RADAR: Runtime-Assisted Dead Region Management for Last-Level Caches

Madhavan Manivannan, Vassilis Papaefstathiou, Miquel Pericàs, Per Stenström
Chalmers University of Technology
{madhavan,vaspap,miquelp,per.stenstrom}@chalmers.se

ABSTRACT

Last-level caches (LLCs) bridge the processor/memory speed gap and reduce energy consumed per access. Unfortunately, LLCs are poorly utilized because of the relatively large occurrence of dead blocks. We propose RADAR, a hybrid static/dynamic dead-block management technique that can accurately predict and evict dead blocks in LLCs. RADAR does dead-block prediction and eviction at the granularity of address regions supported in many of today's task-parallel programming models. The runtime system utilizes static control-flow information about future region accesses in conjunction with past region access patterns to make accurate predictions about dead regions. The runtime system instructs the cache to demote and eventually evict blocks belonging to such dead regions. This paper considers three RADAR schemes to predict dead regions: a scheme that uses control-flow information provided by the programming model (Look-ahead), a history-based scheme (Look-back) and a combined scheme (Look-ahead and Look-back). Our evaluation shows that, on average, all RADAR schemes outperform state-of-the-art hardware dead-block prediction techniques, whereas the combined scheme always performs best.

1. INTRODUCTION

Last-level caches (LLCs) have short access time and consume less energy per data access in comparison to off-chip memory. Therefore, efficient management of LLCs is essential for improving performance and reducing energy consumption. However, *dead blocks* are an important source of inefficiency in LLCs. They tend to stay unused in LLCs for a long time until they are eventually evicted [43, 24] and occupy precious cache space without providing any benefit. Cache efficiency can be improved substantially if dead blocks can be identified accurately and evicted in a timely manner to make space for other blocks [31].

Existing proposals to predict dead blocks broadly fall into two categories: dynamic and static techniques. Dynamic techniques predict dead blocks by using block access history, collected during execution through architectural mechanisms [1, 11, 14, 24, 26, 27, 30, 31]. Static techniques, on the other hand, predict dead blocks by using control-flow information to determine future accesses [6, 7, 18, 42]. Static and dynamic techniques thus have complementary strengths and weaknesses. Dynamic techniques can learn and adapt to changing access patterns but are limited by their inability to leverage information about data accesses in the future. On the contrary, static techniques have the ability to predict future accesses, but are restricted to cases when control-flow information is available during analysis.

This paper proposes RADAR, a framework for exploiting information exchange across the programming model, runtime system and architecture layers to enable completely new optimizations such as dead-block elimination. RADAR builds specifically on modern task data-flow programming models (e.g. OpenMP 4.0.) where dependencies between tasks are statically specified by the programmer using annotations. The annotations specify the *address regions* accessed by a task and provides the basis for dependency resolution between tasks . This dependency information is conveyed to the runtime system allowing it to construct a task data-flow graph during execution that provides a (limited) outlook of future region accesses.

The core of RADAR is a runtime system that collects static region-access information from the programming model and dynamic access history from the architecture. This enables RADAR to predict dead regions using two orthogonal approaches: exploiting knowledge of future region accesses and exploiting information about past region access patterns. The *Look-ahead scheme* peeks into a window of future tasks that are soon to be executed to determine if a region will be accessed in future. On the other hand, the *Look-back scheme* tracks per-region access history and uses it to predict how far into the future the next region access will occur. The orthogonal nature of these two approaches motivates us to also evaluate a combined Look-ahead and Look-back scheme for improved efficiency.

We compare the performance of the proposed RADAR schemes with state-of-the-art dynamic dead-block prediction techniques using detailed architectural simulation. Our evaluation shows that combining static and dynamic information causes RADAR to outperform earlier proposed dead-block prediction techniques that rely only on past eviction patterns to predict dead blocks.

In summary, this work makes three contributions:

- The RADAR framework, in which a runtime system uses static and dynamic information, from the programming model and the architecture, respectively, to predict dead address regions.

978-1-4673-9211-2/16/$31.00

- Specific schemes in the RADAR framework for dead-block elimination: A Look-ahead, a Look-back and a combined Look-ahead and Look-back scheme that utilizes information about past access patterns, future access patterns, and both, respectively.

- A quantitative comparison of the three schemes showing that RADAR enabled schemes can outperform previously proposed dynamic dead-block prediction techniques and that the combined scheme performs best.

As for the rest of the paper, we introduce task-parallel programming and dead region management notations in Section 2. Section 3 then presents RADAR and the associated schemes we propose. Sections 4 and 5 present the experimental methodology and evaluation. Section 6 discusses related work followed by concluding remarks in Section 7.

2. BACKGROUND

We introduce the parallel programming model semantics in Section 2.1 and the notion of live and dead address regions, central to the RADAR approach, in Section 2.2.

2.1 Programming Model Concepts

Several parallel programming models exploit parallelism using the notion of tasks as exemplified by TBB [17], Cilk [8] and OpenMP [35]. Tasks are typically conservatively synchronized using barriers and this imposes unnecessary synchronization between independent tasks. For this reason, many emerging task data-flow programming models [12, 16, 35] only synchronize dependent tasks by relying on dependency annotations to establish inter-task dependencies. Such models have been shown to improve performance and scalability of asynchronous parallel applications as a consequence of the reduction in synchronization overheads [12, 4].

We assume OmpSs [12], a task data-flow programming model that extends OpenMP to support asynchronous parallelism on heterogeneous multi-core architectures. Listing 1 shows the *sparselu* code snippet from the Barcelona Application Repository (BAR) expressed in the OmpSs programming model. Like in the OpenMP 4.0 specification [35], dependencies are specified using *in, out* and *inout* clauses as part of the task creation construct. The dependency clauses specify *address regions* that are read (*in*), updated (*out*) or both (*inout*) by the task and these constitute the working set of a task. Regions enable the programmer to refer to multiple addresses or ranges within an array by using a single expression[1]. Line 3 in the listing indicates that task *lu0* reads and updates the region *A[k][k]* where the region constitutes an array of a certain size. This dependency annotation enables the runtime system to establish inter-task dependencies.

The runtime system establishes dependencies by letting a master thread generate tasks in program order. Tasks are tracked until all their input and output dependencies are resolved. Tasks which do not have any unresolved dependencies are submitted to the ready queue in an out-of-order manner. Worker threads pick up tasks from the ready queue for execution. After a worker thread finishes execution of a task,

[1]In the simplest case a region refers to a single address

Listing 1: sparselu code snippet

```
1   for(k=0; k<NB; k++)
2   {
3  #pragma omp task inout(A[k][k])
4   lu0(A[k][k]);
5         for(j=k+1; j<NB; j++)
6            if(A[k][j] != NULL)
7  #pragma omp task in(A[k][k]) inout(A[k][j])
8       fwd(A[k][k], A[k][j]);
9         for(i=k+1; i <NB; i++)
10           if(A[i][k] != NULL)
11 #pragma omp task in(A[k][k]) inout(A[i][k])
12              bdiv(A[k][k], A[i][k]);
13
14        for(i=k+1; i<NB; i++)
15          for(j=k+1; j<NB; j++)
16            if(A[k][j] != NULL && A[i][k] != NULL) {
17              if(A[i][j] == NULL)
18                  A[i][j] = allocate_block();
19 #pragma omp task in(A[i][k], A[k][j]) inout(A[i][j])
20              bmod(A[i][k], A[k][j], A[i][j]);
21      }
22  }
```

other dependent tasks are released and these in turn are submitted to the ready queue for execution. The set of tasks in the ready queue and dependent tasks with unresolved dependencies together makeup a task data-flow graph which constitutes the *look-ahead window* of tasks. RADAR exploits the look-ahead window to identify live and dead address regions as will be described in Section 3.

2.2 Generational Behavior of Address Regions

Typically, cache blocks are accessed frequently after they are brought into the cache followed by a period of inactivity. Following the last access, blocks eventually move to the LRU position and get evicted. The sheer size of LLC prolongs the period for which the blocks remain inactive in the cache. On a subsequent access, the blocks experience a cache miss and are re-fetched from memory. This is referred to as generational behavior [43]. Generational behavior of cache blocks has been characterized and exploited to enable a wide range of dead-block related optimizations [43, 24].

Unlike prior work on dead-block management, RADAR manages blocks at the granularity of *address regions* specified by task data-flow programming models. An address region is considered to have been accessed when a task reading from or writing to the region completes. An access to a region is classified as a *region miss* when all the cache blocks accessed during a region access have to be (re-)fetched from memory. Conversely, if there is a hit even to a single block in that region, the access is classified as a *region hit*. Our definition of a region miss conservatively requires that all accessed blocks to be re-fetched so as to avoid classifying regions with a small number of cache hits as dead.

As shown in Figure 1, a *generation* extends from a region miss, when the region becomes live, to the last region hit (or end of execution) after which the region becomes dead. Cache blocks which belong to dead regions remain unused in the cache until they are evicted. A new generation begins when the next access following an eviction results in a region

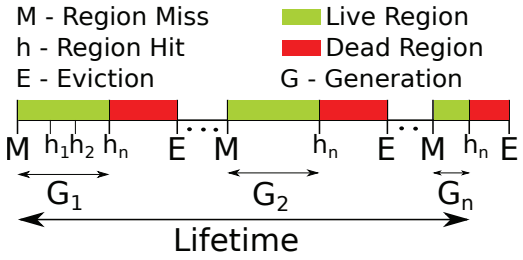

Figure 1: Generational behavior in regions.

miss. The *lifetime* of a region extends from the first access in the first generation to the last access in the last generation and can thus span one or more generations.

3. RUNTIME-MANAGED DEAD REGIONS

We begin by introducing the RADAR framework which builds on interaction across three layers: the programming model, the runtime system and the architecture layers in Section 3.1. We then propose three schemes that are accommodated in the RADAR framework for detecting dead regions. First, the *Look-ahead scheme* described in Section 3.2 predicts dead regions by utilizing the look-ahead window constructed by the runtime system. Second, the *Look-back scheme* described in Section 3.3 predicts dead regions based on region accesses in the past. Third, the combined *Look-ahead and Look-back scheme*, described in Section 3.4 can offer higher prediction accuracy for dead regions by combining past and future information about region access. Finally, in Section 3.5 we introduce architecture support needed for demoting the cache blocks that belong to the dead regions from the LLC.

3.1 RADAR Framework

RADAR is a hybrid static/dynamic technique that can improve LLC efficiency by detecting and evicting cache blocks belonging to dead regions. Conceptually, RADAR adopts the following strategy for dead region management: after each task finishes execution, RADAR's runtime system predicts whether the regions accessed by the task are dead. This is akin to predicting if the regions will experience a region miss on the next access. In case a region is predicted to be dead, RADAR's runtime system hints the LLC to demote the blocks belonging to the dead region to the LRU position.

In a nutshell and as shown in Figure 2, RADAR can be viewed as an interaction across three layers: 1) a programming model layer that conveys static data-dependency information by providing the regions that are used by tasks; 2) a runtime system layer responsible for detecting dead regions during execution and 3) an architecture layer responsible to provide dynamic feedback information about region access and demote the cache blocks that belong to dead regions. In order to predict and demote dead regions, additional functionality is needed in the runtime system and in the architecture layers as illustrated in Figure 2. Next we describe the three schemes that can be implemented on top of the RADAR framework to accurately predict dead regions.

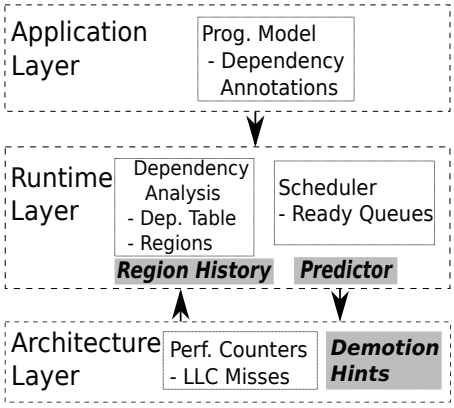

Figure 2: Overview of RADAR and support required across the stack. Boxes represent existing (key) components that RADAR leverages and shaded boxes represent added functionality for RADAR.

3.2 Look-ahead Scheme

3.2.1 Conceptual Approach

The runtime system dynamically constructs the data-flow graph out of tasks dispatched for execution and those waiting for dependency resolution. This provides the runtime system with a look-ahead window for observing future tasks and their region accesses. The Look-ahead scheme uses information from this look-ahead window, typically covering hundreds of tasks, to make a decision: If the region is not accessed by any task in the look-ahead window, the next access to that region is deemed to result in a region miss (dead region). Conversely, if the region is accessed in the look-ahead window, the next access is deemed to result in a region hit (live region).

3.2.2 Implementation

Task data-flow runtime systems track internal state of regions to establish inter-task dependencies during execution. Specifically, the runtime system maintains per-region state – typically in a hash-table– and keeps ordered lists of the producer (writer) and the consumer (reader) tasks that access the region. We leverage this state for finding reuses of a region in the dynamically constructed data-flow graph.

We first illustrate how dependencies are tracked using *sparselu* decomposition on a 3×3 blocked matrix as an example[2]. Figure 3 depicts the task dependency graph and Figure 4 illustrates how the runtime system tracks the state of regions using a dependency table and performs reference counting. For each generated task, the runtime system probes the dependency table on a per-argument basis. In our example, *T0* is the first task and does not have any dependencies, therefore a new entry for *A00* is allocated in the dependency table and the writer field (W in Figure 4) set to *T0*. When tasks *T1, T2, T3, T4* are generated, the runtime system detects a dependency with *A00* and tasks *T1, T2, T3, T4* are

[2]Each element represents an address region that spans multiple cache blocks.

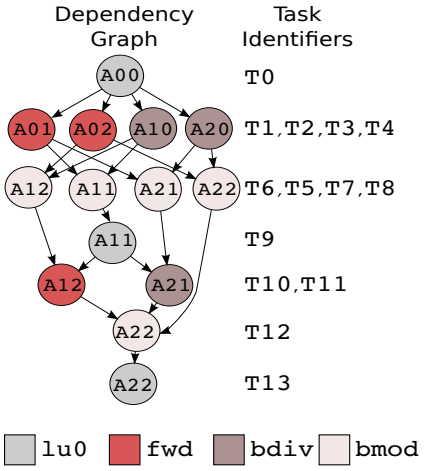

Figure 3: SparseLU dependency graph.

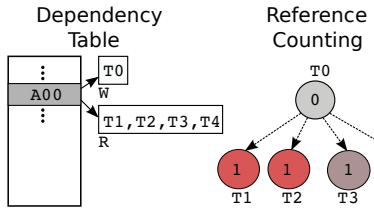

Figure 4: Dependency tracking mechanism.

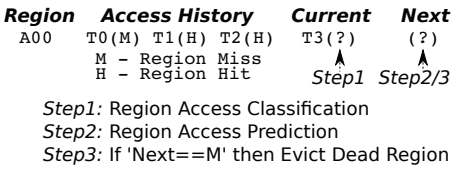

Figure 5: Overview of the Look-back Scheme.

added to the reader's list (R in Figure 4). In addition, the new tasks are added to *T0*'s successor list and the reference count of each new task is incremented by one as shown in Figure 4. When *T0* completes execution, the writer field is cleared and the reference count of each of its successors is decremented by one. Once a successor's reference count becomes zero, it denotes that all its dependencies have been satisfied. The dependent tasks are enqueued in the ready queue for execution. Finally, when tasks *T1, T2, T3, T4* finish execution they are removed from the readers list of *A00*.

The internal state already maintained by the runtime system offers RADAR the opportunity to identify future reuses at almost no additional cost. In the previous example, RADAR can infer that region *A00* is dead after the last task that uses it completes execution and removes the last entry from the readers list. Observing the per-region state (readers and writers) permits the Look-ahead scheme to identify whether further accesses to a region are expected from the tasks in the look-ahead window.

3.2.3 Discussion

The Look-ahead scheme has two limitations. Firstly, although the look-ahead window can confirm reuse it does not provide temporal information, i.e. when the next access will actually happen. Therefore, it does not establish if the next access would be part of the current region generation or if it would result in a new generation as a consequence of a *region miss*. Secondly, even if a reuse is not detected by the look-ahead window it cannot be guaranteed that there would be no further accesses to the region. This could happen because the master thread that generates tasks has not progressed fast enough to populate the look-ahead window with future tasks, thus distant region reuses may not be available in a timely fashion.

3.3 Look-back Scheme

3.3.1 Conceptual Approach

The Look-back scheme exploits the observation that region accesses tend to exhibit a repetitive pattern during their lifetime. This behavior opens up the opportunity to predict future accesses to regions based on past access patterns using the same methodology as branch predictors: (i) classify the current region access as hit/miss (analogous to taken/not taken) and (ii) predict whether the next access to the region will hit/miss based on previous accesses to the region. In case the next access is predicted to be a region miss the region is considered to be dead. Figure 5 uses region *A00* in the *sparselu* example discussed previously to illustrate the different steps involved in predicting if this region is dead.

3.3.2 Implementation

This section presents the approach for classifying the current region access as a hit/miss and predicting the outcome of the next access.

Region Access Classification

While regions are application-level abstractions, their liveness status depends on the capacity of the cache. The Look-ahead scheme, discussed previously, can only establish that a region will be reused but not whether it will be still in cache when reused, as it does not take into account the size of regions and the size of the LLC. In contrast, the Look-back scheme uses architectural feedback to determine if region access is a hit/miss. The feedback metric used is the per-task LLC miss count.

Assuming that the task accesses a single region, the Look-back scheme classifies a region access as a miss if the per-task LLC miss-count corresponds to the number of cache blocks in a region. This information is trivially computed from the region-size specification provided to the runtime system. This approach is accurate when the cache blocks accessed by a task are confined to a single region and the misses are not due to other sources, such as private data or instructions. RADAR establishes the per-task LLC miss-count using performance counters readily available in most modern architectures.

We next consider the case when a task accesses two regions. For simplicity, we assume that both regions are of the same size, each region corresponds to R blocks and the

Listing 2: Pseudo-code for region access classification

```
1  foreach region in task.regions; do
   // in descending region distance order
2    if (miss_count >= region.total_cache_lines)
3      miss_count = miss_count − region.total_cache_lines
4      region.status = REGION_MISS
5    else
6      break // remaining regions are REGION_HIT by default
7  done
```

recorded miss-count for the task is M. First, if $M = 0$, both regions are classified as hit. Second, if $M = 2R$, both regions are classified as miss. Third, if $R > M > 0$, both regions are still classified as hit. Finally, if $2R > M \geq R$, a question arises as to which of the two region accesses should be classified as a miss.

To resolve this issue we use the notion of *region distance*. Region distance is the number of intermediate region accesses between two successive accesses to a region and is computed at region access granularity. Determining the region distance requires keeping track of: i) the total number of regions accessed up to the current access and ii) when each region was last accessed. The Look-back scheme classifies the region with the largest region distance as a miss. The intuition is that the region with the largest region distance is also the oldest, and thus most likely to have missed in the cache [3].

We now generalize this procedure to arbitrary number of regions. Assuming N regions, all the regions are initially classified as region hit. RADAR attributes misses to regions based on descending region distance order as shown in Line 1 in Listing 2. For each of the regions accessed by the task, the classification logic shown in Line 2-4 changes the outcome (denoted by region.status) to a region miss if it satisfies the condition for a region miss, namely the per-task miss count M (denoted by miss_count) is greater than the number of blocks in the region R (denoted by region.total_cache_lines) as shown in Line 2. In that case the region size is subtracted from the per-task LLC miss count as shown in Line 3. As soon as a region hit is detected the algorithm stops. In the rare event that the region distances are equal, misses are attributed in region annotation order. This procedure enables RADAR to classify multiple regions accessed by a task as hit/miss by employing LLC miss count and region distance in conjunction.

Region Access Prediction

Once the current access is classified, RADAR uses the region access history to predict the next region access by leveraging prior work on branch predictors [45]. Region access prediction is analogous to *Taken/Not-Taken* predictions made by the branch predictor because they also use past single-bit outcomes to predict the next outcome. We utilize simple predictors like single-level bimodal predictors and two-level branch predictors (*PAp:per-address history with private pattern table*). This is because the limited number of accesses to a region in its lifetime makes it hard to train large predictor tables. The changes needed to track and predict region accesses are incorporated in the per-region state maintained by the runtime system. We have found that some patterns can be predicted accurately using a two-level predictor whereas others can be predicted accurately by using a single-level predictor. Therefore RADAR exercises both predictors in parallel in order to monitor the accuracy of each one and select the best-performing one on a per-region basis.

For the single-level bimodal predictor, RADAR requires two additional variables, one to act as a 2-bit saturating counter and another to monitor the number of mispredictions, which is needed to choose the predictor with the lowest number of mispredictions. On each access, RADAR compares the prediction with the actual outcome and increments the saturating counter only if the prediction is correct and the access is a region miss. In case of a misprediction or a region hit the saturating counter variable is reset. In order to improve the prediction accuracy, RADAR predicts a miss only when the 2-bit counter value has saturated, i.e. the confidence of the prediction is the highest.

For the two-level predictor, RADAR uses one variable to act as a history register, one to hold the number of mispredictions, and an array of 2-bit saturating counters indexed by the value of the history register. The number of 2-bit saturating counters depends on the length of the history register. We find that tracking last six accesses and using it to index an array of 64 2-bit counters provides accurate results. The mechanism by which individual counters are accessed/updated is similar to the single-level predictor. The performance impact of the modifications to the runtime system by RADAR is presented in Section 4.

Region Specification

The region access classification logic classifies an access as a region miss only if all the cache blocks that are part of the region specification are accessed. However, there are applications in which the region specification is imprecise i.e. overprovisioned. In such cases the classification logic would not be able to classify an access as a miss. In order to improve classification for such misrepresented regions, the Look-back scheme incorporates an additional heuristic to determine how a region access should be classified independently of the LLC miss count. This heuristic is based on the observation that reuses with a region distance over a certain *high threshold* are most likely to be fetched from memory and those with a region distance under a certain *low threshold* are most likely to be fetched from cache. The thresholds are computed during the initial program phase with the help of the dependency table introduced in Section 3.2. To determine the high threshold we need two variables: one to count the number of regions accessed and the other to count the total size of all the unique regions in the dependency table. The former is updated on access and the latter on insertion of a region into the dependency table. The high threshold value corresponds to the number of regions accesses when the size of unique regions matches the size of the LLC (e.g. 8MB). A fraction (25%) of this high threshold is used as the low threshold. The thresholds are used in conjunction with the procedure detailed in Listing 2 for classification.

[3]We do not consider the order of region accesses within a single task as it is unlikely to influence region access classification.

3.3.3 Discussion

Look-back scheme also has two limitations. First, it is prone to inefficiencies introduced by misprediction of region accesses. For instance, mispredicting a region hit as a region miss leads to cache blocks of the region getting demoted and subsequently evicted from the cache. This can have a cascading effect resulting in subsequent accesses (even those with short region distance) appearing as misses. To address this issue, the cache blocks are demoted to one position ahead of the LRU (LRU-1). This gives demoted blocks a chance to stay in the cache slightly longer thereby increasing their probability of being re-accessed in the LLC. Second, this scheme, like all history-based approaches, cannot predict future accesses accurately if the access patterns deviate from those observed in the past. In Section 5.1 we evaluate the prediction accuracy of this scheme.

3.4 Combined Scheme

Having presented two orthogonal schemes for predicting dead regions, it is natural to consider whether the two can be combined to achieve better performance than each scheme in isolation.

Conceptually, the combined Look-ahead and Look-back scheme initially classifies all accessed regions as hit. We define *LA* to be the set of all the region accesses classified by the Look-ahead scheme as misses and *LB* to be the set of all the region accesses classified by the Look-back scheme as misses. The combined scheme then classifies regions using two approaches: The *conservative combined* and the *aggressive combined* approach. In the conservative combined approach, the set of regions that belongs to ($LA \cap LB$), i.e., the intersecting set, is classified as miss. The intuition is that both schemes agree upon that these regions miss which is expected to lead to a high prediction accuracy but may miss opportunities. By contrast, in the aggressive combined approach, the set of regions that belongs to ($LA \cup LB$), i.e., the union of the sets, is classified as miss. The intuition is that by considering the different perspectives of the two schemes, we can identify more dead regions, potentially with a higher misprediction rate.

3.5 Architectural Support

Once the runtime component of RADAR identifies a dead region, hardware support is required to manage the blocks belonging to the dead region. In this work, we demote the cache blocks of the dead region to the LRU or LRU-1 position instead of evicting them in order to alleviate the cost of mispredictions. Since existing architectures do not provide support for demoting a cache block to the LRU position we incorporate architectural support to accomplish this.

One option is to demote one cache block at a time by introducing support for *dbhint* instruction that takes the virtual address and demotes the cache block to the LRU position in the LLC. We do not consider this option because it requires one instruction per cache-block and thus has high software overhead. Furthermore, it also increases the queueing requirements and the traffic to the LLC controller. RADAR instead introduces support for a region demotion instruction (*rdbhint*) that takes the base virtual address, the length of the region that needs to be demoted, the demotion position (LRU or LRU-1) and sends a hint to the L2 cache controller.

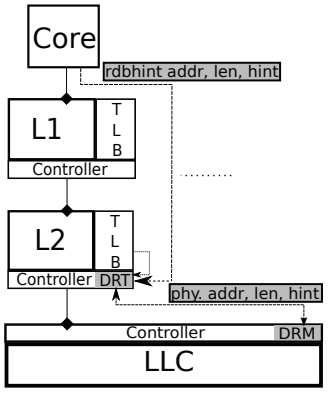

Figure 6: Overview of RADAR's architectural support.

The proposed architectural support shown in Figure 6 consists of an engine at the L2 cache controller called *Dead Region Tracker (DRT)* that has direct access to the L2 TLB in order to support the virtual-to-physical address translation. DRT accepts the virtual address range (base, length), translates it in a per-page basis, and sends a demotion hint to the underlying LLC for each physical range associated with the dead region. The L2 TLB needs to be accessed just once for each page involved in a region. Since a single region may span a few pages the number of translations introduced as a consequence is limited. We evaluate the impact on energy consumption of the memory hierarchy in Section 5.2.3.

The hardware support in the LLC controller consists of an engine called *Dead Region Manager (DRM)* that receives the physical ranges from DRT. The DRM probes the cache on a per-cache-block basis and demotes each cache block that hits in a set. The LLC tags are probed only on idle cycles in order to avoid interfering with demand requests (loads, stores) from the cores. The DRM contains a command queue supporting one outstanding DRT command (physical address range) per core. Once the DRM demotes all the blocks belonging to the physical range it signals the completion of the operation to the corresponding DRT. The DRT supports up-to eight outstanding virtual address ranges from the local core and sends the physical page ranges for demotion to the DRM one by one, when it has an empty slot on the DRM. In addition to incorporating hardware support for demoting cache blocks, support could also be envisioned for dead region prediction in the Look-back scheme as the logic is similar to what is used in branch predictors. We plan to investigate this as part of our future work.

4. EXPERIMENTAL METHODOLOGY

We model a chip multi-processor (CMP) architecture with 8 cores as the baseline. The baseline CMP comprises a private L1 I/D cache, a unified L2 cache per core and a shared LLC. Inclusion is maintained across all levels of the cache. The private caches are kept coherent using a MESI write-invalidate coherence protocol. We use Sniper [10], a parallel multi-core x86 simulator, to model the baseline system using the architectural parameters in Table 1. Our simulations fast-

Core	8 cores, single-issue, x86 ISA
L1 Cache	Private Split, 32K L1 I & D 8-way, LRU, 1 cycle tag and 1 cycle data access
L2 Cache	Unified, 256K L2 8-way, LRU, 4 cycles tag and 10 cycles data access
L3 Cache	Shared, 8MB, 16-way, LRU, 10 cycles tag and 40 cycles data access
Coherence	MESI, write invalidate, 64-byte blocks
Main Memory	300 cycles
CDBP	64K history table
SDBP	3 tables, 8K history each
SHiP	2-bit SRRIP, 16K SHCT, SHiP-PC
NANOS	HT_{min} = 250, HT_{max} = 500, bf scheduler; plain dep. plugin

Table 1: Baseline Architecture.

Application	Input	# Tasks	# Insts.	# Cycles
			(Avg. per Task)	
cholesky	3072x3072 bsize:32	152096	271K	340K
gauss	1536x1536 bsize:32 iter:32	73728	76K	127K
jacobi	1536x1536 bsize:32 iter:32	73728	81K	134K
matmul	1536x1536 bsize:32	110592	277K	336K
redblack	1536x1536 bsize:32 iter:32	73728	91K	156K
sparselu	3072x3072 bsize:32	78448	265K	301K

Table 2: Application Parameters.

forward the initialization phase and then simulate the rest of the application to completion.

The applications we use, listed in Table 2, are from the Barcelona Application Repository[4]. These applications are parallelized using the OmpSs programming model, which like OpenMP 4.0, allows to specify inter-task dependencies using dependency annotations. 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 applications can be classified into two categories based on whether the dependency annotations represent the working set used by the task. In the case of *cholesky, matmul* and *sparselu* the annotations represent the working set precisely whereas for *gauss, jacobi* and *redblack* the annotation misrepresents (i.e. overspecifies) the working set used by the task. Finally, the tasks in the application are sized such that their working set fits in the L1 cache.

We use the Nanos++ runtime system [5] with the breadth first scheduling policy and the hysteresis throttling policy as the default unless specified otherwise. Our evaluation shows that the additional logic added by RADAR (classification of regions, tracking of region history and prediction of region accesses) increases the average runtime instruction count by less than 1.5K instructions per task. As shown in Table 2, this is a small fraction of the execution time of a single task.

Finally, to evaluate the impact of RADAR we use LLC misses as a metric to measure cache efficiency and the sum of execution time of tasks (i.e. work time) as metric to measure performance. We focus on work time because this lets us isolate the task management overheads and focus on time spent in the application. To compute LLC misses we only simulate accesses from the tasks in the application.

5. EVALUATION

The evaluation focuses on assessing the potential improvements that RADAR can provide in terms of cache efficiency and performance. In Section 5.1 we evaluate the effectiveness of the Look-back scheme to predict region misses. Next, in Section 5.2, we evaluate the impact of different schemes used by RADAR to predict dead regions. We then compare RADAR against state-of-the-art dynamic dead block predictors. Finally, in Section 5.3 we study the sensitivity of the approach to parameters of the execution environment.

[4] `https://pm.bsc.es/projects/bar`

	Single-level		Two-level	
	Coverage	**Accuracy**	**Coverage**	**Accuracy**
cholesky	89.9	96.6	67.5	95.5
gauss	6.9	100	85.1	100
jacobi	3.1	100	70.8	100
matmul	93.6	100	76.5	100
redblack	3.5	100	85.4	100
sparselu	22.3	63.2	0.48	3.5

Table 3: Accuracy and coverage for two different predictor strategies under the Look-back scheme.

5.1 Region Access Predictability

We quantify the *accuracy* and *coverage* of predictors in the Look-back scheme to evaluate their effectiveness at predicting region misses. We use only region misses to measure accuracy and coverage since RADAR benefits from accurate *region miss* predictions (no action is taken upon a region hit prediction). *Accuracy* is computed by dividing the total number of correct *region miss* predictions by the total number of *region miss* predictions made by the predictor. *Coverage* is computed by dividing the total number of correct *region miss* predictions by the actual number of *region misses*. Demoting the blocks of a predicted dead region may cause a region which would normally be reused (hit) to actually miss. Therefore, we study predictor accuracy and coverage without applying the dead region optimization.

Table 3 presents the prediction accuracy and coverage for region misses using the single-level bimodal and the two-level *Pap* strategies. The results indicate that single-level predictors are more effective than two-level predictors for *cholesky, matmul* and *sparselu*. Single-level bimodal predictors work effectively when there is a sequence of consecutive region hits or misses. In the cases of *cholesky and matmul* which have almost perfect accuracy and very high coverage we observe that *region misses* are clustered. In the case

of *sparselu*, the clusters are smaller and interspersed with *region hits* leading to low accuracy and coverage. This is due to the nature of sparse matrix algorithms, where NULL blocks may appear in any region of the matrix.

For other applications like *gauss, jacobi* and *redblack* two-level predictors are more effective than single-level predictors. The region miss patterns in these applications are repetitive[5] but not consecutive (e.g. a repeating pattern with one region miss followed by a few region hits). Therefore single-level predictors misbehave since they cannot build confidence (saturate the 2-bit counters) and they predict only for a small subset of the regions (low coverage). On the other hand, the history tables in two-level predictors can accurately capture and predict this behavior. Despite the predictable access pattern, two-level predictors have slightly lower coverage because training them requires several instances of classified region misses before they build good confidence. The learning curve for these predictors does not amortize well when the number of instances is small, which leads to less predictions and thus lower coverage.

5.2 RADAR Evaluation

5.2.1 RADAR Policies

We evaluate the different schemes employed by RADAR to detect dead regions: *i)* Look-ahead (LA), *ii)* Look-back (LB), *iii)* Aggressively Combined Look-ahead with Look-back LA∪LB (AS) and *iv)* Conservatively Combined Look-ahead with Look-back LA∩LB (CS). The runtime component of RADAR predicts, after each region access, if the region is dead using one of the aforementioned schemes. In case a region accessed by a task is predicted to be dead, RADAR signals the LLC to demote the cache blocks that belong to the dead region. In order to evaluate the effectiveness of these schemes we first compare them against the baseline LRU policy. Figure 7 shows LLC misses with different RADAR schemes normalized to the LRU baseline.

LA: We observe that LA reduces LLC misses for all applications. The average reduction in LLC misses is more than 23% when compared to the baseline. The heuristic used by LA performs well across most of the applications because look-ahead windows with several hundreds of tasks allow predicting future region reuses with reasonable accuracy. However, the effectiveness of LA is sensitive to the number of outstanding tasks in the look-ahead window, as detailed in Section 5.3.1.

LB: The results for LB indicate that it consistently outperforms LRU. The average improvement for LB is however smaller than LA. For some applications like *cholesky, gauss and jacobi* the difference between LA and LB is small. This is attributed to the fact that the region predictor for these applications exhibits good coverage and accuracy. In *sparselu* LB does not perform as well as LA because the predictor coverage and accuracy are not sufficiently high as discussed in Section 5.1. *Matmul* is an exception and the improvement offered by LB is not significant even though the predictor achieves almost perfect accuracy and coverage. The reduced benefit is attributed to the long reuse patterns found in *matmul* when it sweeps through the regions of the input matrices. LB classifies the accesses to these regions as misses and demotes them all, leading to reduced benefit.

[5]The repetitive behavior is due to the nearest-neighbor pattern observed in such stencil-based iterative applications.

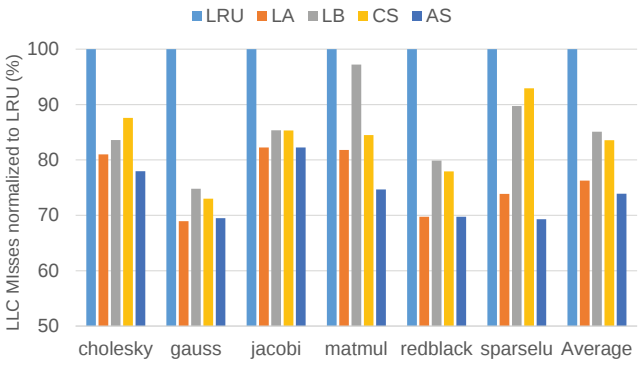

Figure 7: LLC misses for different RADAR policies normalized to the LRU baseline.

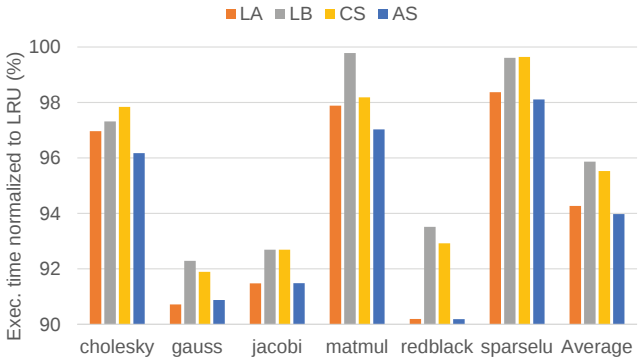

Figure 8: Execution time for different RADAR policies normalized to the LRU baseline.

AS: The aggressive combination is the best performing across all RADAR schemes. For *cholesky, matmul* and *sparselu* the results clearly show the synergistic effect, i.e. each scheme is able to detect some *region misses* that are not detected by the other scheme. Compared to LA, the average improvement in LLC misses is 5% for these three applications. AS does not provide any significant benefit over LA for *gauss, jacobi* and *redblack*. These applications feature predictable iterative behavior which LA and LB can already predict accurately. The average reduction in LLC misses for all applications is more than 26% when compared to the baseline.

CS: The conservative combination underperforms in comparison with the aggressive combination and it does not always benefit from synergistic behavior. The results closely track that of LB with the exception of *matmul*. In the case of *matmul*, CS outperforms LB because it leverages the future view of LA to avoid classifying all accesses to the input regions as misses.

Overall, we show that regardless of the prediction scheme, RADAR is beneficial for all the applications. Among the different schemes, AS performs best and achieves on average

more than 26% reduction in LLC misses over the baseline LRU. Finally, we observe an advantageous synergistic effect when combining LA with LB, thus making a case for combining static and dynamic information about future and past region accesses to predict dead regions accurately.

Figure 8 presents the execution time for different RADAR policies normalized to the LRU baseline. Across all the RADAR policies we observe similar trends as seen previously for the LLC miss rate. However, for *cholesky, matmul* and *sparselu* large improvements in miss rate translate to small improvements in performance because of the compute intensive nature of these applications. The average improvement for *gauss, jacobi* and *redblack* is over 9% since these applications are memory intensive. For the rest of the text, AS is used as the RADAR policy of choice to detect dead regions, unless specified otherwise.

5.2.2 *RADAR vs. Dynamic Dead Block Predictors*

In this section, we compare RADAR against three state-of-the-art techniques: count-based dead block predictor (CDBP) [27], sampling-based dead block predictor (SDBP) [26] and signature-based insertion (SHiP) [44]. We evaluate an updated version of CDBP as proposed by Liu et al. [31] that addresses inefficiencies in the original proposal. The configuration for different techniques is provided in Table 1. Figure 9 uses LLC misses (normalized to LRU baseline) as a metric to compare RADAR against the different techniques.

The results for CDBP in the case of *gauss, jacobi, redblack* and *sparselu* indicate a reduction in LLC misses over LRU. However, *cholesky* and *matmul* show substantial increase in LLC misses. In the case of SDBP *redblack* and *sparselu* alone show reduction in LLC misses comparable to CDBP. SHiP is the best performing among the three techniques we evaluated. Even SHiP experiences an increase in LLC misses over baseline in the case of *matmul*. This increase is caused by untimely demotion of cache blocks. If a cache block gets evicted from the cache before the task that accesses the cache block finishes execution, it may need to be fetched again from memory. This has a detrimental effect on the cache miss rate as shown in Figure 9.

RADAR is immune to this problem since it triggers demotions only after a task finishes execution and thus all accesses to the region have been already performed. Moreover, for the applications for which state-of-the-art techniques reduce the number of LLC misses, RADAR is on average at least 10% better. This shows that RADAR is more effective than existing techniques at managing dead blocks. Figure 10 presents the execution time for the evaluated techniques and RADAR normalized to the LRU baseline. The results show that RADAR consistently outperforms state-of-the-art dynamic dead block predictors.

5.2.3 *Energy Analysis*

We analyze the impact of RADAR on the dynamic energy consumption of the LLC and the DRAM by counting all the LLC (tag, data) and DRAM accesses. Moreover, our energy model incorporates the additional LLC tag accesses made by RADAR to demote cache lines. For obtaining the LLC access energy we use CACTI 6.3 [34] and for DRAM access energy we use DRAM power calculator from Micron.

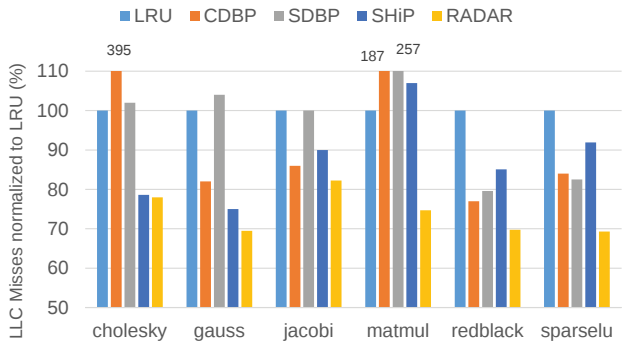

Figure 9: LLC misses for RADAR and state-of-the-art dynamic dead block prediction techniques normalized to the LRU baseline.

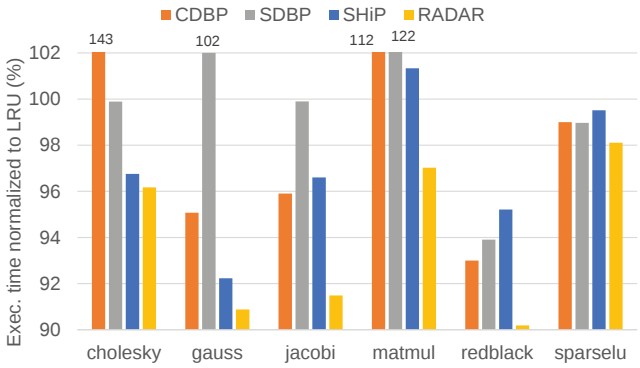

Figure 10: Execution time for RADAR and state-of-the-art dynamic dead block prediction techniques normalized to the LRU baseline.

In Figure 11 we compare the dynamic energy consumption of LLC and DRAM in RADAR against the LRU baseline. Our results suggest that RADAR reduces the dynamic energy consumption in LLC and DRAM (combined) by more than 25% on average owing to the significant reduction in the LLC misses which in turn reduces traffic towards the off-chip DRAM. Although RADAR increases the number of LLC tag accesses, its impact is negligible on the overall dynamic energy consumption since DRAM and LLC data accesses consume the bulk of energy as shown in the figure.

5.3 Impact of Execution Environment

5.3.1 *Impact of Look-ahead Throttling*

As introduced previously, the Look-ahead (LA) scheme uses the look-ahead window (that comprises ready tasks that have not yet executed and tasks that have unresolved dependencies) in order to make predictions about dead regions. Since RADAR utilizes the look-ahead window, we investigate the impact of varying the window size. We use the *hysteresis throttling policy* of the Nanos++ runtime system to adjust the look-ahead window. The hysteresis mecha-

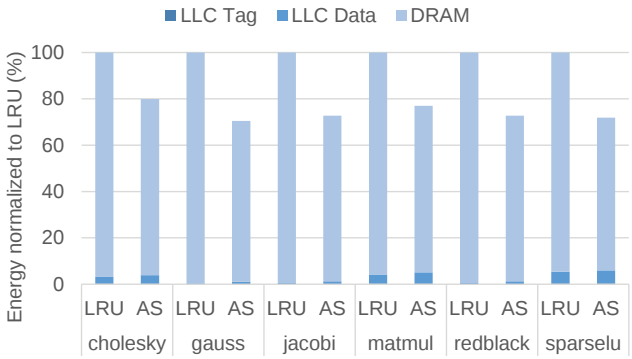

Figure 11: Dynamic energy consumption normalized to the LRU baseline.

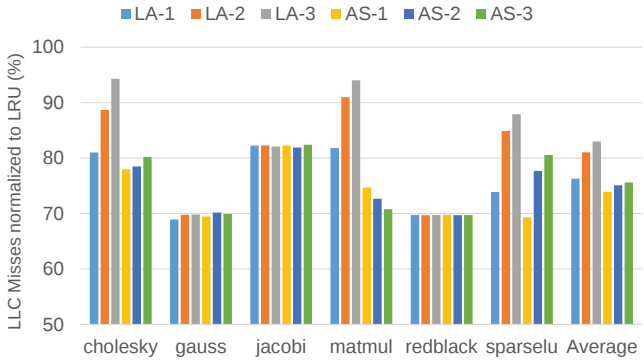

Figure 12: LLC misses using different RADAR schemes with varying look-ahead window size.

nism specifies a minimum (HT_{min}) and a maximum (HT_{max}) threshold for the window size. These thresholds specify a limit on the number of outstanding tasks in the look-ahead window after which the master thread stops generating new tasks and starts executing ready tasks. The number of tasks in the look-ahead window has to be chosen carefully to ensure that the available parallelism of the application is not artificially limited. However, a very large number of tasks in the look-ahead window increases the runtime and task management overheads. We use the default values specified by the runtime for all our previous experiments. The threshold values used for the hysteresis mechanism are specified in Table 1. Figure 12 presents LLC misses normalized to the LRU baseline for LA (LA-1) and AS (AS-1) variants with different look-ahead window sizes. For these experiments we scale the HT_{max} threshold by a factor of two (LA-2, AS-2) and three (LA-3, AS-3).

LA variants: We can observe that for the LA variants, the LLC misses increase as we scale the look-ahead window size for *cholesky, matmul* and *sparselu*. The increase in LLC misses is due to the smaller number of *region miss* predictions induced by a large look-ahead window. A large look-ahead window reduces the number of dead region predictions because it captures distant reuses even if they actually belong to different generations. The difference in LLC misses between LA-1 and LA-2 is higher than the difference between LA-2 and LA-3. The reason is that the master thread cannot easily fill very large look-ahead windows. Additionally, we notice that *gauss, jacobi* and *redblack* are immune to changes in the look-ahead window size. This is because distant reuse of regions only happens between tasks in consecutive phases (iterations). However the phase-oriented nature of these applications inhibits the master from observing such tasks across phase boundaries.

AS variants: For *cholesky* and *matmul* the AS variants are able to hide the impact of scaling the look-ahead window size. This is because AS leverages on the Look-back scheme to accurately predict region misses in the cases where the Look-ahead scheme is not efficient. In the case of *sparselu* AS variants are sensitive to the look-ahead window size – although they are impacted to a smaller extent when compared to LA variants – because in this benchmark the predictors of the Look-back scheme do not perform well in terms of accuracy and coverage as we have seen in Section 5.1.

5.3.2 *Impact of Prefetching*

Task data-flow applications typically have regular access patterns that makes them amenable to prefetching. To evaluate the impact of RADAR in the presence of prefetching we extend the baseline with L2 *stream prefetchers* which can detect strides of varying length and issue requests in advance. The prefetchers are trained by L2 misses and by hits to lines inserted by the prefetcher. Each L2 *stream prefetcher* can track up to eight independent streams and has a prefetch degree of four (without crossing page boundaries).

Since prefetching hides most of the memory latency in these applications, there is very small room for further performance improvement using RADAR. However, RADAR still reduces LLC misses considerably in the presence of prefetching as shown in Figure 13. This is because latency hiding optimizations like prefetching can hide the impact of memory access latency but do not improve LLC efficiency. RADAR on the other hand improves the LLC efficiency by identifying and evicting dead blocks. This improvement in LLC efficiency has a strong influence on dynamic energy consumption since DRAM accesses consume much more energy than LLC accesses (as shown in Figure 11). Consequently, RADAR provides over 20% reduction (on average) in dynamic energy in the presence of prefetching. This can directly be attributed to the reduction in DRAM accesses using RADAR.

5.3.3 *Impact of Scheduling Policies*

The scheduling policy can also affect data reuse characteristics of regions because they influence the way that tasks are scheduled in the application. Consequently, application behavior may vary significantly under different scheduling policies. We study the sensitivity of RADAR to different scheduling policies in Figure 14. We compare the LLC misses for different scheduling policies using RADAR and normalize them to the LRU baseline.

In addition to the default breadth first (BF) scheduling policy, we evaluate the distributed breadth first (DBF) and work first (WF) scheduling policies. RADAR performs equally

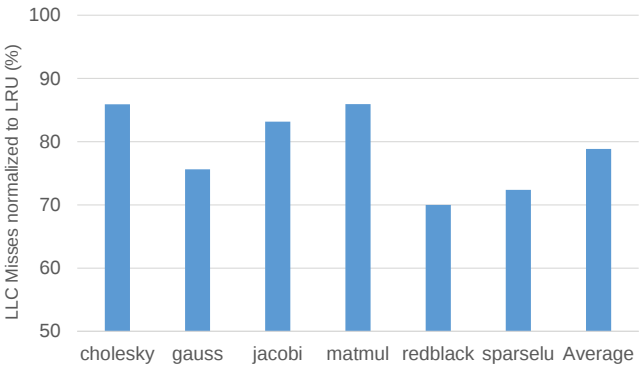

Figure 13: LLC misses using RADAR vs. LRU baseline with prefetching.

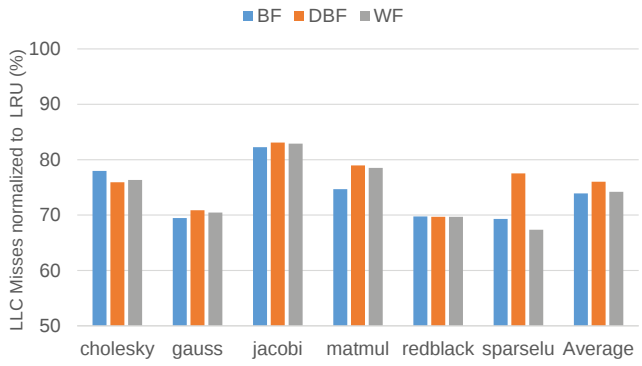

Figure 14: LLC misses using RADAR vs. LRU baseline with different scheduling policies.

well with all schedulers and consistently improves the LLC miss rates. We only observe small differences between the policies. These results confirm the generality and versatility of the approach adopted by RADAR.

6. RELATED WORK

6.1 Dead Block Optimizations

Several hardware techniques have been proposed to improve cache efficiency by identifying and evicting dead blocks. They identify dead blocks by either counting the number of times a block is accessed before eviction [27], by tracking the last PC or trace of instructions to access a block before eviction [26, 30], by tracking the last time a block was accessed [24], observing L2 access pattern [11] or by using a combination of approaches [1, 23, 31]. These schemes observe past eviction patterns and then use it to predict when blocks become dead. Related work has shown some of these proposals are not applicable in the context of LLC [31] because, unlike L1 caches, accesses to LLC are filtered.

Another line of work exploits the observation that a large fraction of blocks tend to be accessed just once before eviction. Several techniques have been proposed attempting to improve cache efficiency by identifying and evicting blocks accessed once (single use) early. Replacement optimizations identify single use blocks and insert them into the cache with low priority so that they are evicted early [20, 37, 44]. A few others identify single use blocks and bypass the LLCs by installing them in the upper cache levels to improve utilization [21, 22, 38]. Other approaches have also been proposed that identify dead blocks by predicting reuse distances dynamically and using them to guide cache replacement [25]. Lastly, specific approaches have been proposed in the context of inclusive caches and exclusive caches to improve cache efficiency by reducing back-invalidations [19, 46] or enabling bypassing [14]. RADAR is different from all these hardware based proposals because RADAR exploits static information made available to the runtime system in task data-flow programming models in conjunction with region access history to detect when address regions become dead. RADAR only requires architectural support to demote cache blocks that belong to the dead region.

Compiler-based schemes proposed in the context of single threaded applications can make use of semantic information in the application [6, 7, 18, 40, 42]. These schemes focus on array accesses inside loops and utilize reuse distance analysis or its refinements to identify the last accesses to a data region before eviction. A major shortcoming of compiler-based approaches is that they cannot adapt to changes in the task schedule typical of dynamically scheduled parallelism.

In the context of managed applications a few optimizations that exploit semantic information about data accesses available to the runtime system have been proposed [9, 39]. Pointy [9] uses object connectivity information available to the runtime to initiate timely prefetches but does not manage dead blocks. Cache Scrubbing [39] exploits the observation that a large fraction of accesses and writebacks to DRAM happen due to temporary objects. This technique utilizes semantic information about when objects die and uses it to avoid wasteful writebacks. While cache scrubbing detects the end of an object lifetime, RADAR learns about the liveness of address regions and uses this knowledge to evict cache lines at the end of each region generation. This allows RADAR to optimize the cache even for regions whose lifetime spans multiple generations.

6.2 Task data-flow Optimizations

Since task data-flow programming models make application semantic information available to the runtime system they can be used to enable several optimizations. A recent proposal makes a case for designing architectures aware of such runtime systems to address several programmability, memory and reliability issues [41]. Our proposal shares the same vision that either the entire runtime system or certain features of it should be part of the chip to facilitate better information exchange across the layers. Task Superscalar proposed moving the runtime system to hardware in order to improve scalability of the runtime system [13].

Some proposals leverage dependency annotations to increase prefetching effectiveness and reduce cache pollution due to prefetching [36, 29]. These annotations have also been used to identify sharing patterns and enable coherence optimizations [33, 32]. RADAR is orthogonal to these pro-

posals as we specifically improve the efficiency of the LLC by managing dead regions effectively.

An alternative strategy that exploits semantic information from the programming model is explicitly managed scratchpad memory [2, 3, 28]. Although explicitly managed scratchpad memory has the potential to be more efficient in terms of reducing coherence overheads, reducing access energy and hiding memory latency by double buffering, they cannot be easily applied to LLCs because they act as a caching substrate for the memory subsystem and are usually shared between different cores and the workloads running on them. The optimizations proposed in RADAR for improving cache efficiency only modify replacement decisions and thus cannot influence correctness. Some proposals reassign a part of the cache to act as a scratchpad and achieve latency improvement on top by performing bulk prefetching and bulk DRAM writebacks [15]. However, such proposals do not manage dead blocks effectively. RADAR can be integrated with such proposals to improve LLC efficiency.

7. CONCLUSION

Effective management of last-level caches (LLCs) in multicore architectures is essential for performance and energy efficiency. Unfortunately, LLCs tend to be used rather inefficiently due to the large fraction of dead blocks that occupy precious cache space without providing any potential benefit. Earlier attempts to identify and evict dead blocks have for the most part used only past access patterns to predict future access by gathering information at the architecture level.

This paper contributes with a new cache management technique taking a radically different approach. The RADAR framework, proposed in this paper, comprises a runtime system that collects static as well as dynamic information to make informed cache management decisions, in particular applied to dead-block prediction. RADAR leverages the notion of address regions specified by task-parallel programming models to define dependency relations between tasks. Static information about region dependencies available in existing runtime systems is one source of information exploited by RADAR. Dynamic information collected at the architecture level about region reuse is another source made available to RADAR.

RADAR, builds on this and contributes with three schemes: 1) The Look-ahead scheme 2) The Look-back scheme and 3) The combined Look-ahead and Look-back scheme. The Look-ahead scheme, uses information in the dynamically constructed data-flow graph to establish blocks that will be accessed in future. By contrast, the Look-back scheme records the past pattern of region accesses and uses it to predict future accesses. Because of their complementary perspectives, this paper contributes with a combined scheme that exploits both approaches.

This study shows that the RADAR approach of collecting static as well as dynamic information yields substantially higher accuracy in predicting dead blocks than what has been achieved in prior work. In fact, this paper shows that all RADAR enabled schemes outperform dead-block schemes proposed in prior art. RADAR shows a novel direction as to how static and dynamic information can be used at the runtime system level to improve cache management.

Acknowledgement

We thank Angelos Arelakis, Bhavishya Goel and Risat Pathan for their feedback on this work. We also acknowledge the anonymous reviewers for their insightful comments and their feedback on improving the quality of the paper. This research was supported by grants from the Swedish Foundation for Strategic Research (SSF) under the SCHEME project (RIT10-0033) and the European Research Council (ERC) under the MECCA project (contract 340328). The simulations were run on the resources provided by the Swedish National Infrastructure for Computing (SNIC) at C3SE.

8. REFERENCES

[1] Jaume Abella, Antonio González, Xavier Vera, and Michael F. P. O'Boyle. Iatac: A smart predictor to turn-off l2 cache lines. *ACM Trans. Archit. Code Optim.*, 2(1):55–77, March 2005.

[2] Lluc Alvarez, Miquel Moretó, Marc Casas, Emilio Castillo, Xavier Martorell, Jesús Labarta, Eduard Ayguadé, and Mateo Valero. Runtime-guided management of hybrid memory hierarchies in multicore architectures. In *Proceedings of the 24th international conference on Parallel architectures and compilation, San Jose, CA, USA, October 18-21, 2015*, 2015.

[3] Lluc Alvarez, Lluís Vilanova, Miquel Moretó, Marc Casas, Marc González, Xavier Martorell, Nacho Navarro, Eduard Ayguadé, and Mateo Valero. Coherence protocol for transparent management of scratchpad memories in shared memory manycore architectures. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*, pages 720–732, 2015.

[4] Abdelhalim Amer, Naoya Maruyama, Miquel Pericàs, Kenjiro Taura, Rio Yokota, and Satoshi Matsuoka. Fork-join and data-driven execution models on multi-core architectures: Case study of the fmm. In *Supercomputing*, pages 255–266. Springer, 2013.

[5] Barcelona Supercomputing Center. NANOS++ runtime system. `https://pm.bsc.es/nanox`.

[6] Kristof Beyls and Erik H D'Hollander. Reuse distance-based cache hint selection. In *Euro-Par 2002 Parallel Processing*, pages 265–275. Springer, 2002.

[7] Kristof Beyls and Erik H D'Hollander. Generating cache hints for improved program efficiency. *Journal of Systems Architecture*, 51(4):223–250, 2005.

[8] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of PPoPP '95*. ACM, July 1995.

[9] Ioana Burcea, Livio Soares, and Andreas Moshovos. Pointy: A hybrid pointer prefetcher for managed runtime systems. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 97–106, New York, NY, USA, 2012. ACM.

[10] Trevor E Carlson, Wim Heirman, and Lieven Eeckhout. 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*, page 52. ACM, 2011.

[11] Mainak Chaudhuri, Jayesh Gaur, Nithiyanandan Bashyam, Sreenivas Subramoney, and Joseph Nuzman. Introducing hierarchy-awareness in replacement and bypass algorithms for last-level caches. In *International Conference on Parallel Architectures and Compilation Techniques, PACT '12, Minneapolis, MN, USA - September 19 - 23, 2012*, pages 293–304, 2012.

[12] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

[13] Yoav Etsion, Felipe Cabarcas, Alejandro Rico, Alex Ramirez, Rosa M. Badia, Eduard Ayguade, Jesus Labarta, and Mateo Valero. Task superscalar: An out-of-order task pipeline. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on*

Microarchitecture, MICRO '43, pages 89–100, Washington, DC, USA, 2010. IEEE Computer Society.

[14] Jayesh Gaur, Mainak Chaudhuri, and Sreenivas Subramoney. Bypass and insertion algorithms for exclusive last-level caches. In *38th International Symposium on Computer Architecture (ISCA 2011), June 4-8, 2011, San Jose, CA, USA*, pages 81–92, 2011.

[15] Jayanth Gummaraju, Mattan Erez, Joel Coburn, Mendel Rosenblum, and William J. Dally. Architectural support for the stream execution model on general-purpose processors. In *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007), Brasov, Romania, September 15-19, 2007*, pages 3–12, 2007.

[16] INRIA. StarPU handbook 1.2. `http://starpu.gforge.inria.fr/doc/starpu.pdf`.

[17] Intel. Thread building blocks 4.3. `https://www.threadingbuildingblocks.org/docs/help/reference/flow_graph.htm`.

[18] Prabhat Jain, Srinivas Devadas, Daniel Engels, and Larry Rudolph. Software-assisted cache replacement mechanisms for embedded systems. In *Proceedings of the 2001 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '01, pages 119–126, Piscataway, NJ, USA, 2001. IEEE Press.

[19] Aamer Jaleel, Eric Borch, Malini Bhandaru, Simon C. Steely Jr., and Joel S. Emer. Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (TLA) cache management policies. In *43rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2010, 4-8 December 2010, Atlanta, Georgia, USA*, pages 151–162, 2010.

[20] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. High performance cache replacement using re-reference interval prediction (rrip). In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 60–71, New York, NY, USA, 2010. ACM.

[21] Jonas Jalminger and Per Stenström. A novel approach to cache block reuse predictions. In *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, pages 294–302. IEEE, 2003.

[22] Teresa L. Johnson, Daniel A. Connors, Matthew C. Merten, and Wen-mei W. Hwu. Run-time cache bypassing. *IEEE Trans. Comput.*, 48(12):1338–1354, December 1999.

[23] Martin Kampe, Per Stenstrom, and Michel Dubois. Self-correcting lru replacement policies. In *Proceedings of the 1st Conference on Computing Frontiers*, CF '04, pages 181–191, New York, NY, USA, 2004. ACM.

[24] Stefanos Kaxiras, Zhigang Hu, and Margaret Martonosi. Cache decay: Exploiting generational behavior to reduce cache leakage power. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ISCA '01, pages 240–251, New York, NY, USA, 2001. ACM.

[25] Georgios Keramidas, Pavlos Petoumenos, and Stefanos Kaxiras. Cache replacement based on reuse-distance prediction. In *Computer Design, 2007. ICCD 2007. 25th International Conference on*, pages 245–250. IEEE, 2007.

[26] Samira Manabi Khan, Yingying Tian, and Daniel A. Jimenez. Sampling dead block prediction for last-level caches. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 175–186, Washington, DC, USA, 2010. IEEE Computer Society.

[27] Mazen Kharbutli and Yan Solihin. Counter-based cache replacement and bypassing algorithms. *IEEE Trans. Comput.*, 57(4):433–447, April 2008.

[28] Rakesh Komuravelli, Matthew D. Sinclair, Johnathan Alsop, Muhammad Huzaifa, Maria Kotsifakou, Prakalp Srivastava, Sarita V. Adve, and Vikram S. Adve. Stash: Have your scratchpad and cache it too. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 707–719, 2015.

[29] Costas Kyriacou, Paraskevas Evripidou, and Pedro Trancoso. Cacheflow: A short-term optimal cache management policy for data driven multithreading. In *Euro-Par 2004 Parallel Processing*, pages 561–570. Springer, 2004.

[30] An-Chow Lai, Cem Fide, and Babak Falsafi. Dead-block prediction and dead-block correlating prefetchers. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ISCA '01, pages 144–154, New York, NY, USA, 2001. ACM.

[31] Haiming Liu, Michael Ferdman, Jaehyuk Huh, and Doug Burger. 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 41, pages 222–233, Washington, DC, USA, 2008. IEEE Computer Society.

[32] Madhavan Manivannan, Anurag Negi, and Per Stenstrom. Efficient forwarding of producer-consumer data in task-based programs. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 517–522, Oct 2013.

[33] Madhavan Manivannan and Per Stenstrom. Runtime-guided cache coherence optimizations in multi-core architectures. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 625–636. IEEE, 2014.

[34] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. Cacti 6.0: A tool to model large caches. *HP Laboratories*, pages 22–31, 2009.

[35] OpenMP Architecture Review Board. OpenMP application program interface version 4.0, 2013.

[36] Vassilis Papaefstathiou, Manolis G.H. Katevenis, Dimitrios S Nikolopoulos, and Dionisios Pnevmatikatos. Prefetching and cache management using task lifetimes. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 325–334. ACM, 2013.

[37] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 381–391, New York, NY, USA, 2007. ACM.

[38] Jude A. Rivers, Edward S. Tam, Gary S. Tyson, Edward S. Davidson, and Matt Farrens. Utilizing reuse information in data cache management. In *Proceedings of the 12th International Conference on Supercomputing*, ICS '98, pages 449–456, New York, NY, USA, 1998. ACM.

[39] Jennifer B Sartor, Wim Heirman, Stephen M Blackburn, Lieven Eeckhout, and Kathryn S McKinley. Cooperative cache scrubbing. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 15–26. ACM, 2014.

[40] Jennifer B Sartor, Subramaniam Venkiteswaran, Kathryn S McKinley, and Zhenlin Wang. Cooperative caching with keep-me and evict-me. In *Interaction between Compilers and Computer Architectures, 2005. INTERACT-9. 9th Annual Workshop on*, pages 46–57. IEEE, 2005.

[41] Mateo Valero, Miquel Moreto, Marc Casas, Eduard Ayguade, and Jesus Labarta. Runtime-aware architectures: A first approach. *Supercomputing frontiers and innovations*, 1(1), 2014.

[42] Zhenlin Wang, Kathryn S. McKinley, Arnold L. Rosenberg, and Charles C. Weems. Using the compiler to improve cache replacement decisions. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, PACT '02, pages 199–, Washington, DC, USA, 2002. IEEE Computer Society.

[43] David A. Wood, Mark D. Hill, and R. E. Kessler. A model for estimating trace-sample miss ratios. In *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '91, pages 79–89, New York, NY, USA, 1991. ACM.

[44] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C. Steely, Jr., and Joel Emer. Ship: Signature-based hit predictor for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 430–441, New York, NY, USA, 2011. ACM.

[45] Tse-Yu Yeh and Yale N Patt. Two-level adaptive training branch prediction. In *Proceedings of the 24th annual international symposium on Microarchitecture*, pages 51–61. ACM, 1991.

[46] Mohamed Zahran. Cache replacement policy revisited. In *The Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD) held in conjunction with the International Symposium on Computer Architecture (ISCA)*, 2007.