A Generalized Theory of Collaborative Caching

Xiaoming Gu Chen Ding

Department of Computer Science
University of Rochester
Rochester, New York, USA
{xiaoming, cding}@cs.rochester.edu

Abstract

Collaborative caching allows software to use hints to influence cache management in hardware. Previous theories have shown that such hints observe the inclusion property and can obtain optimal caching if the access sequence and the cache size are known ahead of time. Previously, the interface of a cache hint is limited, e.g., a binary choice between LRU and MRU.

In this paper, we generalize the hint interface, where a hint is a number encoding a priority. We show the generality in a hierarchical relation where collaborative caching subsumes non-collaborative caching, and within collaborative caching, the priority hint subsumes the previous binary hint. We show two theoretical results for the general hint. The first is a new cache replacement policy, priority LRU, which permits the complete range of choices between MRU and LRU. We prove a new type of inclusion property—non-uniform inclusion—and give a one-pass algorithm to compute the miss rate for all cache sizes. Second, we show that priority hints can enable the use of the same hints to obtain optimal caching for all cache sizes, without having to know the cache size beforehand.

Categories and Subject Descriptors B.3.2 [*MEMORY STRUCTURES*]: Design Styles - Cache memories; D.3.4 [*PROGRAMMING LANGUAGES*]: Processors - Compilers, Optimization

General Terms Algorithms, Performance, Theory

Keywords collaborative caching, cache replacement policy, priority cache hint, priority LRU, optimal size-oblivious hint

1. Introduction

The performance of modern chips is largely determined by cache management. A program has a high performance if the working set can be cached. When the size of the working set is too large, replacement decisions have to be made. LRU replacement policy, the most commonly used one in practice, is rigid in that it caches program data in the same way whether the data is part of the program's working set or not. This may lead to serious under-utilization of cache. An alternative solution is for a program to influence the cache management by providing hints distinguishing the type of data it uses.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ISMM'12, June 15–16, 2012, Beijing, China.
Copyright © 2012 ACM 978-1-4503-1350-6/12/06...$10.00

A number of hardware systems have been built or proposed to provide an interface for software to influence cache management. Examples include cache hints on Intel Itanium [4], bypassing access on IBM Power series [25], and evict-me bit [29]. Wang et al. called a combined software-hardware solution *collaborative caching* [29].

In this paper, we study collaborative caching in the framework shown in Figure 1. Given an execution trace, hints are added by annotating each memory access with a numerical priority. Data accessed with a higher priority (smaller numerical value) takes precedence than data accessed with a lower priority (larger value). We number the priority this way to match the numbering of memory hierarchy, where the layers of caches are numbered top down starting with L1 cache at the highest level.

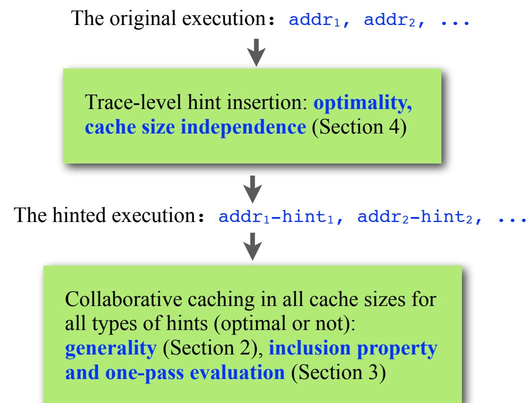


Figure 1. The framework of collaborative caching. Hints are added in software to influence hardware cache management. The interaction raises questions concerning the benefit of software intervention and the stability of the hardware cache under such intervention. We answer the questions in this paper for the generalized priority hints.

In traditional caching methods such as LRU, the hardware infers the "importance" of data and manages cache based on the inferred priority. Collaborative cache enables software intervention, which specifies the "importance" of data and changes the priority in which the hardware manages the data. In previous types of hints including evict-me [29], cache bypass [1, 25], and LRU-MRU hints [11, 12], they all have a single bit and exert a binary effect—it gives a data block either the highest or the lowest priority. Priority hints generalizes the interface and allows software to choose any priority.

In this paper, we address mainly three issues concerning this generalization:

- We categorize the effect of priority hints on cache management. The complete effect includes four cases at a cache hit and two

cases at a cache miss. The many cases make it difficult to analyze caching properties such as cache inclusiveness. We go through a lengthy but thorough proof to consider all these cases.

- We show a unique phenomenon of non-uniform inclusion, caused by the conflicts between the hinted priority and the inferred priority (by the cache) from the past accesses. One cannot simulate such cache using a priority list, as has been done for all other types of inclusive cache including LRU, OPT, and collaborative LRU-MRU. We describe a more general representation called span and give a new, one-pass simulation algorithm based on spans.
- Previous single-bit hints for optimal performance are cache-size dependent. A consequence is that the inability to optimize a memory hierarchy with multiple layers of cache. We show how to obtain size-oblivious optimal hints using priority hints.

The results in this paper are mostly theoretical (except for the cost of stack simulation). There are significant obstacles preventing a practical use. First, for a clean theory we consider fully associative cache. The same idea may be applied to each set of set associative cache. It improves efficiency since the number of priorities will be limited by the set associativity rather than the cache size. Second, we assume unit size cache blocks.[1] Third, we insert hints at the trace level. A program-level method may be devised as done for LRU-MRU hints [12]. Despite of the limitations, the theoretical results are valuable. Previous schemes of LRU, MRU and collaborative LRU-MRU are all but a few special cases of priority hints. The range of choices is far greater in this general case. It exposes and solves a fundamental problem in collaborative caching—the conflict between the priorities stated in the hints and those implied in the access sequence.

The rest of the paper is organized as follows. Section 2 introduces the basic concepts. Section 3 categorizes the six cases of cache accesses, proves the inclusion property and gives the algorithm for calculating the priority LRU stack distance. Cache hints for optimal performance for all cache sizes are discussed in Section 4 followed by the related work in Section 5 and a summary.

2. Basic Concepts and Generalized Caching

Inclusive cache The inclusion property is first characterized by Mattson et al. in their seminal paper in 1970 [19]. The property states that a larger cache always contain the content of a smaller cache. The property is fundamental for three reasons.

i) In inclusive caches, the miss ratio is a monotone function of the cache size. Belady anomaly does not occur [2].

ii) The miss ratio of an execution can be simulated in one pass for all cache sizes. Mattson's algorithms used a stack and hence are called stack algorithms [19].

iii) Stack simulation provides a metric called stack distance. Stack distance is useful for program analysis because it is independent of specific cache sizes. In particular, the reuse distance, which is the LRU stack distance, has been extensively used in improving program and system locality [37].

Stack algorithms An inclusive cache can be viewed as a stack—data elements at the top c stack positions are the ones in a cache of size c. The stack position defines the priority of the stored data. Stack simulation is to simulate cache of an infinite size. All accessed data are ordered by their priority in a *priority list*. The stack distance gives the minimal cache size to make an access a cache hit [19]. A stack distance is defined for each type of inclusive cache and computed by simulating that type of cache of an infinite size.

The following are examples of inclusive but non-collaborative cache.

- *LRU* The data in an LRU cache is prioritized by the most recent access time. The data element with the least recent access time has the lowest priority (highest position number) and is evicted when a replacement happens. Most hardware implements pseudo-LRU for efficiency [27]. The LRU stack distance is called reuse distance in short. It measures the amount of data accessed between two consecutive uses of the same data element. Reuse distance is measured in near constant time by organizing the priority list as a dynamically compressed tree [37].
- *MRU* The data in an MRU cache is also prioritized by the most recent access time. Unlike LRU, the lowest priority is the data element with the most recent access time.
- *OPT* The data in an OPT cache is prioritized by the next access time. The data element with the furthest reuse has the lowest priority. OPT is impractical because it requires future knowledge. It is used as the upper bound of cache performance. The fastest method for calculating the OPT stack distance is the one-pass algorithm by Sugumar and Abraham [28].

We are the first to formalize the inclusion property in collaborative cache [12]:

- *LRU-MRU* A hint indicates whether an access is LRU or MRU. The inclusion property holds even when LRU and MRU accesses are mixed arbitrarily. To calculate the LRU-MRU stack distance, the following priority scheme is used. An LRU access is assigned the current access time as the priority, while an MRU access is assigned the negation of the current access time. A stack algorithm can compute the LRU-MRU stack distance [12].

The inclusive cache management hierarchy We organize the commonly used inclusive caching methods into the following three categories. They form a hierarchy based on the "implemented-by" relation, as explained below and shown pictorially in Figure 2.

- *Level 1, non-collaborative caching*, including LRU, MRU and OPT. The priority is entirely inferred from the access sequence.
- *Level 2, limited collaborative caching*, including cache bypass, evict-me bit, and LRU-MRU. The priority is specified by a hint. The specified priority is either the highest or the lowest. It is easy to see that LRU-MRU subsumes LRU and MRU. It also subsumes OPT as we have shown that LRU-MRU hints can obtain optimal caching [11].
- *Level 3, generalized collaborative caching.* The priority hint is the only member of this category. A priority hint is a number encoding a priority. Since it allows a hint to specify any priority, it subsumes the limited collaborative schemes in Level 2.

Being general, the priority hint not only can implement other cache hints but also can create cache management scenarios not possible with any prior method of inclusive caching. We will show such an example and describe proofs and solutions for this general scheme.

3. The Priority LRU Cache Replacement Policy

For this study, the cache is fully associative and organized as a stack. The default scheme is LRU, where the data element is placed

[1] Petrank and Rawitz showed that optimal data placement (in non-unit size cache blocks) cannot be solved or well approximated in polynomial time unless P=NP [21].

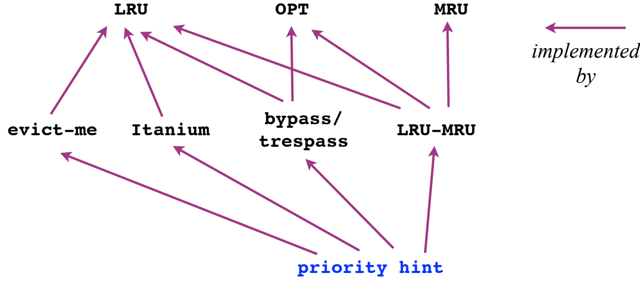


Figure 2. Common inclusive caching methods organized in a hierarchy based on the "implemented-by" relation. Limited collaborative caching of LRU-MRU [12] subsumes non-collaborative schemes of LRU, MRU and OPT [19]. Priority hint subsumes LRU-MRU and other prior collaborative methods.

at the top of the stack at position 1 and displaced at the bottom of the stack at position c, which is the cache size. Priority hint changes the default scheme. In this section, the priority value directly specifies the stack position to insert the associated data element. The phrase "a data element has a priority p" is used interchangeably with "a data element is at position p in the cache stack".

In priority LRU, an access is a pair (d, p), which means that the accessed data element d is to be inserted at position p in the cache stack. The priority p can be any positive integer. If p is always 1, priority LRU becomes LRU. If p is the cache size, priority LRU is the same as MRU. If p is greater than the cache size, the access is a cache bypass. If p can be either 1 or the cache size, priority LRU is the same as the collaborative LRU-MRU cache [12].

As an interface, priority hints may be used in arbitrary ways, sometimes optimal but probably suboptimal most times and even counter productive. In this section, we derive the properties for collaborative caching under all possible priority hints. The problem of optimal hint insertion will be discussed in Section 4.

We categorize priority LRU accesses into six classes, illustrated in Figure 3 and 4. Consider an access to w with the priority i, i.e. (w, i), arriving in the size-m cache. If w is in cache, the access is a hit. Otherwise, the access is a miss. Let the current stack position be j. A priority LRU access falls into one of the six classes, which is determined by the relations between i, j, m. The hit has four cases and the miss has two cases. To describe the change in priority, we use the terms up move, no move, and down move. We should note that the move is conceptual and may not be physical. The change in "position" requires only an update on the associated position bits.

i) A hit up move ($1 \leq i < j \leq m$)—Figure 3(a) shows that w is moved up to the position i, and the data elements between S_i and S_{j-1} are moved one position lower.

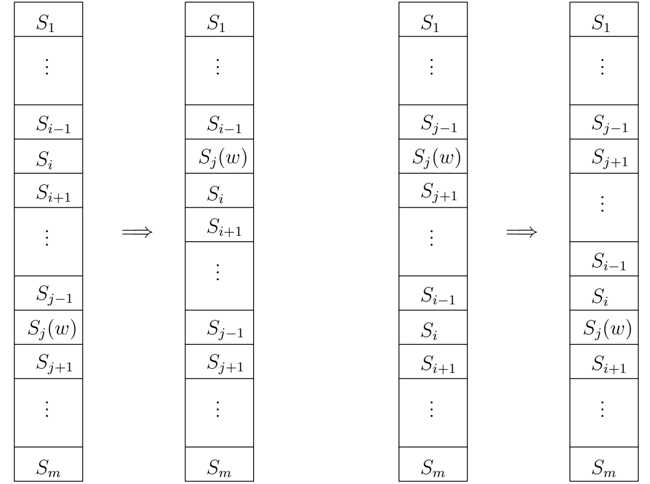
ii) A hit no move ($1 \leq j = i \leq m$)—Figure 3(c) shows that all data elements including w do not change their positions.

iii) A hit down move ($1 \leq j < i \leq m$)—Figure 3(b) shows that w is moved down to the position i in the cache, and the data elements between S_{j+1} and S_i are moved one position higher.

iv) A hit bypass ($1 \leq j \leq m < i$)—Figure 3(d) shows that w is moved out of the cache, and the data elements between S_{j+1} and S_m are moved one position higher. We also refer to this case as a voluntary eviction.

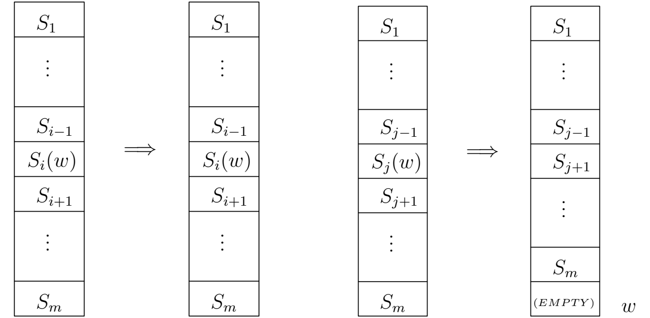
v) A miss insertion ($j = \infty$ and $1 \leq i \leq m$)—We take $j = \infty$ when the accessed data element w is not in the cache. Figure 4(a) shows that w is moved into the cache at the position i. The data elements between S_i and S_{m-1} are moved one position lower. The lowest priority element S_m is evicted.

vi) A miss bypass ($j = \infty$ and $i > m$)—We assume that the accessed data elements can be accessed without being stored in the cache. Figure 4(b) shows that w bypasses the cache. The data elements in the cache are unaffected.



(a) Priority i is higher than initial position j (numerically $i < j$): w is moved up to position i

(b) Priority i is lower than j ($i > j$): w is moved down to position i



(c) $i == j$: no move

(d) Priority i is lower than the cache size m ($i > m$): w is moved out of the cache

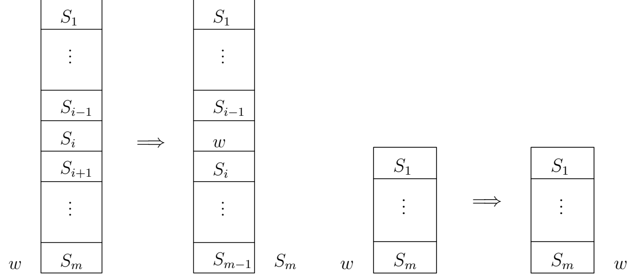
Figure 3. Four cases of data hit in the priority cache when the data block w, at position j in cache, is accessed with a priority i.

We make a few observations of the above operations of priority LRU:

i) A cache bypass can happen either for a hit or for a miss. In the hit bypass, the accessed data element voluntarily vacates its space in the cache. Neither case of bypass happens in LRU or LRU-MRU.

ii) A forced eviction only happens in a miss insertion. The victim is the data element with the lowest priority at LRU position before the insertion. This is the same as LRU and LRU-MRU. If the LRU position is unoccupied, the eviction does not happen.

iii) Only a hit bypass or a miss insertion can change the content of the cache. No data element is moved into or out of cache in the other four cases.



(a) Priority i is higher than the cache size m ($i \leq m$): w is moved into the cache at position i

(b) Priority i is lower than m ($i > m$): w bypasses the cache

Figure 4. Two cases of data miss in the priority cache when the data block w, not in cache before the access, is accessed with priority i.

iv) No data position is changed at the hit no-move case or the miss bypass case. The cache stack stays unchanged.

3.1 The Inclusion Property

THEOREM 1. *Let the access trace be executed on two priority LRU caches C_1 and C_2 ($|C_1| < |C_2|$). At each access, every data element in C_1 locates at the same or a lower position compared with the position of the corresponding element in C_2.*

Proof Let the access trace be $P = (x_1, x_2, ..., x_n)$. Let $C_i(t)$ be the collection of data in cache C_i after x_t. A function $loc()$ returns the location of a data element in the cache stack—$loc_i^t(d) = p$ ($1 \leq p \leq |C_i|$) means that the data element d is at the position p of C_i after x_t. In other words, $loc()$ returns the priority of a data element. We let $loc_i^t(d) = \infty$ if d is not in C_i after x_t. The initial situation is $C_1(0) = C_2(0) = \emptyset$, in which the theorem holds. Now we prove the theorem by induction on t.

Assume any $d, d \in C_1(t) \rightarrow loc_1^t(d) \geq loc_2^t(d)$. Suppose there is a data element d satisfying $d \in C_1(t)$ and $d \notin C_2(t)$. Then we have $loc_1^t(d) < loc_2^t(d) = \infty$—a contradiction of the assumption. So we have 9 possible cases for the next access $x_{t+1}(d', p')$ shown in Table 1. We prove for any d, $d \in C_1(t+1) \rightarrow loc_1^{t+1}(d) \geq loc_2^{t+1}(d)$ for each case. We do not have to check all data elements in $C_1(t+1)$ but only the ones moved up in C_1 or moved down in C_2.

	x_{t+1} hits in C_1 and C_2	x_{t+1} misses in C_1 but hits in C_2	x_{t+1} misses in C_1 and C_2
$1 \leq p' \leq \lvert C_1 \rvert$	I	II	III
$\lvert C_1 \rvert < p' \leq \lvert C_2 \rvert$	IV	V	VI
$p' > \lvert C_2 \rvert$	VII	VIII	IX

Table 1. The 9 cases for the next access x_{t+1} to d' with a priority p'.

I. From the assumption, we know that $loc_1^t(d') \geq loc_2^t(d')$. There are six sub-cases of x_{t+1} as shown in Table 2.

	up move in C_2	no move in C_2	down move in C_2
up move in C_1	i	ii	iii
no move in C_1	IMPOSSIBLE	iv	v
down move in C_1	IMPOSSIBLE	IMPOSSIBLE	vi

Table 2. The 6 sub-cases of Case I in Table 1: the access x_{t+1} is a hit in both C_1 and C_2. A hit can be one of the cases shown in Figure 3 except the bypass case.

i) x_{t+1} is a hit up move in both C_1 and C_2, which means $p' < loc_2^t(d') \leq loc_1^t(d')$. The only data element moved up in C_1 is d', which goes to the same position p' in C_2. A data element d moved down in C_2 satisfies $p' \leq loc_2^t(d) < loc_2^t(d')$: ① if $p' \leq loc_1^t(d) < loc_1^t(d')$, given $loc_1^t(d) \geq loc_2^t(d)$, we have $loc_1^{t+1}(d) \geq loc_2^{t+1}(d)$ because $loc_1^{t+1}(d) = loc_1^t(d) + 1$ and $loc_2^{t+1}(d) = loc_2^t(d) + 1$; ② if $loc_1^t(d) > loc_1^t(d')$, given $loc_1^t(d) \geq loc_2^t(d)$, we have $loc_1^{t+1}(d) \geq loc_2^{t+1}(d)$ because $loc_2^{t+1}(d) \leq loc_2^t(d') \leq loc_1^t(d') < loc_1^t(d) = loc_1^{t+1}(d)$. The induction holds in this case.

ii) x_{t+1} is a hit up move in C_1 but a hit no move in C_2. The only data element moved up in C_1 is d', which goes to the same position p' in C_2. No other data location is changed in C_2. The induction holds.

iii) x_{t+1} is a hit up move in C_1 but a hit down move in C_2. d' is the only data element moved up in C_1 or the only one moved down in C_2, which goes to the same position p' in C_2. The induction holds.

iv) x_{t+1} is a hit no move in both C_1 and C_2. No data location is changed in either C_1 or C_2. The induction holds.

v) x_{t+1} is a hit no move in C_1 but a hit down move in C_2. No data location is changed in C_1. The only data element moved down in C_2 is d', which goes to the same position p' in C_1. The induction holds.

vi) x_{t+1} causes a down move in both C_1 and C_2, which means $loc_2^t(d') \leq loc_1^t(d') < p'$. A data element d moved up in C_1 satisfies $loc_1^t(d') < loc_1^t(d) \leq p'$: ① if $loc_2^t(d') < loc_2^t(d) \leq p'$, given $loc_1^t(d) \geq loc_2^t(d)$, we have $loc_1^{t+1}(d) \geq loc_2^{t+1}(d)$ because $loc_1^{t+1}(d) = loc_1^t(d) - 1$ and $loc_2^{t+1}(d) = loc_2^t(d) - 1$; ② if $loc_2^t(d) < loc_2^t(d')$, given $loc_1^t(d) \geq loc_2^t(d)$, we have $loc_1^{t+1}(d) \geq loc_2^{t+1}(d)$ because $loc_2^{t+1}(d) = loc_2^t(d) < loc_2^t(d') \leq loc_1^t(d') \leq loc_1^{t+1}(d)$. The only data element moved down in C_2 is d', which goes to the same position p' in C_1. The induction holds again as in all previous five cases.

II. There are three sub-cases about x_{t+1} as shown in Table 3.

	up move in C_2	no move in C_2	down move in C_2
a miss insertion in C_1	i	ii	iii

Table 3. The 3 sub-cases of case II in Table 1: the access x_{t+1} misses in C_1 but hits in C_2. The hit and miss cases are shown in Figures 3 and 4.

i) x_{t+1} is a miss insertion in C_1 but a hit up move in C_2. No data element is moved up in C_1 except that d' is moved into C_1, which goes to the same position p' in C_2. A data element d moved down in C_2 satisfies $p' \leq loc_2^t(d) < loc_2^t(d')$: ① if $d \in C_1(t)$ and $p' \leq loc_1^t(d) < |C_1|$, given $loc_1^t(d) \geq loc_2^t(d)$, we have $loc_1^{t+1}(d) \geq loc_2^{t+1}(d)$ because $loc_1^{t+1}(d) = loc_1^t(d)+1$ and $loc_2^{t+1}(d) = loc_2^t(d) + 1$; ② if $loc_1^t(d) = |C_1|$, we do not have to worry about this case because d is evicted and not in $C_1(t+1)$; ③ if $d \notin C_1(t)$, we do not have to worry about this case either because d is not in $C_1(t+1)$. The induction holds.

ii) x_{t+1} is a miss insertion in C_1 but a hit no move in C_2. No data element is moved up in C_1 except that d' is moved

time	1	2	3	4	5	6	7	8	9
access & hint	A-2	B-2	C-5	D-1	B-6	D-6	A-4	C-1	A-4
stack: 1				D	D			C	C
2	A	B	B			A			
3		A	A	B	A				
4				A			A		A
5			C					A	

(a) Cache size is 5. A is in position 5 after time 8.

time	1	2	3	4	5	6	7	8	9
access & hint	A-2	B-2	C-5	D-1	B-6	D-6	A-4	C-1	A-4
stack: 1				D	D			C	C
2	A	B	B			A			
3		A	A	B	A		C		
4				A		C	A	A	A
5			C		C	B	B	B	B
6				C	B	D	D	D	D

(b) Cache size is 6. A is in position 4 after time 8.

Figure 5. An example of non-uniform inclusion. The priority LRU observes the inclusion principle but permits data to reside in different positions in the smaller cache than in the larger cache. In this example, after time 8, A locates at a lower position in the size-5 cache than in the size-6 cache.

into C_1, which goes to the same position p' in C_2. No data element changes location in C_2. The induction holds.

iii) x_{t+1} is a miss insertion in C_1 but a hit down move in C_2. No data element is moved up in C_1 except that d' is moved into C_1, which goes to the same position p' in C_2. And d' is the only data element moved down in C_2. The induction holds.

III. x_{t+1} is a miss insertion in both C_1 and C_2. No data element is moved up in C_1 except that d' is moved into C_1, which goes to the same position p' as in C_2. A data element d moved down in C_2 satisfies $loc_2^t(d) \geq p'$: ① if $d \in C_1(t)$ and $p' \leq loc_1^t(d) < |C_1|$, given $loc_1^t(d) \geq loc_2^t(d)$, we have $loc_1^{t+1}(d) \geq loc_2^{t+1}(d)$ because $loc_1^{t+1}(d) = loc_1^t(d)+1$ and $loc_2^{t+1}(d) = loc_2^t(d) + 1$; ② if $loc_1^t(d) = |C_1|$, we do not have to worry about this case because d is evicted and not in $C_1(t+1)$; ③ if $d \notin C_1(t)$, we do not have to worry about this case either because d is not in $C_1(t+1)$. The induction holds.

IV. From the assumption, we know that $loc_1^t(d') \geq loc_2^t(d')$. So x_{t+1} is a hit bypass in C_1 but a hit down move in C_2, in which we have $loc_2^t(d') \leq loc_1^t(d') < p'$. A data element d moved up in C_1 satisfies $loc_1^t(d) > loc_1^t(d')$: ① if $loc_2^t(d') < loc_2^t(d) \leq p'$, given $loc_1^t(d) \geq loc_2^t(d)$, we have $loc_1^{t+1}(d) \geq loc_2^{t+1}(d)$ because $loc_1^{t+1}(d) = loc_1^t(d) - 1$ and $loc_2^{t+1}(d) = loc_2^t(d) - 1$; ② if $loc_2^t(d) < loc_2^t(d')$, given $loc_1^t(d) \geq loc_2^t(d)$, we have $loc_1^{t+1}(d) \geq loc_2^{t+1}(d)$ because $loc_2^{t+1}(d) = loc_2^t(d) < loc_2^t(d') \leq loc_1^t(d') \leq loc_1^{t+1}(d)$. The only data element moved down in C_2 is d', which is moved out of C_1. The induction holds.

V. There are three sub-cases about x_{t+1} as shown in Table 4.

	a hit up move in C_2	a hit no-move in C_2	a hit down move in C_2
a miss bypass in C_1	i	ii	iii

Table 4. The 3 sub-cases of x_{t+1} of case V

i) x_{t+1} is a miss bypass in C_1 but a hit up move in C_2. No data location is changed in C_1. A data element d moved down in C_2 satisfies $p' \leq loc_2^t(d) < loc_2^t(d')$: we do not have to worry about this case because $d \notin C_1(t+1)$. Otherwise, $d \in C_1(t+1)$ implies $d \in C_1(t)$ because x_{t+1} is a miss bypass in C_1, from which we get $loc_2^t(d) \leq loc_1^t(d) \leq |C_1| < p'$—a contradiction of the assumption $loc_2^t(d) \geq p'$. The induction holds.

ii) x_{t+1} is a miss bypass in C_1 but a hit no move in C_2. No data element changes location in either C_1 or C_2. The induction trivially holds.

iii) x_{t+1} is a miss bypass in C_1 but a hit down move in C_2. No data element changes location in C_1. The only data element moved down in C_2 is d', which is not in $C_1(t+1)$. The induction again holds.

VI. x_{t+1} is a miss bypass in C_1 but a miss insertion in C_2. No data element changes location in C_1. A data element d moved down in C_2 satisfies $loc_2^t(d) \geq p'$: we do not have to worry about this case because d is not in $C_1(t+1)$. Otherwise, $d \in C_1(t+1)$ implies $d \in C_1(t)$ because x_{t+1} is a miss bypass in C_1, from which we get $loc_2^t(d) \leq loc_1^t(d) \leq |C_1| < p'$—a contradiction of the assumption $loc_2^t(d) \geq p'$. The induction holds.

VII. x_{t+1} is a hit bypass in both C_1 and C_2. A data element d moved up in C_1 satisfies $loc_1^t(d) \geq loc_1^t(d')$: ① if $loc_2^t(d) > loc_2^t(d')$, given $loc_1^t(d) \geq loc_2^t(d)$, we have $loc_1^{t+1}(d) \geq loc_2^{t+1}(d)$ because $loc_1^{t+1}(d) = loc_1^t(d) - 1$ and $loc_2^{t+1}(d) = loc_2^t(d) - 1$; ② if $loc_2^t(d) < loc_2^t(d')$, given $loc_1^t(d) \geq loc_2^t(d)$, we have $loc_1^{t+1}(d) \geq loc_2^{t+1}(d)$ because $loc_2^{t+1}(d) = loc_2^t(d) < loc_2^t(d') \leq loc_1^t(d') \leq loc_1^{t+1}(d)$. No data element is moved down in C_2 except that d' is moved out in both C_1 and C_2. The induction therefore holds.

VIII. x_{t+1} is a miss bypass in C_1 and a hit bypass in C_2. No data element changes location in C_1. No data element is moved down in C_2 except that d' is moved out. The induction is preserved.

IX. x_{t+1} is a miss bypass in both C_1 and C_2. No data changes location in either C_1 or C_2. The induction trivially holds.

With the above long list, we have covered all possible cases. The theorem is proved. ∎

The inclusion property is shown in the following corollary.

COROLLARY 1. *An access trace is executed on two priority LRU caches—C_1 and C_2 ($|C_1| < |C_2|$). At any access, the content of cache C_1 is always a subset of the content of cache C_2.*

Proof Suppose a data element d is in $C_1(t)$ but not in $C_2(t)$. Then we have $loc_1^t(d) < loc_2^t(d) = \infty$—a contradiction of Theorem 1. The supposed situation is impossible. Priority LRU preserves the inclusion property. ∎

3.2 Uniform vs Non-uniform Inclusion

The generality of priority LRU can create cache management scenarios not possible in the past. In particular, the stack layout may differ based on cache size—the same data element may reside in a lower position in the smaller cache than in the larger cache. We call this case *non-uniform inclusion*. In comparison, all previous inclusive caching schemes, e.g. LRU and LRU-MRU, have *uniform inclusion*, in which the same data element has the same position regardless of the cache size.

Figure 5 shows an example of non-uniform inclusion. The stack layout at each access is shown in Figure 5(a) for cache size 5 and Figure 5(b) for cache size 6. The non-uniformity happens after the access at time 8—the data element A locates at the position 5 in the smaller cache but at the position 4 in the larger cache. The reason has to do with the data element C. Before time 8, C is out of the size-5 cache but in the size-6 cache. When C is accessed again at time 8, the element A is moved down by one position in the size-5 cache but stays in situ in the size-6 cache, creating different stack layouts. The example shows that the difference is allowed by priority LRU but does not violate the inclusion property.

The non-uniform inclusion is shown formally by Theorem 1, which allows for the data to locate in a lower position in a smaller cache. Previous inclusive caching schemes have the stronger property that the data has to be in the same position in caches of different sizes.

Non-uniform inclusion uncovers a subtle distinction between the inclusion property and the stack layout, which is that the inclusion principle does not have to imply identical placement. The inclusion property can hold without requiring different caches to have the same stack layout. Priority LRU represents this new category of non-uniform inclusive caching. For this new type of caching, computing the stack distance becomes problematic, as we discuss next.

3.3 The Priority LRU Stack Distance

For an access trace running on a priority LRU cache, for each access, a minimal cache size exists to make the access a hit because of the inclusion property. This critical minimal cache size is called *stack distance* [19]. With a one-pass stack distance analyzer, we can compute miss ratios for all cache sizes without doing cache simulations repeatedly for each cache size.

time	1	2	3	4	5	6	7	8	9
trace	A-2	B-2	C-5	D-1	B-6	D-6	A-4	C-1	A-4
stack 1				D	D			C	C
2	A	B	B			A			
3		A	A	B	A				
4				A			A		(A)

Figure 6. For the same trace in Figure 5, the access at time 9 is a miss in the size-4 cache.

3.3.1 Priority List ... No Longer Works

A priority list is the core data structure in the original stack algorithms [19]. Different stack algorithms are identical in construction and maintenance of the priority list. The only difference is the priority used. For example, the priority used for LRU is the most recent access time but the one for OPT is the next access time. While Mattson et al. considered only non-collaborative caches, this solution extends to the case of limited collaboration in particular the LRU-MRU cache. Indeed, an important finding by us is a way to assign a "dual" priority based on the LRU-MRU hint to maintain a single priority list [12].

Because of non-uniform inclusion, a single priority list no longer works for priority LRU. Since the stack position changes depending on the cache size, so does the priority. We cannot maintain a single priority list to represent the layout for all cache sizes.

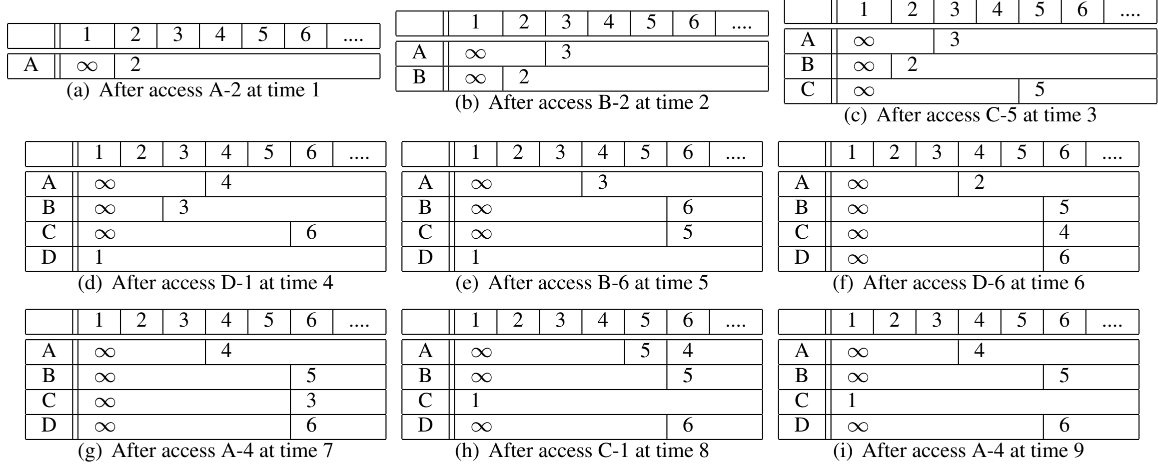
Still, can we solve the problem by simulating an infinitely large cache and use the lowest position as the stack distance? We can show a counter example as follows. Take the example trace in Figure 5(b). It is the same as a simulation of infinite cache size. The lowest position of A before the access at time 9 is 4 in the infinitely large cache. However, this access is a miss in the size-4 cache, as shown in Figure 6. The lowest stack position, 4, is not the right stack distance.

3.3.2 Span

We generalize the classic stack algorithms by replacing the priority list with the notion of span. The purpose is to track the position of a data element in all cache sizes (not just the infinite size). A span is denoted as (d, c_1, c_2, loc), which means the data element d is at position loc when cache size is between c_1 and c_2. An inherent constraint for a span is that $loc \leq c_1 \leq c_2$ when $loc \neq \infty$. If $loc = \infty$, d is not in the cache with the specific cache sizes. The span leverages the fact that a data element usually locates at the same position in multiple cache sizes.

In the following paragraphs, several cases of span update are discussed in details with an example. Figure 7 shows how spans work on an example trace with 9 accesses. Each step is a table showing spans for all data elements. Unlike previous stack algorithms that use an infinite cache size, the spans in these tables shows data positions in all cache sizes. The first column of the table lists all data. The first row shows all cache sizes. We show the sizes from 1 to 6 separately and the rest are compacted into a single "size" with ellipses. Each of the following rows with several spans is for a data element. For example, there are two spans $(A, 1, 1, \infty)$ and $(A, 2, \infty, 2)$ in Figure 7(a)—the former means that A is not in cache with a size-1 cache and the latter means that A is at the position 2 for all cache sizes no less than 2. In this way, the locations of the same data element for all cache sizes are represented. From the column view, each column for a cache size indicates how data elements locate in the cache with this specific cache size. Based on these spans, we are able to simulate all size caches at the same time. Because the spans accurately represent all stack layouts, the correctness is ensured.

At the beginning, all caches are empty. The access at time 1 is about creating spans for itself—$(A, 1, 1, \infty)$ and $(A, 2, \infty, 2)$ shown in Figure 7(a). For the first access to a data element, the stack distance is infinity since it is a compulsory miss [13]. The first access has an infinite stack distance. For the access at time 2, it is a miss bypass with a cache size 1 but a miss insertion for larger caches. So the span $(A, 1, 1, \infty)$ is unchanged because A stays outside the size-1 cache. The other span $(A, 2, \infty, 2)$ is first changed to $(A, 2, \infty, 3)$ because moving B to the position 2 makes A one position lower. Then the new $(A, 2, \infty, 3)$ is split into $(A, 2, 2, 3)$ and $(A, 3, \infty, 3)$—the former is updated to $(A, 2, 2, \infty)$ to indicate A is out of the size-2 cache. The two adjacent spans with the same



	1	2	3	4	5	6	....
A	∞	2					

(a) After access A-2 at time 1

	1	2	3	4	5	6	....
A	∞		3				
B	∞	2					

(b) After access B-2 at time 2

	1	2	3	4	5	6	....
A	∞		3				
B	∞	2					
C	∞				5		

(c) After access C-5 at time 3

	1	2	3	4	5	6	....
A	∞			4			
B	∞		3				
C	∞					6	
D	1						

(d) After access D-1 at time 4

	1	2	3	4	5	6	....
A	∞			3			
B	∞					6	
C	∞					5	
D	1						

(e) After access B-6 at time 5

	1	2	3	4	5	6	....
A	∞			2			
B	∞					5	
C	∞					4	
D	∞					6	

(f) After access D-6 at time 6

	1	2	3	4	5	6	....
A	∞			4			
B	∞					5	
C	∞					3	
D	∞					6	

(g) After access A-4 at time 7

	1	2	3	4	5	6	....
A	∞				5	4	
B	∞					5	
C	1						
D	∞					6	

(h) After access C-1 at time 8

	1	2	3	4	5	6	....
A	∞			4			
B	∞					5	
C	1						
D	∞					6	

(i) After access A-4 at time 9

Figure 7. An example of priority LRU stack simulation. The trace has 9 accesses to 4 data elements. A data element may locate at different stack positions depending on cache sizes. All possible positions for each data element are tracked by its *span* list, shown in each row. Cache sizes are shown by the header row.

loc, $(A, 1, 1, \infty)$ and $(A, 2, 2, \infty)$, are merged into a single span $(A, 1, 2, \infty)$. At last, we create the spans for B: $(B, 1, 1, \infty)$ and $(B, 2, \infty, 2)$. The updated all-size cache snapshot is in Figure 7(b). The second access is also a compulsory miss and has an infinite stack distance.

At time 5, the all-size cache snapshot is in Figure 7(d). First we update the spans for other data elements except for B. There are two spans for A—$(A, 1, 3, \infty)$ and $(A, 4, \infty, 4)$: the former stays the same, and the latter is updated to $(A, 4, \infty, 3)$ because moving B to the position 6 makes A one position higher. We update the spans of C and D in the same way and obtain the new all-size cache snapshot in Figure 7(e). When a data element is accessed again, the stack distance equals to the c_1 of the leftmost span with a finite loc, which is the minimal cache size to keep the accessed data element in cache. For this access, B is accessed again and its leftmost span with a finite loc is $(B, 3, \infty, 3)$. So the stack distance is 3. At last, we update the spans of B to $(B, 1, 5, \infty)$ and $(B, 6, \infty, 6)$.

At time 9 with the lower position exception, we have to look back into the access at time 8. In the cache snapshot in Figure 7(g), A has two spans $(A, 1, 3, \infty)$ and $(A, 4, \infty, 4)$. Moving C to the position 1 has a different impact on A for different cache sizes. The span $(A, 1, 3, \infty)$ stays the same. For the other span $(A, 4, \infty, 4)$, A is moved one position lower when the cache size is 4 or 5 but stays the same when the cache size is 6 or greater. This span is updated into two spans: $(A, 4, 5, 5)$ and $(A, 6, \infty, 4)$. The new span $(A, 4, 5, 5)$ is then split and merged with $(A, 1, 3, \infty)$ in the same way as in the case at time 2. Finally, A has three spans after the access at time 8: $(A, 1, 4, \infty)$, $(A, 5, 5, 5)$, and $(A, 6, \infty, 4)$. When the access at time 9 to A happens, the left most span with a finite loc of A is $(A, 5, 5, 5)$. The stack distance is 5.

Algorithm 1: `process_one_access()`: compute the priority LRU stack distance for an access

Input: d is accessed with a priority p.
Output: returns the priority LRU stack distance of this access.

```
1  process_one_access(d,p)
2  begin
3      if There is no span list for d then
4          for Each current span list (list_iter) do
5              Update the span list by calling
               update_one_list(list_iter,NULL,p)
6          end
7          Create a new list for d
8          Create a new span for the current access with
           c1 = loc = p and insert it into the new list
9          Return an infinite stack distance
10     else
11         Set the_list to the span list for d
12         for Each current span list (list_iter) do
13             if list_iter ≠ the_list then
14                 Update the span list by calling
                   update_one_list(list_iter,the_list,p)
15             end
16         end
17         Save the c1 value of the first span in the_list to a
           temporary
18         Delete all the current spans in the_list
19         Create a new span for the current access with
           c1 = loc = p and insert it into the_list
20         Return the saved temporary as the stack distance
21     end
22 end
```

3.3.3 The One-pass Simulation Algorithm

In the algorithm, each data element has a list of spans once it is accessed. A node in the list is a span but only with two fields for c_1 and loc. The c_2 equals to the c_1 of the next span minus one. For the last span, the c_2 equals to ∞. Only the spans with a finite value for loc show up in a span list, which implies the corresponding data element could not be contained in cache when the cache size is less than the c_1 of the first span node.

Function `process_one_access()` in Algorithm 1 is the top-level function to compute a stack distance. It mainly consists of two cases: one for first-time accesses (compulsory misses) and the other for the other accesses. Both cases follow a similar procedure: update all the spans except the ones for the accessed data element; update the spans for the accessed data element; and return the stack distance. The first step is done by calling Function `update_one_list()` in Algorithm 2.

Function `update_one_list()` has three arguments providing sufficient information for span updates. The `while` loop traverses

Algorithm 2: `update_one_list()`: update the span list for a data element for all cache sizes

```
Input: updated_list is the span list for a data element to be updated;
       accessed_list is the span list for the accessed data element,
       which has not been updated yet; new_priority is the new
       priority for the accessed data element.
1  update_one_list(updated_list,accessed_list,new_priority)
2  begin
3      Set updated_span to the last span of updated_list
4      if accessed_list ≠ NULL then
5          Set accessed_span to the last span of accessed_list
6      else
7          Set accessed_span to NULL
8      end
9      while updated_span ≠ NULL do
10         if accessed_span = NULL then
11             Do updates for updated_span including changing c1
               and loc values and merging unnecessary adjacent
               spans
12             Set updated_span to its predecessor
13         else
14             if The c1 of updated_span is no less than the c1 of
               accessed_span then
15                 Do updates for updated_span including
                   changing c1 and loc values and merging
                   unnecessary adjacent spans
16                 if The c1 of accessed_span equals to the c1 of
                   updated_span then
17                     Set accessed_span to its predecessor
18                 end
19                 Set updated_span to its predecessor
20             else
21                 Create a new span and set its c1 to c1 of
                   accessed_span and loc to loc of
                   updated_span
22                 Insert the new span as the successor of the
                   updated_span
23                 Set updated_span to this new span
24                 Do updates for updated_span including
                   changing c1 and loc values and merging
                   unnecessary adjacent spans
25                 Set accessed_span to its predecessor
26                 Set updated_span to its predecessor
27             end
28         end
29     end
30 end
```

and updates the spans of a data element. The traversal is associated with another traversal through the span list of the accessed data element to make sure that span updates are done for the same cache sizes. The two correlated traversals both are done in the reverse order, from the last to the first, to make it easier to merge adjacent spans. The span updates, done in line 11, 15, and 24, have been demonstrated in the example in Figure 7.

In line 21 and 22, a span is split into two if neither the condition in line 10 nor the one in line 14 is satisfied. An example is the access at time 7 in Figure 7 when the span (A, 4, ∞, 4) is first split into (A, 4, 5, 4) and (A, 6, ∞, 4). The span splitting aligns the spans to be updated with the spans of the accessed data element. The updating operation becomes simpler since the current access has the same impact for all cache sizes within the being updated span. In this example, (A, 4, 5, 4) is updated to (A, 4, 5, 5) and (A, 6, ∞, 4) remains unchanged.

The span splitting is not always necessary. However, an unnecessary span can be merged with its successor shortly. For example, suppose we have only two spans (A, 5, ∞, 5) and (B, 8, ∞, 8), and the next access is B-1. The span (A, 5, ∞, 5) is first split into (A, 5, 7, 5) and (A, 8, ∞, 5). Then the new spans are updated to (A, 5, 7, 6) and (A, 8, ∞, 6). The span (A, 5, 7, 6) is split into (A, 5, 5, ∞) and (A, 6, 7, 6). The former span (A, 5, 5, ∞) is abandoned since we do not store a span with an infinite loc. The latter span (A, 6, 7, 6) is merged with its successor (A, 8, ∞, 6) into (A, 6, ∞, 6). It is possible to remove unnecessary span splittings with a more complex algorithm.

3.3.4 The Space and Time Overhead

The space cost per data element is proportional to the number of spans, which is bounded by the maximal priority M in a hint and the data set size D. The number of spans for a data item equals to the number of different priorities in all cache sizes. Because the possibly maximal priority for a data item is $M + D$, the possibly maximal number of spans is also $M + D$. The overall space cost is $\mathcal{O}(D \cdot (M + D))$. The bound is high in theory but not as formidable in practice. In the following empirical evaluation, the number of spans for a data element is only a few and much less than $M + D$.

The time cost consists primarily the operations involved in the span updates at line 11, 15, and 24 in Algorithm 2. The number of operations is proportional to the total number of spans of the datum being updated. If the number of spans is bounded by $M + D$, the time bound for each access is $\mathcal{O}(D \cdot (M + D))$. For LRU cache, there is only one span for each data element, so the time cost is $\mathcal{O}(D)$ per access and matches the cost of the original stack algorithm [19].

An Experiment To give a sense of the number of spans in practice, we have implemented the stack distance algorithm for priority LRU and tested it on a random trace with randomly generated accesses and priorities. The data size is set to 1024 and the trace length is 10 million. For the number of priorities, we choose to vary from 1 to 1 million in numbers that are powers of two. Instead of measuring the physical time and space, we use two logical metrics. The space is measured by the number of spans being stored. The time is measured by the number of span updates.

The columns in Table 5 shows 13 out of the 20 results on different priority ranges. We omit the cost results of other priority ranges because their measured costs are nearly equal to the computed numbers obtained by interpolating using the costs of the neighboring ranges shown in the table.

When the priority number is always 1, priority LRU degenerates into LRU. A priority list is enough to obtain the stack distance. Each data element has only one span. The space overhead for all data is 1024. For each access, the worst time cost is 2046, because the algorithm needs to do 2 updates on the span list for each of the remaining 1023 data elements. [2] The average is 1534.

In the other extreme when the maximal priority is 10 million, much greater than the data size, the overall cost is on average 1026 for space and 1537 for time, nearly identical to the cost of LRU. The highest average overall cost is 4317 for space and 6556 for time, incurred when the range of the priority is up to 1024, the size of data set. The costs in all other cases are at most half of the highest costs. If the priority is up to 512, the average overall space and time costs are 1024 and 1535, near identical to LRU.

From the results of the random access trace, we can make the following observations on the number of spans in practice. First, the number is mostly constant, close to the single span in LRU, in most cases. Second, in the worst case, the maximal number of

[2] Two updates are needed for a single-span list because of an unnecessary span splitting.

max priority		1 (i.e. LRU)	16	64	256	512	1K	2K	4K	8K	16K	64K	256K	1M
space	avg overall	1024	1024	1024	1024	1024	4317	2015	1472	1239	1129	1050	1030	1026
	avg per data element	1.0	1.0	1.0	1.0	1.0	4.2	2.0	1.4	1.2	1.1	1.0	1.0	1.0
overhead	max overall	1024	1025	1026	1035	1058	11731	5391	2859	2038	1500	1145	1057	1040
	max per data element	1	2	2	3	5	37	23	16	12	8	6	4	3
time	overall avg	1534	1534	1534	1534	1535	6556	3014	2203	1856	1692	1573	1544	1537
overhead	overall max	2046	2046	2046	2050	2072	33660	18361	12155	9961	6640	5257	4256	3667

Table 5. The measured overhead of Algorithm 1 when computing the priority LRU stack distance over a random-access trace with 10 million accesses to 1024 data elements with random priorities. The maximal priority number ranges from 1 to 1 million. The space is measured by the number of being stored spans. The time is measured by the number of calls to a span update. In most columns, the time and space costs are close to LRU stack simulation. The highest cost is incurred when the priority is up to 1024, but this worst cost is still far smaller than the theoretical upperbound.

spans per data item is far smaller than the theoretical upperbound, 37 vs. 2048.

4. Optimal Size-oblivious Hint Insertion

4.1 Cache Size-dependent Hint Insertion

We previously proposed an optimal collaborative caching scheme called Program-assisted Optimal Caching (P-OPT) for LRU-MRU cache, in which an OPT cache simulation is used to decide the access type, LRU or MRU, for each access [11]. The process is the same as shown in the introduction in Figure 1. By default, all accesses are initialized as LRU. During the OPT simulation, an access is changed to MRU if the next access to the same data element is a cache miss. In other words, an access is selected as MRU if it does not lead to a data reuse in the OPT cache with the given cache size. The new trace tagged with the single-bit cache hints has the same minimal number of misses as OPT.

The details to obtain the optimal cache hints is shown in Figure 8. We run an offline OPT simulation on a trace from a_1 to a_n with a given cache size. At a_j, we find out that the data element X is evicted. Then a_i, the most recent access to X, is selected to use MRU.

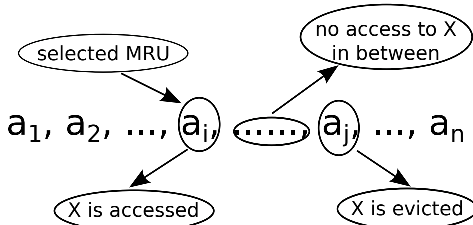


Figure 8. a_i is selected to use MRU for a given cache size during an OPT cache simulation

Compared with OPT, P-OPT encodes the future information by passing hints to the collaborative LRU-MRU cache. From the point view of hardware implementation, an LRU-MRU cache is much simpler than an OPT cache. In P-OPT, the access type for an access may change for different cache sizes in order to stay optimal. The hint insertion requires running another OPT cache simulation when the cache size is changed.

4.2 The Size-oblivious Hint Insertion

In P-OPT, cache hints may change with the cache size. However, the change is single directional similar to the inclusion property, as stated in the following Lemma 1.

LEMMA 1. *In P-OPT, an access is selected to use MRU in a smaller cache if it is selected to use MRU in a larger cache.*

Proof Assume we have two caches C_1 and C_2 ($|C_1| > |C_2|$) and an access trace. An access a_i to the data element X is selected to use MRU in C_1 in P-OPT, which means that the next access to X is a miss in an OPT cache with size $|C_1|$. Because of the inclusion property of OPT, the next access to X is also a miss in $|C_2|$. So a_i is selected as an MRU access in C_2 since a_i does not bring a cache reuse in OPT with size $|C_2|$. The lemma is proved. ∎

Lemma 1 indicates that a minimal cache size C exists for every access, which makes an access be selected to use MRU with the cache size no greater than C. Theorem 2 shows that the critical cache size is tightly correlated with the forward OPT stack distance, which is the minimal cache size to make the next reuse a cache hit in OPT.

THEOREM 2. *In P-OPT, an access is selected to use MRU if and only if the given cache size is less than its forward OPT stack distance.*

Proof Given an access a_i to a data element X, assume a_i has forward OPT stack distance d and the next access to X is a_j. From the definition of forward OPT stack distance, X is evicted between a_i and a_j if and only if the OPT cache size is less than d. Hence a_i is selected as MRU if and only if the cache size is less than d. The theorem is proved. ∎

A special case for Theorem 2 is the last accesses to data, which have infinite forward OPT stack distances. We use infinity as the critical cache size to select MRU for these last data accesses since none of them brings a cache reuse in any cache size.

The critical cache size serves well for all cache sizes to achieve optimal caching. We may encode forward OPT stack distances into priority hints for a dynamic cache control scheme, as shown in Figure 9. Like priority LRU, each access is associated with a priority hint. The cache logic in hardware dynamically compares the priority with the cache size and then chooses either the LRU or the MRU position for placing the accessed data element. As a result, a program is optimized for all cache sizes instead of a specific one. We need not know the cache size beforehand and the optimal hints are oblivious to the cache size.

5. Related Work

Cache hints and replacement policies The ISA of Intel Itanium extends the interface of the memory instruction to provide source and target hints [1]. The source hint suggests where data is expected, and the target hint suggests which level cache the data

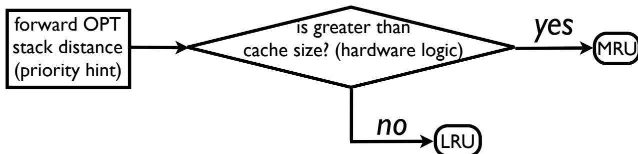


Figure 9. The dynamic cache control applies the optimal priority hint for a specific cache size.

should be kept. The target hint changes the cache replacement decisions in hardware. IBM Power processors support bypass memory access that do not keep the accessed data in cache [25]. Wang et al. proposed an interface to tag cache data with evict-me bits [29]. Our previous LRU-MRU studies considered single-bit hints for LRU and MRU [11, 12]. Recently, Ding et al. developed ULCC which uses page coloring to partition cache to separately store high locality and low locality data [8]. It may be used to approximate LRU-MRU cache management in software on existing machines. In this paper, a priority hint may place the accessed data element at an arbitrary position in cache, which is more general.

Mattson et al. established the property of inclusion and the metrics of stack distance [19]. The miss ratio of inclusive cache is monotonically non-increasing as the cache gets larger (whereas the Belady anomaly, more misses in larger cache, is impossible) [2]. Stack distance can be used to compute the miss ratio for cache of all sizes. They presented a collection of algorithms based on a priority list. The LRU stack distance, i.e. reuse distance in short, can be computed asymptotically faster (in near linear time for a guaranteed precision) using a (compression) tree [37]. The cost can be further reduced by sampling [36]. Recent work has developed multicore reuse distance to model the locality of multi-threaded programs [24] and the LRU-MRU stack distance to measure the performance of collaborative caching [12].

In this paper, we generalize the concept of inclusive cache and establish the new category of non-uniform inclusion. Previous algorithms cannot handle non-uniform inclusion. We give a new algorithm based on the notion of spans instead of the priority list or tree.

Collaborative caching Collaborative caching was pioneered by Wang et al. [29] and Beyls and D'Hollander [3, 4]. The studies were based on a common idea, which is to evict data whose forward reuse distance is larger than the cache size. Wang et al. used compiler analysis to identify self and group reuse in loops [20, 29, 30] and select array references to tag with the evict-me bit. They showed that collaborative caching can be combined with prefetching to further improve performance.

Beyls and D'Hollander used profiling analysis to measure the reuse distance distribution for each program reference. They added cache hint specifiers on Intel Itanium and improved average performance by 10% for scientific code and 4% for integer code [3]. Profiling analysis is input specific. Fang et al. showed a technique that accurately predicts how the reuse distances of a memory reference change across inputs [9]. Beyls and D'Hollander later developed a static analysis called reuse-distance equations and obtained similar improvements without profiling [4]. Compiler analysis of reuse distance was also studied by Cascaval and Padua for scientific code [5] and Chauhan and Shei for Matlab programs [6].

The prior methods used heuristics to identify data in small-size working sets for caching. It is unclear whether cache utilization could be further improved. We showed the theoretical potential of LRU-MRU collaborative caching to achieve optimal cache performance [12]. Our approach used the OPT replacement policy to gain insights into program behavior, as shown in Figure 8. The advantage is that we can partition a large working set and partially cache it to fully utilize the available cache space.

Two recent papers show the benefits of collaborative caching on current x86 processors. Yang et al. used non-temporal writes for zero initialization in JVM to reduce cache pollution [33]. Rus et al. used non-temporal prefetches and writes to specialize string operations like `memcpy()`, based on the data reuse information in certain static program contexts [23].

Virtual machine, operating system and hardware memory management Garbage collectors may benefit from the knowledge of application working set size and the affinity between memory objects. For LRU cache, reuse distance has been used by virtual machine systems to estimate the working set size [32] and to group simultaneously used objects [35]. There have been much research in operating systems to improve beyond LRU. A number of techniques used last reuse distance instead of last access time in virtual memory management [14, 26, 38] and file caching [15]. The idea of evicting dead data or least reused data early has been extensively studied in hardware cache design, including deadblock predictor [18], forward time distance predictor [10], adaptive cache insertion [22], less reuse filter [31], virtual victim cache [17], and globalized placement [34].

These techniques do not require program changes but they could only collect program information by passive observation. They were evaluated for specific cache sizes. Our work complements them in two ways. First in theory, we show the conditions for these techniques to maintain the inclusion property, for either LRU-MRU [12] or the general priority in this work. Second in practice, we show that program information can be used to obtain optimal caching for caches of all sizes.

Optimal caching Optimal caching is difficult purely at the program level. Kennedy and McKinley [16] and Ding and Kennedy [7] showed that optimal loop fusion is NP hard. Petrank and Rawitz showed that given the order of data access and cache management, the problem of optimal data layout is intractable unless P=NP [21]. We showed that collaborative caching, in particular, bypass LRU and trespass LRU [11], LRU-MRU [12], and now priority LRU can be used to obtain optimal cache management. Sugumar and Abraham gave an efficient algorithm for simulating OPT [28]. We used their algorithm in off-line training to select optimal LRU-MRU hints [12]. With priority LRU, we can now encode optimal hints for caches of all sizes.

6. Summary

In this paper, we have presented priority LRU and generalized the theory of collaborative caching. We proved the inclusion property by a careful consideration of all possible effects of priorities on cache management. More interestingly, through the theorem (and an example), we show non-uniform inclusion, which is a new category of inclusive cache that has not been explored in the previous literature. We give an algorithm to compute the priority LRU stack distance. The algorithm is radically different from previous solutions and can solve the problem of non-uniform inclusion. Finally, we show that the same priority hints can obtain optimal caching for all cache sizes, without having to know the cache size beforehand. This removes a limitation in the previous work and provides new ways for dealing with the remaining difficulties in practice.

Acknowledgments

The idea of priority hints was suggested by Kathryn McKinley, which was the starting point of this entire work. We also wish to thank Michael Scott, Engin Ipek, Tongxin Bai, and anonymous reviewers for their helpful comments.

References

[1] *IA-64 Application Developer's Architecture Guide*. May 1999.

[2] L. A. Belady, R. A. Nelson, and G. S. Shedler. An anomaly in space-time characteristics of certain programs running in a paging machine. *Communications of ACM*, 12(6):349–353, 1969.

[3] K. Beyls and E. D'Hollander. Reuse distance-based cache hint selection. In *Proceedings of the 8th International Euro-Par Conference*, Paderborn, Germany, Aug. 2002.

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

[5] C. Cascaval and D. A. Padua. Estimating cache misses and locality using stack distances. In *Proceedings of the International Conference on Supercomputing*, pages 150–159, 2003.

[6] A. Chauhan and C.-Y. Shei. Static reuse distances for locality-based optimizations in MATLAB. In *Proceedings of the International Conference on Supercomputing*, pages 295–304, 2010.

[7] C. Ding and K. Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. *Journal of Parallel and Distributed Computing*, 64(1):108–134, 2004.

[8] X. Ding, K. Wang, and X. Zhang. ULCC: a user-level facility for optimizing shared cache performance on multicores. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 103–112, 2011.

[9] C. Fang, S. Carr, S. Önder, and Z. Wang. Instruction based memory distance analysis and its application. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, pages 27–37, 2005.

[10] M. Feng, C. Tian, C. Lin, and R. Gupta. Dynamic access distance driven cache replacement. *ACM Transactions on Architecture and Code Optimization*, 8(3):14, 2011.

[11] X. Gu, T. Bai, Y. Gao, C. Zhang, R. Archambault, and C. Ding. P-OPT: Program-directed optimal cache management. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, pages 217–231, 2008.

[12] X. Gu and C. Ding. On the theory and potential of LRU-MRU collaborative cache management. In *Proceedings of the International Symposium on Memory Management*, pages 43–54, 2011.

[13] M. D. Hill. *Aspects of cache memory and instruction buffer performance*. PhD thesis, University of California, Berkeley, Nov. 1987.

[14] S. Jiang, F. Chen, and X. Zhang. CLOCK-Pro: An effective improvement of the clock replacement. In *USENIX Annual Technical Conference, General Track*, pages 323–336, 2005.

[15] S. Jiang and X. Zhang. LIRS: an efficient low inter-reference recency set replacement to improve buffer cache performance. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, Marina Del Rey, California, June 2002.

[16] K. Kennedy and K. S. McKinley. Typed fusion with applications to parallel and sequential code generation. Technical Report TR93-208, Dept. of Computer Science, Rice University, Aug. 1993. (also available as CRPC-TR94370).

[17] S. M. Khan, D. A. Jiménez, D. Burger, and B. Falsafi. Using dead blocks as a virtual victim cache. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 489–500, New York, NY, USA, 2010. ACM.

[18] A.-C. Lai, C. Fide, and B. Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *ISCA*, pages 144–154, 2001.

[19] R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM System Journal*, 9(2):78–117, 1970.

[20] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[21] E. Petrank and D. Rawitz. The hardness of cache conscious data placement. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Portland, Oregon, Jan. 2002.

[22] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. S. Jr., and J. S. Emer. Adaptive insertion policies for high performance caching. In *Proceedings of the International Symposium on Computer Architecture*, pages 381–391, San Diego, California, USA, June 2007.

[23] S. Rus, R. Ashok, and D. X. Li. Automated locality optimization based on the reuse distance of string operations. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 181–190, 2011.

[24] D. L. Schuff, M. Kulkarni, and V. S. Pai. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, pages 53–64, 2010.

[25] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. Power5 system microarchitecture. *IBM J. Res. Dev.*, 49:505–521, July 2005.

[26] Y. Smaragdakis, S. Kaplan, and P. Wilson. The EELRU adaptive replacement algorithm. *Perform. Eval.*, 53(2):93–123, 2003.

[27] K. So and R. N. Rechtschaffen. Cache operations by MRU change. *IEEE Transactions on Computers*, 37(6):700–709, 1988.

[28] R. A. Sugumar and S. G. Abraham. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, Santa Clara, CA, May 1993.

[29] Z. Wang, K. S. McKinley, A. L.Rosenberg, and C. C. Weems. Using the compiler to improve cache replacement decisions. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, Charlottesville, Virginia, 2002.

[30] M. E. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.

[31] L. Xiang, T. Chen, Q. Shi, and W. Hu. Less reused filter: improving L2 cache performance via filtering less reused lines. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 68–79, New York, NY, USA, 2009. ACM.

[32] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. CRAMM: Virtual memory support for garbage-collected applications. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, pages 103–116, 2006.

[33] X. Yang, S. M. Blackburn, D. Frampton, J. B. Sartor, and K. S. McKinley. Why nothing matters: the impact of zeroing. In *OOPSLA*, pages 307–324, 2011.

[34] M. Zahran and S. A. McKee. Global management of cache hierarchies. In *Proceedings of the 7th ACM international conference on Computing frontiers*, CF '10, pages 131–140, New York, NY, USA, 2010. ACM.

[35] C. Zhang and M. Hirzel. Online phase-adaptive data layout selection. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 309–334, 2008.

[36] Y. Zhong and W. Chang. Sampling-based program locality approximation. In *Proceedings of the International Symposium on Memory Management*, pages 91–100, 2008.

[37] Y. Zhong, X. Shen, and C. Ding. Program locality analysis using reuse distance. *ACM Transactions on Programming Languages and Systems*, 31(6):1–39, Aug. 2009.

[38] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 177–188, 2004.