TCAM) to perform this operation. These structures can be implemented in numerous ways [30], [71], [73], [74], [92], with different latency, area and power consumption trade-offs. Thus, the implementation of the RRTs can be adapted to meet the desired design constraints. The proposed design assumes a latency of 1 cycle, and Section V-E evaluates the performance impact of varying the latency of the RRTs between 0 and 4 cycles.

2) Registering and De-registering RRT Entries: The runtime system manages the contents of the RRTs by executing the `tdnuca_register` and `tdnuca_invalidate` instructions. When the `tdnuca_register` instruction is executed, the address range specified in the instruction is registered in the RRT of the executing core. The specified address range is in the virtual address space, so it must be translated to the physical address space before being registered in the RRT. Figure 5 shows an example of this process.

To translate the virtual address range to a physical address range, the execution of the `tdnuca_register` instruction 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. This iterative process can take multiple cycles to register a memory region in the RRT. The example in Figure 5 requires 4 TLB accesses and registers 2 collapsed regions in the RRT.

The RRTs have a limited size so, when a virtual address range of a dependency spans over multiple physical address ranges, some physical address ranges may not fit in the RRT. The RRTs do not implement any replacement policy, so no RRT entry is evicted to make room for the exceeding physical address ranges. Instead, when there are no free entries in the RRT, the exceeding physical address ranges are not registered in the RRT, so they are not tracked by TD-NUCA. This loses opportunities for optimization, but it does not break the functionality of TD-NUCA because the cache blocks belonging to these physical address ranges fall back to a static address interleaved allocation across banks.

RRT entries are de-registered by the runtime system using the `tdnuca_invalidate` instruction. The execution of this instruction follows the same process of virtual to physical

address translation as the `tdnuca_register` instruction. By iterating over the virtual address range specified in the instruction parameters, the respective physical addresses are looked-up and removed from the RRTs.

3) Memory Accesses Under TD-NUCA: Memory accesses issued by a core first look for the data in the private cache of the core. Upon private cache misses, the RRT of the core is consulted 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 access is not found in the RRT, the memory reference proceeds as it would in the baseline S-NUCA system. If the address of the memory access is found in the RRT, the BankMask of the RRT entry is checked to identify the LLC bank the request will be sent to. The runtime system guarantees that the contents of the BankMask field in the RRT entries conforms to one of the following conditions:

- If all the bits of the BankMask are set to zero, the memory access bypasses the LLC. TD-NUCA triggers an LLC bypass variant of the coherence transaction directly to the memory controller, and this replies to such a request by forwarding the cache block directly to the requesting private cache, bypassing the LLC. Note that modern architectures already support uncacheable memory accesses.
- If exactly one bit of the BankMask is set, the memory access is served by that LLC bank. To communicate with the LLC bank, the private cache sets the destination bank for its coherence message to the position of the single set bit in the BankMask and sends the request to that LLC bank.
- If exactly 4 bits of the BankMask are set, the task dependency is spread among a cluster of 4 LLC banks. The cache blocks belonging to the task dependency are address interleaved among the 4 banks of the cluster, and the last two bits of the block address of the memory access identify the bank where the block is mapped to within the cluster. Based on this, the hardware calculates the destination LLC bank and sends to it the corresponding coherence message.

Note that the RRT is also checked before initiating coherence messages and writebacks from the private caches to the LLC to identify the LLC bank the requests will be sent to.

4) Cache Flushing: The `tdnuca_flush` instruction flushes the cache blocks of a task dependency from the specified cache level and cores. The `initial_address` and `size` identify the task dependency to be flushed, the `CoreMask` parameter indicates the tiles which will be flushed, and the `cache_level` parameter specifies whether the flush is applied to the LLC banks or to the private caches of the specified tiles. The address range of the dependency is translated from the virtual to the physical address space and a flush transaction for all the cache blocks belonging to the physical address range is initiated in the specified caches.

To synchronize with the `tdnuca_flush` instructions, TD-NUCA uses a memory mapped register that keeps 1 bit per core. The bit of a core is set when a flush on that core starts, and the bit is unset when the flush finishes. The memory mapped register is accessible by all the threads. The runtime system uses a polling loop on this register to check which bits are set and wait for the completion of the desired flushes.

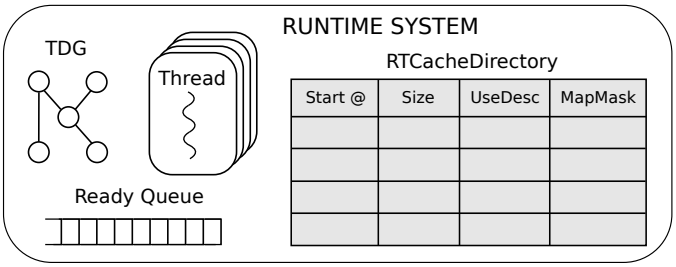

Fig. 6: Runtime system with additions for TD-NUCA.

C. Runtime System Extensions

TD-NUCA extends the runtime system of task dataflow programming models with a new software structure that captures relevant characteristics of the dependencies and with new logic that uses this information to drive NUCA management decisions and communicates them to the hardware layer.

1) Runtime System Data Structures: Figure 6 shows the components of the runtime system, with the new component added by TD-NUCA as shaded. The runtime system is augmented with the RTCacheDirectory structure to track access and reuse patterns of task dependencies. The information stored in the RTCacheDirectory is used by the runtime system as outlined in Section III-C2 to make decisions about which TD-NUCA operations to apply to each task dependency.

The RTCacheDirectory contains a unique entry for each task dependency, and it has 4 fields: the start address and the size of the dependency, the MapMask is a bitvector that encodes which LLC banks the dependency is mapped to, and the *use descriptor (UseDesc)* counts how many times the dependency will be used in the future. The UseDesc is incremented when a task using the dependency is created and decremented when a task using the dependency starts to execute.

2) Runtime System Operational Model: TD-NUCA extends the operational model of task dataflow programming models to decide which TD-NUCA data placement to apply to task dependencies. After the runtime system has scheduled a task to a core, but before the task starts executing, it iterates over the dependencies of the task. For each dependency, the runtime system decides the data placement for the dependency and then communicates to the hardware the characteristics of the dependency through the `tdnuca_register` instruction: the dependency start address, the dependency size, and the dependency BankMask. 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 7.

LLC Bypass: If the use descriptor of the dependency shows that there are no outstanding tasks in the TDG that use the dependency (i.e., UseDesc=0), then the runtime system chooses the Bypass LLC data placement for the dependency.

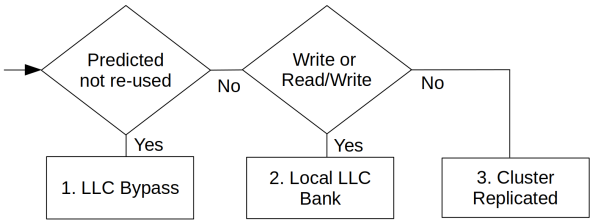

Fig. 7: Runtime decision for LLC data placement.

Zero bits are set in the BankMask communicated with the `tdnuca_register` instruction and in the MapMask of the RTCacheDirectory entry for that dependency. On task end, the runtime system executes the `tdnuca_flush` instruction to flush the dependency from the L1 cache of the executing core, and then it executes the `tdnuca_invalidate` instruction to clear the entry from the RRT of that core.

Local LLC Bank Mapping: If the dependency is Output (write-only) or Input/Output (read-write), the runtime system chooses to map the dependency in the local LLC bank of the core that is going to execute the task. Before the task starts, the `tdnuca_register` instruction is issued with exactly 1 bit set in the BankMask, specifically the bit in the mask position corresponding to the local LLC bank, and the MapMask field of the RTCacheDirectory is updated accordingly. On task end, the runtime system executes the `tdnuca_flush` instruction to flush the dependency from the LLC bank and the private caches of the core the data is mapped to (specified in the MapMask of the RTCacheDirectory field), and then it executes a `tdnuca_invalidate` instruction to clear the corresponding RRT entry of that same core.

Cluster Replicated Mapping: If the dependency has not been assigned to LLC Bypass or Local LLC Bank Mapping, then the dependency is an Input (read-only) that is reused in the future, and the runtime system applies the Cluster Replicated Mapping. Before the task execution, the runtime system issues the `tdnuca_register` instruction with 4 bits set in the BankMask, which correspond to the 4 LLC banks of the local cluster of the core executing the task. These bits are also set in the MapMask field of the RTCacheDirectory entry for that dependency. At the end of the task, the mapping of the dependency remains in the RRT to be used by future tasks.

This operational model, combined with the pre-existing task synchronization 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. In contrast, the cluster replicated data is invalidated lazily. When a dependency transitions from read-only to written, the instructions `tdnuca_invalidate` and `tdnuca_flush` are issued to invalidate all the cache blocks from all the caches and all the RRT entries from all the cores associated with that dependency. Thus, neither the caches nor the RRTs contain stale data when a subsequent task uses the same dependency.

D. Additional Considerations

Task dataflow programming models guarantee no data races will occur for data that is only accessed from within tasks that 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 TD-NUCA

To ensure correct operation, TD-NUCA only operates on cache blocks that are entirely contained within the start and end address of a dependency. To comply with this requirement, if the dependency start or end address is not aligned to a cache block boundary, the runtime system leaves the first and last cache block outside of the address range of the dependency. This ensures only cache blocks that are entirely contained within the bounds of the task dependency start and end address have their behavior modified in the caches. Note that this mechanism loses optimization opportunities for a negligible amount of the data specified in the dependencies, since they typically have sizes of at least hundreds of kilobytes and this mechanism excludes at most 128 bytes (two cache blocks).

TD-NUCA does not require any modifications in the application source code. All the changes in the software are enclosed in the runtime system, so applications only need to link against the library (re-compiling or using library interposition) to capitalize on the benefits of TD-NUCA. For legacy code and non task dataflow programs, the hardware support of TD-NUCA can be powered down, so the only overhead introduced is the small area of the RRTs.

The proposed hardware support for TD-NUCA can be extended to support context switches and multiprogrammed workloads by tagging the RRTs with the OS process ID. This way, different processes can use the RRTs concurrently and the RRTs do not need to be saved and restored at context switches. When the OS migrates a thread between cores, the RRT entries belonging to the thread must also be migrated, and the data in the private cache of the source core must be invalidated with a `tdnuca_invalidate` instruction.

IV. Experimental Framework

We evaluate TD-NUCA with the gem5 [16] simulator, using the x86 out-of-order CPU model and the Ruby memory model. The execution-driven, full-system and cycle-accurate simulations model in detail all the components of the architecture, including the cache coherence protocol and the architectural support for TD-NUCA, and they run a complete software stack including the benchmarks, the runtime system, libraries, and the OS. Table I specifies the main architectural parameters.

The software environment runs Ubuntu 14.04 with kernel version 4.3. The default Linux page allocator is simulated in

TABLE I: gem5 simulator configuration parameters.

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 64 entries fully-associative, 1 cycle
LLC	Inclusive shared unified 32MB, banked 2MB/core 64B/line, 15 cycles, 16-way, pseudoLRU
Coherence Protocol	MESI with blocking states, silent evictions
Cache Directory	Total 512K entries, banked 32K entries/core 15 cycles, 16-way, pseudoLRU
NoC	4x4 mesh, link 1 cycle, router 1 cycle
RRT	64 entries/core, 1 cycle access time

TABLE II: Benchmarks, problem and task sizes.

Bench	Problem set	Input set size (MB)	Num tasks	AVG task size (KB)
Gauss	2D Matrix N^2 = 58982400, 2 iters.	488.04	3200	294
Histo	1500x1500 pixels, 50x50 blocks, 50 bins	478.75	1800	528
Jacobi	2D Matrix N^2 = 16777216, 5 iters.	264.34	320	4112
Kmeans	450000 pts., 90 dims, 6 clusters, 1 iter.	314.37	228	1404
KNN	512/229376 training/input pts, 8 classes	85.01	448	318
LU	2D Matrix N^2 = 9437184	73.45	1188	318
MD5	128 $\times$ 4 MB buffers	513.39	128	4096
Redblack	N^2 = 28901376, 5 iters.	223.96	320	3549

detail. We use the Nanos++ 0.10 [39] runtime system, which supports OpenMP 4.0 [72]. The runtime system communicates with TD-NUCA using the instructions described in Section III. For the new TD-NUCA instructions, we extend the ISA and simulate their execution and their latency in cycle-by-cycle detail, including the iterative process for the virtual to physical address translation of the address regions and the cache flushes triggered by the `tdnuca_flush` instruction.

The evaluation uses a set of task dataflow parallel benchmarks programmed with OpenMP 4.0. 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. Table II shows the benchmarks and the input set sizes used in the experiments. The input set sizes of all the benchmarks exceed the total capacity of the LLC. For the task granularities, we do an exhaustive exploration for each benchmark and select the best performing configuration on the S-NUCA baseline. The benchmarks are run from start to completion, and the evaluation results are collected from the entire post-initialisation parallel execution phase of the benchmarks. The benchmarks are compiled with Mercurium 1.99 source-to-source compiler [10], using gcc 4.6.4 as backend.

Power consumption is evaluated with McPAT [62] 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.* [94] to improve the accuracy of the models. The hardware structures of TD-NUCA are modeled using CACTI 6.0 [70].

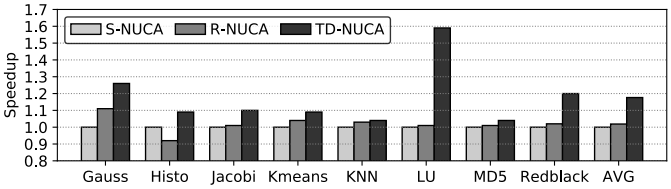

Fig. 8: Performance speedup normalized to S-NUCA.

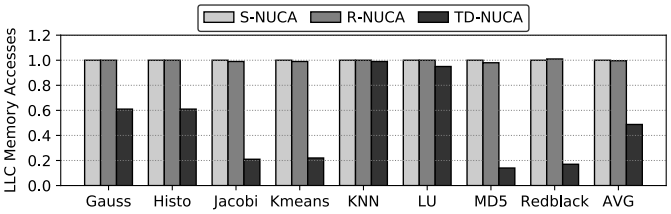

Fig. 9: LLC accesses normalized to S-NUCA.

V. Evaluation

This section evaluates TD-NUCA, comparing it to S-NUCA and R-NUCA. Unless stated otherwise, all the results of this section are normalized to a baseline S-NUCA LLC system.

The original R-NUCA proposal [48] only allows replication of shared read-only cache blocks belonging to instruction pages. To make a fair and more competitive comparison, we augment R-NUCA to also perform replication for shared read-only data pages. An important difference between instruction pages and read-only data pages is that the latter may transition to shared read-write during the execution. When a shared read-only data page is written, the cache blocks belonging to the page are flushed from all the caches. This mechanism is consistent with the approach used in R-NUCA for flushing the caches on private to shared read-write page transitions. This enhancement allows R-NUCA to optimize data placement for shared read-only data and further reduce the NUCA distance.

A. Performance Improvements and Cache Behavior

Figure 8 reports the speedup for R-NUCA and TD-NUCA over S-NUCA. TD-NUCA greatly outperforms S-NUCA in 3 of the 8 benchmarks, Gauss, LU and Redblack, with respective speedups of 1.26x, 1.59x and 1.20x. In 3 other benchmarks, Histo, Jacobi and Kmeans, TD-NUCA achieves significant speedups of 1.09x to 1.10x, while in KNN and MD5 TD-NUCA achieves moderate speedups of 1.04x. The speedups of R-NUCA are lower in all cases, at 1.11x for Gauss and below 1.05x in the rest of benchmarks. On average across all benchmarks, TD-NUCA and R-NUCA achieve respective speedups of 1.18x and 1.02x over S-NUCA.

Figure 9 shows the number of accesses to the LLC in R-NUCA and TD-NUCA compared to S-NUCA. TD-NUCA reduces the number of accesses required versus both S-NUCA and R-NUCA in all the benchmarks, from 0.99x in KNN down to 0.14x in MD5. This is because TD-NUCA bypasses the LLC for accesses to dependencies predicted to not be reused. The number of accesses to the LLC in R-NUCA is within

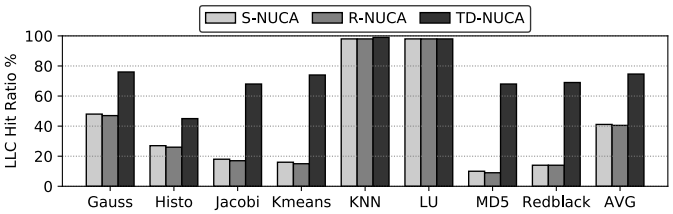

Fig. 10: LLC hit ratio.

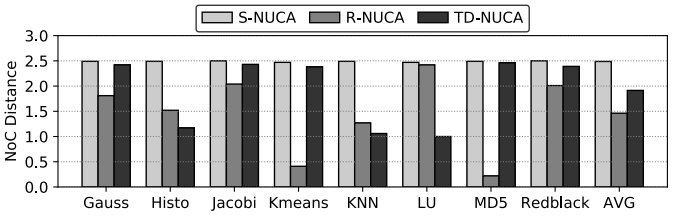

Fig. 11: Average NUCA distance.

0.02x those of S-NUCA in all benchmarks. On average over all the benchmarks, TD-NUCA performs 0.48x the accesses to the LLC compared to S-NUCA, whereas R-NUCA makes 0.99x the number of accesses to the LLC as S-NUCA.

Figure 10 shows the LLC hit ratio achieved by S-NUCA, R-NUCA and TD-NUCA, without any normalization. It can be observed that there are very small differences between S-NUCA and R-NUCA. TD-NUCA however shows significantly higher hit ratios in 6 of the 8 benchmarks. This is due to the lower demand of TD-NUCA for LLC capacity due to LLC bypassing, which causes the non-bypassed data allocated in the LLC to experience fewer conflicts and, thus, an increased hit ratio. In LU and KNN all three approaches have very high hit ratios, close to 100% and within 2% of each other. On average, the LLC hit ratio in TD-NUCA is 74%, while in S-NUCA and R-NUCA it is 41% and 40% respectively.

The TLB hit ratios for S-NUCA, R-NUCA and TD-NUCA are over 99% across all benchmarks. TD-NUCA incurs additional TLB accesses to perform the virtual to physical address translations required to manage the RRTs. However, the amount of TLB accesses added is negligible, below 0.01% in all benchmarks. In addition, we observe that most of these TLB accesses do not cause extra misses because most of the virtual to physical translations triggered by the TD-NUCA instructions happen immediately before or after the execution of a task that requires the same translations. On average across the benchmarks, TD-NUCA marginally decreases the TLB hit ratio from 99.8646% to 99.8636%, so the TLB accesses introduced by TD-NUCA have negligible impact.

B. NUCA Distance and Data Movement

Figure 11 shows the NUCA distance travelled for messages from the requesting cores to the LLC banks, without any normalization. In this metric, a core accessing its local LLC bank counts as distance 0, a core accessing a neighbor LLC bank (i.e., directly adjacent to its north, south, east, or west) is counted as distance 1, and so on for greater distances.

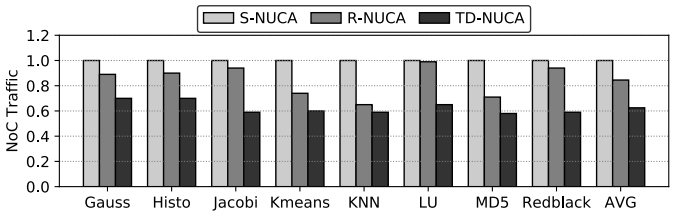

Fig. 12: Data movement in the NoC normalized to S-NUCA.

Accesses to blocks that bypass the LLC in TD-NUCA are not included in the calculation of the metric. Results show that S-NUCA achieves a perfectly uniform distribution of cache blocks among the LLC banks, with an average NUCA distance of 2.49. Note that the theoretical average NUCA distance in a 4x4 mesh is 2.5. R-NUCA reduces the NUCA distance, down to 1.46 on average, by mapping private blocks into the local LLC bank of the accessing cores and by replicating shared read-only blocks in multiple LLC banks. The average NUCA distance for TD-NUCA is 1.91, which represents an important improvement over S-NUCA. However, the NUCA distance for TD-NUCA is higher than R-NUCA because this metric does not count the blocks that bypass the LLC, which make up a majority of the working set in Gauss, Jacobi, Kmeans, MD5 and Redblack, as shown in Figure 3. Although R-NUCA achieves a lower NUCA distance than TD-NUCA in these benchmarks, bypassing the LLC provides larger performance improvements. In the benchmarks with smaller proportions of bypassed blocks (Histo, KNN and LU), TD-NUCA achieves significantly lower average NUCA distances than R-NUCA.

Figure 12 presents the total data movement in the NoC. This metric is computed as the aggregate number of bytes transferred through all routers in the NoC, including LLC bypassed blocks under TD-NUCA that travel from the DRAM to the private L1 caches. TD-NUCA reduces the data movement in the NoC by between 0.58x (MD5) and 0.70x (Gauss and Histo) versus S-NUCA, achieving an average reduction over all benchmarks of 0.62x. On average, R-NUCA only achieves 0.84x the data movement in S-NUCA. These reductions of both TD-NUCA and R-NUCA are mostly caused by the reduced average NoC distances between the LLC banks and the requesting cores, and by the reduction of LLC misses, which saves data requests and writebacks to the DRAM.

C. Energy Consumption

TD-NUCA and R-NUCA modify the operation of the LLC, which impacts the energy consumption within both the LLC and the NoC. Figures 13 and 14 present, respectively, the dynamic energy consumed in the LLC and in the NoC.

Figure 13 shows that TD-NUCA significantly reduces the dynamic energy consumed in the LLC for all benchmarks except LU. The largest saving is achieved in Jacobi, which consumes only 0.10x the dynamic energy in the LLC consumed by S-NUCA. This is due to the LLC bypass feature of TD-NUCA, which also reduces dynamic energy significantly in Gauss, Histo, Kmeans, MD5 and Redblack. In contrast,

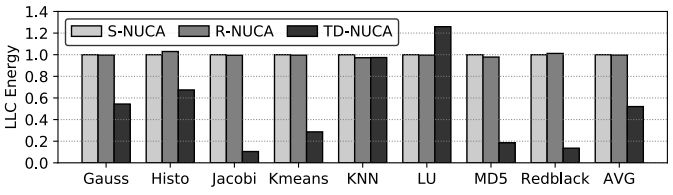

Fig. 13: LLC dynamic energy normalized to S-NUCA.

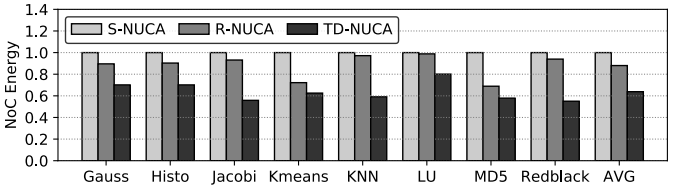

Fig. 14: NoC dynamic energy normalized to S-NUCA.

the replication of read-only data causes an increased energy consumption in LU. Although this operation is costly in terms of energy, it enables less data movement in the NoC and better performance, as shown previously. On average across all the benchmarks, TD-NUCA consumes 0.52x the dynamic energy of S-NUCA. Conversely, R-NUCA consumes the same dynamic energy as S-NUCA in the LLC on average.

Figure 14 shows the dynamic energy consumption in the NoC. TD-NUCA consumes between 0.55x (Redblack) and 0.80x (LU) the energy compared to S-NUCA for an average of 0.64x, while R-NUCA consumes between 0.68x (MD5) and 0.98x (LU) over S-NUCA for an average of 0.88x. These results follow the same trends as those for data movement in the NoC, which is the main cause of the energy consumption.

D. Effectiveness of LLC Bypassing

Figure 15 quantifies the effectiveness of the LLC bypassing feature of TD-NUCA. The plot compares the performance of S-NUCA, TD-NUCA, and a variant of TD-NUCA that only performs LLC bypassing for the cache blocks that the runtime system predicts are not going to be reused. This variant, which is labeled as TD-NUCA (Bypass Only) in the figure, does not map private cache blocks to the local LLC banks nor performs cluster replication of read-only cache blocks.

Results show that the TD-NUCA variant that only does LLC bypassing does not provide any benefit in Histo, KNN and LU. As shown in Figure 3, these 3 benchmarks present a low percentage of cache blocks predicted not to be reused. In these cases the LLC bypass optimization is not effective, while the complete TD-NUCA design is able to achieve notable performance gains by mapping private blocks to the local LLC banks and by applying cluster replication to read-only blocks. The opposite situation happens in Jacobi, Kmeans, MD5 and Redblack, where the percentage of cache blocks predicted not to be reused is over 97%, and the speedups obtained by the TD-NUCA variant that only does LLC bypassing match the speedups obtained by the complete TD-NUCA design. In Gauss we observe that the complete TD-NUCA is much

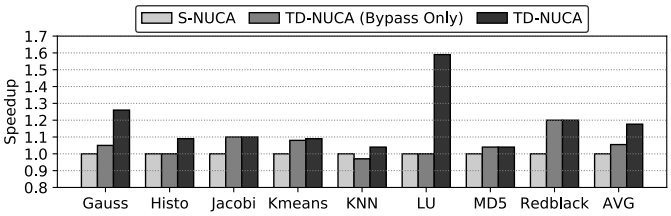

Fig. 15: Performance speedup of a TD-NUCA variant that only performs LLC bypassing normalized to S-NUCA.

more effective than the TD-NUCA variant that only does LLC bypassing. In this benchmark, 94% of the cache blocks are predicted not to be reused, but only 32% of the L1 cache misses are caused by accesses to these blocks, while there is a 2% of unique cache blocks that are used both as input and output dependencies which are responsible for 41% of the total L1 cache misses. Due to this circumstance, the LLC bypassing policy of TD-NUCA provides some benefits in Gauss, but its combination with the cluster replication and the local LLC bank mapping policies further increases the performance gains. On average across all the benchmarks, the speedup of the TD-NUCA variant that only does LLC bypassing is 1.06x, while the complete TD-NUCA design reaches 1.18x.

E. TD-NUCA Design Trade-offs and Overheads

TD-NUCA introduces minimal hardware overheads. In terms of storage requirements, TD-NUCA introduces a 64-entry RRT per core, which require a chip-wide total storage of 12.5KB. We model each RRT in CACTI as an SRAM and we multiply its energy consumption by 30 [92] to approximate the cost of a real TCAM implementation. We observe that all the RRTs combined add negligible overheads in energy consumption, as they consume less than 0.1% of the total energy consumed by the architecture.

Although the RRTs add a delay of 1 cycle to the private cache misses and writebacks, this has a negligible impact on performance. Compared to an ideal RRT with zero latency, the average performance overhead is 0.1%, and augmenting the latency of the RRTs to 2, 3 and 4 cycles adds average performance overheads of 0.5%, 1.1%, and 1.9%, respectively.

The performance impact of the cache flushing required by TD-NUCA is also negligible. We observe that, in all benchmarks but Histo, less than 0.1% of the total execution time is spent in flushing cache blocks. Histo spends 0.49% of the total execution time in flushing cache blocks, as it has the highest proportion of Out dependencies of any benchmark.

The occupancy of the RRTs is 14.71 entries per RRT on average during the whole execution of all the benchmarks. In Gauss, Histo, Kmeans and KNN, the maximum occupancy of any RRT never exceeds 23 entries. However, in the other benchmarks (Jacobi, LU, MD5, and Redblack) we observe a higher occupancy of the RRTs, up to 59 entries of the RRT of a core in Redblack. The occupancy of the RRTs is directly related to the size of the tasks, shown in Table II. In LU, which uses task sizes of 318KB on average, the RRT occupancy

reaches a maximum of 37 entries due to the presence of shared read-only data, which is not invalidated at the end of the execution of the tasks and, thus, adds capacity pressure on the RRTs. In Jacobi, MD5 and Redblack, which use larger task sizes of around 4MB, some dependencies require multiple entries in the RRT because the virtual address ranges of the dependencies are not contiguous in the physical address space. Altogether, using 64-entry RRTs per core is sufficient to always track all the address ranges of all the dependencies. In addition, our setup uses 4KB pages, so larger page sizes could be leveraged to reduce the capacity pressure on the RRTs.

Finally, we measure the performance overhead of the runtime systems extensions of TD-NUCA. To do so, we modify the runtime system to include all the TD-NUCA extensions to manage the RTCacheDirectory and to decide the LLC placement of task dependencies, but it never executes the TD-NUCA ISA instructions. Thus, the cache hierarchy behaves as in S-NUCA. The overhead of the runtime system extensions is 0.01% on average and it is below 0.03% in all benchmarks. The biggest source of overhead is the algorithm that maps the task dependencies to the LLC banks before tasks execute.

VI. Related Work

A. Hardware-Managed NUCA Caches

Many works propose to improve the management of NUCA caches using specific operations at the microarchitecture level.

Some works propose relocating cache blocks in NUCA LLCs. Beckmann et al. [14] show that block relocations benefit performance but increase the complexity of lookups, and they propose a two-step lookup that first checks if the block is present in the close banks and, if this search fails, the lookup is multicasted to the remaining banks. Kandemir et al. [56] present relocation algorithms to improve block placement, and Ricci et al. [78] use Bloom Filters to optimize the lookups.

Other works propose to replicate cache blocks across banks. NuRAPID [31] and CMPNuRAPID [32] introduce block replication by decoupling physical placement from logical organization. To do so, tags are re-organized in per-core private storage so that they point to their associated blocks in the NUCA banks, which requires forwards and backwards pointers from the tag entries to the cache blocks. Victim Replication [98] replicates blocks evicted from the L1 in the local LLC bank if there is spare capacity, providing a constrained level of replication at low cost. Huh et al. [50] add a configurable degree of replication and a directory to manage coherence among the replicas within the LLC. ASR [13] controls the amount of replication of shared read-only blocks with a probabilistic cost-benefit measure, and SLIP [35] introduces block reuse counters to avoid power-inefficient migrations and replications of blocks that are not reused.

Other works classify private and shared data at the microarchitecture level to allocate private blocks near their requesting cores at the granularity of pages [45], coarse-grained [67], [96] or fine-grained [40] memory regions. Zhao et al. [100] dedicate different cache ways to private and shared data.

B. Software-Managed NUCA Caches

Some works propose to manage NUCA caches by identifying shared and private data at the OS level. Cho et al. [33] change the virtual to physical page mappings to place shared and private virtual pages in particular LLC banks. Chaudhuri et al. [29] and Awasthi et al. [8] extend this scheme by allowing page migrations and formalising the cost/benefit decision making for page spreading. In these approaches, the NUCA operations require relocating and invalidating pages in the physical address space, which is very costly. To minimize overheads, R-NUCA and other works use the OS to identify shared and private pages and use this information in the hardware to perform the NUCA operations without changing the virtual to physical page mappings. Jigsaw [15] proposes a system-level algorithm that uses OS-based page classification and extensive hardware monitoring to orchestrate the placement, relocation and replication of pages in LLC banks. The decisions made by the algorithm are communicated to the hardware and stored in a structure similar to the RRTs of TD-NUCA. Jenga [91] extends Jigsaw to manage DRAM cache banks and to consider memory bandwidth in the algorithm. Jumanji [83] extends Jigsaw in the context of data centers, improving tail latency by reserving enough cache space for latency-critical applications to meet their deadlines, and reinforcing security by placing data from untrusted applications in different LLC banks. The main limitation of OS-based approaches is that they suffer from inaccurate classification of shared and private pages.

Other works propose compile-time approaches to minimize the NUCA distance. Zhang et al. [99] propose combining array tiling, computation-to-core mapping and layout customization in parallelized loop nests. Tang et al. [89] partition the computations of loop nests into subcomputations and schedule the resulting subcomputations on cores. Kislal et al. [60] refine the computation-to-core mapping strategy by also considering the distances to the memory controllers and by using an inspector-executor paradigm for irregular applications.

C. Task Dataflow Programming Models

Although TD-NUCA targets OpenMP 4.0 [72], it can be applied to any runtime-managed task dataflow programming model that specifies task dependencies either using real data addresses or some abstraction from which the runtime system can extract the addresses [7], [11], [39], [46], [55], [84], [101]. Similar properties are present in streaming programming models [3], [12], [90] and in offload programming models [1], [38], which could also benefit from TD-NUCA. However, in task programming models where the runtime system does not know the addresses of the data that is going to be accessed by the tasks [17], [77], TD-NUCA is not directly applicable.

Many works exploit the characteristics of task dataflow programming models to perform optimizations [24], [93]. The runtime system can transparently manage GPUs [7], [76], FPGA accelerators [18], [85], multi-node clusters [20], [27], [28], heterogeneous memories [4], [63], scratchpad memories [5], NUMA [81], [82] and cache coherent NUMA [21], [23] systems. Adding hardware support, the runtime system

can guide cache replacement [37], [65], cache coherence deactivation [22], cache prefetching [47], [75], cache communication mechanisms in producer-consumer task relationships [64], [66], reliability and resilience [51]–[53], value approximation [19], and DVFS to accelerate critical tasks [26].

Other works [25], [44], [61], [69], [87], [88], [95] propose implementing performance critical elements of the runtime system of task dataflow programming models in hardware instead of in software. These approaches drastically reduce the overheads of the runtime system, which can significantly improve the performance of task-based applications. More importantly, implementing the runtime system in hardware increases the viability of using sophisticated parallelization strategies that today are seldom used due to the high overheads they incur, such as speculative parallelization [2], [6], [54], [97] or fine-grained nested parallelization [86].

All together, task dataflow programming models bring the opportunities for architectural optimizations to an unprecedented level. Crucially, TD-NUCA and most of the aforementioned proposals are highly compatible, so one can envision an architectural solution that integrates many of these techniques. Such architecture would be specifically designed for high-performance computing systems, and it could capitalize on the benefits of advanced architectural techniques while being easy to program with task dataflow programming models.

VII. Conclusions

This paper proposes TD-NUCA, a runtime-driven NUCA management approach for task dataflow programming models. State-of-the-art solutions to manage NUCA caches are not effective in the presence of dynamic task schedulers, given the strong limitations of OS-based techniques to detect data sharing and reuse patterns. Instead, the runtime system of task dataflow programming models can exploit the rich information it possesses to accurately identify these properties. TD-NUCA leverages this information to decide the best allocation strategy for cache blocks in NUCA caches, guiding placement, migration and replication operations. The decisions made by the runtime system are communicated to the hardware using new ISA instructions, and the architecture only requires a simple directory per core to direct memory accesses to the appropriate LLC bank. As a result, TD-NUCA provides multiple benefits compared to S-NUCA and R-NUCA, achieving an average speedup of 1.18x over S-NUCA and R-NUCA while reducing the utilization of the NoC to 0.62x of that caused by S-NUCA.

Acknowledgements

This work has been supported by the Spanish Ministry of Science and Technology (contract PID2019-107255GB-C21) and the Generalitat de Catalunya (contract 2017-SGR-1414). M. Casas has been partially supported by the Grant RYC-2017-23269 funded by MCIN/AEI/10.13039/501100011033 and ESF 'Investing in your future'. M. Moreto has been partially supported by the Spanish Ministry of Economy, Industry and Competitiveness under Ramon y Cajal fellowship No. RYC-2016-21104.

References

[1] "The OpenACC Application Programming Interface. 2015."

[2] M. Abeydeera and D. Sanchez, "Chronos: Efficient speculative parallelism for accelerators," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020, p. 1247–1262.

[3] 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.

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

[5] L. Alvarez, M. Moretó, 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.

[6] S. Apostolakis, Z. Xu, G. Chan, S. Campanoni, and D. I. August, "Perspective: A sensible approach to speculative automatic parallelization," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020, p. 351–367.

[7] 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.

[8] 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 *International Symposium on High Performance Computer Architecture (HPCA)*, 2009, pp. 250–261.

[9] M. Azimi, N. Cherukuri, D. N. Jayasimha, A. Kumar, P. Kundu, S. Park, I. Schoinas, and A. S. Vaidya, "Integration challenges and tradeoffs for tera-scale architectures," *Intel Technology Journal*, vol. 11, no. 3, pp. 173–184, Aug. 2007.

[10] 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.

[11] 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.

[12] 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.

[13] B. M. Beckmann, M. R. Marty, and D. A. Wood, "Asr: Adaptive selective replication for cmp caches," in *International Symposium on Microarchitecture (MICRO)*, 2006, pp. 443–454.

[14] B. M. Beckmann and D. A. Wood, "Managing wire delay in large chip-multiprocessor caches," in *International Symposium on Microarchitecture (MICRO)*, 2004, pp. 319–330.

[15] N. Beckmann and D. Sanchez, "Jigsaw: Scalable software-defined caches," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013, pp. 213–224.

[16] 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.

[17] 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.

[18] J. Bosch, X. Tan, A. Filgueras, M. Vidal, M. Mateu, D. Jiménez-González, C. Álvarez, X. Martorell, E. Ayguade, and J. Labarta, "Application acceleration on fpgas with ompss@fpga," in *International Conference on Field-Programmable Technology (FPT)*, 2018, pp. 70–77.

[19] 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.

[20] 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.

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

[22] P. Caheny, L. Alvarez, M. Valero, M. Moretó, and M. Casas, "Runtime-assisted cache coherence deactivation in task parallel programs," in *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, 2018, pp. 35:1–35:12.

[23] 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.

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

[25] E. Castillo, L. Alvarez, M. Moretó, M. Casas, E. Vallejo, J. L. Bosque, R. Beivide, and M. Valero, "Architectural support for task dependence management with flexible software scheduling," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 283–295.

[26] E. Castillo, M. Moretó, M. Casas, L. Alvarez, E. Vallejo, K. Chronaki, R. 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.

[27] D. Chasapis, M. Casas, M. Moretó, M. Schulz, E. Ayguadé, J. Labarta, and M. Valero, "Runtime-guided mitigation of manufacturing variability in power-constrained multi-socket numa nodes," in *International Conference on Supercomputing (ICS)*, 2016.

[28] D. Chasapis, M. Moretó, M. Schulz, B. Rountree, M. Valero, and M. Casas, "Power efficient job scheduling by predicting the impact of processor manufacturing variability," in *International Conference on Supercomputing (ICS)*, 2019, pp. 296–307.

[29] M. Chaudhuri, "Pagenuca: Selected policies for page-grain locality management in large shared chip-multiprocessor caches," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2009, pp. 227–238.

[30] H. Che, Z. Wang, K. Zheng, and B. Liu, "Dres: Dynamic range encoding scheme for tcam coprocessors," *IEEE Trans. Comput.*, vol. 57, no. 7, pp. 902–915, 2008.

[31] Z. Chishti, M. D. Powell, and T. N. Vijaykumar, "Distance associativity for high-performance energy-efficient non-uniform cache architectures," in *International Symposium on Microarchitecture (MICRO)*, 2003, pp. 55–66.

[32] Z. Chishti, M. D. Powell, and T. N. Vijaykumar, "Optimizing replication, communication, and capacity allocation in CMPs," in *International Symposium on Computer Architecture (ISCA)*, 2005, pp. 357–368.

[33] S. Cho and L. Jin, "Managing distributed, shared l2 caches through os-level page allocation," in *International Symposium on Microarchitecture (MICRO)*, 2006, pp. 455–468.

[34] 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.

[35] S. Das, T. M. Aamodt, and W. J. Dally, "Slip: Reducing wire energy in the memory hierarchy," in *International Symposium on Computer Architecture (ISCA)*, 2015, pp. 349–361.

[36] 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.

[37] V. Dimic, M. Moretó, M. Casas, and M. Valero, "Runtime-assisted shared cache insertion policies based on re-reference intervals," in *European Conference on Parallel and Distributed Computing (Euro-Par)*, 2017, pp. 247–259.

[38] 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.

[39] 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.

[40] H. Dybdahl and P. Stenström, "An adaptive shared/private nuca cache partitioning scheme for chip multiprocessors," in *High Performance Computer Architecture (HPCA)*, 2007, pp. 2–12.

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

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

[43] 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.

[44] Y. Etsion, F. Cabarcas, A. Rico, A. Ramirez, R. M. Badia, E. Ayguadé, J. Labarta, and M. Valero, "Task superscalar: An out-of-order task pipeline," in *International Symposium on Microarchitecture (MICRO)*, 2010, pp. 89–100.

[45] B. Falsafi and D. A. Wood, "Reactive numa: A design for unifying s-coma and cc-numa," in *International Symposium on Computer Architecture (ISCA)*, 1997, pp. 229–240.

[46] 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.

[47] V. Garcia, A. Rico, C. Villavieja, P. Carpenter, N. Navarro, and A. Ramirez, "Adaptive runtime-assisted block prefetching on chip-multiprocessors," *International Journal of Parallel Programming*, pp. 1–21, 2016.

[48] 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.

[49] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Near-optimal cache block placement with reactive nonuniform cache architectures," *IEEE Micro Top Picks*, vol. 30, no. 1, pp. 20–28, 2010.

[50] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler, "A NUCA substrate for flexible CMP cache sharing," in *International Conference on Supercomputing (ICS)*, 2005, pp. 31–40.

[51] L. Jaulmes, M. Casas, M. Moretó, E. Ayguadé, J. Labarta, and M. Valero, "Exploiting asynchrony from exact forward recovery for DUE in iterative solvers," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[52] L. Jaulmes, M. Moretó, E. Ayguade, J. Labarta, M. Valero, and M. Casas, "Asynchronous and exact forward recovery for detected errors in iterative solvers," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, pp. 1961–1974, 2018.

[53] L. Jaulmes, M. Moretó, M. Valero, and M. Casas, "A vulnerability factor for ECC-protected memory," in *International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 07 2019, pp. 176–181.

[54] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez, "A scalable architecture for ordered parallelism," in *International Symposium on Microarchitecture (MICRO)*, 2015, p. 228–241.

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

[56] M. Kandemir, F. Li, M. J. Irwin, and S. W. Son, "A novel migration-based nuca design for chip multiprocessors," in *Conference on Supercomputing (SC)*, 2008, pp. 1–12.

[57] S. M. Khan, Y. Tian, and D. A. Jimenez, "Sampling dead block prediction for last-level caches," in *International Symposium on Microarchitecture (MICRO)*, 2010, p. 175–186.

[58] C. Kim, D. Burger, and S. W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002, pp. 211–222.

[59] 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.

[60] O. Kislal, J. Kotra, X. Tang, M. T. Kandemir, and M. Jung, "Enhancing computation-to-core assignment with physical location information," in *Conference on Programming Language Design and Implementation (PLDI)*, 2018, pp. 312–327.

[61] S. Kumar, C. J. Hughes, and A. Nguyen, "Carbon: Architectural support for fine-grained parallelism on chip multiprocessors," in *International Symposium on Computer Architecture (ISCA)*, 2007, pp. 162–173.

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

[63] H. Liu, Y. Chen, X. Liao, H. Jin, B. He, L. Zheng, and R. Guo, "Hardware/software cooperative caching for hybrid DRAM/NVM memory architectures," in *International Conference on Supercomputing (ICS)*, 2017, pp. 26:1–26:10.

[64] 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.

[65] 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.

[66] M. Manivannan and P. Stenstrom, "Runtime-guided cache coherence optimizations in multi-core architectures," in *International Parallel and Distributed Processing Symposium (IPDPS)*, 2014, pp. 625–636.

[67] J. Merino, V. Puente, P. Prieto, and J. A. Gregorio, "SP-NUCA: A cost effective dynamic non-uniform cache architecture," *SIGARCH Comput. Archit. News*, vol. 36, no. 2, pp. 64–71, May 2008.

[68] S. Mittal, "A survey of cache bypassing techniques," *Journal of Low Power Electronics and Applications*, vol. 6, no. 2, 2016.

[69] L. Morais, V. Silva, A. Goldman, C. Alvarez, J. Bosch, M. Frank, and G. Araujo, "Adding tightly-integrated task scheduling acceleration to a risc-v multi-core processor," in *International Symposium on Microarchitecture (MICRO)*, 2019, p. 861–872.

[70] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "CACTI 6.0: A tool to understand large caches," Tech. Rep., 2009.

[71] H. Noda, K. Inoue, M. Kuroiwa, F. Igaue, K. Yamamoto, H. Mattausch, T. Koide, A. Amo, A. Hachisuka, S. Soeda, I. Hayashi, F. Morishita, K. Dosaka, K. Arimoto, K. Fujishima, K. Anami, and T. Yoshihara, "A cost-efficient high-performance dynamic tcam with pipelined hierarchical searching and shift redundancy architecture," *IEEE Journal of Solid-State Circuits*, vol. 40, no. 1, pp. 245–253, 2005.

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

[73] K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (cam) circuits and architectures: a tutorial and survey," *IEEE Journal of Solid-State Circuits*, vol. 41, no. 3, pp. 712–727, 2006.

[74] R. Panigrahy and S. Sharma, "Reducing tcam power consumption and increasing throughput," in *Symposium on High Performance Interconnects*, 2002, pp. 107–112.

[75] 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.

[76] 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.

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

[78] R. Ricci, S. Barrus, D. Gebhardt, and R. Balasubramonian, "Leveraging bloom filters for smart search within NUCA caches," in *Workshop on Complexity-Effective Design (WCED)*, Jun. 2006.

[79] 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.

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

[81] I. Sánchez-Barrera, M. Casas, M. Moretó, E. Ayguadé, J. Labarta, and M. Valero, "Graph partitioning applied to dag scheduling to reduce numa effects," in *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2018, pp. 419–420.

[82] I. Sánchez-Barrera, M. Moretó, E. Ayguadé, J. Labarta, M. Valero, and M. Casas, "Reducing data movement on large shared memory systems by exploiting computation dependencies," in *International Conference on Supercomputing (ICS)*, 2018, pp. 207–217.

[83] B. C. Schwedock and N. Beckmann, "Jumanji: The case for dynamic nuca in the datacenter," in *International Symposium on Microarchitecture (MICRO)*, 2020, pp. 665–680.

[84] 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.

[85] L. Sommer, J. Korinth, and A. Koch, "Openmp device offloading to fpga accelerators," in *International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2017, pp. 201–205.

[86] S. Subramanian, M. C. Jeffrey, M. Abeydeera, H. R. Lee, V. A. Ying, J. Emer, and D. Sanchez, "Fractal: An execution model for fine-grain nested speculative parallelism," in *International Symposium on Computer Architecture (ISCA)*, 2017, p. 587–599.

[87] X. Tan, J. Bosch, D. Jiménez-González, C. Álvarez Martínez, E. Ayguadé, and M. 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.

[88] X. Tan, J. Bosch, M. Vidal, C. Álvarez, D. Jiménez-González, E. Ayguadé, and M. Valero, "General purpose task-dependence management hardware for task-based dataflow programming models," in *International Parallel and Distributed Processing Symposium (IPDPS)*, 2017, pp. 244–253.

[89] X. Tang, O. Kislal, M. Kandemir, and M. Karakoy, "Data movement aware computation partitioning," in *International Symposium on Microarchitecture (MICRO)*, 2017, pp. 730–744.

[90] 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.

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

[92] Z. Ullah, M. K. Jaiswal, and R. C. C. Cheung, "Z-tcam: An sram-based architecture for tcam," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, no. 2, pp. 402–406, 2015.

[93] M. Valero, M. Moretó, 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.

[94] 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.

[95] F. Yazdanpanah, C. Álvarez, D. Jiménez-González, R. M. Badia, and M. Valero, "Picos: A hardware runtime architecture support for ompss," *Future Generation Computing Systems*, vol. 53, pp. 130–139, 2015.

[96] T. Y. Yeh and G. Reinman, "Fast and fair: Data-stream quality of service," in *International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2005, pp. 237–248.

[97] V. A. Ying, M. C. Jeffrey, and D. Sanchez, "T4: Compiling sequential code for effective speculative parallelization in hardware," in *International Symposium on Computer Architecture (ISCA)*, 2020, p. 159–172.

[98] M. Zhang and K. Asanovic, "Victim replication: maximizing capacity while hiding wire delay in tiled chip multiprocessors," in *International Symposium on Computer Architecture (ISCA)*, 2005, pp. 336–345.

[99] Y. Zhang, W. Ding, M. Kandemir, J. Liu, and O. Jang, "A data layout optimization framework for nuca-based multicores," in *International Symposium on Microarchitecture (MICRO)*, 2011, pp. 489–500.

[100] L. Zhao, R. Iyer, M. Upton, and D. Newell, "Towards hybrid last level caches for chip-multiprocessors," *SIGARCH Comput. Archit. News*, vol. 36, no. 2, pp. 56–63, May 2008.

[101] 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.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

We run the benchmarks detailed in the paper using the gem5 simulator (https://github.com/gem5/gem5) and the Nanos++ runtime system from the OmpSs release 16.06 (https://pm.bsc.es/ftp/ompss/releases/). The modifications made to the publicly available gem5 simulator and the publicly available Nanos++ runtime system to support the submission are proprietary to the authors' affiliated institutions.

In gem5 we use a 16 core x86 out-of-order CPU model with the Ruby memory system model with MESI coherence. L1I and L1D sizes are 32KB each, L2 is 2MB per core and the NoC is a 4x4 mesh. The execution-driven, full-system and cycle-accurate simulations model in detail all the components of the architecture, including the cache coherence protocol and the architectural support for TD-NUCA, and they run a complete software stack including the benchmarks, the runtime system libraries, and the OS. Power consumption is evaluated with McPAT using a process technology of 22 nm, voltage of 0.6V and the default clock gating scheme. The hardware structures of TD-NUCA are modeled using CACTI 6.0.

For the evaluation we use a set of task dataflow parallel benchmarks programmed with OpenMP 4.0. Table I shows the benchmarks and the input set sizes used in the experiments. The benchmarks are run from start to completion, and the evaluation results are collected from the entire post-initialisation parallel execution phase of the benchmarks. The benchmarks are compiled with Mercurium 1.99 source-to-source compiler using gcc 4.6.4 as backend.

Table 1: Benchmarks, problem and task sizes.

2*Bench	2*Problem set	Input set size (MB)	Num tasks	AVG task size (KB)
Gauss	2D Matrix N^2 = 58982400, 2 iters.	488.04	3200	294
Histo	1500x1500 pixels, 50x50 blocks, 50 bins	478.75	1800	528
Jacobi	2D Matrix N^2 = 16777216, 5 iters.	264.34	320	4112
Kmeans	450000 pts., 90 dims, 6 clusters, 1 iter.	314.37	228	1404
KNN	512/229376 training/input pts, 8 classes	85.01	448	318
LU	2D Matrix N^2 = 9437184	73.45	1188	318
MD5	128 × 4 MB buffers	513.39	128	4096
Redblack	N^2 = 28901376, 5 iters.	223.96	320	3549

Reproduction of the artifact without container: TD-NUCA modifications to the specified baseline runtime system and specified baseline simulator are proprietary to the authors affiliated institutions.