Runtime-Assisted Shared Cache Insertion Policies Based on Re-reference Intervals

Vladimir Dimić[1,2(✉)], Miquel Moretó[1,2], Marc Casas[1,2], and Mateo Valero[1,2]

[1] Barcelona Supercomputing Center (BSC), Barcelona, Spain
{vladimir.dimic,miquel.moreto,marc.casas,mateo.valero}@bsc.es
[2] Universitat Politécnica de Catalunya, Barcelona, Spain

Abstract. Processor speed is improving at a faster rate than the speed of main memory, which makes memory accesses increasingly expensive. One way to solve this problem is to reduce miss ratio of the processor's last level cache by improving its replacement policy. We approach the problem by co-designing the runtime system and hardware and exploiting the semantics of the applications written in data-flow task-based programming models to provide hardware with information about the task types and task data-dependencies. We propose the Task-Type aware Insertion Policy, TTIP, which uses the runtime system to dynamically determine the best probability per task type for bimodal insertion in the recency stack and the static Dependency-Type aware Insertion Policy, DTIP, that inserts cache lines in the optimal position taking into account the dependency types of the current task. TTIP and DTIP perform similarly or better than state-of-the-art replacement policies, while requiring less hardware.

Keywords: Shared cache · Replacement policy · Runtime system · Task-based programming model · Hardware-software co-design

1 Introduction

Throughout the last decades, main memory performance has been improving with a slower rate than the performance of CPUs, which has been described as the *Memory Wall* [28]. Misses happening in last level caches (LLC) result in memory accesses, which cause CPU to wait for the data. Non-blocking caches try to mitigate this problem by being able to serve several outstanding misses, but they cannot hide the memory latency in all cases. One way to approach this problem is to reduce the miss rate of the LLC. Optimizing the LLCs usage is a complex problem and requires identifying the important factors that impact its performance.

The access pattern of an application together with the memory hierarchy configuration (i.e. cache size, associativity, replacement policy, etc.) are some of these factors. Most commonly used applications can have several fundamental access patterns. Memory accesses with high spatial and temporal locality are

F.F. Rivera et al. (Eds.): Euro-Par 2017, LNCS 10417, pp. 247–259, 2017.
DOI: 10.1007/978-3-319-64203-1_18

cache friendly and usually have good hit rates. Streaming access patterns are characterised by sequential or strided access of vectors in memory. In general, they have no reuse and, therefore, the choice of the cache placement policy has a low impact on the miss rate. Thrashing access patterns are the ones that have reuse distances bigger than the cache associativity. Repeated, circular, accesses to the same sequence of addresses cause the circular eviction of the cache lines, thus making all accesses resulting in misses. Many applications show more complex access patterns that are a combination of the simple ones. Choosing an appropriate replacement policy is important for achieving good performance when executing these applications.

The majority of modern CPUs are multi-core and have a multi-level cache hierarchy with shared LLC. The sequences of memory accesses coming from threads executing on different cores arrive into the LLC. If the LLC uses a replacement policy that does not take into account the priority of certain access patterns, such as LRU, accesses generated by a thread may trash the working set of another. Giving more priority to the lines of the trashed thread will improve its performance while not hurting the performance of the thrashing thread. There are several state-of-the-art replacement policies designed by taking access pattern priorities into account, such as DIP [19], DRRIP [9] and SHiP [27]. However, they do not consider application semantics, which can give useful information about access patterns for different memory regions and different code segments in the application. Using this information in designing a replacement policy for LLC can bring benefits in performance.

Applications written in a task-based programming model can provide more information about the semantics. Many programming models use the notion of task as a unit of work, such as OpenMP 4.0 [15], Cilk [5], Chapel [3], Intel TBB [21], Charm++ [11] and OmpSs [25]. In task-based data-flow programming models, task data-dependencies are used for synchronisation, meaning that tasks consuming a dependency cannot start executing before the task producing that dependency finishes. Special directives, inserted by programmers, instruct the compiler how to parallelise the code. The compiler translates these directives to calls to the runtime library, which manages the execution of the application.

Tasks perform different functions and access their dependencies in different ways. We argue that using information about task types and dependency types can help designing better replacement policies in the LLC. To approach the problem, we use a recently proposed idea [4,26], where architectures and runtime systems collaborate in order to achieve better performance in modern and future computer systems. We exploit application semantics available in the runtime system and provide the processor with the necessary information to optimise the behaviour of the LLC. In this paper, we propose two insertion policies that utilise information about task types (TTIP) and task data-dependency types (DTIP), which are described in more detail in Sect. 3. TTIP dynamically determines the best probability per task-type for probabilistic insertion achieving 5.1% and 0.8% better execution time than LRU and DRRIP, respectively. DTIP statically assigns different insertion positions based on data-dependency type and is faster 4.8% and 0.3% than LRU and DRRIP, respectively.

2 Related Work

Cache replacement policies are a set of algorithms that maintain the logical order of cache lines inside a cache set. Insertion policies determine the position of the line in the logical queue on its insertion into the cache. Promotion policies update the line's position when it is accessed. Eviction policies decide which line to remove from the cache when a space for a new line is needed.

The optimal replacement policy, the Belady's MIN algorithm [1], evicts the line that is going to be referenced farthest in the future. It is unusable in real systems because it requires the knowledge of the future. Least-Recently Used (LRU) policy evicts the line that was used farthest in the past, while on insertion and promotion lines are moved to the top of the LRU stack. The cost of maintaining LRU states is increasing with set size, so it is not used in caches with high associativity (LLC). Pseudo LRU policies [7,14] sacrifice precision while simplifying the state management. On average, they perform similarly to LRU in caches with high associativity.

LRU performs poorly with scanning and thrashing workloads. Qureshi et al. [19] propose several insertion policies that try to reduce thrashing. Bimodal Insertion Policy (BIP) inserts a tunable percentage of lines in the LRU position, and the rest of the lines in the Most-Recently Used (MRU) position. LRU and BIP outperform each other in different cases, so Dynamic Insertion Policy (DIP) chooses the best of the two via sampling-based adaptive replacement [20].

Jaleel et al. [9] propose a cache replacement policy that uses Re-Reference Interval Prediction (RRIP) in order to prevent cache pollution by lines that are not going to be referenced for a long time. For managing the logical order of the lines, two bits per cache line are used to encode 4 different states: 00 (*immediate*), 01, 10 (*long*) and 11 (*distant*). Multiple lines can be in the same state. On a cache hit, the accessed line is promoted to *immediate* position. On eviction, a line with *distant* state is removed. If no such line exists, states of all lines are increased by one until at least one of the lines is in *distant* position. On insertion, a line is assigned a position that depends on the insertion policy.

Static RRIP (SRRIP) always assigns *long* re-reference interval to new lines, which protects lines with shorter re-reference interval from being evicted by scanning access patterns. In thrashing workloads, SRRIP performs poorly. Bimodal RRIP (BRRIP) solves this by inserting the majority of new lines with *distant*, and the rest with *long* re-reference interval. Both policies implement promotion and eviction similarly to NRU [14]. Dynamic RRIP (DRRIP) uses set dueling [18] to dynamically select the best performing policy. Finally, Wu et al. [27] propose Signature-based Hit Predictor (SHiP) that extends RRIP by predicting the re-reference interval of an incoming cache line based on its history. Cache line's PC-based signature is used for tracking the history of hits.

There have been several proposals that use the runtime system of task-based data-flow programming models to optimize the LLC performance. Papaefstathiou et al. [17] propose a prefetching scheme in which the runtime provides hardware with information about task data-dependencies. To minimize cache pollution, quotas are assigned to current and future tasks based

on their footprint. The current task is given the highest priority, thus keeping its lines in cache. Manivannan and Stenstrom [13] propose cache coherence protocol, in which the runtime system exposes to the hardware which are the lines that will be reused. The coherence protocol can then reduce coherence traffic by invalidating or downgrading lines precisely. Pan and Pai [16] propose a runtime-assisted cache partitioning technique. Runtime knowledge about task inter-dependencies and future tasks is used in order to preserve useful data in the cache, while removing the data that will not be re-referenced in the future.

RADAR [12], a runtime-assisted scheme for dead-block management, consists of two independent algorithms for dead-block prediction, which are combined to give the better final algorithm. The first, *Look-ahead scheme*, uses information about task dependencies and current state of the task dependency graph to determine whether certain blocks of data will reused. The second, *Look-back scheme*, uses previous outcomes of cache accesses to estimate whether certain cache lines will be dead.

3 Runtime-Assisted Insertion Policies in the LLC

An important factor that affects the LLC performance is the memory access pattern. Runtime systems that support task-based data-flow programming models have information about task types and task data-dependencies of the application. We aim to utilise this information at the hardware level to improve the LLC performance by optimizing its insertion policy.

3.1 Task Type Aware Probabilistic Insertion

BRRIP is designed to overcome thrashing in access patterns with longer re-reference interval than the cache associativity. It achieves so by selecting the new line's insertion position in the recency stack based on a pre-determined static probability.

If multiple access patterns, including thrashing ones, meet in the LLC, it is beneficial to assign them different insertion probabilities. We assign probabilities per task type, thus giving them different priorities in the LLC. A higher probability means inserting more lines into the *long* position, so the lines have more chances to be preserved in cache. Tasks that show more locality in their accesses should get a higher insertion probability than the tasks having scanning access patterns. Moreover, using a larger set of probabilities instead of a fixed one gives more opportunities to tune the tasks' performance in complex scenarios where many different task types compete for the LLC resources. The optimal probability depends on the co-runners that share the LLC at the moment. In complex applications, instances of a given task type may execute with different co-runners in different phases of the application, which means that the optimal probability may change.

We develop a dynamic mechanism that aims to determine the best probability per task type during the execution of an application. The proposed mechanism alternates between two phases, training phase and stable phase, for each

task type independently. The goal of the training phase is to find the optimal probability for the given task type which is then used during the stable phase.

At the beginning of the application, all task types are set to run in the training phase. When a task instance is scheduled on a given core, the algorithm selects a probability from the pool of preselected probabilities $\mathcal{P}$ and instructs the LLC to use that probability for all accesses issued by this task instance. Upon completion of each task instance, the algorithm records the number of misses generated by that task with the selected probability. The same probability is used for $\boldsymbol{K}$ **task instances**. Probabilities are selected sequentially from the pool until all probabilities are evaluated. In total, $K \times |\mathcal{P}|$ task instances are used for training during one training phase for one task type. This concludes the training phase and the stable phase begins. The algorithm then selects the probability that induced lowest average number of misses and uses it for the next $\boldsymbol{N}$ **instances** of the given task type. After that, the whole process repeats until the end of the application's execution.

Consequently, TTIP is able to select the best probability parameter for a given task type appropriate for the current conditions in the LLC.

3.2 Dependency Type Aware Insertion

We reason in Sect. 1 that task data-dependencies show different access patterns. Input dependencies are read-only data useful for the current task instance. Output dependencies are generated by a task in order to be consumed by its successors in the task dependency graph. Therefore, it may be beneficial to insert cache lines belonging to outputs in higher positions of the recency stack, thus giving them more chances to stay in the cache until the moment they are required by the consumer task. A similar reasoning applies to dependencies denoted as *inouts*, as they are also inputs of a future task. Non-dependencies are the local variables in the call stack and the global variables that are not specified as task dependencies. In some of our benchmarks, like CG, they are predominantly accesses to large global variables that have streaming-like access patterns. In other benchmarks, where this is not true, decisions that we make for non-dependencies do not harm the performance.

We develop an insertion policy that inserts lines in positions based on which dependency type they belong to. We call this policy Dependency Type aware Insertion Policy (DTIP). The policy configuration can be formally defined as a function $f : \mathcal{DT} \rightarrow \mathcal{IP}$, where $\mathcal{DT} = \{input, output, inout, non\text{-}dependency\}$ and $\mathcal{IP} = \{immediate, long, distant\}$. To determine the impact of mapping dependency types to specific positions in the recency stack, we perform an exhaustive design space exploration where we try all possible functions f. Number of different policy configurations per benchmark is $|\mathcal{IP}|^{|\mathcal{DT}|} = 3^4 = 81$. For all benchmarks, we run 405 simulations. On average, the best performing policy is the one that inserts *inputs* and *non-dependencies* on the *distant* position in the recency stack and *outputs* and *inouts* on *long* or *immediate* positions. This is consistent with the intuitive expectations described above.

3.3 Implementation

In this section, we describe the hardware and runtime extensions necessary for implementation of our policies. The cost of their implementation is discussed in detail in Sect. 4.5.

Hardware Extensions. To be able to use different insertion probabilities for different task types, TTIP requires a small and fast hardware structure in the LLC that maps a hardware thread ID to the appropriate probability. It is designed as a SRAM memory containing probabilities and is addressed by the hardware thread ID, which is already required to enforce coherence in the LLC. The mapping table is accessed on a new miss, in parallel with creation of a new MSHR entry. The read probability from the table is used to calculate the insertion position, which is cached in the newly created MSHR entry. The runtime modifies the structure via memory-mapped registers. To track the performance under each probability, we use one hardware counter per hardware thread for misses to LLC. The counters are exposed to the runtime as a set of registers.

To identify the dependency type of an access, which is necessary for DTIP, we add a special hardware structure that stores the mappings of dependency regions to the dependency type for all running tasks. We assume that only one task is executing concurrently on any given hardware thread. If tasks are switched, the runtime or the operating system updates the mapping table with dependency regions of the new task. There might be several tasks using the same region at the same time, but the runtime scheduler inherently guarantees that the region will have the same dependency type in all these tasks.

The mapping table is read on every occurrence of a miss in the LLC to determine the dependency type of the missing line. The missing line's address is fed to the table, which simultaneously compares all stored region boundaries and selects the entry containing the dependency type corresponding to the region of the missed address. This is done in parallel with creating a new MSHR entry, thus not introducing any additional latency. The dependency type of the line is stored in the newly created MSHR entry. Upon serving the request from main memory, the new line is inserted into the position in recency stack determined by the stored dependency type.

Hardware structures of both TTIP and DTIP are centralised and located in the LLC. They are accessed by the core via special requests through the memory hierarchy. The requests are propagated to the LLC and do not change the contents of the private caches.

Runtime System Extensions. To implement TTIP, several runtime modifications are required. The runtime system contains a per-task data structure that tracks the performance in terms of number of misses for each probability. When a task starts, before its user code starts executing, the runtime decides which probability to use for that task instance and writes it in the probability table on the position specified by the core ID on which the task is scheduled to

run. At the end of execution of a task, the number of misses produced by that task in the LLC is read by the runtime and stored in the software data structure mentioned above.

DTIP requires several changes in the runtime system. When a new task starts executing on a core, the runtime system updates the mapping table with the information for the new task by issuing store instructions to the memory-mapped registers. This does not require changes to the ISA, since many modern processors have a support for memory-mapped registers. If there are several consecutive dependency regions of the same type, the runtime may perform two optimizations to reduce the storage requirements in the mapping table. The first optimization merges the consecutive dependency regions of the same dependency type into one. The second does not insert the region if it already exists in the table, which happens if two or more tasks are sharing the same region. Since the mapping table is not readable by the runtime to simplify the hardware design, the runtime keeps a software copy of the mapping information.

4 Evaluation

4.1 Simulation Infrastructure

We use TaskSim [23], a trace-driven [22] computer architecture simulator that simulates applications written in data-flow task-based programming models. The simulated system is a 4-core processor with a cache hierarchy consisting of 3 levels, two of which are private, L1 (4-way, 32 KB) and L2 (8-way, 256 KB), and the LLC is shared (16-way, 8 MB). All caches are write-back and write-allocate. Access latencies are 4, 10 and 24 cycles, respectively. Each cache can serve up to 16 outstanding misses and 4 write-back requests which are served when the bus is not in use. Private caches use LRU replacement policy. The size of a cache line in all caches is 64 B. Only memory instructions are simulated in detail while other instructions are simulated on a simple CPU model. Inter-dependencies of memory accesses are respected. The reorder buffer contains 128 entries. The main memory has a latency of 200 ns and a bandwidth of 2.4 GB/s per core.

4.2 Benchmarks

In order to evaluate our proposals in relevant scenarios, we use benchmarks that cover a wide range of modern applications and kernels used in HPC and show variability in task sizes and dependency types. PARSECSs [6] is a task-based implementation of widely-accepted benchmark suite of parallel applications, PARSEC [2]. Benchmarks that fulfil our requirements are facesim and ferret. We use *simlarge* input set, the largest input set suitable for simulation. Moreover, we use two HPC applications used in previous works [22,23], specfem3D and stap. The inputs are selected to balance between simulation time and LLC footprint. Finally, we use benchmark CG, a conjugate gradient method [24], implemented in OmpSs by Jaulmes et al. [10]. The input is the matrix *qa8fm* from The University of Florida Sparse Matrix Collection [8]. The algorithm is decomposed in 8 blocks and runs until convergence (97 iterations).

4.3 TTIP Parameters Space Exploration

TTIP's performance depends on two parameters K and N, which are described in Sect. 3.1. These parameters determine how many task instances per probability are used in training, and how many instances for running with the best probability in the stable phase. We explore the set of configurations (K, N) where $K \in \{1, 2, 4, 8, 16\}$ and $N \in \{10, 50, 100, 500, 1000, \infty\}$. Configurations where $N = \infty$ have only one training phase which is followed by one stable phase that lasts until the end of execution. Intuitively, choosing a larger K offers better precision by having more time to evaluate one probability. However, too large K can hurt the overall performance if certain probabilities perform badly. Configurations with larger N use the best probability for a longer period of time, but are less able to adapt to potential changes in application behaviour. Using a smaller N can be bad for the final performance because a larger percentage of the execution is spent in the training phase.

Figure 1 shows the performance of TTIP in terms of MPKI depending on the choice of parameters K and N. For most benchmarks except specfem3D we can observe a performance improvement as N increases. This is due to the fact that, in the majority of benchmarks, instances of the same task type have similar behaviour. For cg, we can notice the trend of performance degradation when increasing K for a constant N. Similar behaviour can be noticed for stap. Stap highly benefits from configurations where $N = \infty$ due to having a large number of task instances. Having many training phases in case of stap means repeatedly evaluating sub-optimal probabilities, thus hurting the overall performance. Ferret does not show significant sensitivity to K and N. Facesim obtains better performance with larger K due to having a lot of small task instances and, therefore, needing more instances per probability to properly evaluate the performance of each probability. The configuration that performs the best on average for all our benchmarks is $(N, K) = (\infty, 8)$, which we will use for further evaluation of TTIP in the remaining of the paper.

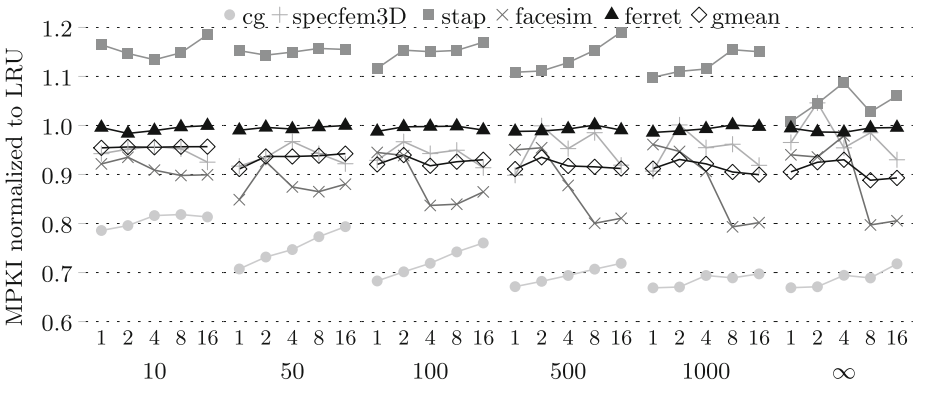

Fig. 1. TTIP sensitivity to $N \in \{10, 50, 100, 500, 1000, \infty\}$ and $K \in \{1, 2, 4, 8, 16\}$

4.4 Performance Results

Figure 2 compares TTIP and DTIP with LRU and state-of-the-art SRRIP, BRRIP and DRRIP in terms of MPKI and speedup normalised to LRU. For BRRIP we use the probability for inserting into the *long* position $\epsilon = 1/32$ and for DRRIP SDM with 32 sets.

TTIP upgrades BRRIP by supporting multiple probability values and being able optimize the probability per task type. It achieves up to 32.1% and on average 11.2% reduction in MPKI compared to LRU. The speedup over LRU is up to 12.3% and on average 5.1%. TTIP performs similarly as DRRIP, having 3.3% higher MPKI and being 0.8% faster than DRRIP. However, it does not need the hardware for Set Dueling, but instead uses a small mapping table described in Sect. 3.3 and whose cost is discussed in the Sect. 4.5.

DTIP improves MPKI over LRU for up to 33.3% and on average 16.8%. The largest contribution of improvement in MPKI comes from specfem3D, where misses to *output* dependencies of the largest task are reduced by inserting *outputs* in *immediate* position. This decision does not significantly impact the number of misses to *inputs* and *non-dependencies*. DTIP is faster than LRU for up to 12.1% and on average 4.8%. Compared to SRRIP, which is another static RRIP policy, DTIP achieves up to 29.1% (12.8% on average) lower MPKI and performs up to 10.5% (3.7% on average) faster. The improvement over SRRIP comes from the fact that DTIP differentiates the cache lines by their data-dependency types. DTIP is able to benefit from this information by inserting the new lines in a more optimal position in the recency stack so that different access patterns that

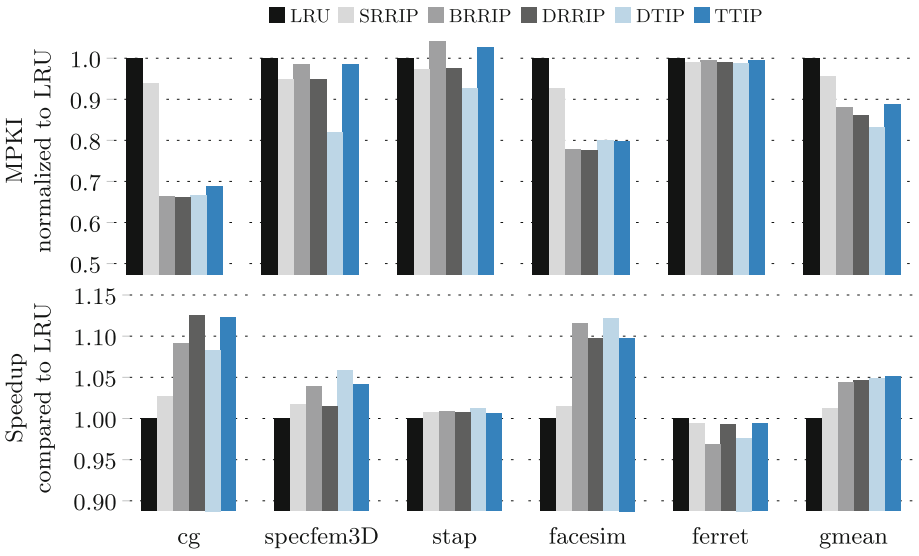

Fig. 2. Performance of TTIP and DTIP

collide in LLC have least possible negative effects on each other. DTIP reduces MPKI by 3.1% on average and is faster 0.3% than DRRIP.

Even though it shows higher MPKI than DTIP on average, TTIP achieves better execution time. The contributor to this effect is cg, where DTIP fails to achieve a speedup comparable to TTIP and DRRIP. The largest task type, which performs a matrix-vector multiplication, is the main source of MPKI improvement of DTIP over TTIP. However, three smaller, but still important tasks, show higher execution time with DTIP due to increased number of misses to *inputs* and *non-dependencies*. The improvement in execution time achieved in the largest task is not enough to compensate losses in three smaller tasks, because hits in the largest task are hidden by the unavoidable misses to the matrix.

4.5 Design Costs

To store the state of the recency stack, both TTIP and DTIP need $2n$ bits per cache set, the same as DRRIP, whereas LRU requires $n \log n$ bits per set, where n is the cache associativity. In the system evaluated in this work ($n = 16$), RRIP policies consume $2\times$ less space than LRU.

The mapping table required by TTIP has 4 entries, one for each core. Probabilities are stored with resolution of 6 bits, making the size of the structure $4 \times 6\,\text{bit} = 3\,\text{B}$. In addition, TTIP requires 4 hardware counter registers, each one being 32 bit long. The total additional hardware cost required by TTIP is $3\,\text{B} + 4 \times 32\,\text{bit} = 19\,\text{B}$. After each task instance, the runtime reads the corresponding hardware counter and potentially sets the new probability for the new task instance, which incurs overhead of few instructions. Calculating the best probability after the training period takes less than hundred instructions.

The mapping table for DTIP technique contains 32 pairs of 48-bit physical addresses, thus providing each core with 8 entries, which is more than enough to cover the most demanding tasks in regards to number of data-dependencies. In the case of larger demand for mapping table entries, smaller, less important dependencies can be omitted or merged with another dependency of the same type without degrading the performance. The total size of the mapping table is $32 \times 2 \times 48\,\text{bit} = 348\,\text{B}$. When a new task instance is scheduled for execution, the mapping table is updated with data-dependencies of the task. Upon completion of a task, the runtime clears the entries from the mapping table that belong only to that task. Both actions require several tens of instructions. The total runtime overhead in terms of number of instructions is negligible when compared with the total number of instructions of any benchmark that we use.

5 Conclusions

Improving LLC performance is of great importance in modern and future systems. In multi-core processors, threads generating various access patterns are competing for LLC resources. To achieve best performance, it is necessary to protect certain access patterns from being thrashed by accesses coming from

another thread. In this paper we aim to exploit semantic information about applications written in data-flow task-based programming model to better manage the LLC. The runtime system provides the information about task types and task data-dependencies to the LLC in order to improve the insertion policy. We propose two techniques:

TTIP - Task Type aware Insertion Policy tries to determine the best probability for inserting lines in the recency stack by using runtime-guided dynamic approach that evaluates the performance of several pre-set probabilities and chooses the best performing one.
DTIP - Dependency Type aware Insertion Policy is a static policy that inserts lines in the recency stack based on the type of data-dependency they belong to. Data that will be used by the next tasks is given more chance to stay in cache by inserting it in higher positions of the recency stack, while read-only data is given less priority.

Our policies use the runtime system for providing the hardware with the necessary information for determining appropriate insertion positions, which simplifies hardware design. The overheads of the runtime extensions are negligible. The performance benefits compared to LRU are significant for both policies. TTIP performs slightly worse than DRRIP, but uses simpler hardware. DTIP performs better than DRRIP on average, which proves the benefits of using runtime information about the application in designing LLC replacement policies. In comparison with DRRIP, our policies do not use set dueling monitors and do not require a decoder for determining *dedicated follower* sets.

Possible improvements for TTIP include discarding probabilities that perform badly from the training process. DTIP can be extended to distinguish between dependencies, since different dependencies of the same type may have slightly different access patterns that benefit from different insertion positions. Further benefits could be obtained by also taking into account task type.

Acknowledgments. This work has been supported by the RoMoL ERC Advanced Grant (GA 321253), by the European HiPEAC Network of Excellence, by the Spanish Ministry of Science and Innovation (contract TIN2015-65316-P), by Generalitat de Catalunya (contracts 2014-SGR-1051 and 2014-SGR-1272). V. Dimić has been partially supported by AGAUR of the Government of Catalonia (contract 2017 FI_B 00855). M. Moretó has been partially supported by the Ministry of Economy and Competitiveness under Juan de la Cierva postdoctoral fellowship number JCI-2012-15047. M. Casas has been supported by the Secretary for Universities and Research of the Ministry of Economy and Knowledge of the Government of Catalonia and the Cofund programme of the Marie Curie Actions of the 7th R&D Framework Programme of the European Union (contract 2013 BP B 00243).

References

1. Belady, L.A.: A study of replacement algorithms for a virtual-storage computer. IBM Syst. J. **5**, 78–101 (1966)
2. Bienia, C.: Benchmarking modern multiprocessors. Ph.D. thesis, Princeton (2011)
3. Blumofe, R., Joerg, C., Kuszmaul, B., et al.: Cilk: an efficient multithreaded runtime system. J. Parallel Distrib. Comput. **37**, 55–69 (1995)
4. Casas, M., et al.: Runtime-aware architectures. In: Träff, J.L., Hunold, S., Versaci, F. (eds.) Euro-Par 2015. LNCS, vol. 9233, pp. 16–27. Springer, Heidelberg (2015). doi:10.1007/978-3-662-48096-0_2
5. Chamberlain, B., Callahan, D., Zima, H.: Parallel programmability and the Chapel language. Int. J. High Perform. Comput. Appl. **21**, 291–312 (2007–2008)
6. Chasapis, D., Casas, M., Moretó, M., Vidal, R., Ayguadé, E., Labarta, J., Valero, M.: PARSECSs: evaluating the impact of task parallelism in the PARSEC benchmark suite. In: TACO (2015)
7. Chen, W., Liu, P., Stelzer, K.: Implementation of a pseudo-LRU algorithm in a partitioned cache, US Patent 7,069,390 (2006)
8. Davis, T., Hu, Y.: The University of Florida sparse matrix collection. ACM Trans. Math. Softw. **38**(1), 1 (2011)
9. Jaleel, A., Theobald, K.B., Steely Jr., S.C., Emer, J.: High performance cache replacement using re-reference interval prediction (RRIP). SIGARCH Comput. Arch. News **38**, 60–71 (2010)
10. Jaulmes, L., Casas, M., Moretó, M., et al.: Exploiting asynchrony from exact forward recovery for due in iterative solvers. In: SC (2015)
11. Kale, L.V., Krishnan, S.: CHARM++: a portable concurrent object oriented system based on C++. In: OOPSLA (1993)
12. Manivannan, M., Papaefstathiou, V., Pericas, M., Stenstrom, P.: RADAR: runtime-assisted dead region management for last-level caches. In: HPCA (2016)
13. Manivannan, M., Stenstrom, P.: Runtime-guided cache coherence optimizations in multi-core architectures. In: IPDPS (2014)
14. Sun Microsystems: UltraSPARC T2 supplement to the UltraSPARC architecture 2007, draft D1.4.3 (2007)
15. OpenMP Arch. Rev. Board: OpenMP Application Program Interface, v4.0 (2013)
16. Pan, A., Pai, V.S.: Runtime-driven shared last-level cache management for task-parallel programs. In: SC (2015)
17. Papaefstathiou, V., Katevenis, M.G., Nikolopoulos, D.S., Pnevmatikatos, D.: Prefetching and cache management using task lifetimes. In: ICS (2013)
18. Qureshi, M., Jaleel, A., Patt, Y., Steely, S., Emer, J.: Set-dueling-controlled adaptive insertion for high-performance caching. In: Micro. IEEE (2008)
19. Qureshi, M.K., Jaleel, A., Patt, Y.N., Steely, S.C., Emer, J.: Adaptive insertion policies for high performance caching. In: ISCA (2007)
20. Qureshi, M.K., Lynch, D.N., Mutlu, O., Patt, Y.N.: A case for MLP-aware cache replacement. In: ISCA (2006)
21. Reinders, J.: Intel Threading Building Blocks. First edn. (2007)
22. Rico, A., Duran, A., Cabarcas, F., Etsion, Y., Ramirez, A., Valero, M.: Trace-driven simulation of multithreaded applications. In: ISPASS (2011)
23. Rico, A., Cabarcas, F., Villavieja, C., et al.: On the simulation of large-scale architectures using multiple application abstraction levels. In: TACO (2012)
24. Shewchuk, J.R.: An introduction to the conjugate gradient method without the agonizing pain. Technical report (1994)

25. Teruel, X.: OmpSs quick overview, a practical approach (2013)
26. Valero, M., Moreto, M., Casas, M., Ayguade, E., Labarta, J.: Runtime-aware architectures: a first approach. Supercomp. Front. Innov. **1**, 29–44 (2014)
27. Wu, C.J., Jaleel, A., Hasenplaugh, W., et al.: SHiP: signature-based hit predictor for high performance caching. In: MICRO (2011)
28. Wulf, W.A., McKee, S.A.: Hitting the memory wall: implications of the obvious. SIGARCH Comput. Arch. News **23**, 20–24 (1995)