Global Dead-Block Management for Task-Parallel Programs

MADHAVAN MANIVANNAN, MIQUEL PERICÀS, VASSILIS PAPAESTATHIOU, and
PER STENSTRÖM, Chalmers University of Technology

Task-parallel programs inefficiently utilize the cache hierarchy due to the presence of dead blocks in caches. Dead blocks may occupy cache space in multiple cache levels for a long time without providing any utility until they are finally evicted. Existing dead-block prediction schemes take decisions *locally* for each cache level and do not efficiently manage the entire cache hierarchy. This article introduces *runtime-orchestrated global dead-block management*, in which static and dynamic information about tasks available to the runtime system is used to effectively detect and manage dead blocks across the cache hierarchy. In the proposed global management schemes, static information (e.g., when tasks start/finish, and what data regions tasks produce/consume) is combined with dynamic information to detect when/where blocks become dead. When memory regions are deemed dead at some cache level(s), all the associated cache blocks are evicted from the corresponding level(s). We extend the cache controllers at both private and shared cache levels to use the aforementioned information to evict dead blocks. The article does an extensive evaluation of both inclusive and non-inclusive cache hierarchies and shows that the proposed global schemes outperform existing local dead-block management schemes.

CCS Concepts: • **Computer systems organization** → **Embedded systems**;

Additional Key Words and Phrases: Multi-core architecture, cache hierarchy, dead blocks, task parallelism, runtime system

ACM Reference format:

Madhavan Manivannan, Miquel Pericàs, Vassilis Papaefstathiou, and Per Stenström. 2018. Global Dead-Block Management for Task-Parallel Programs. *ACM Trans. Archit. Code Optim.* 15, 3, Article 33 (August 2018), 25 pages.

<https://doi.org/10.1145/3234337>

1 INTRODUCTION

Efficient utilization of caches is key to scalable performance of parallel programs on multi-core architectures. However, prior work has shown that caches are used inefficiently because of the occurrence of dead blocks (Wood et al. 1991; Lai et al. 2001; Liu et al. 2008). A number of techniques to detect and evict dead blocks have been proposed (Kharbutli and Solihin 2008; Khan et al. 2010; Wu et al. 2011). These techniques track cache accesses and evictions to train hardware predictors

This research was supported by the European Research Council (ERC) under the MECCA project (Contract No. 340328). The simulations were run on the resources provided by the Swedish National Infrastructure for Computing (SNIC) at C3SE. Authors' addresses: M. Manivannan, M. Pericàs, V. Papaefstathiou, and P. Stenström, Department of Computer Science and Engineering, Chalmers University of Technology, SE 41296, Sweden; emails: {madhavan, miquelp, vaspap, per.stenstrom}@chalmers.se.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM

1544-3566/2018/08-ART33 \$15.00

<https://doi.org/10.1145/3234337>

at run-time and mainly target single-threaded and multi-programmed workloads. Moreover, they limit dead-block detection and eviction to a single level of the cache hierarchy, e.g., the last-level cache (LLC) in a multicore system. We refer to them collectively as *local dead-block management* schemes.

Managing only a single level in isolation can lead to *inefficient* utilization of the private cache capacity in multi-level cache hierarchies. For example, in a non-inclusive hierarchy, blocks that become dead and will never be accessed again can remain in the private caches even if the LLC correctly identifies the blocks as dead and evicts them. Furthermore, *uncoordinated* decisions that do not consider the state of the block in other cache levels can render dead-block management ineffective. One example of this is the writeback of dirty blocks from the upper-level cache causing evicted dead blocks to be reinserted in the lower-level caches in non-inclusive hierarchies. Another example of such uncoordinated decisions is when hardware dead block predictors evict blocks from the LLC without considering if the blocks are live in the upper levels of an inclusive hierarchy. This necessitates a solution that can manage dead blocks in the entire cache hierarchy, efficiently utilizing private cache capacity and enabling coordinated action across multiple cache levels. We refer to this approach as *global dead-block management*.

A naive approach to facilitate global dead-block management would be to apply local schemes individually to each of the private caches and the shared LLC. Such an approach, however, is inadequate, since it cannot solve the aforementioned problems of inefficiency and uncoordination. For example, to manage blocks that become dead in multiple cache levels, each predictor must individually establish this fact and evict the block. Furthermore, because hardware-based predictors do not share information, this arrangement would also not address the problem of uncoordinated evictions.

Another challenge for hardware-based dead block predictors are parallel applications (Natarajan and Chaudhuri 2013). Load balancing techniques, such as work stealing and data sharing, introduce unpredictable branching in the control path, which disturbs the history tables used by hardware predictors for training. With incomplete history, hardware predictors are prone to make poor eviction decisions. All these drawbacks together motivate a *runtime-orchestrated global dead-block prediction* approach for managing the entire cache hierarchy in modern multi-core architectures.

In this article, we introduce global dead-block management to improve cache efficiency in a multi-level, multi-core cache hierarchy for task-based parallel programs. The key to global management is to obtain an outlook of future reuses for each block. We achieve this by exploiting readily available runtime information about future reuse of data regions from emerging task data-flow parallel programming models (e.g., OpenMP 4.0 (OpenMP Architecture Review Board 2013)). Specifically, we exploit static (compile-time) information about when tasks start/finish and what regions (data structures) tasks consume (read) and produce (write) along with dynamic information about when and where tasks will execute to predict region reuse. The proposed scheme leverages this information to establish a global view of region reuse to manage dead blocks in the cache hierarchy.

The proposed global scheme effectively manages dead blocks in task-based parallel programs. Local hardware-based schemes, however, while generic, are only effective for single-threaded and multi-programmed workloads. For this reason, we combine both approaches and propose a *hybrid dead-block management scheme*, which leverages dynamic information from both the runtime and from local dead-block management schemes. The goal of the hybrid scheme is to ensure that the runtime-based prediction scheme can function effectively even in the presence of hardware-based predictors when running task-based parallel programs. The hybrid scheme improves the scope of the dead block prediction by supporting also single-threaded and multi-programmed workloads, in addition to task-based parallel programs.

We evaluate our proposed runtime-orchestrated global dead-block management schemes and show that they outperform state-of-the-art local dead-block schemes for task-based parallel programs both in the context of inclusive and non-inclusive cache hierarchies. Overall, our techniques lead to an average 26.4% reduction in LLC misses, 6.4% reduction in execution time, and 22% reduction in energy consumption compared to the baseline using the LRU replacement policy.

In summary, the contributions of this article are fourfold:

1. We propose, for the first time, global dead-block management, supported by a cooperative HW/SW scheme that can efficiently deem blocks dead in the entire cache hierarchy by coordinating actions in different cache levels.
2. We propose to leverage the task graph administered by task-parallel runtime systems to infer future region reuse and apply it to globally manage the cache hierarchy.
3. We propose a hybrid scheme that uses both the future outlook of region accesses and the past access patterns identified by existing local hardware-based predictors to achieve accurate global dead-block management for task-based parallel programs. The same scheme can support general (sequential) workloads.
4. We evaluate our proposed global dead-block management schemes and show that they outperform state-of-the-art local dead-block management schemes in both inclusive and non-inclusive hierarchies.

The rest of the article is organized as follows. Section 2 presents an overview of the targeted programming model and introduces concepts specific to our proposed scheme. Section 3 shows how runtime systems can establish when and where cache blocks become dead and what architectural support is needed to manage dead blocks. Section 4 proposes a hybrid scheme that synergistically combines the runtime-orchestrated global scheme with hardware-based local dead-block schemes. Section 5 presents the methodology we use for our evaluation and Section 6 discusses the results. We then discuss related work in Section 7 followed by concluding remarks in Section 8.

2 BACKGROUND

We introduce the basics of the underlying task-based programming model and runtime system support in Section 2.1. We then discuss basic concepts and terminology that are important for our runtime-orchestrated schemes in Section 2.2.

2.1 Programming Model Overview

We focus on dynamically scheduled parallel programs written in modern task data-flow programming models such as OpenMP 4.0 (OpenMP Architecture Review Board 2013) or OmpSs (Duran et al. 2011), and run on multi-core architectures with per-core private caches and a shared last-level cache (LLC). In these programming models, programmers expose parallelism by identifying sections of code known as *tasks*. Inter-task dependencies are specified by annotating each task with *in*, *out*, or *inout* clauses. *Out* from task A and *in* to task B for region X specifies a true dependency between tasks A and B with respect to region X, whereas *inout* for region X specifies that a task both reads and writes to region X.

These annotations specify the memory *regions* accessed by a task and constitute its working set.¹ A region can refer to single/multiple (contiguous) addresses or sub-ranges of a multi-dimensional array.

The runtime system uses information about the regions accessed by each task to dynamically infer task dependencies and schedule them for execution. By introducing point-to-point

¹Note that the term *Region* is used in OmpSs. In OpenMP 4.0 the equivalent concept is called *array-section*.

Fig. 1. (a) Sample task graph. (b) Dependency resolution overview.

dependencies, task data-flow models allow relaxing the strict synchronization of traditional models, such as thread barriers (OpenMP Architecture Review Board 2005) or nested fork/join synchronization (Frigo et al. 1998). Task data-flow models have been studied comprehensively in the past. This research has led to optimized software implementations with low overheads (Vandierendonck et al. 2011; Perez et al. 2008). In the following, we assume the OmpSs implementation of the task data-flow model (Perez et al. 2008).

We illustrate how annotations are specified and how the runtime system utilizes them to infer inter-task dependencies in Figure 1(a). Consider an application with three tasks T1, T2, and T3 that read and update regions R1, R2, and R3, respectively. In addition, tasks T2 and T3 read the updated version of R1. To specify this dependency, a programmer would annotate T1 with `inout(R1)`, T2 with `in(R1) inout(R2)`, and T3 with `in(R1) inout(R3)`.

In the task data-flow model, a master thread independently creates tasks in program order and establishes inter-task dependencies based on the annotations. Tasks without any unresolved dependencies are submitted to a main ready-queue to be executed by the worker threads in an out-of-order manner. Tasks with unresolved dependencies are submitted to the ready queues only after their dependencies are satisfied. Each worker thread also has a local ready queue. To exploit locality, these local queues hold ready tasks released by predecessor tasks that executed on the same worker. Worker threads look for work in their local queues and perform work stealing in case their local queue is empty. The combination of depth-first task creation and work stealing generates cache-efficient schedules similar to the PDF (*parallel depth first*) scheduler (Blelloch et al. 1999; Chen et al. 2007). In this example, the runtime system dispatches T1 immediately, but T2 and T3 have to wait until T1 finishes execution. T2 and T3 are then dispatched for execution to the queue of the worker thread that executed T1.

2.2 Concepts and Terminology

Tasks in the ready queue and those with unresolved dependencies provide an *outlook* of future tasks in the data-flow graph that we refer to as the *look-ahead window*. We consider a region to have been *accessed* when the task with an annotation to that region finishes execution. We define a region access to be a *region miss* in a particular level of the cache hierarchy if all of the cache blocks that span the region result in a cache miss and have to be fetched from the next level of the cache/memory. We classify all accesses that are not region misses to be *region hits*. A region is considered *dead* in a specific level of the cache hierarchy if the next access to that region will result in a region miss in that level and all blocks spanning the region need to be fetched from the subsequent level(s).

3 GLOBAL DEAD REGION MANAGEMENT

The runtime system manages dead blocks at the granularity of regions instead of individual cache lines. The management of dead regions comprises two steps and can be summarized as follows: After a task completes, the runtime system predicts the next reuse for all regions accessed by this task based on the look-ahead window. On the basis of predicted reuse in the look-ahead window, the runtime system supplies eviction hints to the private cache and/or the shared cache for regions that are deemed dead. These hints instruct the replacement policy to demote blocks belonging to the dead region in the recency stack and thereby prioritize them for eviction.

Section 3.1 presents our proposed reuse categories for a region, discusses challenges with accurate reuse estimation, and presents our heuristic-based approach to region reuse prediction. Section 3.2 establishes how the inter-task dependency analysis framework in the runtime system can be leveraged to implement the heuristics. In addition, it also presents how hints are generated by taking into account the predicted reuse and the characteristics of the cache hierarchy. Finally, Section 3.3 presents our proposed architectural support for runtime-orchestrated dead block management and our proposed strategies to evict dead blocks from the entire cache hierarchy based on eviction hints.

3.1 Reuse Categorization and Prediction

In the following, we refer to the L2 cache as *private cache* and to the L3 cache as *shared cache* (henceforth, LLC). We do not predict dead regions in the L1 cache because of the limited opportunity provided by its small capacity (32–64 KB) and the short lifetime of cache-blocks in the L1 cache. For the assumed cache hierarchy, we define three distinct region reuse categories:

- **Private Reuse:** the next access to this region will experience a region hit in the private cache and the blocks spanning the region do not have to be fetched from the shared cache.
- **Shared Reuse:** the next access to this region from the same core will experience a region miss in the private cache and a region hit in the shared cache and the blocks spanning the region do not have to be fetched from memory.
- **No Reuse:** the next access to this region will experience a region miss in both the private and the shared cache. Irrespective of the core that accesses the region, it will result in an off-chip access.

Assuming oracular knowledge of all the future tasks, including: (i) where they are executed, (ii) the partial (per-core) and total task order, (iii) the region(s) accessed by each task, and (iv) its (their) size(s), the reuse category for the next access to the region can be determined using coarse-grain reuse-distance estimation techniques proposed for off-line analysis (Pericàs et al. 2014). In reality, however, the look-ahead window does not provide perfect knowledge of all future tasks nor does it have precise information about the schedule, since tasks are dynamically selected for execution. As a consequence, our scheme relies on simple and, as will be shown, effective heuristics that leverage inter-task dependencies to predict an appropriate reuse category for the next access to a region. It does not take into account the size of regions. In particular, the heuristics are based on the observation that a region that will be accessed by a task with unresolved dependencies (not ready) is not likely to be reused soon, whereas a region that is to be accessed by a task in the ready queue(s) (satisfied dependencies) is more likely to be reused soon. We describe the operation of these heuristics below:

Private Reuse: Two conditions determine if the predicted reuse for the next access to a region falls in this category. The first condition is that a task has a dependency with a currently running task. The rationale for this is that schedulers typically improve data locality by scheduling a consumer

task on the same core as the associated producer task. Consequently, if a task releases one or multiple tasks for execution, the region responsible for this dependency is predicted to have private reuse. The second condition is that a task is ready. The rationale for this is that tasks in ready queue(s) usually execute before other tasks that have unresolved dependencies. If the examined region will be accessed by any task that fulfills one of these two conditions, then the region is predicted to have *Private Reuse*.

Shared Reuse: If all tasks that access a region have unresolved dependencies in the look-ahead window, then the next access to a region is predicted to have *Shared Reuse*. This prediction is based on the observation that tasks with unresolved dependencies tend to take a longer time before they are submitted to the ready queue. Since the private cache is smaller than the shared cache and the likelihood that the next task with unresolved dependency will execute on the same core is low, such reuses are more likely to be captured by the shared cache.

No Reuse: We predict the next access to a region to be beyond the capacity of the on-chip cache if the runtime system does not observe any reuse for the region in the look-ahead window. The look-ahead window comprises hundreds of tasks and provides a view of future region accesses. The combined working set of these future tasks usually exceeds the capacity of the cache hierarchy. Therefore, it serves as a good estimate to identify regions whose next reuse cannot be captured on-chip.

We analyze the effectiveness of our proposed heuristics in predicting reuse categories in Section 6.3.

3.2 Runtime System Support

We implement the proposed heuristics for reuse prediction inside the task dependency resolution framework of the baseline runtime system. We generate appropriate hints on the basis of the predicted reuse, region access type (read/write), and the characteristics of the cache hierarchy. Section 3.2.1 describes the mechanisms we leverage in the baseline runtime system. Sections 3.2.2 and 3.2.3 then show how the heuristics are integrated and what hints are communicated to the architecture.

3.2.1 Dependency Resolution Framework. We use the sample task graph depicted in Figure 1(a) to illustrate how the runtime performs inter-task dependency resolution. These are standard mechanisms available in task-based runtimes that support dependency annotations. The master thread first creates T1 and inserts the accessed regions into the dependency table. Each entry in the dependency table tracks a region and maintains a list of reader tasks (*readers list*), the number of pending writer tasks (*numWriter*), and the last-writer task (*lastWriter*) that accesses the region. R1's *numWriter* is set to one and *lastWriter* is set to T1. Initially, the runtime does not detect any dependency with R1 and thus T1 is submitted to the ready queue for execution. The master thread then detects a dependency for tasks T2 and T3 with R1. A reference counting mechanism ensures that dependent tasks (T2, T3) are submitted for execution only after their input dependencies are resolved (T1 finishes execution). The reference counting mechanism sets the *predecessor count* for tasks T2 and T3 to one and adds these two tasks to the *successor list* of T1 as shown in Figure 1(b). When T1 finishes execution, the *numWriter* is decremented and the *lastWriter* for R1 is cleared as there are no writes pending to this region. The *predecessor count* of each task in the *successor list* of T1 is also decremented. The tasks in the *successor list* are submitted to the current worker's ready queue when their *predecessor count* becomes zero. Once they execute the *lastWriter* for R2 and R3 is cleared and their *numWriter* is decremented. Note that in schemes like these, where parallelism is achieved via work stealing, successor tasks are by default executed by the same thread that executed the predecessor. This enables data reuse via private caches. Data communication

across the lower levels of the cache hierarchy occurs only in the event that an idle thread “steals” a successor task for execution. Note, however, that our heuristics do not take task stealing into account. This is subject to future research.

3.2.2 Implementation of Heuristics. The runtime predicts the reuse category for the next access to a region by leveraging information about the current access type (read/write), the readers list, number of writers, the last writer, the predecessor count, and the successor list. All this information is readily available from the dependency resolution framework and is updated in the background between task executions to manage inter-task dependencies. We describe below our runtime methods to predict each reuse category.

Private Reuse: A region access by a dependent consumer task is detected when the predecessor count of tasks in the successor list is decremented to zero. In the example, after T1 accesses R1, we predict R1 to have Private Reuse, because T2 and T3 are successors of T1 and depend on R1. In addition to the first condition, any region accessed by a task in the ready queue is also predicted as Private Reuse. To identify such region reuses, we check the readers list for tasks with predecessor count zero, which means that they do not have any unresolved dependencies and are ready to execute. Note that R1 will be correctly predicted even if it is the first time it is accessed by T2 and T3 as prediction happens when T1 finishes execution.

Shared Reuse: Regions accessed by tasks that have unresolved dependencies, i.e., when the predecessor count is greater than zero, are tagged as Shared Reuse. We distinguish between two cases depending on the type of access, read or write. For reads, we check the predecessor count of the tasks in the readers list, and for writes, we check the count in the successors list. In both cases, if the predecessor count for the task is greater than zero, then the region is accessed by tasks with unresolved dependencies and we predict Shared Reuse.

No Reuse: The case of a region that is not reused from tasks in the look-ahead window can be detected by simply analyzing the size of the readers list and the state of the last-writer task. If the readers list and the last-writer field are empty, then the runtime predicts that this region will not be reused.

3.2.3 Runtime Hint Generation. The runtime system, using hints, conveys information to the cache hierarchy (to be discussed in Section 3.3) about the region (base address and size) and the specific cache level(s) in which the region is dead and the action that needs to be taken by the cache hierarchy. The hints are decided based on the predicted reuse category and the current region access type (read/write). Where to send the hints, if necessary, and what actions to take is summarized as follows:

- **Private Reuse:** No eviction hints are sent to the private cache and eviction decisions are delegated to the replacement policy.
- **Shared Reuse:** An eviction hint is sent to the private cache to evict all blocks belonging to this region.
- **No Reuse:** Eviction hints are sent to both the private and the shared cache to evict all blocks belonging to this region.

If *No Reuse* is predicted for a region that has not been written, then the region is evicted from all cache levels. However, if *No Reuse* is predicted for a region that has been written, the runtime instructs the cache hierarchy to perform eager write-back of blocks that span the region from the upper cache level(s) before it is evicted from the lower cache level. This is done to ensure that dirty blocks (that span the dead regions) residing in the upper cache level(s) are written back before the blocks are evicted from the lower cache level. The need for performing eager write-backs is communicated along with the eviction hint.

Fig. 2. Architectural support for global dead-block management. Shaded boxes indicate additional support required for global dead-block management.

Finally, if the underlying cache hierarchy is non-inclusive, it permits blocks to reside in the upper levels (L1, L2) without enforcing residence in the lower level (LLC). To exploit this behavior the runtime system sends an eviction hint to the shared cache while retaining the region in the private cache. This is done when the access is a write and Private Reuse is predicted, since this helps exploit cache capacity by keeping a single copy of the block in the hierarchy and capturing reuse in the private cache without introducing the risk of silent eviction from the cache hierarchy, which may cause the block to be re-fetched from memory, if needed. If the underlying cache hierarchy is inclusive, then this hint is disabled, as this would trigger back-invalidation from the upper cache level(s).

3.3 Architectural Support

Figure 2 provides an overview of the architectural support needed for communicating information about regions (base address and size) and their predicted reuse category and for evicting cache blocks that span a region from different levels of the cache hierarchy. A region eviction instruction (*rdbhint*) is introduced in the ISA to convey the base virtual address of the region, the size of the region, and the reuse hint from the runtime system. In addition, each private cache (L2) controller is equipped with a Dead Region Tracker (DRT) and a Dead Region Manager (DRM) and the shared cache (LLC) controller is equipped with a single DRM. DRT and DRM together are responsible for evicting dead blocks from the cache hierarchy based on the supplied hints and their operation is described in detail below.

Upon predicting reuse, a single *rdbhint* instruction is used, if needed, to convey information about the region base address, region size, and the eviction hint to the Dead Region Tracker (DRT) in the corresponding private cache. The instruction itself has non-blocking semantics and a short execution latency, i.e., the cycles it takes to notify the L2 cache controller. The DRT performs virtual to physical address translation for the addresses spanning a region. To keep the address translation overheads minimal and avoid interfering with the access-critical L1 TLBs, we employ two optimizations: (i) we use the second-level (L2) TLB for address translations and thus avoid pressure in the first-level TLBs and (ii) we access the L2 TLB only once for each page in a region. However, the *rdbhint* instruction is ignored on a L2 TLB miss, because L2 TLB misses happens rarely and ignoring them does not cause any correctness issues. The DRT then sends the physical address range on a per-page basis to the respective DRM located at either the private (L2) and/or the shared cache (LLC) controllers based on the hint.

In case the hint indicates an eviction without eager write-back, the Dead Region Manager (DRM) at the specific cache level receives this request, probes the cache blocks specified by the address range within a physical page and proceeds to demote each block to the LRU position in the recency stack (or lowest priority in the case of pseudo-LRU). The rationale behind performing demotion instead of eviction is that demotion offers a possibility for the line to be found in the cache even after the dead block predictor issues a wrong prediction. Moreover, current cache invalidation instructions, like Intel's CLFLUSH, introduce considerable overheads for each cache block and are usually preferred for ensuring correctness rather than for optimizing performance.

The DRM leverages the observation that lower level caches experience higher levels of idleness owing to filtering of requests by upper level caches and consequently operates only during idle cycles to avoid interfering with regular load/store requests issued from the cores. As a consequence the latency overheads associated with demoting individual blocks that span a region is outside of the critical path of demand (cache) accesses. Note, however, that demotions increase the number of tag accesses and the associated energy overheads. Once all the blocks in the address range within a physical page have been demoted, the DRM informs the DRT in the corresponding core. The DRT then sends the physical address range for the next page that is part of the region, if available.

If the hint indicates an eviction with eager write-back, then the previous sequence of actions is preceded by the sending of a downgrade coherence request to the upper level cache(s), including L1, to ensure that dirty data are eagerly written-back before the block is evicted from the current level.

4 HYBRID DEAD BLOCK MANAGEMENT

The upside of local schemes is their proven effectiveness for single-threaded, multi-programmed and statically-scheduled (e.g., parallel loop-based) multi-threaded workloads (Khan et al. 2010; Kharbutli and Solihin 2008; Liu et al. 2008; Wu et al. 2011). Apart from taking a global approach to dead-block management, our proposed global approach differs from local schemes in three essential ways. First, the global approach works on the granularity of regions whereas local schemes work on the granularity of single blocks. Second, the global approach does delay the eviction of blocks belonging to a region until a task finishes execution. In contrast, local schemes evict blocks immediately. For this reason, the global approach *lazily* deems a block dead after a task finishes execution. By contrast, existing local hardware-based approaches *eagerly* deem a block dead at the granularity of individual memory accesses. Finally, while our proposed global approach bases its predictions on future outlook, existing schemes base their prediction on past outcomes.

To see whether global schemes can benefit from local schemes, we propose a hybrid scheme using the lazy approach of the global scheme as a baseline. The hybrid scheme leverages local schemes by taking past access patterns into account when the global scheme generates hints to the cache hierarchy. As such, instead of directly deeming blocks dead in the cache hierarchy, local schemes act as advisers to the global scheme.

4.1 Concept

The hybrid scheme is coordinated by the global scheme and makes informed decisions using the local schemes at each level of the cache hierarchy. When the local scheme predicts a block to be dead, and this block is part of an annotated region, the block is tracked instead of being eagerly evicted. The decision whether to evict the block is taken lazily after receiving hints from the run-time system about the blocks in the region. We study two options to combine these decisions. First, the *aggressive hybrid scheme* deems a block dead when any (OR) of the two schemes predicts the block to be dead. In contrast, the *conservative hybrid scheme* predicts a block to be dead only when both (AND) the local and the global schemes predict the block to be dead. Further combinations

Fig. 3. (a) Architectural support for hybrid management. (b) Overview of hybrid dead-block prediction strategies.

are possible whereby the runtime, based on accesses to regions, determines when one approach would be more efficient than the other and picks a suitable candidate scheme for that region. We defer exploration of such additional schemes to future work.

4.2 Architectural Support

Since the hybrid scheme builds on the local dead-block prediction scheme and the proposed runtime-orchestrated global scheme it assumes the existence of architectural support for both these schemes. There are, however, additional requirements for the hybrid scheme: (i) disable eager eviction by the local scheme, (ii) enable tracking of blocks predicted dead by the local scheme, and (iii) lazy decision-making for blocks that are part of annotated regions. Figure 3(a) provides an overview of how the proposed hybrid approach predicts a dead block.

To disable eager eviction of blocks that are part of an annotated region we exploit the observation that local dead-block predictors deem a block dead upon access. To delay the classification of region accesses, every access to the private cache and the shared cache is tagged with a *region bit* in the request that identifies if the address is part of the annotated region accessed by the task. To identify such addresses, we add this *region bit* in each Data TLB entry (not shown in the figure) and a mechanism to set/reset the bit from the runtime system. To set/reset the region bit in the Data TLB we utilize a variant of the *rdbhint* instruction that, when executed, instructs the TLB to set/reset the region bit in the entry corresponding to the page indicated by the address. Since the region-bit is not part of the page table and only exists in the Data TLB, no changes to the OS kernel are needed. Moreover, the region bit is only a hint to the hardware predictors. There are no correctness issues if the TLB entry is evicted at some point in time. This bit is set by the runtime system at the beginning of the task for the pages that span the annotated regions of the task. At the completion of the task, these bits are cleared from the TLB entries.

Once lazy eviction is enabled, the next step is to track the blocks that are deemed dead by the local dead-block predictors in the cache hierarchy. To enable tracking, we propose adding one *dead bit* to every tag entry. This bit is set when a local dead-block predictor predicts a block (that is part of the annotated region—accessed with the region bit) to be dead upon access and is reset when the block is moved to the bottom of the recency stack as shown in Figure 3(a). This region bit helps ensure that blocks that belong to non-annotated regions are evicted eagerly.

Finally, when the global scheme provides a hint (P2 in the figure) to manage dead regions using the architectural support discussed in Section 3.3 the local prediction (P1 in the figure) is also taken into account to determine if the block is dead and if eviction needs to be triggered. As shown in Figure 3(b) this decision is taken based on either the conservative (AND) or the aggressive (OR) strategy.

Table 1. Baseline Architecture Parameters

Core	8 cores, single-issue, x86 ISA
L1 Cache	Private Split, 32KB L1 I & D 8-way, LRU, single cycle tag and data access and virtually indexed
L2 Cache (inclusive baseline)	Unified, 256KB L2 8-way, LRU, 4 cycles tag and 8 cycles data access and physically indexed
L2 Cache (non-inclusive baseline)	Unified, 512KB L2 8-way, LRU, 4 cycles tag and 10 cycles data access and physically indexed
L3 Cache	Shared, 8MB, 16-way, LRU, 10 cycles tag and 40 cycles data access and physically indexed
Coherence	MESI, write invalidate, 64-byte blocks
Main Memory	300 cycles
CDBP (Kharbutli and Solihin 2008)	64K history table
SDBP (Khan et al. 2010)	3 tables, 8K history each
SHiP (Wu et al. 2011)	3-bit SRRIP, 16K SHCT, SHiP-PC
SHiP++ (Young et al. 2017)	3-bit SRRIP, 16K SHCT, SHiP-PC
Hawkeye (Jain and Lin 2016)	3-bit SRRIP, 16K SHCT with 5-bit counter, 16K 4-byte sampling entries, 128 occ. vector with 128 entries each
NANOS	dbf scheduler; plain dep. plugin

5 EXPERIMENTAL METHODOLOGY

We model a multi-core architecture consisting of eight cores, each comprising private L1 and L2 caches and a shared LLC, using the Sniper simulator (Carlson et al. 2011). The architectural parameters we use for our baseline systems are presented in Table 1. The impact of our proposal is evaluated assuming two baseline systems: an inclusive and a non-inclusive cache hierarchy. Both modeled baseline systems insert a block at all the levels of the cache hierarchy. Unlike the inclusive hierarchy, the non-inclusive hierarchy does not trigger back-invalidation at the upper levels upon eviction from the lower levels. Moreover, in the baseline architecture, we assume that all blocks are silently evicted if clean and that evictions trigger a writeback (and insertion in the next level of the cache/write to DRAM, if needed) only if the block is dirty. The inclusive L2 caches are chosen smaller than the non-inclusive L2 caches. The rationale is that all data in inclusive L2 caches are included in L3. Hence, larger L2 caches waste L3 cache capacity. Our technique targets performance improvement mainly for memory-intensive workloads. The performance of such workloads is dominated by memory access latencies, which cannot be effectively hidden by out-of-order execution. Hence, the impact of our techniques is not very sensitive to the core microarchitecture. Based on this observation, we decided to use one-IPC cores (single-issue, in-order cores) as the microarchitectural core model. Moreover, simplifying the core model allows us to run dynamically-scheduled multithreaded workloads to completion, which is necessary to obtain representative results.

The applications we evaluate are taken from the Barcelona Application Repository and are implemented using the OmpSs programming model, which is an academic precursor to OpenMP 4.0. The tasks in the application are annotated with *in*, *out* and *inout* annotations that allow the runtime system to orchestrate parallelism. As in the OpenMP 4.0 specification, we assume that tasks must refer to either identical or disjoint regions and do not support partial overlaps between regions. The application parameters are presented in Table 2. *gauss*, *jacobi*, and *redblack* are comparatively memory-intensive, with MPKI rates between 1 and 2. Task working sets, identified by the *in*, *out*

Table 2. Application Parameters: Input, Block Size (BS), Iteration Count (Iter), Task Count, and Per-task Averages for Instructions and Cycles

Application	Input	BS	Iter	# Tasks	# Insts.	# Cyc.
cholesky	3,072×3,072	32	—	152,096	271K	340K
gauss	1,536×1,536	32	32	73,728	76K	127K
jacobi	1,536×1,536	32	32	73,728	81K	134K
matmul	1,536×1,536	32	—	110,592	277K	336K
redblack	1,536×1,536	32	32	73,728	91K	156K
sparselu	3,072×3,072	32	—	78,448	265K	301K

and *inout* region annotations, are sized to fit the L1 cache. For these applications, regions are typically 1D or 2D arrays spanning two memory pages (8KB). The total working set of each application is much larger than the size of the LLC.

We skip the initialization phases and simulate the rest of the application until completion. We use the NANOS++ runtime system configured with per-thread work queues as the baseline for our evaluation (*dbf* scheduler). We measure the sum of execution time of tasks as a measure of performance to emphasize on the time spent executing tasks in the application and to isolate the task management overheads. Likewise, we only simulate memory accesses from tasks and do not consider memory accesses from the runtime system. The rationale is that the impact of the runtime system on performance is marginal in comparison with task execution.

Our scheme is first compared against RADAR, a recently proposed local scheme that leverages information available to the runtime system in task-parallel programs to manage dead blocks in the LLC only (Manivannan et al. 2016). We also compare our scheme against three hardware-based local dead-block management schemes that do not coordinate writebacks from upper cache levels: The Count-Based Dead-Block Predictor (CDBP) (Kharbutli and Solihin 2008), the Sampling Dead-Block Predictor (SDBP) (Khan et al. 2010), and the Signature-based Hit Predictor (SHiP) (Wu et al. 2011) and two recent hardware-based dead-block management schemes that additionally coordinate writebacks from upper cache levels: SHiP++ (Young et al. 2017) and Hawkeye (Jain and Lin 2016). We evaluate two different configurations for each of the hardware-based predictors: (i) the predictors are used only in the LLC (L3), and (ii) the predictors are used in each of the private caches and in the shared cache (L2+L3). The different configurations for SHiP, SHiP++ and Hawkeye that we evaluate use 2-bit SRRIP as the replacement for the private caches. Finally, we evaluate the proposed hybrid scheme by combining GLOBAL decisions with information provided by CDBP-L2+L3. We evaluate both the aggressive approach (Hybrid-A) and the conservative approach (Hybrid-C).

6 EVALUATION

The goal of our evaluation is threefold. The first goal is to show that our proposed global scheme can manage dead blocks more effectively than state-of-the-art techniques that manage dead blocks in the LLC alone. The second goal is to demonstrate that using multiple local dead-block predictors in an uncoordinated manner to manage dead blocks is considerably less effective in comparison with the global approach to manage dead blocks in the cache hierarchy. The third goal is to assess whether the proposed hybrid scheme that leverages prediction decisions made by local hardware-based dead-block predictors to assist global schemes can be more effective at reducing off-chip accesses than the local approach alone by exploiting synergies between the two approaches. To this end, in Section 6.1, we evaluate the impact of our proposed global and hybrid scheme on misses and execution time and compare it against state-of-the-art local schemes both for the shared and

Fig. 4. LLC misses for different schemes normalized to the non-inclusive baseline.

Fig. 5. L2 misses for different schemes normalized to the non-inclusive baseline.

the private cache. In Section 6.2, we evaluate the influence of our proposal on execution time. We then analyze the effectiveness of our proposed heuristic-based approach by looking at the coverage and accuracy of its predictions in Section 6.3 and evaluate the impact of the proposed scheme in the presence of prefetching in Section 6.4. Finally, in Section 6.5, we present the impact of the proposed scheme on energy consumption.

6.1 Impact on Cache Misses

We first evaluate our proposal with a non-inclusive cache hierarchy (a common baseline in many dead-block prediction studies) in Section 6.1.1 and then with an inclusive cache hierarchy in Section 6.1.2. For each case, we first study the impact of our proposed global schemes (referred to as GLOBAL) on the number of misses in the shared and the private cache.

6.1.1 Non-inclusive Baseline. Figure 4 shows LLC misses for all the schemes normalized to the non-inclusive baseline that manages blocks using LRU. Figure 5 shows L2 misses for all the schemes normalized to the non-inclusive baseline.

Runtime Schemes: GLOBAL vs. RADAR. LLC Misses: The average reduction in LLC misses we obtain with GLOBAL by managing dead blocks in the cache hierarchy is 26.4%. In contrast, the average reduction obtained by RADAR that only manages the LLC is 10.1%. This result demonstrates that managing dead blocks globally provides greater reduction in LLC misses

Fig. 6. On-chip hits for different schemes normalized to the non-inclusive baseline.

compared to managing dead blocks in the LLC using existing local approaches in isolation as in RADAR.

The reduction in misses can be attributed to two reasons. First, global management of dead blocks results in an increased number of hits observed in the private caches, thereby reducing the total number of LLC misses. Second, local schemes, such as RADAR, that only evict dead blocks from the LLC in a non-inclusive cache, suffer when the block is dirty in the upper levels of the cache hierarchy and is later written back (after the block has been deemed dead and evicted from the LLC). Such write-backs can reduce the utility of doing dead-block prediction, a problem that RADAR suffers from. To attribute the gains, we introduce a new design point labeled GLOBAL-EWB. GLOBAL-EWB is a reduced version of GLOBAL that, unlike GLOBAL, only responds to blocks that are dead in the entire cache hierarchy and implements eager writeback assisted by the runtime for such blocks. The results show that GLOBAL-EWB decreases the number of LLC misses by 12% compared to RADAR. Although both schemes operate only on blocks predicted to have no future reuses in the program, RADAR is fundamentally limited by its local view. To achieve the coordinated management of dead blocks required by the eager writeback scheme, a global view is necessary. The support for global management is implemented in GLOBAL via runtime extensions to differentiate dead reads and writes (Section 3.2.3), an ISA supporting eager eviction hints (Section 3.3), and dead region managers (DRM) at all cache levels (Section 3.3). The multi-level capacity management enabled by GLOBAL provides an additional 4.3% reduction in LLC misses on top of GLOBAL-EWB through utilizing private caches efficiently. Overall, the reduction in number of LLC misses for GLOBAL over RADAR is 16.3%, on average, with *sparselu*, *gauss*, *jacobi*, and *redblack* experiencing most reduction and *cholesky* and *matmul* experiencing a more modest reduction.

L2 Misses: As expected, RADAR does not provide any reduction in L2 misses over the baseline, because it only manages dead blocks in the LLC. Despite the huge reduction in LLC misses, the average reduction in L2 misses with GLOBAL is modest at 1.5%. Intuitively, L2 misses should improve, however, dynamic task scheduling by the runtime has some non-trivial effects. For instance, if a task that accesses a region gets scheduled on a different core in a subsequent iteration, the benefit of retaining cache blocks would not manifest directly as a reduction in L2 misses. This trend can also be observed when using hardware-based dead-block predictors for managing dead blocks in the private caches as shown by CDBP-L2+L3, SDBP-L2+L3, and SHIP-L2+L3.

To better understand the benefit of managing dead blocks in the private cache, we study the number of hits in the on-chip private and LLC in Figure 6. The L2 hits consider both private cache

hits and remote private cache hits. We consider an access to be a remote private cache hit if the line (both tag and data) does not exist in the shared LLC but only resides in the remote private cache. For *gauss*, *jacobi*, *redblack*, and *sparselu*, we can see a marked increase in the number of L2 hits for GLOBAL over RADAR. This shows how global management effectively uses remote private caches to better utilize the available on-chip cache capacity. These are applications for which GLOBAL improves the number of LLC misses substantially over RADAR as discussed previously. In addition to the increase in the L2 hits, we also see an increase in the number of LLC hits for these applications over RADAR. This increase in LLC hits is attributed to better handling of write-backs in GLOBAL that addresses the inefficiencies of RADAR.

GLOBAL vs. Hardware-based Local Schemes. LLC Misses: Among the different hardware-based dead-block predictors that we evaluate, there is no clear winner across all applications. SHiP++-L2+L3 on average reduces LLC misses on average by 16.3% while Hawkeye-L2+L3, which also employs a similar strategy to handle writebacks effectively, manages to reduce LLC misses on average by 19.4%. In comparison, GLOBAL provides an average LLC miss reduction of 26.4%. The difference can be attributed to two reasons. GLOBAL can identify blocks that are deemed to be globally dead and evict them from the entire hierarchy, whereas local-only hardware predictors, which even coordinate with private caches, rely upon dead blocks being identified individually at each cache level before they can be evicted from there. Furthermore, the heuristics that GLOBAL and hardware-based predictors use to identify dead blocks are radically different. The former predicts dead blocks at a region granularity by utilizing information about the task-graph available to the runtime system while the latter utilizes information about cache access patterns at cache block granularity.

L2 Misses: Existing predictors experience almost no reduction for *cholesky*, *gauss*, *jacobi*, and *redblack*. In the case of *sparselu*, existing predictors end up increasing the number of L2 misses over the baseline. The notable exception is *matmul*, where CDBP-L2+L3 provides a considerable reduction in L2 misses. CDBP performs well for *matmul* as it is able to capture a part of a row of an input matrix in the private cache without thrashing the private cache. When the tasks finish accessing the first row and move to accessing subsequent rows, part of the subsequent rows are captured. This behavior is neither addressed by SDBP nor SHiP, since they only use PC bits to index into the predictor table thereby resulting in conflicts at the same row. Since GLOBAL predicts region-reuse based on information about unresolved dependencies, it cannot classify among reuse patterns while accessing different rows of the same matrix. With the exception of *sparselu*, we can always observe an increase in the L2 hits when managing dead blocks in both the private and the shared cache as shown in Figure 6.

Hybrid Schemes. LLC Misses: Hybrid-A is beneficial for *cholesky* and *matmul*, where GLOBAL is slightly behind the hardware-based dead-block predictors. By combining the predictions made by both the hardware-based and the GLOBAL scheme, we can obtain more reduction in LLC misses than GLOBAL alone. For *matmul* the aggressive scheme exploits synergistic behavior and provides more reduction in LLC misses than each scheme alone. For these two applications, our proposed reuse heuristics offer room for improvement in terms of prediction coverage as shown in Figure 11. Accommodating information from hardware-based predictors helps bridge the coverage gap, thereby resulting in improvement. For applications in which GLOBAL performs best the Hybrid-A scheme does not provide any additional benefit over GLOBAL. The results also demonstrate that Hybrid-A outperforms hardware-based schemes considerably. Hybrid-C, on average, underperforms in comparison to the worst performing scheme among the hardware-based, GLOBAL and Hybrid-A. This demonstrates that dead blocks predicted by different schemes exhibit a significant degree of non-overlap.

Fig. 7. LLC misses for different schemes normalized to the inclusive baseline.

L2 Misses: Hybrid-A shows a similar trend in L2 misses as GLOBAL with the exception of *sparselu*. CDBP-L2+L3 introduces more misses than the baseline and this impacts the hybrid scheme. In contrast to LLC misses, Hybrid-C performs significantly better than Hybrid-A scheme for *matmul*. This is because Hybrid-C demotes only when both GLOBAL and local schemes agree and, like CDBP-L2+L3, it targets dead blocks when tasks access different rows of the input matrix in a thrashing manner, as discussed previously.

Summary of Results for Non-inclusive Caches. Overall, we have shown that GLOBAL outperforms local schemes. In addition, by combining GLOBAL with local schemes, we show that the global scheme can synergistically leverage information from local schemes. Concretely, we have shown that compared to RADAR, GLOBAL provides an additional 16.3% reduction in LLC misses. Compared to the best performing hardware-based, local dead-block prediction scheme, GLOBAL reduces LLC misses by an additional 7%. This clearly demonstrates that our proposed global approach is able to reduce LLC misses significantly more than state-of-the-art local approaches. Finally, Hybrid-A outperforms the best-performing state-of-the-art hardware-based local scheme and shows potential for exploiting synergistic behavior by, in some cases, reducing miss rates more than the best-performing schemes.

6.1.2 Inclusive Baseline. Figure 7 shows LLC misses for all the schemes we evaluate normalized to the inclusive baseline that manages blocks using LRU. Similarly, Figure 8 shows L2 misses for all the schemes we evaluate normalized to the inclusive baseline.

Runtime Schemes—GLOBAL vs. RADAR. LLC Misses: On average, GLOBAL does not provide any marked improvement over RADAR demonstrating that managing private caches does not have considerable impact on LLC misses for inclusive caches. This can be attributed to the fact that the effective capacity of the cache hierarchy in an inclusive baseline is the size of the LLC. Although managing dead blocks in L2s can retain more live blocks, these blocks will be back-invalidated eventually when the block gets evicted from the LLC, thereby ruling out the possibility of improving remote hits. Moreover, inclusion guarantees that dirty write-backs to LLC cannot happen once a block has been evicted from the LLC. Overall, GLOBAL reduces LLC misses on average by 23% compared to 22.7% by RADAR.

L2 Misses: The results for L2 misses show that RADAR is able to reduce L2 misses by 1%, on average, in spite of not managing dead-blocks in the private caches. We attribute this reduction to

Fig. 8. L2 misses for different schemes normalized to the inclusive baseline.

an observed increase in back-invalidations as a consequence of dead block eviction from the LLC. GLOBAL is able to reduce L2 misses even further, on average by 3%, as a consequence of managing dead blocks in the private cache. This demonstrates that handling dead blocks in multiple levels of an inclusive cache hierarchy has a small impact on misses in the private caches and virtually no impact on LLC misses or execution time.

GLOBAL vs. Hardware-based Local Schemes. LLC and L2 Misses: The LLC misses for different configurations of CDBP, SDBP, SHiP++, and Hawkeye show that these local schemes perform significantly worse than the baseline, on average, in the context of inclusive caches. SHiP also experiences a drop in effectiveness, while still being able to perform better than the baseline. This trend is amplified in *cholesky*, *matmul* and *sparselu*. These applications experience a significant increase in both L2 and LLC misses as a consequence of an observed increase in the number of back-invalidations. This stems from the fact that hardware-based dead block predictors do not have knowledge about the lifetime of a task and as a consequence prematurely deem blocks to be dead before a task completes. The negative effect of premature back-invalidations caused by dead-block management can also be observed in the number of L2 misses.

Hybrid Schemes. LLC and L2 Misses: To ascertain that the lack of effectiveness of hardware-based predictors is due to premature invalidations, we evaluate a test scheme (labeled CDBP-Lazy in Figures 7 and 8) that demotes all the blocks deemed dead by CDBP-L2+L3 but does so only after the task boundary. The result demonstrates that lazily evicting dead blocks by exploiting the knowledge of task boundary is able to substantially improve the performance over CDBP-L2+L3 where blocks are evicted eagerly. Overall, we can observe a trend for Hybrid-A and Hybrid-C that is similar to what has been observed for the non-inclusive baseline. Hybrid-C does better than Hybrid-A in terms of L2 misses (see Figure 8), because the latter induces a lot of premature evictions (due to hardware-based predictors) from LLC that in turn result in back-invalidation from the private caches thereby increasing the number of misses. Hybrid-A is still able to perform better than CDBP-Lazy in terms of LLC misses. In addition, Hybrid-A is also able to exploit synergistic effects and delivers additional reduction for *cholesky* and *matmul*, which are compute-intensive, over GLOBAL, whereas it does slightly worse than GLOBAL for the other applications, which are memory intensive.

Summary of Results for Inclusive Caches. The results for the inclusive baseline suggests that GLOBAL provides an average reduction of 23% in LLC misses. The improvement over RADAR

Fig. 9. Execution time for different schemes normalized to the non-inclusive baseline.

Fig. 10. Execution time for different schemes normalized to the inclusive baseline.

is marginal at 0.3% and this can be attributed to the inclusion property, which precludes dead block management done in L2 from having any effect on the number of remote L2 hits, unlike non-inclusive caches. Moreover, delayed write-backs from private caches do not influence dead block prediction decisions in the shared cache due to back-invalidations in inclusive hierarchies. Hardware-based schemes, with the exception of SHiP, perform worse than the baseline, on average, as a consequence of premature invalidation of data before the task has finished execution. Incorporating the knowledge of task boundary into decisions made by existing predictors is able to reduce the misses significantly as demonstrated by the CDBP-Lazy scheme. Finally, similar to the non-inclusive baseline, Hybrid-A is able to outperform hardware-based local scheme but shows limited improvement as a consequence of exploiting synergy while even doing slightly worse for some applications.

6.2 Impact on Execution Time

Figures 9 and 10 show the impact of all the previously evaluated schemes on execution time normalized to the non-inclusive and inclusive LRU baseline, respectively.

Fig. 11. Accuracy and Coverage for predicting LLC and L2 region miss.

6.2.1 Non-inclusive Baseline. For the compute-intensive applications, such as *cholesky*, *matmul* and *sparselu*, the reduction in misses translates only to modest improvement of 1.9% in execution time for GLOBAL, on average. However, memory-intensive applications like *gauss*, *jacobi*, and *redblack* observe, on average, a greater reduction in execution time of 10.8%. RADAR provides an average improvement of 1.4% for compute-intensive applications and 1.7% for memory-intensive applications. Overall, GLOBAL is able to improve execution time by 6.4% and RADAR is able to improve execution time by 1.5%, on average. The hardware-based cache management schemes, such as SHiP++-L2+L3 and Hawkeye-L2+L3, that handle writeback efficiently reduce execution time on average by 3.3% and 4.5%, respectively. This demonstrates that managing dead blocks in the entire cache hierarchy is still beneficial in the context of non-inclusive cache hierarchy. It also demonstrates that the improvement achieved by global management of the cache hierarchy is more significant than what is achieved by adopting local predictors at each level. Although Hybrid-A exploits a synergistic effect and reduces miss rates over GLOBAL, its impact is marginal, since the additional reduction is only achieved for compute-intensive applications. Hybrid-A improves execution time by 6.4% and Hybrid-C improves execution time by 2.7%, on average. In general, we observe that the trends in execution time correlate with the trends in miss rates observed for the different schemes evaluated.

6.2.2 Inclusive Baseline. The results for the inclusive baseline suggest that GLOBAL does not provide much benefit over RADAR as they both have similar improvements in execution time of 5%, on average. Although GLOBAL reduces the number of L2 misses compared to RADAR this does not reflect on execution time. Overall, GLOBAL provides an improvement of 2.2% for compute-intensive applications and 7.8% for memory-intensive applications. Hardware-based schemes, with the exception of SHiP, perform worse than the baseline, on average, as a consequence of premature invalidation of data before the end of a task's lifetime. Hybrid-A exhibits a similar trend in execution time as observed for non-inclusive caches.

6.3 Reuse Prediction Accuracy and Coverage

To understand the effectiveness of the heuristics in predicting region misses in the on-chip caches, we compute the accuracy and coverage for the GLOBAL scheme assuming a non-inclusive baseline. Coverage determines the fraction of the overall region misses in the L2 and LLC predicted correctly by the runtime over the actual number of region misses, i.e., how much opportunity did the runtime exploit, and accuracy determines how often region misses in the L2 and LLC predicted by the runtime are correct over all the predictions. We compute these numbers on the baseline without enabling dead region optimization to avoid our scheme from interfering with the results, and to understand the heuristics' effectiveness. Figure 11 shows the accuracy and coverage with

Fig. 12. Impact of prefetching on performance and DRAM with BL and GLOBAL.

which the runtime heuristics are able to predict region misses in the L2 and the LLC. The results indicate that the proposed heuristics are able to predict regions misses from on-chip caches with reasonable accuracy in spite of being simple and thus make an effective case for global dead-block management. Pursuing more advanced heuristics is thus left as future work. There are certain applications, such as *cholesky* and *sparselu*, where a slightly lower coverage is achieved. The lower coverage can mainly be attributed to the irregular task dependency graph.

6.4 Impact of Prefetching

To evaluate the impact of the GLOBAL scheme in the presence of prefetching, we model one L2 *stream prefetcher* alongside each private L2 cache. This prefetcher detects strides of varying length and issues requests in advance and is trained by L2 misses and by hits to lines inserted by the prefetcher. Each prefetcher can track up to eight independent streams, has a prefetch degree of four and cannot issue prefetches in case the addresses cross page boundaries.

Figure 12 shows the impact of enabling prefetching on performance and DRAM accesses assuming a non-inclusive baseline. The results demonstrate that the baseline with prefetching provides an average reduction in execution time of 21.7%, while increasing the DRAM accesses on average by 2%. The effectiveness of prefetching can be attributed to the regular access patterns associated with accessing cache lines that span a region. GLOBAL, in the presence of prefetching, provides an average reduction in execution time of 21.9%, which translates to just a 0.2% additional reduction over BL. However, GLOBAL ends up reducing the number of DRAM accesses on average by 17.7%. The reduction in DRAM accesses can be attributed to better utilization of on-chip cache capacity through efficient management of dead blocks in the cache hierarchy. Since the reduction in the number of DRAM accesses also helps reduce inter-application interference and energy consumption it makes the proposed predictor desirable even in the presence of prefetching.

6.5 Impact on Energy

The process of evicting a region involves performing a tag check and updating the replacement status bits for each of the cache blocks that belongs to the region. This increases the number of tag accesses performed compared to the baseline. To understand the impact of GLOBAL on the dynamic energy consumption in the memory hierarchy, we provide a breakdown of the energy consumption for the different constituent components. For this analysis, we obtain cache access energy numbers from CACTI 6.5 (Muralimanohar et al. 2007) and DRAM power numbers from CACTI-D (Thoziyoor et al. 2008). Figure 13 provides a breakdown of overall dynamic energy consumption in the private cache, shared cache and the DRAM for the non-inclusive baseline (labeled BL), GLOBAL scheme, and Hybrid-A. The power consumed due to demotion of blocks (i.e., extra tag accesses) is included in the private/shared cache power numbers.

Fig. 13. Dynamic energy consumption of private cache, shared cache, and DRAM normalized to the non-inclusive baseline.

These results assuming a non-inclusive baseline indicate that the biggest contributor to dynamic energy consumption by a large margin is DRAM accesses (reads and writebacks). LLC data accesses are the next biggest contributor. The L2 tag, data, and LLC tag accesses only consume a tiny fraction of the energy. The results indicate that GLOBAL clearly provides significant reduction (22% on average) in energy consumption owing to its ability to reduce the number of LLC misses. The additional improvement in energy consumption over RADAR is 13.5% on average. In addition, for applications like *cholesky* and *matmul* the Hybrid-A scheme, as a consequence of exploiting synergistic effects, is able to reduce energy consumption even further. The results also indicate that for *redblack* the Hybrid-A scheme does not fare as well as the GLOBAL scheme. As part of future work, we want to explore an adaptive mechanism where the runtime system, on a per-application basis, chooses the most beneficial scheme.

7 RELATED WORK

Several techniques, both static and dynamic, have been proposed to manage dead blocks. We first position our work to these techniques and then to previous work on runtime-guided cache management.

Static Techniques: Static techniques leverage semantic information and have been proposed mainly for single-threaded applications (Beyls and D'Hollander 2002, 2005; Jain et al. 2001; Sartor et al. 2005; Wang et al. 2002). They mainly focus on array accesses inside loops and utilize reuse distance analysis, data dependence analysis, and data-flow analysis, among other techniques. They rely on ISA hints to manage the entire cache hierarchy by guiding placement of blocks and by providing hints about blocks that have to be retained/evicted. Static techniques, however, have limited applicability in the context of dynamically scheduled task-based parallel applications.

Dynamic Techniques: Dynamic techniques, in contrast, monitor past accesses to identify when blocks become dead and evict such blocks. They can be grouped under two categories, depending on whether they predict (i) multi-use blocks (Lai et al. 2001; Kharbutli and Solihin 2008; Khan et al. 2010; Keramidas et al. 2007) or (ii) single-use blocks (Wu et al. 2011; Jaleel et al. 2010b; Qureshi et al. 2007) to be dead.

Existing multi-use predictors can be classified along two attributes: where accesses are monitored and how blocks are predicted to be dead. To deem a block dead, some techniques monitor accesses at just the specific cache level (Kharbutli and Solihin 2008; Khan et al. 2010), others monitor accesses at the upper cache level (Chaudhuri et al. 2012), or both (Tian et al. 2013). Different approaches have been proposed to predict when blocks become dead: counting the number of times a block is accessed (Kharbutli and Solihin 2008), tracking the last PC or trace of instructions

to access a block (Khan et al. 2010; Lai et al. 2001), tracking when a block was last accessed (Kaxiras et al. 2001), or using a combination of approaches (Abella et al. 2005; Kampe et al. 2004; Liu et al. 2008). Techniques have also been proposed that bypass single-use blocks from the LLCs and install them directly at upper cache levels to improve utilization (Jalminger and Stenström 2003; Johnson et al. 1999; Rivers et al. 1998).

A drawback of the aforementioned dead-block schemes is that they apply their predictions locally (mostly LLC). Global management has been investigated in the limited context of avoiding back-invalidation of hot data from the L1 cache as a consequence of eviction from the LLC (Zahran 2007; Jaleel et al. 2010a; Tian et al. 2013) and for optimizing placement of cache blocks in a two-level cache hierarchy in uni-processor systems (Zahran and McKee 2010). In contrast, our work introduces global dead-block management, whereby dead-blocks are managed in a coordinated manner across the entire cache hierarchy. In addition, the proposed hybrid scheme shows that global and local schemes can cooperate synergistically.

Runtime Management: Some proposals utilize information about objects available to the runtime system to improve cache utilization. Pointy (Burcea et al. 2012) proposes an efficient approach to prefetching that utilizes object connectivity information available to the runtime system for initiating timely prefetches. Cache Scrubbing (Sartor et al. 2014) eliminates dead blocks and avoids write-back of blocks that belong to dead objects. The lifetime of a region spans from the first time a region is accessed to the last time it is accessed. During a regions lifetime it may become dead in the on-chip caches several times, i.e., several *generations*. Cache scrubbing can only detect the last access to the region in the lifetime. However, our approach can detect if a region is dead after every access and not just after the last access to the region.

Recent works (Dimić et al. 2017; Manivannan et al. 2016; Pan and Pai 2015) leverage information about tasks and its working set in task-based programming models to improve efficiency of the shared LLC. However, like other local schemes, they only address cache inefficiency at a single level of the cache hierarchy. This article shows that managing dead blocks in the entire cache hierarchy offers substantially higher gains and proposes a framework for coordination with local schemes.

In addition to these, several orthogonal optimizations have been proposed to leverage information about tasks and data accesses available to the runtime system to improve execution efficiency (Valero et al. 2014; Alvarez et al. 2015a, 2015b; Papaefstathiou et al. 2013; Kyriacou et al. 2004; Manivannan and Stenstrom 2014; Manivannan et al. 2013).

8 CONCLUSION

Existing dead-block management schemes have to evict a dead block at each cache level locally before a block can be globally deemed dead in the entire hierarchy. In this article, we propose, for the first time, a global approach whereby the run-time system helps evicting an address range—a region—from the entire cache hierarchy simultaneously, if the region is deemed to be globally dead. This is achieved by leveraging semantic information from the programming model about region accesses. Our evaluation shows that the proposed global approach to dead-block management is more effective than state-of-the-art local approaches in improving the efficiency across all levels of the cache hierarchy. In addition, we also propose and evaluate a hybrid scheme in which local schemes provide hints to the runtime-orchestrated global dead-block management to further improve the accuracy of the decisions.

The proposed global approach, although evaluated here specifically for dead-block management, can also be applied to other aspects of cache management, including placement, lookup, and write-backs. A fruitful direction is to study how this approach can be used to improve other aspects of cache management and the utility of the entire cache hierarchy.

REFERENCES

- Jaume Abella, Antonio González, Xavier Vera, and Michael F. P. O'Boyle. 2005. IATAC: A smart predictor to turn-off L2 cache lines. *ACM Trans. Archit. Code Optim.* 2, 1 (Mar. 2005), 55–77.
- Lluc Alvarez, Miquel Moretó, Marc Casas, Emilio Castillo, Xavier Martorell, Jesús Labarta, Eduard Ayguadé, and Mateo Valero. 2015a. Runtime-guided management of hybrid memory hierarchies in multicore architectures. In *Proceedings of the 24th International Conference on Parallel Architectures and Compilation*.
- Lluc Alvarez, Lluís Vilanova, Miquel Moretó, Marc Casas, Marc González, Xavier Martorell, Nacho Navarro, Eduard Ayguadé, and Mateo Valero. 2015b. Coherence protocol for transparent management of scratchpad memories in shared memory manycore architectures. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 720–732.
- Kristof Beyls and Erik H. D'Hollander. 2002. Reuse distance-based cache hint selection. In *Proceedings of the European Conference on Parallel Processing (Euro-Par'02)*. Springer, 265–275.
- Kristof Beyls and Erik H. D'Hollander. 2005. Generating cache hints for improved program efficiency. *J. Syst. Architect.* 51, 4 (2005), 223–250.
- Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. 1999. Provably efficient scheduling for languages with fine-grained parallelism. *J. ACM* 46, 2 (Mar. 1999), 281–321.
- Ioana Burcea, Livio Soares, and Andreas Moshovos. 2012. Pointy: A hybrid pointer prefetcher for managed runtime systems. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*. ACM, New York, NY, 97–106.
- Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. 2011. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 52.
- Mainak Chaudhuri, Jayesh Gaur, Nithiyanandan Bashyam, Sreenivas Subramoney, and Joseph Nuzman. 2012. Introducing hierarchy-awareness in replacement and bypass algorithms for last-level caches. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*. 293–304.
- Shimin Chen, Phillip B. Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E. Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd C. Mowry, and Chris Wilkerson. 2007. Scheduling threads for constructive cache sharing on CMPs. In *Proceedings of the Symposium on Parallel Algorithms and Architectures (SPAA'07)*.
- Vladimir Dimić, Miquel Moretó, Marc Casas, and Mateo Valero. 2017. Runtime-assisted shared cache insertion policies based on re-reference intervals. In *Proceedings of the European Conference on Parallel Processing (Euro-Par'17)*. 247–259.
- Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. 2011. Ompps: A proposal for programming heterogeneous multi-core architectures. *Parallel Process. Lett.* 21, 02 (2011), 173–193.
- Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The implementation of the cilk-5 multithreaded language, In *Proceedings of the ACM Special Interest Group on Programming Languages (SIGPLAN'98)*.
- Akansha Jain and Calvin Lin. 2016. Back to the future: Leveraging Belady's algorithm for improved cache replacement. In *Proceedings of the ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA'16)*. 78–89.
- Prabhat Jain, Srinivas Devadas, Daniel Engels, and Larry Rudolph. 2001. Software-assisted cache replacement mechanisms for embedded systems. In *Proceedings of the 2001 IEEE/ACM International Conference on Computer-aided Design (ICCAD'01)*. IEEE Press, Piscataway, NJ, 119–126.
- Aamer Jaleel, Eric Borch, Malini Bhandaru, Simon C. Steely Jr., and Joel S. Emer. 2010a. Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (TLA) cache management policies. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'10)*. 151–162.
- Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. 2010b. High performance cache replacement using re-reference interval prediction (RRIP). In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*. ACM, New York, NY, 60–71.
- Jonas Jalminger and Per Stenström. 2003. A novel approach to cache block reuse predictions. In *Proceedings of the International Conference on Parallel Processing*. IEEE, 294–302.
- Teresa L. Johnson, Daniel A. Connors, Matthew C. Merten, and Wen-mei W. Hwu. 1999. Run-time cache bypassing. *IEEE Trans. Comput.* 48, 12 (Dec. 1999), 1338–1354.
- Martin Kampe, Per Stenstrom, and Michel Dubois. 2004. Self-correcting LRU replacement policies. In *Proceedings of the 1st Conference on Computing Frontiers (CF'04)*. 181–191.
- Stefanos Kaxiras, Zhigang Hu, and Margaret Martonosi. 2001. Cache decay: Exploiting generational behavior to reduce cache leakage power. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA'01)*. ACM, New York, NY, 240–251.
- Georgios Keramidas, Pavlos Petoumenos, and Stefanos Kaxiras. 2007. Cache replacement based on reuse-distance prediction. In *Proceedings of the 25th International Conference on Computer Design (ICCD'07)*. IEEE, 245–250.

- Samira Manabi Khan, Yingying Tian, and Daniel A. Jimenez. 2010. Sampling dead block prediction for last-level caches. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'10)*. IEEE Computer Society, Washington, DC, 175–186.
- Mazen Kharbutli and Yan Solihin. 2008. Counter-based cache replacement and bypassing algorithms. *IEEE Trans. Comput.* 57, 4 (Apr. 2008), 433–447. DOI: <http://dx.doi.org/10.1109/TC.2007.70816>
- Costas Kyriacou, Paraskevas Evripidou, and Pedro Trancoso. 2004. Cacheflow: A short-term optimal cache management policy for data driven multithreading. In *Proceedings of the European Conference on Parallel Processing (Euro-Par'17)*. Springer, 561–570.
- An-Chow Lai, Cem Fide, and Babak Falsafi. 2001. Dead-block prediction and dead-block correlating prefetchers. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA'01)*. ACM, New York, NY, 144–154. DOI: <http://dx.doi.org/10.1145/379240.379259>
- Haiming Liu, Michael Ferdman, Jaehyuk Huh, and Doug Burger. 2008. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'08)*. IEEE Computer Society, Washington, DC, 222–233.
- Madhavan Manivannan, Anurag Negi, and Per Stenstrom. 2013. Efficient forwarding of producer-consumer data in task-based programs. In *Proceedings of the 42nd International Conference on Parallel Processing (ICPP'13)*. 517–522.
- Madhavan Manivannan, Vassilis Papaefstathiou, Miquel Pericas, and Per Stenstrom. 2016. RADAR: Runtime-assisted dead region management for last-level caches. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'16)*. IEEE, 644–656.
- Madhavan Manivannan and Per Stenstrom. 2014. Runtime-guided cache coherence optimizations in multi-core architectures. In *Proceedings of the IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 625–636.
- Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. 2007. Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'07)*. IEEE Computer Society, Washington, DC, 3–14.
- Ragavendra Natarajan and Mainak Chaudhuri. 2013. Characterizing multi-threaded applications for designing sharing-aware last-level cache replacement policies. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'13)*. 1–10.
- OpenMP Architecture Review Board. 2005. OpenMP API Version 2.5.
- OpenMP Architecture Review Board. 2013. OpenMP API Version 4.0.
- Abhisek Pan and Vijay S. Pai. 2015. Runtime-driven shared last-level cache management for task-parallel programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15)*. ACM, New York, NY, 11:1–11:12.
- Vassilis Papaefstathiou, Manolis G. H. Katevenis, Dimitrios S. Nikolopoulos, and Dionisios Pnevmatikatos. 2013. Prefetching and cache management using task lifetimes. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*. ACM, 325–334.
- J. M. Perez, R. M. Badia, and J. Labarta. 2008. A dependency-aware task-based programming environment for multi-core architectures. In *Proceedings of the IEEE International Conference on Cluster Computing (Cluster'08)*. 142–151.
- Miquel Pericàs, Kenjiro Taura, and Satoshi Matsuoka. 2014. Scalable analysis of multicore data reuse and sharing. In *Proceedings of the 28th ACM International Conference on Supercomputing (ICS'14)*. ACM, New York, NY, 353–362.
- Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. 2007. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA'07)*. ACM, New York, NY, 381–391.
- Jude A. Rivers, Edward S. Tam, Gary S. Tyson, Edward S. Davidson, and Matt Farrens. 1998. Utilizing reuse information in data cache management. In *Proceedings of the 12th International Conference on Supercomputing (ICS'98)*. ACM, New York, NY, 449–456.
- Jennifer B. Sartor, Wim Heirman, Stephen M. Blackburn, Lieven Eeckhout, and Kathryn S. McKinley. 2014. Cooperative cache scrubbing. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*. ACM, 15–26.
- Jennifer B. Sartor, Subramaniam Venkiteswaran, Kathryn S. McKinley, and Zhenlin Wang. 2005. Cooperative caching with keep-me and evict-me. In *Proceedings of the 9th Annual Workshop on Interaction Between Compilers and Computer Architectures (INTERACT'05)*. IEEE, 46–57.
- Shyamkumar Thoziyoor, Jung Ho Ahn, Matteo Monchiero, Jay B. Brockman, and Norman P. Jouppi. 2008. A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA'08)*. IEEE Computer Society, Washington, DC, 51–62.
- Yingying Tian, Samira M. Khan, and Daniel A. Jiménez. 2013. Temporal-based multilevel correlating inclusive cache replacement. *ACM Trans. Archit. Code Optim.* 10, 4 (Dec. 2013).

- Mateo Valero, Miquel Moreto, Marc Casas, Eduard Ayguade, and Jesus Labarta. 2014. Runtime-aware architectures: A first approach. *Supercomput. Frontiers Innovat.* 1, 1 (2014).
- H. Vandierendonck, G. Tzenakis, and D. S. Nikolopoulos. 2011. A unified scheduler for recursive and task dataflow parallelism. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'11)*. 1–11.
- Zhenlin Wang, Kathryn S. McKinley, Arnold L. Rosenberg, and Charles C. Weems. 2002. Using the compiler to improve cache replacement decisions. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'02)*. IEEE Computer Society, Washington, DC, 199.
- David A. Wood, Mark D. Hill, and R. E. Kessler. 1991. A model for estimating trace-sample miss ratios. In *Proceedings of the ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'91)*. ACM, New York, NY, 79–89.
- Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C. Steely, Jr., and Joel Emer. 2011. SHiP: Signature-based hit predictor for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'11)*. ACM, New York, NY, 430–441.
- Vinson Young, Chien Chen, Aamer Jaleel, and Moin Qureshi. 2017. SHiP++: Enhancing signature-based hit predictor for improved cache performance. In *Proceedings of the Cache Replacement Championship (CRC'17) held in Conjunction with the International Symposium on Computer Architecture (ISCA'17)*.
- Mohamed Zahran. 2007. Cache replacement policy revisited. In *Proceedings of the Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD'07) held in conjunction with the International Symposium on Computer Architecture (ISCA'07)*.
- Mohamed Zahran and Sally A. McKee. 2010. Global management of cache hierarchies. In *Proceedings of the 7th Conference on Computing Frontiers*.

Received August 2017; revised May 2018; accepted May 2018