Optimal Collaborative Caching: Theory and Applications

by

Xiaoming Gu

Submitted in Partial Fulfillment

of the

Requirements for the Degree

Doctor of Philosophy

Supervised by

Professor Chen Ding

Department of Computer Science
Arts, Sciences & Engineering
Edmund A. Hajim School of Engineering & Applied Sciences

University of Rochester
Rochester, New York

2014
Biographical Sketch

Xiaoming Gu was born in Handan, Hebei, China, in 1980. He started his study in computer science at the University of Science and Technology of China in 1998 and received his B.S. degree there in 2003. In 2006, he received his first M.S. degree at the Institute of Computing Technology, Chinese Academy of Sciences. In 2008, he received his second M.S. degree at the University of Rochester. After one-year of work at Intel Beijing, he continued his Ph.D. study at the University of Rochester in 2009 under the direction of Professor Chen Ding. He did a summer internship at AMD in 2011. Since the summer of 2012, he has been a full-time software engineer at Azul Systems.

Much of this dissertation is from the following four workshops and conference papers on optimal collaborative caching:

- **P-OPT: Program-directed Optimal Cache Management**, Xiaoming Gu, Tongxin Bai, Yaoqing Gao, Chengliang Zhang, Roch Archambault, and Chen Ding, 21st Workshop on Languages and Compilers for Parallel Computing (LCPC’08)

- **On the Theory and Potential of LRU-MRU Collaborative Cache Management**, Xiaoming Gu and Chen Ding, 10th International Symposium on Memory Management (ISMM’11)

- **A Generalized Theory of Collaborative Caching**, Xiaoming Gu and Chen Ding, 11th International Symposium on Memory Management (ISMM’12)
Other than the findings described in this dissertation, the author has studied several other areas:

- Reuse distance distribution for random access
  
  *Reuse Distance Distribution in Random Access (position abstract)*, Xiaoming Gu and Chen Ding, 5th Workshop on Memory Systems Performance and Correctness (MSPC’08)

- Spatial locality modeling for program tuning
  
  *A Component Model of Spatial Locality*, Xiaoming Gu, Ian Christopher, Tongxin Bai, Chengliang Zhang, and Chen Ding, 8th International Symposium on Memory Management (ISMM’09)

- Software parallelization by hint
  
  *Continuous Speculative Program Parallelization in Software (poster paper)*, Chao Zhang, Chen Ding, Xiaoming Gu, Kirk Kelsey, Tongxin Bai, and Chen Ding, 15th Symposium on Principles and Practice of Parallel Programming (PPoPP’10)

- Resource-based memory management
  
  *Waste Not, Want not: Resource-based Garbage Collection in a Shared Environment*, Matthew Hertz, Stephen Kane, Elizabeth Keudel, Tongxin Bai, Chen Ding, Xiaoming Gu, and Jonathan Bard, 10th International Symposium on Memory Management (ISMM’11)
Acknowledgments

First of all, I would like to sincerely thank my advisor, Professor Chen Ding. Without him, I cannot imagine that I would have been able to finish my Ph.D. study. He is a role model for me in many ways.

Professor Hertz, Professor Ipek, and Professor Scott have sat patiently on my thesis committee for years. I appreciate their insightful suggestions and invaluable time investment very much.

I received help from all the people in the department. They made my study much smoother. In particular, I would like to thank the students in the compiler group: Chengliang Zhang, Kirk Kelsey, Tongxin Bai, Bin Bao, Xiaoya Xiang, Hao Luo, and Jacob Brock. In this dissertation, Jacob Brock made a significant contribution to the loop splitting part, and Bin Bao performed the tests on real hardware. I greatly appreciate their invaluable efforts. I also received considerable help from Hao Zhang, Arrvindh Shriraman, Xiao Zhang, Qi Ge, Daphne Liu, Amal Fahad, Licheng Fang, Zhuan Chen, Li Lu, and Marzieh Bazrafshan.

Most of this dissertation comes from my previous publications. I would like to thank Jorge Albericio, Roch Archambault, Luke K. Dalessandro, Sandhya Dwarkadas, Yaoqing Gao, Li Shen, Xipeng Shen, Lingxiang Xiang, and the anonymous reviewers at LCPC’08, ISMM’11, ISMM’12, and ISMM’13. Special appreciation goes to Kathryn McKinley, who suggested the idea of priority hints.

I would also like to give my sincerest thanks to my colleagues at Azul Systems. They helped me a lot in my full-time job. I cannot imagine that I would have had
time for my dissertation without their help. I really enjoy working with them, especially my manager Bean Anderson. I am proud of working with these great people.

This dissertation is dedicated to my parents and sister. I offer them my thanks for their unconditional love, which kept me warm. My final thanks belong to my girlfriend Zhou Yi, who brings a great deal of fun into my life.
Abstract

On most modern computers, most memory accesses happen in cache, and its usage increasingly affects the performance, stability, and energy consumption of the whole system. The traditional solution divides the memory problem and conquers the pieces separately: software to improve cache locality and hardware to improve cache speed and capacity. A recent approach called collaborative caching breaks the rigid division by allowing software to provide cache hints to influence hardware cache management. In traditional caching, the hardware infers the “importance” of data and manages cache based on the inferred priority. In collaborative caching, software specifies the “importance” of data and changes the priority in which the hardware manages the data.

The dissertation presents the theory and applications of optimal collaborative caching. Through software-hardware collaboration, the goal is not to improve existing solutions but to obtain the optimal solution. Toward this goal, I tackle both theoretical and practical issues. I show in theory that efficient hardware similar to the existing cache is sufficient to produce optimal results. Based on the theory, I describe practical techniques on the software side and evaluate the combined solution on both simulated and real hardware.
Contributors and Funding Sources

This work was supervised by a dissertation committee consisting of Professors Chen Ding, Engin Ipek, and Michael Scott of the Department of Computer Science at the University of Rochester and Professor Matthew Hertz of the Department of Computer Science at Canisius College. Section 4.3 was conducted in part by Jacob Brock and Bin Bao. All other work conducted for the dissertation was completed by Xiaoming Gu independently. Graduate study was supported by NSF awards CNS-0834566 and CCF-0963759, and by an IBM fellowship #TOR08003-12.
Table of Contents

Biographical Sketch ii

Acknowledgments iv

Abstract vi

Contributors and Funding Sources vii

List of Figures xii

List of Tables xvi

1 Introduction 1

1.1 The Thesis ................................................. 1

1.2 Memory Hierarchy and Cache .......................... 2

1.3 From LRU to Optimal ................................. 4

1.3.1 Previous Solutions ................................. 5

1.3.2 From OPT Cache to Collaborative Cache ........... 6

1.3.3 The Belady Anomaly .............................. 8

1.3.4 From One-Size to All-Size Optimization .......... 8

1.4 Contributions and Organization ....... 10
2 Related Work

2.1 Collaborative Cache Hardware ........................................... 12
2.2 Collaborative Caching Software ........................................... 13
   2.2.1 Hint Insertion ......................................................... 13
   2.2.2 Cache Partitioning .................................................... 14
2.3 Non-collaborative Solutions .............................................. 15
   2.3.1 Inclusion Property and Stack Distance .............................. 15
   2.3.2 Program Analysis and Optimization ................................ 15
   2.3.3 Non-LRU Cache ...................................................... 16
   2.3.4 Memory Management ................................................ 17
2.4 Optimal Caching ............................................................ 17

3 Theoretical Properties of Collaborative Caching ...................... 19

3.1 Introduction .............................................................. 19
3.2 Background on Non-collaborative Caching ............................. 22
3.3 LRU-MRU Cache .......................................................... 26
   3.3.1 Cache Design ......................................................... 26
   3.3.2 Optimal LRU-MRU Hints ............................................ 28
   3.3.3 Optimality ............................................................ 30
   3.3.4 Multi-size Optimality ............................................... 33
   3.3.5 The OPT* Algorithm ............................................... 35
   3.3.6 Inclusion Property ................................................ 36
   3.3.7 LRU-MRU Stack Distance ........................................... 38
3.4 Trespass LRU Cache ...................................................... 46
   3.4.1 Optimality ............................................................ 47
### 3.4.2 Multi-size Optimality

Priority LRU Cache

3.5.1 Inclusion Property

3.5.2 Non-uniform Inclusion

3.5.3 Priority LRU Stack Distance

3.5.4 Optimality of Priority Hint for All Cache Sizes

3.6 Inclusive Cache Hierarchy

3.7 Summary

---

### 4 Pacman: Program-assisted Cache Management

4.1 Introduction

4.2 Reference-based Pacman

4.2.1 The Design

4.2.2 Experiment Setup

4.2.3 The LRU-OPT Gap

4.2.4 The Effect of Reference Hints

4.2.5 The Effect of Program Input

4.2.6 The Impact of the MRU Ratio Threshold

4.2.7 A Closer Look at SOR

4.2.8 The Miss Ratio Curves of LRU, OPT, and Pacman

4.3 Loop-based Pacman

4.3.1 Forward OPT Distance Profiling

4.3.2 Pattern Recognition

4.3.3 Loop Splitting and Hint Insertion

4.3.4 Cross-Input Pattern Prediction
4.3.5 Evaluation Setup ......................... 107
4.3.6 Optimal Caching of Group Spatial Reuse in SOR ....... 107
4.3.7 Program-directed Cache Allocation in Swim ............ 111
4.3.8 Six Other Programs .......................... 113
4.3.9 Comparison to Dynamic Insertion Policy ............... 117
4.3.10 Pacman on Real Hardware .................. 119
4.4 Summary ..................................... 122

5 Conclusions and Future Work .................. 124

Bibliography .................................... 127
List of Figures

1.1 LRU & OPT miss ratios of a streaming application on power of 2 cache sizes from 64KB to 32MB. OPT may divide a large working set for caching but LRU does not. ........................................ 4

3.1 Normal LRU at a miss: \( w \) is placed at the top of the stack, evicting \( S_m \). .................................................. 26

3.2 Normal LRU at hit: \( w \), assuming at entry \( S_3 \), is moved to the top of the stack. ........................................ 26

3.3 Bypass MRU at a miss: the bypass posits \( w \) at the bottom of the stack, evicting \( S_m \). .................................................. 27

3.4 Bypass MRU at a hit: the bypass moves \( S_3(w) \) to the bottom of the stack. .................................................. 27

3.5 An example of optimal LRU-MRU hint insertion. For each access, the forward OPT distance is the OPT distance for the following access to the same data element, and the optimal hint for each access is LRU if and only if the forward OPT distance is equal to or less than the cache size. ........................................ 29

3.6 Comparing LRU, MRU, OPT, and LRU-MRU. LRU-MRU is optimal for the targeted cache size. .................. 30
3.7 $a_i$ is selected for bypass for a given cache size during an OPT cache simulation. .................................................. 31
3.8 Trespass MRU at a miss: the trespass posits $w$ at the top of the stack, evicting $S_1$ ................................. 47
3.9 Trespass MRU at a hit: the trespass raises $S_3(w)$ to the top of the stack, evicting $S_1$ ................................. 47
3.10 Element $a_i$ is selected for trespass for a given cache size during an OPT cache simulation .................... 47
3.11 Two cases of data hit in the priority cache when the data block $w$, at position $j$ in cache, is accessed with a priority $i$. ............... 54
3.12 Two cases of data hit in the priority cache when the data block $w$, at position $j$ in cache, is accessed with a priority $i$. ............... 55
3.13 Two cases of data miss in the priority cache when the data block $w$, not in cache before the access, is accessed with priority $i$. . . . 56
3.14 An example of non-uniform inclusion. The Priority LRU observes the inclusion principle but permits data to locate 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. .................................................. 63
3.15 For the same trace in Figure 3.14, the access at time 9 is a miss in the size-4 cache. .................................................. 65
3.16 An example of Priority LRU stack simulation. The trace has nine accesses to four 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. ............... 66
3.17 The following five steps of the example of Priority LRU stack simulation appear in Figure 3.16. 

3.18 The Inclusive Cache Hierarchy: Inclusive caches are organized in a hierarchy based on the “implemented-by” relationship. Limited collaborative caching of LRU-MRU in Section 3.3 subsumes non-collaborative schemes of LRU, MRU and OPT [41]. Priority LRU subsumes LRU-MRU and other prior collaborative caches. 

4.1 An example of reference-based pacman hint insertion. 

4.2 The miss curves of 189.lucas on fully associative caches 

4.3 The miss curves of 434.zeusmp on fully associative caches 

4.4 The miss curves of 171.swim on two different inputs. The curves have an identical shape but cover different cache-size ranges: between 1KB and 4MB in the upper graph and between 1KB and 16MB in the lower graph. 

4.5 The impact of the MRU ratio threshold for 173.applu at 512KB. 

4.6 The miss curves of 173.applu on fully associative caches 

4.7 The SOR kernel computation 

4.8 The gap between LRU and OPT in SOR 

4.9 The SOR kernel loop in SSA form with Pacman transformation. \( M = N = 512 \) and \( \text{NUM_STEPS} = 10 \). 

4.10 The MRU ratio curves of SOR on fully associative caches with cache line size 64B. 

4.11 The miss curves of SOR on fully associative caches with cache line size 64B 

4.12 172.mgrid 

4.13 183.equake
4.14  410.bwaves ................................................................. 96
4.15  433.milc ................................................................. 97
4.16  437.leslie3d ......................................................... 97
4.17  An example of loop-based Pacman hint insertion. .............. 98
4.18  The OPT distances exhibited by a reference in an execution of Swim.101
4.19  The percent reduction in cache misses of SOR by Pacman and OPT
      over LRU ................................................................. 110
4.20  Pacman is tested on swim when the input size (b), array shape (c)
      and both (d) change. The executions in (a,b) use the same loop
      splitting points, so do the executions (c,d). ....................... 112
4.21  The improvements by Pacman and OPT over LRU ................. 114
4.22  The improvements by Pacman and OPT over LRU ................. 115
4.23  One of the representative OPT distance patterns in applu. The two
      graphs show the same reference with the same series of the OPT
      distances as the y coordinate but with different x coordinates as
      the iteration count in two of the five enclosing loops. ............ 116
4.24  An OpenMP example: the inner loop updates the array element by
      element; the outer loop corresponds to the time step. ............ 119
4.25  The performance comparison on Intel Xeon E5520 with hardware
      prefetching ............................................................... 120
4.26  The performance comparison on Intel Xeon E5520 without hard-
      ware prefetching ........................................................ 121
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>An example for LRU cache</td>
<td>23</td>
</tr>
<tr>
<td>3.2</td>
<td>An example for MRU cache</td>
<td>25</td>
</tr>
<tr>
<td>3.3</td>
<td>An example for OPT cache</td>
<td>25</td>
</tr>
<tr>
<td>3.4</td>
<td>An example showing optimal LRU-MRU with cache size 2</td>
<td>31</td>
</tr>
<tr>
<td>3.5</td>
<td>An example of LRU-MRU with cache size 3. The memory trace is the same one used in Table 3.4. The two examples together show that multi-size optimal LRU-MRU does not have the inclusion property.</td>
<td>34</td>
</tr>
<tr>
<td>3.6</td>
<td>Example one-pass simulation of LRU-MRU cache</td>
<td>42</td>
</tr>
<tr>
<td>3.7</td>
<td>Two examples showing Trespass LRU can be optimal and multi-size optimal Trespass LRU holds inclusion property (unlike multi-size optimal LRU-MRU)</td>
<td>48</td>
</tr>
<tr>
<td>3.8</td>
<td>The nine cases for the next access $x_{t+1}$ to $d'$ with a priority $p'$</td>
<td>57</td>
</tr>
<tr>
<td>3.9</td>
<td>In the six subcases of Case I in Table 3.8, 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 3.11 and 3.12 except the bypass case.</td>
<td>57</td>
</tr>
<tr>
<td>3.10</td>
<td>In the three subcases of case II in Table 3.8, the access $x_{t+1}$ misses in $C_1$ but hits in $C_2$. The hit and miss cases are shown in Figures 3.11, 3.12 and 3.13.</td>
<td>59</td>
</tr>
</tbody>
</table>
3.11 The three subcases of $x_{t+1}$ of case V ........................................... 60
3.12 The measured overhead of Algorithm 3.2 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 the worst cost is still far smaller than the theoretical upper bound. . . 73

4.1 The 10 test programs ................................................................. 83
4.2 The LRU-OPT gap and the Pacman improvement. The average improvement is the arithmetic mean of the improvement for all cache sizes between 1KB and data size. ................................. 85
4.3 The statistics of the 8 workloads ............................................. 108
4.4 Pacman makes full utilization of cache space and gradually reduces the miss ratio as the cache size increases ................................. 110
1 Introduction

1.1 The Thesis

Since the late 1990s, an increasing number of hardware systems have been built or proposed to provide a cache hint interface for software to influence cache management. Examples include cache hints on Intel Itanium [11], bypassing and non-temporal accesses on IBM Power series [50], evict-me bit [57], and non-temporal stores on Intel x86 processors. Wang et al. called a combined software-hardware solution collaborative caching [57].

As a form of memory, cache only sees the past and present data accesses. The traditional, non-collaborative management is inherently limited by what it has seen. Collaborative caching allows a program to provide hints about its data usage so that hardware can improve the cache utilization by keeping the most active program data in cache. It exploits the synergy between the hardware considering past behavior and the software supplying the information about the future.

The techniques of collaborative caching were pioneered by a pair of studies. Wang et al. used the dependence information to label “evict-me” bits for data sets too large to fit in cache [57]. Beyls and D'Hollander placed data into different
levels of the cache based on their reuse distances [11]. In these and many other studies, the goal is to improve the existing cache management. The effect of past work depends on specific designs and design parameters. It is unclear what the ultimate potential of collaborative caching is. While significant improvements have been made, it is not clear how much of potential remains.

It is the author’s belief that collaborative caching holds the key to solving the long-standing problem of optimal cache management. Toward this goal, this dissertation studies optimal collaborative caching.

**Thesis Statement.** *Optimal cache management can be done efficiently through software-hardware collaboration. Collaborative hardware can be as simple and efficient as existing cache and as robust against incorrect software control. Collaborative software can be general and optimized for all cache sizes.*

### 1.2 Memory Hierarchy and Cache

In computing, there is a fundamental conflict between computing speed and data capacity.\(^1\) A signal cannot travel faster than the speed of light; consequently, the faster the data access is, the smaller the amount of data it can reach. The solution is to trade space for speed. The data is stored in a memory hierarchy. At each level, the capacity increases while the access speed decreases.

There are two other ways for modern computers to trade space for speed. Firstly, computer cache uses a special structure called the SRAM, which is ten times faster than DRAM, the structure used for main memory, at the cost of storing only one fourth or one sixth as much data. Secondly, to store data more densely, it is not a good idea to run a wire to each memory or cache cell, so the bandwidth of data access is limited in exchange for a higher data capacity.

---

\(^1\)The characterization in the first two paragraphs is based primarily on Peter Sanders in Section 1.1 in [43].
Memory hierarchy provides fast access to active data. Data is active if it is being used by a running program. Denning defined the set of data that a program uses frequently as the working set [18,19]. Depending on different levels of frequencies, there are multiple levels or multiple working sets.

It is an unavoidable problem for any computing system to find ways to manage active data in a limited space. There are two basic sub-problems: data placement and data replacement. When a piece of data is needed, it has to be brought in. If the available space is full, some resident data must be selected and evicted.

Computer cache is a level of memory hierarchy where hardware manages the data. It improves programmability because the decisions of data placement and replacement are done in hardware. Software sees only a uniform memory. There is not need to change a program when the program is run on a machine with a larger cache.

The cache interface represents a fundamental division of labor: the consumption of data in software and the management by hardware. To remove the interdependence, the division is ”air tight”; that is, software has no direct control over data management. So far, the division is effective and pervasive. The problems of cache implementation are well encapsulated, and computer industry has found highly optimized solutions.

The dominant solution is based on the management policy called Least Recently Used (LRU). At eviction, the data that is least recently used (remains unused the longest) is replaced. On the software side, LRU keeps the most recently used data and favors the working set of a program. It supports programs with good locality. The currently used data or their neighbors are likely reused in the near future. On the hardware side, LRU is simple and can be implemented efficiently. Set associativity has a fraction of the cost of the full LRU but can have the same effect once the associativity is reasonably high [12]. Pseudo-LRU takes only $N - 1$ bits per $N$-way cache set instead of $\log_2 N$! bits in LRU [53].
Research on collaborative caching breaks through this long accepted limitation and opens new channels of control for software over hardware. Before making the move, we must have compelling reasons and must have a good understanding of the difficulties that are involved.

1.3 From LRU to Optimal

LRU has a well-known thrashing problem. If the working set of a program is too large to cache, no cache can provide reuse for the whole data. The problem with LRU is that it may not provide reuse for any data.

Consider a streaming program that repeatedly traverses a 5MB array 1000 times. Figure 1.1 shows the miss ratio curves for cache sizes at all power-of-two numbers between 64KB and 32MB. The first is LRU cache, 16-way set associative with 64-byte cache blocks. When the cache size is smaller than 5MB, LRU has no effective cache reuse. It has the same high miss ratio at the 4MB size as it is at...
the 64KB size. The reason is thrashing. The previous data is constantly evicted by the new data. Only when the cache size exceeds 5MB does the miss ratio drop to zero. The sharp knee of the miss ratio curve shows the size of the working set.

A better solution is to keep a portion of the working set in cache. If the cache size is 4MB, we can store 4MB of the 5MB in cache and leave the remaining 1MB outside. This cannot be done by LRU, which has to store the 1MB portion when it is accessed. The second curve shows another replacement policy called OPT, which stores a part of the working set that fits in cache. The miss ratio drops proportionally with the increase in cache size.

The example shows that software may better manage cache memory by targeting specific parts of working sets. Despite the theorized performance bounds between LRU and optimal, this is an example showing that LRU can be arbitrarily inferior to what is obtainable by incorporating software knowledge.

1.3.1 Previous Solutions

The thrashing problem in LRU has been extensively studied. Well-established solutions in program optimization include cache blocking, loop fusion, many other loop nest transformations and structure splitting, and many other cache conscious data and code layout techniques [6,21,70]. In hardware cache design, the solutions include dead-block predictor, forward time distance predictor, adaptive cache insertion, less reuse filter, virtual victim cache, and globalized placement.

While compiler optimization is purely software, the non-LRU cache design is purely hardware. A software solution can change the data usage but not the management policy in hardware. A hardware solution can change the management policy but not the data usage pattern in software. They are fundamentally limited by the rigid software-hardware border in the traditional cache interface.
An increasing number of modern machines have augmented the traditional cache interface to permit software control. Examples include the non-temporal MOVNTQ SSE instruction on x86 in 1999, the MOVNTPD instruction for non-temporal writes in SSE2 in 2001, the placement hints on Intel Itanium processors since 2001, and bypass memory instructions on IBM machines since Power 5 in 2005. These instructions are called cache hints.

A number of techniques have been developed using these hints. They include the pioneering work in collaborative caching as a compiler optimization in 2002 [10, 11, 57] and more recently specialized uses in re-initializing memory in a garbage collector [66] and in string processing [47] in 2011. These studies had the goal of improving the LRU management with program knowledge and were effective in doing so. However, the significance is more profound than the practical improvements. By implicating software, cache management is no longer limited by the traditional software-hardware division. Inspired by and complementary to these earlier studies, this dissertation explores the limit of this synergy—the range of possibilities that arise when combining software flexibility and hardware efficiency.

1.3.2 From OPT Cache to Collaborative Cache

This work aims not to improve LRU but to optimize cache management. The goal is not a better solution but the best possible solution. There are challenges in theory and in practice.

First, we need a theory of optimal collaborative caching. The past work makes incremental improvement over LRU. The claim of improvement can be demonstrated by comparing these improvements against one or more baseline designs. The claim of optimality, however, must be accompanied by a proof that no other cache management can do better.
Secondly, for a theory to matter in practice, there must be a way to realize it. Consider the OPT replacement policy, which is to replace the data whose next use is the furthest in time. OPT is known to be optimal since 1960s. No OPT cache has ever been built nor will it be because OPT is impractical.

There are three main obstacles for hardware OPT. First, OPT requires clairvoyance. It needs to know the future of data accesses. Second, assuming clairvoyance, OPT is time and space inefficient to implement. Each data item has to remember its next access time, and each access triggers a comparison of access times. In comparison, LRU wastes no space in storing any time tags and naturally no operations in comparing them. Considering how cache must be optimized for space and speed, it is questionable whether we can add these overheads and still produce performance competitive to the ultra-optimized and dense LRU. Third, beside the complexity of management inside the cache, another problem is the cost of communication between software and hardware, if we use collaborative caching. Conveying the access time requires the encoding it and adding extra bits to every memory access.

Now we consider the three problems of OPT cache can be solved by collaborative cache. In collaborative caching, the burden of clairvoyance is moved to software. The problem is actually easier. While a hardware solution would have to look through possibly billions of data accesses to find the next reuse, a compiler can derive the information from the program code. If the code is complex and not amenable to program analysis, program behavior analysis can infer the data usage patterns in large executions by analyzing the traces of smaller training runs.

The next two problems are the internal complexity and the interface complexity of the collaborative cache. We will show that collaborative cache can use simple variations of LRU in the place of OPT, using the software hints to eliminate the need for storing access times or performing comparison logic. The cache interface can use just one extra bit for each memory access. Chapter 3 gives the
formal proofs of the optimality of the collaborative cache.

1.3.3 The Belady Anomaly

The Belady anomaly happens if a larger cache incurs more misses than a smaller cache does [7]. It is easy to see that optimal cache management rules out the Belady anomaly. A larger cache can at least be used as a smaller cache, so the best solution is at least as good as before.

We will review the background of traditional, non-collaborative cache in Section 3.2 and show that LRU does not have this anomaly either. When there is more cache space, the performance of LRU cache increases or stays the same but does not decrease.

In collaborative cache, the Belady anomaly would mean that the collaboration on one cache size becomes counter productive in other sizes in that the collaborative cache cannot utilize the additional space. Worse, the addition of space causes it to manage the existing space less efficiently.

Intuitively, the anomaly should not happen. In LRU, the hardware infers the “importance” of data and manages cache based on the inferred priority. In collaborative cache, software specifies the “importance” of data and changes the priority in which the hardware manages the data. As the cache size increases, the data important at the smaller cache remains important. The collaborative cache should be able to cache them at least as well. Indeed, Chapter 3 will prove that this is indeed the case for the majority of collaborative cache designs, where the Belady anomaly cannot happen.

1.3.4 From One-Size to All-Size Optimization

An inherent limitation in optimal collaborative cache is the size dependence of the optimization. For efficiency, we insist on a single bit per access in communicat-
ing between software and hardware. As a result, the collaborative cache cannot optimize for all cache sizes based on this single bit of information. The same hinted execution may be the best one-size solution. Ideally, we want a program be optimized by an all-size solution.

Section 3.5.4 gives a solution where hardware transforms multi-bit priority hints into a single-bit cache hints and enables the use of the same hints to optimize for all cache sizes. Still, it is desirable to solve the all-size optimization problem using only single-bit hints between software and hardware. Chapter 4 presents a practical implementation called Pacman to solve this problem.

Pacman uses off-line profiling analysis. It analyzes the pattern of optimal cache management for all cache sizes. Chapter 4 describes two techniques. The first chooses the hint for each memory reference. It is generally applicable but treats all accesses of the same reference in the same way. The second technique targets loops. It transforms a loop to create differential treatment of its memory accesses based on the iteration counter. The loop-based Pacman automatically changes the software hints for different size caches.

There are many factors affecting a practical implementation. Cache blocks contain multiple data items. Loop analysis must consider both spatial and temporal data reuse in cache management. The cache is set associative, not fully associative. Even within a single set, modern cache is not exactly LRU. We have to evaluate through experimentation. The experiments should compare cache performance for not just a single cache size but as many sizes as possible. The memory usage of a program changes with its input. Pacman should optimize for different inputs or at least provide a robust performance across all inputs. Finally, better caching does not necessarily mean faster running speed. Performance testing on a real machine will consider all factors, including the cost incurred when using special memory instructions. Chapter 4 will address these issues.
1.4 Contributions and Organization

This dissertation makes the following contributions:

1. It lays down the theoretical foundation for collaborative caching:
   - Optimality—LRU-MRU caching can be optimal, and the optimal cache hints can be independent of cache size.
   - LRU-MRU stack distance—LRU-MRU caching is proved as a stack algorithm holding inclusion property. A new stack distance algorithm is devised for it.
   - Generalization—Cache hints are extended to a generalized form from 1-bit for LRU-MRU. Priority LRU, a new stack algorithm based on the generalized priority hint, is discovered holding inclusion property but in a nonuniform way. A new stack distance algorithm is devised for Priority LRU. Generalized cache hints can be used to achieve optimal caching for all cache sizes.

2. It explores the applications of collaborative caching:
   - Reference hints—a heuristic-based solution called Pacman is presented to use optimal trace-level hints to decide program-level reference hints in a binary executable.
   - Refinement by loop splitting—a refined Pacman with loop splitting is proposed to separate run-time accesses with different hints at program level. The cache-size dependent limitation for optimal caching is also removed by using priority hint.

This dissertation is organized as follows. Chapter 2 presents related work. Chapter 3 shows in theory how collaborative caching can use efficient hardware
and still be able to optimize cache management. Chapter 4 presents the Pacman system, which enables a program to determine, efficiently at run time for each memory access, whether the accessed data should be cached or not and to efficiently communicate this decision to hardware at run time at each access. The hints are made robust across input and cache sizes, and there is no trace analysis or code generation at run time. The last chapter is the conclusions and future work.
2 Related Work

2.1 Collaborative Cache Hardware

The ISA of Intel Itanium extends the interface of the memory instruction to provide source and target hints [5]. The source hint suggests where data is expected, and the target hint suggests which level cache the data should be kept. The target hint changes the cache replacement decisions in hardware. IBM Power processors support bypass memory accesses that do not keep the accessed data in cache [50]. Wang et al. proposed an interface to tag cache data with evict-me bits [57]. Another way for software control is cache partitioning. Ding et al. developed a system called ULCC (User Level Cache Control), which uses virtual-to-physical page mapping to partition the cache to separately store high locality and low locality data [23].

The goal in these studies was to allow software control to improve cache management. They do not explore the limit of collaborative caching, i.e. whether the cache can be optimal. This dissertation will show what type of collaborative cache can attain optimal management.
2.2 Collaborative Caching Software

2.2.1 Hint Insertion

Collaborative caching was pioneered by Wang et al. [57] and Beyls and D’Hollander [10, 11]. 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 [42, 57, 58] 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 [10]. 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 [26]. Beyls and D’Hollander later developed a static analysis called reuse-distance equations and obtained similar improvements without profiling [11]. Compiler analysis of reuse distance was also studied by Cascaval and Padua for scientific code [14] and Chauhan and Shei for Matlab programs [15].

The prior methods used reuse distance to identify data in small-size working sets for caching. It was unclear whether and how much cache utilization could be further improved. The goal of this work is optimal collaborative caching. The practical solution, Pacman, uses the OPT distance instead of the reuse distance for caching analysis. In the case when a larger working set is too large, the accesses have the same reuse distance but different OPT distances. By analyzing the difference, Pacman can partition the large working set, choose a partial set to cache and hence utilize the available cache space fully.

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 [66]. Rus et al. used non-temporal prefetches and writes to specialize string operations like \texttt{memcpy()}, based on the data reuse information in certain static program contexts [47]. They showed that significant improvements are already possible by exploiting collaborative caching on current hardware.

### 2.2.2 Cache Partitioning

Cache partitioning as done by ULCC provides practical improvements through software-hardware collaboration [23, 39]. It does not need access hints. Hence there is no additional overhead in memory access or cache management. When we used ULCC to cache 5MB working set in 4MB cache (by assigning the first 3.5MB to use 3.8MB cache and the other 1.5MB to just 128KB cache), we observed a 37% reduction in the execution time. Despite of the large improvement in this case, cache partitioning may not obtain optimal cache management in general. OPT, for example, does not partition the cache for exclusive use. The full cache space is available to all data at all times. Based on the OPT distance, Pacman may choose to cache a piece of data at one time and then choose to keep the same data out of the cache at another time.

Cache partitioning is data based. Collaborative caching is access based. As an allocation scheme, the latter is a form of prioritization rather than partitioning. Rather than allocating cache explicitly between data, Pacman designates some data to be of higher priority than other data. Taking an analogy in operating systems, Pacman is more like CPU scheduling than virtual memory management. When the priority is wrong, the cache space is still utilized. If we partition the cache, an incorrect partition can lead to unutilized space.
2.3 Non-collaborative Solutions

2.3.1 Inclusion Property and Stack Distance

Mattson et al. established the inclusion property and the metrics of stack distance [41]. The miss ratio of inclusive cache is monotonically non-increasing as the cache gets larger (whereas the Belady anomaly [8], more misses in larger cache, is impossible). 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 [70]. The cost can be further reduced by statistical modeling [24, 25, 35, 49], sampling [9,13,48,55,69], footprint-based conversion [20,64], and parallelization on MPI [44] and GPU [16,28]. Recent work has adapted the reuse distance analysis to model the locality in multi-threaded programs [22,35,48,60–62].

Reuse distance and other stack distances cannot be used to analyze the collaborative cache. In fact, it was unknown before this work whether some form of stack distance exists for collaborative cache. This dissertation will prove the inclusion property for two types of collaborative cache and give the algorithms to measure their stack distances. Two new stack distances will be studied. In addition, the thesis will show a new type of inclusion property that gives rise to a new category of caching algorithms.

2.3.2 Program Analysis and Optimization

Much research has been done on improving program locality. Locality was initially defined qualitatively. The textbook concepts of temporal and spatial locality refer to the tendency for the currently accessed data or its neighbors to be accessed in the near future. To measure locality quantitatively, we often use the reuse
distance. A data access is a capacity miss if its reuse distance is greater than the cache size, so the distribution of reuse distances gives the probability of a miss at each access, i.e. the miss ratio, for a fully associative LRU cache.

Locality depends on cache management. Reuse distance assumes that the cache is managed by replacing the least recently used data (LRU). Under this policy, every data element loaded between a pair of data reuses would stay in the cache if the reused data element does. The intermediate data all contribute to the reuse distance. Under optimal cache management (OPT), the intermediate data may or may not be kept in the cache, so the OPT locality can improve over the LRU locality.

OPT locality has often been used to study the potential of caching. In this work, we use OPT distance to direct cache management — to choose which data to cache at what time and to allocate the available cache among different data. With collaborative caching, a program can still have good (OPT) locality even if it cannot have good LRU locality (because of the dependence or other difficulties).

### 2.3.3 Non-LRU Cache

The idea of evicting dead data or least reused data early has been extensively studied in hardware cache design, including dead block predictor [38], forward time distance predictor [27], adaptive cache insertion [46], less reuse filter [63], virtual victim cache [37], and globalized placement [67]. These techniques do not require program changes but they could only collect program information by passive observation. Hardware cache by nature only sees the past and present data access and is inherently limited by what it has seen. Like non-LRU cache, collaborative cache uses non-LRU data management. However, the control is by software. Previous work was aimed to improve the cache not optimize it. For example, Qureshi et al. described a “dueling” strategy to choose the better policy...
between MRU and LRU. This work aims at optimal caching, which is to find the best policy and must require software collaboration.

2.3.4 Memory Management

Garbage collectors may benefit from the knowledge of application working set size and the affinity between memory objects. Reuse distance has been used by virtual machine systems to estimate the working set size [65] and to group simultaneously used objects [68]. 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 [34, 52, 71] and file caching [33]. If collaborative caching is effective for hardware cache, similar solutions may help to improve memory management as well.

2.4 Optimal Caching

The problem of optimal caching has been studied mainly in three forms. The first is the hardware question of optimal cache management: For a given program execution, i.e. a fixed instruction sequence and data layout, how to minimize the number of misses. Here the program is fixed, and the cache is to be optimized. If we fix the cache, we have the software question of program optimization. Given a given cache memory, i.e. a fixed management scheme, how to reorganize a program to minimize the number of misses. In the second problem, the cache is fixed, the program, its computation and data layout, is to be optimized. If we consider both optimization in software organization and hardware management, the ultimate problem is known as minimal I/O complexity. Given a computation with data $D$ stored on disk and loaded into memory of size $M$ ($M < D$) when being computed, how to minimize the number of communication between the memory and the disk.
The three problems are increasingly difficult. Cache management is the only one that is known to be solvable in polynomial time. Before this work, the optimal solution is theoretical. The optimal cache would need the future information that the hardware does not have and require complex and costly operations that the hardware cannot afford to do. This dissertation shows how to solve these two problems using optimal collaborative caching. We will show how software can work with simple hardware to obtain the optimal goal in theory and approach this goal in practice.

Program optimization is more complex because of the dependences that must be preserved in a program execution. Kennedy and McKinley [36] and Ding and Kennedy [21] showed that optimal loop fusion is NP hard. Surprisingly for data layout where there is no constraint on data ordering, Petrank and Rawitz showed that given the order of data access and cache management, the problem of optimal data layout is intractable unless \( P = NP \) [45]. From these results, it is easy to deduce that the I/O complexity problem is NP-hard. The I/O lower-bound has been solved for specific problems by Hong and Kung for matrix-vector and matrix-matrix multiplication, FFT, and odd-even transposition sort [31] and by Vitter for permutation [56]. An extensive overview of the algorithmic issues and solutions can be found in Meyer et al. [43] Given the intractability of program optimization and I/O minimization, optimal cache management is the best possible in theory. This dissertation will establish the theory and explore its applications.
3 Theoretical Properties of Collaborative Caching

This chapter presents three types of collaborative cache and establishes three theoretical properties: optimality, inclusion property, and generality.

3.1 Introduction

In cache as well as memory management, the least recently used replacement policy (LRU) is a common starting point. In theory, Sleator and Tarjan showed that LRU is within a bounded factor of optimal [51]. In practice, all modern microprocessors use some variation of LRU.

For collaborative cache, we use LRU as the basis. To be able to revise the default LRU policy, we add non-LRU access to the cache interface. A program can then use a mix of non-LRU accesses and the default LRU accesses to change the cache management. Cache management becomes collaborative: the default LRU still considers the past history, and the non-LRU variation enables software control based on future information. The non-LRU access is called a software hint.¹

¹LRU accesses are also hints since collaborative caching selects both types of accesses, and the optimal management comes out of the combined effect.
There are two questions with this approach: how much the collaboration via non-LRU accesses can improve over traditional cache and how these special accesses may affect the formal properties of cache, whether they are used judiciously or not.

The chapter studies three designs of collaborative cache, each with a different type of non-LRU access.

- **LRU-MRU cache.** The cache is bipartite where the data blocks accessed normally are managed by LRU, and the data blocks accessed by non-LRU are managed by MRU (which replaces the most recently accessed data upon eviction). The MRU access is a bypass in that the data block will not stay in cache (if the cache size is smaller than the data size).

- **Trespass LRU cache.** The non-LRU access is a trespass in that it causes the cache to evict the most recently accessed data block rather than the least recently accessed data block as in the default policy.

- **Priority LRU cache.** The accessed data block is associated with a numerical priority for hardware to prioritize its cache management.

The first two types are intended for practical implementation. They change the default LRU design only peripherally. The third type is a generalization that can implement all known inclusive caching algorithms, collaborative or not.

For each type, the chapter examines two formal properties. The first is optimality. We will show that they can all be used to effect optimal cache management. Bypass and trespass hints are specific to a given cache size while the same priority hints can be used to optimize for all cache sizes.

The second is inclusion property. The inclusion property was first characterized by Mattson et al. in their seminal paper in 1970 [41]. The property states that
a larger cache always contains the content of a smaller cache. The property is important for at least three reasons.

i) In inclusive caches, the miss ratio is a monotone function of the cache size. There can be no Belady anomaly [8].

ii) The miss ratio of an execution can be simulated in one pass for all cache sizes, known as the stack simulation (Section 3.2).

iii) Most importantly for software analysis, there exists a distance metric known as stack distances (Section 3.2). An access misses in cache if and only if its stack distance exceeds the cache size. For example, the LRU stack distance has been called the reuse distance and used extensively in software and system optimization.

In this chapter, we show that the inclusion property holds for collaborative cache. As an interface, the non-LRU access may be used in arbitrary ways, sometimes optimal but probably suboptimal most times and even counter productive. The inclusion property holds regardless of the usage. The chapter also gives the algorithms to compute the stack distance for collaborative cache: the LRU-MRU distance and the Priority LRU distance.

Being the most general, Priority LRU shows a new type of inclusion which is not uniform. Non-uniform inclusion gives rise to a completely new category of cache not known before this study. It also necessitates a significantly different algorithm to measure its stack distance.

The theoretical findings are the foundation for the application in the next chapter. While in pure theory, we are not concerned with any measure of empirical efficiency, i.e. analysis time and specific cache miss ratio, and will leave these practical considerations for the next chapter.
3.2 Background on Non-collaborative Caching

Mattson’s algorithms use a stack to do the simulation and hence are called stack algorithms [41].

Stack simulation provides a metric called stack distance. Stack distance is useful for program analysis because it is independent of specific cache sizes.

An inclusive cache can be viewed as a stack or a priority list. Data elements at the top $c$ stack positions are the ones in a cache of size $c$. The stack position defines the importance of the stored data. Stack simulation is to simulate cache of an infinite size. Stack distance gives the minimal cache size to make an access a cache hit [41]. A stack distance is defined for each type of inclusive cache and computed by simulating that type of cache in an infinite size.

The following are examples of inclusive cache.

**Least Recently Used (LRU)**

The priority used in LRU cache is 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 [53]. The LRU stack distance is called reuse distance for short. It measures the amount of data accessed between two consecutive uses of the same data element. Reuse distance can be measured in near constant time by organizing the priority list as a dynamically compressed tree [70].

Table 3.1 is an example for LRU cache. The priority list shows the cache content snapshots after each access. For example, the content of a cache of size 3 is the data elements sitting at the top three positions in the priority list. If the data element visited by the next access is in the current top three positions, then

---

2The newly discovered Priority LRU is an exception and will be presented in Section 3.5.
the next access is a hit. Otherwise, it is a miss. For the example trace, there are only three hits with an LRU cache in size 3: access 5, 6, and 7.

The reuse distances are shown at the lowest row in Table 3.1. For example, the reuse distance of No.7 access to data element $b$ is 3 because there are 3 different data elements accessed between the two consecutive accesses to $b$: No.2 and No.7 accesses. This type of reuse distance is called backward reuse distance because the distance looks back from the last access of a reuse pair. On the contrary, forward reuse distance looks forward from the first access of a reuse pair. There are only three reuse distances are no greater than 3—only three hits with an LRU cache of size 3.

<table>
<thead>
<tr>
<th>access time</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>accessed data</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>d</td>
<td>c</td>
<td>b</td>
<td>a</td>
<td>d</td>
<td>c</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>priority</td>
<td>1</td>
<td>a:1</td>
<td>b:2</td>
<td>c:3</td>
<td>d:4</td>
<td>d:5</td>
<td>c:6</td>
<td>b:7</td>
<td>a:8</td>
<td>d:9</td>
<td>c:10</td>
<td>b:11</td>
</tr>
<tr>
<td>list</td>
<td>2</td>
<td>a:1</td>
<td>b:2</td>
<td>c:3</td>
<td>d:5</td>
<td>c:6</td>
<td>b:7</td>
<td>a:8</td>
<td>d:9</td>
<td>c:10</td>
<td>b:11</td>
<td>a:12</td>
</tr>
<tr>
<td>(data-priority)</td>
<td>3</td>
<td>a:1</td>
<td>b:2</td>
<td>b:2</td>
<td>d:5</td>
<td>c:6</td>
<td>b:7</td>
<td>a:8</td>
<td>d:9</td>
<td>c:10</td>
<td>b:11</td>
<td></td>
</tr>
<tr>
<td>reuse distance</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 3.1: An example for LRU cache

The General Rules for Priority List Adjustment

The priority list in the above LRU example is easy to understand because the priority list is ordered. However, LRU is a special case, and a priority list is not necessarily ordered by the used priorities. Two examples of MRU and OPT in Table 3.2 and Table 3.3 shows unordered priority lists.

The general rules for priority list adjustment were first devised by Matterson et al. in their seminal paper [41]. $s_t(i)$ and $v_t(i)$ denote the priority at position $i$ and the victim priority for a cache of size $i$ after No.$t$ access respectively. $p_t$ denotes the priority for No.$t$ access. The general rules for No.$t$ access are:

$$s_t(1) = p_t$$  \hspace*{1cm} (3.1)
$v_t(1) = s_{t-1}(1)$  \hspace{1cm} (3.2)

$s_t(i) = MAX_{or\_MIN}(v_t(i - 1), s_{t-1}(i)) \hspace{0.5cm} (2 \leq i \leq K)$  \hspace{1cm} (3.3)

$v_t(i) = MAX_{or\_MIN}(v_t(i - 1), s_{t-1}(i)) \hspace{0.5cm} (2 \leq i \leq K - 1)$  \hspace{1cm} (3.4)

Equations 3.1 and 3.2 make sure that the data element visited by the current access is brought into the cache for all cache sizes. Equations 3.3 and 3.4 make adjustment from No.2 position to No.$K$ position cooperatively. No.$K$ position is the place where the data element visited by the current access sits before the current access happens. All the data elements in the priority list are adjusted if the current access is a compulsory miss—an access to a new data element [29]. $MAX_{or\_MIN}$ is $MAX$ or $MIN$, which depends on the cache replacement algorithm. The one used in Equation 3.4 is the opposite of the one in Equation 3.3. For LRU, $MAX$ is the one used in Equation 3.3 but $MIN$ in Equation 3.4.

The above equations do not cover the case when $p_t = s_{t-1}(1)$. That case is trivial and no adjustment is needed for the priority list.

**Most Recently Used (MRU)**

The priorities used in MRU are the same as the ones used in LRU—the most recent access time. Unlike LRU, $MIN$ is the one used in Equation 3.3 and $MAX$ in Equation 3.4.

The priority list for MRU is not necessarily ordered. An example is in Figure 3.2. For the same memory trace, there are five hits with an MRU cache of size 3: access 5, 7, 8, 10, and 11. The stack distance row shows backward MRU
stack distances. There are five distances no greater than 3; five hits with an MRU cache of size 3.

### Optimal (OPT)

The priorities used in OPT are the next access time. *MAX* is the one used in Equation 3.3 and *MIN* in Equation 3.4. The setup guarantees the in-cache data element with the furthest reuse is the victim if an eviction is needed. OPT is impractical because it requires future knowledge. It serves 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 [54].

The priority list for OPT is not necessarily ordered. An example is shown in Figure 3.3. For the same memory trace, there are six hits with an OPT cache of size 3: access 5, 6, 7, 9, 10, and 12. Because OPT is optimal, the maximal

<table>
<thead>
<tr>
<th>access time</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>accessed data</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>c</td>
<td>b</td>
<td>a</td>
<td>d</td>
<td>c</td>
<td>b</td>
</tr>
<tr>
<td>priority</td>
<td>1</td>
<td>a:1</td>
<td>b:2</td>
<td>c:3</td>
<td>d:4</td>
<td>d:5</td>
<td>c:6</td>
<td>b:7</td>
<td>a:8</td>
<td>d:9</td>
<td>c:10</td>
<td>b:11</td>
</tr>
<tr>
<td>list</td>
<td>2</td>
<td>a:1</td>
<td>a:1</td>
<td>a:1</td>
<td>a:1</td>
<td>b:7</td>
<td>c:6</td>
<td>b:7</td>
<td>c:6</td>
<td>b:9</td>
<td>d:9</td>
<td>d:9</td>
</tr>
<tr>
<td>(data-</td>
<td>3</td>
<td>b:2</td>
<td>b:2</td>
<td>b:2</td>
<td>c:6</td>
<td>c:6</td>
<td>b:7</td>
<td>b:7</td>
<td>c:10</td>
<td>c:10</td>
<td>c:10</td>
<td>c:10</td>
</tr>
<tr>
<td>priority)</td>
<td>4</td>
<td>c:3</td>
<td>c:3</td>
<td>d:5</td>
<td>d:5</td>
<td>a:8</td>
<td>a:8</td>
<td>a:8</td>
<td>a:12</td>
<td>a:12</td>
<td>a:12</td>
<td>a:12</td>
</tr>
</tbody>
</table>

| MRU stack distance | ∞ | ∞ | ∞ | ∞ | 1 | 4 | 3 | 2 | 4 | 3 | 3 | 4 |

Table 3.2: An example for MRU cache

<table>
<thead>
<tr>
<th>access time</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>accessed data</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>c</td>
<td>b</td>
<td>a</td>
<td>d</td>
<td>c</td>
<td>b</td>
</tr>
<tr>
<td>next access time</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>priority</td>
<td>1</td>
<td>a:8</td>
<td>b:7</td>
<td>c:6</td>
<td>d:5</td>
<td>d:9</td>
<td>c:10</td>
<td>b:11</td>
<td>a:12</td>
<td>d:∞</td>
<td>c:∞</td>
<td>b:∞</td>
</tr>
<tr>
<td>list</td>
<td>2</td>
<td>a:8</td>
<td>b:7</td>
<td>c:6</td>
<td>d:5</td>
<td>d:9</td>
<td>c:10</td>
<td>b:11</td>
<td>a:12</td>
<td>a:12</td>
<td>a:12</td>
<td>b:∞</td>
</tr>
<tr>
<td>(data-</td>
<td>3</td>
<td>a:8</td>
<td>b:7</td>
<td>b:7</td>
<td>b:7</td>
<td>c:10</td>
<td>c:10</td>
<td>d:∞</td>
<td>c:∞</td>
<td>c:∞</td>
<td>c:∞</td>
<td>c:∞</td>
</tr>
<tr>
<td>priority)</td>
<td>4</td>
<td>a:8</td>
<td>a:8</td>
<td>a:8</td>
<td>a:8</td>
<td>b:11</td>
<td>b:11</td>
<td>b:11</td>
<td>d:∞</td>
<td>d:∞</td>
<td>d:∞</td>
<td>d:∞</td>
</tr>
</tbody>
</table>

| OPT stack distance | ∞ | ∞ | ∞ | ∞ | 1 | 2 | 3 | 4 | 2 | 3 | 4 | 2 |

Table 3.3: An example for OPT cache
number of hits for the example trace running with a cache of size 3 is six. The stack distance row shows backward OPT stack distances. There are six distances no greater than 3; that is, six hits with an OPT cache of size 3.

### 3.3 LRU-MRU Cache

#### 3.3.1 Cache Design

In LRU-MRU, an access can be normal LRU or bypass MRU, which are illustrated in Figure 3.1, Figure 3.2, Figure 3.3, and Figure 3.4.

- **Normal LRU access** uses the most recently used position for placement and the least recently used position for replacement.

\[
\begin{array}{ccc}
S_1 & w & S_1 \\
S_2 & S_1 & S_2 \\
S_3 & S_2 & S_3(w) \\
\vdots & \vdots & \vdots \\
S_{m-1} & S_{m-1} & S_{m-1} \\
S_m & S_m & S_m
\end{array}
\]

Figure 3.1: Normal LRU at a miss: \( w \) is placed at the top of the stack, evicting \( S_m \).

\[
\begin{array}{ccc}
S_1 & w & S_1 \\
S_2 & S_1 & S_2 \\
S_3 & S_2 & S_3(w) \\
\vdots & \vdots & \vdots \\
S_{m-1} & S_{m-1} & S_{m-1} \\
S_m & S_m & S_m
\end{array}
\]

Figure 3.2: Normal LRU at hit: \( w \), assuming at entry \( S_3 \), is moved to the top of the stack.

- **Miss:** Evict the data element \( S_m \) at the LRU position (bottom of the stack) if the cache is full, shift other data elements down by one position, and place \( w \), the visited element, in the MRU position (top of the stack). See Figure 3.1.

- **Hit:** Find \( w \) in cache, shift the elements over \( w \) down by one position, and re-insert \( w \) at the MRU position. See Figure 3.2. Note that search
cost is constant in associative cache where hardware checks all entries in parallel.

- **Bypass MRU access** uses the LRU position for placement and the same position for replacement. It is similar to the bypass instruction in IA64 [5] except that its bypass demotes the visited element to LRU position when hit.

\[
\begin{array}{c|c|c|c}
S_1 & S_1 & S_1 & S_1 \\
S_2 & S_2 & S_2 & S_2 \\
S_3 & S_3 & S_3 & S_3 \\
\vdots & \vdots & \vdots & \vdots \\
S_{m-1} & S_{m-1} & S_{m-1} & S_{m-1} \\
w & w & S_m & S_m \\
\end{array}
\quad \Rightarrow \quad
\begin{array}{c|c|c|c}
S_1 & S_1 & S_1 & S_1 \\
S_2 & S_2 & S_2 & S_2 \\
S_3 & S_3 & S_3(w) & S_3(w) \\
\vdots & \vdots & \vdots & \vdots \\
S_{m-1} & S_{m-1} & S_m & S_m \\
w & S_m & S_m & S_m \\
\end{array}
\]

Figure 3.3: Bypass MRU at a miss: the bypass posits \( w \) at the bottom of the stack, evicting \( S_m \).

\[
\begin{array}{c|c|c|c}
S_1 & S_1 & S_1 & S_1 \\
S_2 & S_2 & S_2 & S_2 \\
S_3 & S_3 & S_3 & S_3 \\
\vdots & \vdots & \vdots & \vdots \\
S_{m-1} & S_{m-1} & S_{m-1} & S_{m-1} \\
w & S_m & S_m & S_m \\
\end{array}
\quad \Rightarrow \quad
\begin{array}{c|c|c|c}
S_1 & S_1 & S_1 & S_1 \\
S_2 & S_2 & S_2 & S_2 \\
S_3(w) & S_3(w) & S_3(w) & S_3(w) \\
\vdots & \vdots & \vdots & \vdots \\
S_{m-1} & S_m & S_m & S_m \\
\end{array}
\]

Figure 3.4: Bypass MRU at a hit: the bypass moves \( S_3(w) \) to the bottom of the stack.

- Miss: Evict \( S_m \) at the LRU position if the LRU position is taken and insert \( w \) into the LRU position. See Figure 3.3.
- Hit: Find \( w \), lift the elements under \( w \) by one position, and place \( w \) in the LRU position. See Figure 3.4.

LRU-MRU cache differs from conventional cache in three ways:

- **LRU-MRU content.** The cache stack is divided into two parts: the upper part for LRU data and the lower part for MRU data. Either part may be missing, and the cache is entirely LRU or MRU.

- **Capacity-dependent placement.** The MRU data is placed at the bottom. The location depends on the size of the cache.
- Hybrid priority. The LRU part is prioritized by the LRU order; that is, the last accessed is last replaced. The MRU part is by the MRU order; that is, the last accessed is first replaced.

In comparison, conventional non-collaborative cache manages data using a single priority order, for example, LRU by the last access time and OPT by the next access time. The placement depends on the priority order and not on cache size. The single priority naturally gives rise to the inclusion property and its practical benefits.

In LRU-MRU cache, an access could be either a normal LRU or a bypass MRU. A 1-bit cache hint indicates whether an access is LRU or MRU. As an interface, it may be used in arbitrary ways, sometimes optimal as shown in Section 3.3.2 and proved in Section 3.3.3 but probably suboptimal in most times and even counter productive. OPT*, a faster implementation of the original OPT, is presented in Section 3.3.5 for the training pass to find optimal hints.

It is known that both LRU and MRU are stack algorithms holding the inclusion property. In Section 3.3.6, it is proved that the inclusion property holds even in an LRU-MRU cache. To calculate the LRU-MRU stack distance, a hybrid priority scheme is devised in Section 3.3.7.

### 3.3.2 Optimal LRU-MRU Hints

Given an execution trace and an LRU-MRU collaborative cache of size $c$, the optimal hint for each access is LRU if the forward OPT distance is equal to or less than $c$; otherwise, it is MRU. For each access, the forward OPT distance is the OPT distance of the following access to the same data element. The OPT distance is infinite if there is no follow up reuse.

As an example, consider the data access trace in Figure 3.5. To make it interesting, the trace has mixed locality: two blocks $xy$ are reused frequently,
having high locality; while the other seven blocks abcd
defg (highlighted in red) have a streaming pattern and low locality, as shown in the first row. The second and third rows show the OPT and the forward OPT distances. Assuming a cache size $c = 5$, the optimal LRU-MRU hints are given in the last row.

<table>
<thead>
<tr>
<th>trace</th>
<th>xyaxybxyxcxydxyexyfxygxxybxyxcxydxye</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPT distance</td>
<td>---23-23-23-23-23-23-23-234235236237238</td>
</tr>
<tr>
<td>forward OPT distance</td>
<td>234235236237238239234235236237238239</td>
</tr>
<tr>
<td>optimal hint (c=5)</td>
<td>LLLLLLMMMMLMMLMLMLMLMLMLMLMLMLL</td>
</tr>
</tbody>
</table>

Figure 3.5: An example of optimal LRU-MRU hint insertion. For each access, the forward OPT distance is the OPT distance for the following access to the same data element, and the optimal hint for each access is LRU if and only if the forward OPT distance is equal to or less than the cache size.

When the cache size is 5, the optimal management chooses to store high-locality data, that is, $xy$, in cache, as indicated by the small (forward) OPT distances 2 and 3. It stores two of the low-locality streaming blocks, which have OPT distances of 4 and 5 in the remaining space. The rest is not stored in cache.

The hints replicate the behavior of the optimal cache on the LRU-MRU cache. High-locality $xy$ are accessed by LRU and so are two of the low-locality blocks to utilize the residual space. The rest are by MRU. Through the hints, the simple LRU-MRU cache imitates the optimal cache and stores as much data in the cache as can benefit from it.

As a comparison, Figure 3.6 gives the stack distances for LRU (that is, the reuse distance), MRU, OPT, and LRU-MRU. From the distances we can compute the number of capacity misses for all cache sizes. An access is a miss under a management policy if the distance exceeds the cache size (miss iff $dis > c$).

The distances explain the inner workings of the cache management. The high distances in LRU show its inability to cache any low-locality data. The high distances in MRU show its problem with caching high-locality data. Both OPT and LRU-MRU treat the two working sets separately and always have low dis-
LRU distance ---33-33-33-33-33-33-119119119119 5
MRU distance ---23-34-45-56-67-78-892924345456567 10
OPT distance ---23-23-23-23-23-23-234235236237238 3
LRU-MRU distance ---33-33-32-32-32-32-325334336327328 3

Figure 3.6: Comparing LRU, MRU, OPT, and LRU-MRU. LRU-MRU is optimal for the targeted cache size.

The varying distances are effectively priorities through which these policies select program data to cache.

When the cache size is 5, the miss counts are 5 and 10 for LRU and MRU but just 3 for OPT and LRU-MRU. In fact, for the trace shown above, the hints give the optimal performance for any LRU-MRU cache of size 3 or higher. When the size is 2, the LRU-MRU cache has 3 more misses than OPT. To see this, observe that the collection of LRU-MRU distances differs from the collection of OPT distances only in the number of 2s and 3s. The frequency of the occurrences of other distances is the same. There is one 4 and one 5 in both LRU-MRU and OPT but in a different order. This shows that the optimal cache management is not unique. LRU-MRU is optimal but does not always make identical replacement decisions as OPT.

At the trace level, LRU-MRU must be optimal for the targeted cache size, but the same hints may not be optimal for all cache sizes. At program level, however, we may use program transformation to change the LRU-MRU hints to automatically optimize for all cache sizes, which is discussed in Chapter 4.

3.3.3 Optimality

The type of each access is determined in the OPT training. The details to obtain the optimal cache hints are shown in Figure 3.7. 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 bypass. After training, the unselected accesses still use normal LRU. The training result is specific to the cache size being used.

Figure 3.7: $a_i$ is selected for bypass for a given cache size during an OPT cache simulation.

An example of optimal LRU-MRU is shown in Table 3.4 to demonstrate that optimal LRU-MRU has the same result as OPT.

<table>
<thead>
<tr>
<th>accessed time</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>accessed data</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>d</td>
<td>c</td>
<td>e</td>
<td>b</td>
<td>e</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>OPT cache</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>d</td>
<td>c</td>
<td>e</td>
<td>b</td>
<td>e</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>c</td>
<td>d</td>
<td>c</td>
<td>e</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>misses</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bypasses</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>optimal LRU-MRU cache</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>d</td>
</tr>
<tr>
<td>misses</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.4: An example showing optimal LRU-MRU with cache size 2

We next prove the optimality for all traces.
Lemma 3.1. If the bottom element in the optimal LRU-MRU stack is most recently visited by a normal access, then all cache elements are most recently visited by some normal accesses.

Proof. If some data elements are most recently visited by bypass accesses, then they appear only at the bottom of the stack. They can occupy multiple positions but cannot be lifted up over an element most recently visited by a normal access. Therefore, if the bottom element is most recently visited by a normal access, all elements in the cache must also be. \qed

Theorem 3.1. Optimal LRU-MRU generates no more misses than OPT. In particular, optimal LRU-MRU has a miss only if OPT has a miss.

Proof. We show that there is no access that is a cache hit in OPT but a miss in optimal LRU-MRU. Suppose the contrary were true. Let \( z' \) be the first access in the trace that hits in OPT but misses in optimal LRU-MRU. Let \( d \) be the element accessed by \( z' \), \( z \) be the most recent access to \( d \) before \( z' \), and the reference trace between them be \((z, ..., z')\). The access \( z \) can be one of the two cases:

- \( z \) is a normal access. For \( z' \) to miss in optimal LRU-MRU, there should be a miss \( y \) in \((z, ..., z')\) that evicts \( d \). From the assumption that \( z' \) is the earliest access that is a miss in optimal LRU-MRU but a hit in OPT, \( y \) must be a miss in OPT. Consider the two possible cases of \( y \):
  - \( y \) occurs when the OPT cache is partially full. Because the OPT cache is always full after the loading of the first \( M \) elements, where \( M \) is the cache size, this case can happen only at the beginning. However, when the cache is not full, OPT will not evict any element. Hence this case is impossible.
  - \( y \) occurs when the OPT cache is full. The element \( d \) is at the LRU position before the access of \( y \). According to Lemma 3.1, the optimal
LRU-MRU cache is full, and the most recent accesses of all data elements in cache are normal accesses. Let the set of elements in cache be $T$ for optimal LRU-MRU and $T^*$ for OPT. At this time (before $y$), the two sets must be identical. The reason is a bit tricky. If there is an element $d'$ in the optimal LRU-MRU cache but not in the OPT cache, $d'$ must be replaced by OPT before $y$. However, by the construction of the algorithm, the previous access of $d'$ before $y$ should be labeled a bypass access. This contradicts to the lemma, which says the most recent access of $d'$ (and all other elements in $T$) is normal. Since both caches are full, they must be identical; as a result, we have $T = T^*$. Finally, $y$, in the case of OPT, must evict some element. However, evicting any element other than $d$ would violate our lemma. Hence, such a $y$ cannot exist and this case is impossible.

- $z$ is a bypass access in optimal LRU-MRU. There must be an access $y \in (z, ..., z')$ in the case of OPT that evicts $d$; otherwise $z$ cannot be designated as a bypass. However, in this case, the next access to $d$, $z'$ cannot be a cache hit in OPT, contradicting the assumption that $z'$ is a cache hit in OPT.

Considering both cases, it is impossible for the same access to be a hit in OPT but a miss in optimal LRU-MRU.

### 3.3.4 Multi-size Optimality

If we find LRU-MRU for each cache size, we have the optimal LRU-MRU for all sizes. It has to use different hints for different cache sizes. If we view multi-size optimal LRU-MRU as a single policy, it does not have the inclusion property, which is shown using a counter example. By comparing the two examples of optimal LRU-MRU in Table 3.4 and Table 3.5, we see that at the first access to
e, the stack content, given in bold letters, is \((e,d)\) in the smaller cache and \((e,c,b)\) in the larger cache. Hence the inclusion property does not hold [41].

<table>
<thead>
<tr>
<th>accessed time</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>accessed data</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>d</td>
<td>c</td>
<td>e</td>
<td>b</td>
<td>e</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>OPT</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>d</td>
<td>c</td>
<td>e</td>
<td>b</td>
<td>e</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>cache</td>
<td>a</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>c</td>
<td>c</td>
<td>b</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>misses</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

| bypasses      | X | X | X | X |   |   |   |   |   |   |
| optimal       |   |   |   |   |   |   |   |   |   |   |
| LRU-MRU       |   |   |   |   |   |   |   |   |   |   |
| cache         |   |   |   |   |   |   |   |   |   |   |
| misses        | X | X | X | X | X | X |   | X |   |   |   |

Table 3.5: An example of LRU-MRU with cache size 3. The memory trace is the same one used in Table 3.4. The two examples together show that multi-size optimal LRU-MRU does not have the inclusion property.

Because OPT is optimal, we have the immediate corollary that multi-size optimal LRU-MRU has the same number of misses as OPT and is therefore optimal. In fact, the misses happen for the same accesses in multi-size optimal LRU-MRU and in OPT. Lastly, we show that multi-size optimal LRU-MRU has a peculiar feature.

**Corollary 3.1.** *Multi-size optimal LRU-MRU does not have the inclusion property, but it does not suffer from Belady anomaly [8], in which the number of misses sometimes increases when the cache size becomes larger.*

*Proof.* OPT is a stack algorithm since the stack content for a smaller cache is a subset of the stack content for a larger cache [41]. The number of misses of an access trace does not increase with the cache size. Because multi-size optimal LRU-MRU has the same number of misses as OPT, it has the same number of misses as OPT and does not suffer from Belady anomaly. □
3.3.5 The OPT* Algorithm

A faster implementation of OPT called OPT* is designed to do the OPT training. OPT* is asymptotically faster than the original OPT.

Given a memory access sequence, the original OPT algorithm has two passes [41]:

- First pass: Compute the forward reuse distance for each access through a backward scan of the trace.

- Second pass: Incrementally maintain a priority list based on the forward reuse distance of the cache elements. The pass has two steps. First, if the visited element is not in cache, find its place in the sorted list based on its forward reuse distance. Second, after each access, update the forward reuse distance of each cache element.

The update operation is costly and unnecessary. To maintain the priority list, it is sufficient to use the next access time instead of the forward reuse distance. At each point $p$ in the trace, the next access time of data $x$ is the logical time of the next access of $x$ after $p$. Because the next access time of data $x$ changes only at each access of $x$, OPT* stores a single next access time at each access in the trace, which is the next access time of the element being accessed. OPT* collects next access times through a single pass traversal of the trace. The revised algorithm OPT* is as follows:

- First pass: Store the next reuse time for each access through a backward scan of the trace.

- Second pass: Maintaining the priority list based on the next reuse time. It has a single step. If the visited element is not in cache, find its place in the sorted list based on its next access time.
The cost per operation is $O(\log M)$ for cache of size $M$, if the priority list is maintained as a heap. It is asymptotically more efficient than the $O(M)$ per access cost of OPT. The difference is computationally significant when the cache is large. While OPT* is still costly, it is used only for pre-processing and adds no burden to on-line cache management.

OPT* is an enhanced implementation of the original OPT. In the following discussion, we still use OPT to name the training pass since any OPT implementation works.

### 3.3.6 Inclusion Property

If the collaborative cache is used optimally, the performance is the same as OPT shown in Section 3.3.3. In general, however, the cache may not be used optimally. The selection of MRU accesses may be arbitrary. The following proof is for all uses of LRU-MRU cache, including the extreme cases (when all accesses are normal, i.e. LRU caching, and when all accesses are special, i.e. MRU caching), the optimal use, and everything in between. In a sense, the proof subsumes the individual conclusions for LRU, MRU, and OPT [41].

We prove that for any sequence of LRU and MRU accesses, the LRU-MRU cache obeys the inclusion principle.

**Lemma 3.2.** *If the bottom element in the LRU-MRU cache stack is most recently accessed by a normal LRU access, then all elements in cache are most recently accessed by normal LRU accesses.*

The Lemma 3.2 follows from the fact that MRU data are placed at the bottom of the stack and only replaced by LRU data (never pushed up except by other MRU data). The formal proof is in Lemma 3.1 on page 31. Next we prove the inclusion property.
Theorem 3.2. A trace $P$ is being executed on two LRU-MRU caches of sizes $|C_1|$ and $|C_2|$ ($|C_1| < |C_2|$). At every access, the content of cache $C_1$ is always a subset of the content of cache $C_2$.

Proof. Let the access trace be $P = (x_1, x_2, ..., x_n)$. Let $C_1(t)$ and $C_2(t)$ be the set of elements in cache $C_1$ and $C_2$ after access $x_t$. The data element visited by $x_t$ is $d(x_t)$. The initial cache contents are $C_1(0) = C_2(0) = \emptyset$. The inclusion property holds. We now prove the theorem by induction on $t$.

Assume $C_1(t) \subseteq C_2(t)$ $(0 \leq t \leq n - 1)$. It is easy to see that if $x_{t+1}$ is a hit in $C_2$ ($x_{t+1} \in C_2(t)$), the inclusion property holds. We now consider the case that $x_{t+1}$ is a miss in $C_2$. Since $C_1$ is included in $C_2$, $x_{t+1}$ is also a miss in $C_1$.

Let the evicted elements be most recently accessed at $x_p$ in $C_1$ and $x_q$ in $C_2$. After the cache miss, we have $C_1(t+1) = C_1(t) - d(x_p) + d(x_{t+1})$ and $C_2(t+1) = C_2(t) - d(x_q) + d(x_{t+1})$. Since $C_1(t) \subseteq C_2(t)$, the only possibility for $C_1(t+1) \not\subseteq C_2(t+1)$ is that $C_2$ evicts $d(x_q)$, and $C_1$ has $d(x_q)$ but does not evict it, so $d(x_q) \in C_1(t+1)$ but $d(x_q) \notin C_2(t+1)$.

First, we assume $d(x_p)$ exists (a cache miss does not mean a cache eviction. see the next case). The eviction in $C_1$ happens at the LRU position regardless of whether $x_p$ is a LRU or MRU access. So, $d(x_p)$ is at the bottom in $C_1$ before access $x_{t+1}$. At the same time, $d(x_q)$ is at the bottom in $C_2$. To violate the inclusion property, we must have $d(x_q) \in C_1(t)$ in a position over $d(x_p)$. From the inductive assumption, $d(x_p) \in C_2(t)$, and it is in a position over $d(x_q)$. Therefore, both $C_1$ and $C_2$ contain $d(x_p)$ and $d(x_q)$ but in an opposite order.

The two accesses, $x_p$ and $x_q$, may be LRU or MRU accesses. There are four cases:

I. $x_p$ and $x_q$ are both LRU accesses. Because $d(x_q)$ is at a higher position than $d(x_p)$ in $C_1$, we have $p < q$. Similar reasoning from $C_2$ requires $q < p$, which makes this case impossible.
II. $x_p$ is an LRU access but $x_q$ is an MRU access. Using Lemma 3.2 on $C_1$, we see that this case is impossible—$x_q$ has to be an LRU access because $d(x_q)$ resides over $d(x_p)$ that is most recently accessed by an LRU access in $C_1$.

III. $x_p$ is an MRU access but $x_q$ is an LRU access. Using Lemma 3.2 on $C_2$, we see that this case is impossible—$x_p$ has to be an LRU access because $d(x_p)$ resides over $d(x_q)$ that is most recently accessed by an LRU access in $C_2$.

IV. $x_p$ and $x_q$ are both MRU accesses. Because $d(x_q)$ is at a higher position than $d(x_p)$ in $C_1$, we have $p > q$. Similar reasoning from $C_2$ requires $q > p$, which makes the last case impossible.

There is no eviction in $C_1$ if the bottom cache line is unoccupied when $x_{t+1}$ is accessed, and $d(x_q)$ is at the bottom of $C_2$. Regardless of whether $x_q$ is LRU or MRU, $C_2$ is filled. Because $|C_2| > |C_1|$, there must have been enough data access to fill $C_1$, making it impossible for its bottom spot to remain unoccupied. Hence, by induction, the inclusion property holds for every access in the trace. 

The inclusion property holds for any access trace with mixed LRU and MRU accesses, regardless of how these two types of accesses are interleaved.

### 3.3.7 LRU-MRU Stack Distance

The inclusion property implies the existence of the LRU-MRU stack distance. An access has a distance $k$ if it is a cache hit in caches of sizes $k$ and up and a miss in caches of size $k - 1$ and down. Given a program trace with mixed LRU-MRU accesses, Algorithm 3.1 computes the stack distance for each access. Effectively, the algorithm simulates LRU-MRU caches of all sizes—top $C$ elements in the priority list are always the content of a cache with size $C$. We call the algorithm $bi-sim$ in short for LRU-MRU cache simulation.
Algorithm 3.1: Bi-sim: computing the stack distance of LRU-MRU cache

**Input:** $x$ is accessed at time $t$ with flag $f = \{LRU, MRU\}$. The cache is organized as a priority list, with data $d_i$ and priority $p_i$, $i = 1, \ldots, m$. No two priorities are the same, that is, $\forall i$ and $j$, $p_i \neq p_j$ if $i \neq j$. The list may not have been sorted.

**Output:** It returns the LRU-MRU stack distance and updates the priority $p_x$ of $x$ (first adding it to the priority list if it was not included). The priority $p_x$ is unique.

```plaintext
1 bi_sim(x,t,f)
2 begin
3     if $f == LRU$ then
4         $p_x = t$
5     else
6         $p_x = -t$
7     end
8     /* adjust the priority list */
9     if $x \not\in \{d_i : i = 1, \ldots, m\}$ then
10        /* $x$ is a miss */
11           /* $d_m$ is at the bottom of the cache */
12               if $p_m < 0$ then
13                  Remove $d_m$ from the list
14               end
15               Insert $x$ at the front of the list
16               Return $\infty$
17     else
18        /* $x$ is a hit */
19           Find out $d_k = x$
20           for $i = 1; i < k; i++$ do
21              if $p_i < p_{i+1}$ then
22                 Swap $d_i$ and $d_{i+1}$
23              end
24           end
25           Move $x$ to the front of the list
26           Return $k$
27     end
28 end
```
For access $x$ at time $t$, Algorithm 3.1 computes the stack distance and updates the priority list. The algorithm has three parts:

- The first part, lines 3 to 7, sets the priority for $x$ to be $t$ or $-t$ depending on whether $x$ is LRU or MRU. The purpose is to handle mixed priority. By negating $t$, the priority of MRU data is reversed to the access order. The MRU in the access order becomes LRU in the priority order. In addition, the negative priority means that all MRU data has a lower priority than all LRU data. Finally, all priority numbers remain distinct. As a result, all data in the cache are prioritized with no ties.

- The second part, lines 11 to 21, handles cache replacement at a miss when $x$ is not in the priority list. The element with the lowest priority is shifted down to the bottom. It is removed if its priority is negative (an MRU data element). Element $x$ is inserted to become the new head of the list.

- The third part, lines 24 to 31, handles a hit at location $k$, that is, $d_k = x$. The element of the lowest priority in $d_1, \ldots, d_k$ is shifted down to replace $d_k$. Element $x$ is moved to the front of the list as in the second part.

The update process, swapping and then inserting, is similar to Mattson et al. [41] but with two notable qualities. First, the priority list of bi-sim is not completely sorted. In comparison, the priority list in LRU simulation is always totally sorted. Second, bi-sim may remove an element from the priority list (line 16), even if it is simulating a cache of an infinite size. The stack simulation of previous caching methods such as LRU and OPT never removes elements when simulating for all cache sizes.

An example An example depicting bi-sim in action is given in Table 3.6. The access trace and the access types are listed in the second and third columns. The
priority list (after each access) is shown in the next column. The last column
is the stack distance returned by Algorithm 3.1: \( \infty \) always means a miss, and \( k \)
means a cache hit if cache size \( C \geq k \) and a miss otherwise. The priority lists in
the table show only the priority numbers \( p_x \). A reader can find the data element
from the \( p_x \)th row of the table (the \( p_x \)th access in the trace).

The example shows two notable characteristics of the bi-sim algorithm. The
priority list is not completely sorted because of the negative priority numbers of
MRU accesses. An MRU element may be removed from cache even when there is
space, as happens at access 3. These are necessary to measure the miss ratios of
all cache sizes in a single pass.

The cost and its reduction The asymptotic cost of Algorithm 3.1 is \( O(M) \) in
time and space for each access, where \( M \) is the number of distinct data elements
in the input trace. The main time overhead comes from the two swap loops at
lines 11-15 and 25-29. To improve performance, we divide the priority list into
partially sorted groups. For example, there are 4 groups at the 25th access in the
example in Table 3.6: [26], [22, -23], [20, -24], and [17, 14, 11, 1]\(^3\). The swap loops
are changed to iterate over the groups. The minimal element of a group is simply
the last element. Grouping in priority lists was first invented by Sugumar and
Abraham for simulating OPT [54]. A difference between OPT and bi-sim is that
the accessed data element can be in the middle of a group in bi-sim. For OPT,
the accessed data element always stays at the front of a group.

The Equivalence Proof

So far we have presented the LRU-MRU cache and its simulation. We now show
that the simulation algorithm is correct; that is, the elements of the priority list
\( d_1, d_2, \ldots, d_C \) in the algorithm are indeed the content of an LRU-MRU cache of

\(^3\)For convenience, the top element is always put into a separate window.
<table>
<thead>
<tr>
<th>access no.</th>
<th>access trace</th>
<th>LRU or MRU</th>
<th>the priority list (top (\rightarrow) bottom)</th>
<th>stack distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>h</td>
<td>L</td>
<td>1</td>
<td>(\infty)</td>
</tr>
<tr>
<td>2</td>
<td>f</td>
<td>M</td>
<td>-2 1</td>
<td>(\infty)</td>
</tr>
<tr>
<td>3</td>
<td>i</td>
<td>L</td>
<td>3 1</td>
<td>(\infty)</td>
</tr>
<tr>
<td>4</td>
<td>i</td>
<td>M</td>
<td>-4 1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>c</td>
<td>L</td>
<td>5 1</td>
<td>(\infty)</td>
</tr>
<tr>
<td>6</td>
<td>b</td>
<td>L</td>
<td>6 5 1</td>
<td>(\infty)</td>
</tr>
<tr>
<td>7</td>
<td>b</td>
<td>M</td>
<td>-7 5 1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>e</td>
<td>M</td>
<td>-8 5 1</td>
<td>(\infty)</td>
</tr>
<tr>
<td>9</td>
<td>d</td>
<td>M</td>
<td>-9 5 1</td>
<td>(\infty)</td>
</tr>
<tr>
<td>10</td>
<td>b</td>
<td>L</td>
<td>10 5 1</td>
<td>(\infty)</td>
</tr>
<tr>
<td>11</td>
<td>g</td>
<td>L</td>
<td>11 10 5 1</td>
<td>(\infty)</td>
</tr>
<tr>
<td>12</td>
<td>b</td>
<td>L</td>
<td>12 11 5 1</td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td>e</td>
<td>L</td>
<td>13 12 11 5 1</td>
<td>(\infty)</td>
</tr>
<tr>
<td>14</td>
<td>d</td>
<td>L</td>
<td>14 13 12 11 5 1</td>
<td>(\infty)</td>
</tr>
<tr>
<td>15</td>
<td>a</td>
<td>L</td>
<td>15 14 13 12 11 5 1</td>
<td>(\infty)</td>
</tr>
<tr>
<td>16</td>
<td>c</td>
<td>L</td>
<td>16 15 14 13 12 11 5 1</td>
<td>6</td>
</tr>
<tr>
<td>17</td>
<td>e</td>
<td>L</td>
<td>17 16 15 14 12 11 5 1</td>
<td>4</td>
</tr>
<tr>
<td>18</td>
<td>a</td>
<td>L</td>
<td>18 17 16 14 12 11 5 1</td>
<td>3</td>
</tr>
<tr>
<td>19</td>
<td>c</td>
<td>L</td>
<td>19 18 17 14 12 11 5 1</td>
<td>3</td>
</tr>
<tr>
<td>20</td>
<td>i</td>
<td>L</td>
<td>20 19 18 17 14 12 11 5 1</td>
<td>(\infty)</td>
</tr>
<tr>
<td>21</td>
<td>f</td>
<td>L</td>
<td>21 20 19 18 17 14 12 11 5 1</td>
<td>(\infty)</td>
</tr>
<tr>
<td>22</td>
<td>b</td>
<td>L</td>
<td>22 21 20 19 18 17 14 12 11 5 1</td>
<td>7</td>
</tr>
<tr>
<td>23</td>
<td>a</td>
<td>M</td>
<td>-23 22 21 20 19 17 14 11 1 5</td>
<td>5</td>
</tr>
<tr>
<td>24</td>
<td>f</td>
<td>M</td>
<td>-24 22 -23 20 19 17 14 11 1 3</td>
<td>3</td>
</tr>
<tr>
<td>25</td>
<td>c</td>
<td>M</td>
<td>-25 22 -23 20 -24 17 14 11 1 5</td>
<td>5</td>
</tr>
<tr>
<td>26</td>
<td>c</td>
<td>L</td>
<td>26 22 -23 20 -24 17 14 11 1 1</td>
<td>1</td>
</tr>
<tr>
<td>27</td>
<td>e</td>
<td>M</td>
<td>-27 26 22 20 -23 -24 14 11 1 6</td>
<td>6</td>
</tr>
<tr>
<td>28</td>
<td>i</td>
<td>M</td>
<td>-28 26 22 -27 -23 -24 14 11 1 4</td>
<td>4</td>
</tr>
<tr>
<td>29</td>
<td>c</td>
<td>L</td>
<td>29 -28 22 -27 -23 -24 14 11 1 2</td>
<td>2</td>
</tr>
<tr>
<td>30</td>
<td>f</td>
<td>L</td>
<td>30 29 -27 -28 14 11 1 6</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 3.6: Example one-pass simulation of LRU-MRU cache
size $C$. We show the equivalence in two steps. First, we show that the algorithm observes the inclusion property. Then we show that the two are equivalent at each cache size.

Proving the inclusion property is easier for the algorithm than for LRU-MRU cache because we can use its algorithmic design directly. We first define a property in cache replacement. Let two caches of size $s, s+1$ be $C_s, C_{s+1}$, which are also the data sets in cache. Assume that $C_s, C_{s+1}$ are filled with data, and $z$ is the element in $C_{s+1}$ but not in $C_s$. At a cache miss, $C_s$ evicts element $y_s$, and $C_{s+1}$ evicts $y_{s+1}$. The eviction invariance is a property that requires

$$y_{s+1} = y_s \lor y_{s+1} = z$$

Mattson et al. [41] showed the following result:

**Lemma 3.3.** Eviction invariance is a necessary and sufficient condition for maintaining the inclusion property.

**Proof.** First, we show the necessity. If $y_{s+1} \neq y_s \land y_{s+1} \neq z$, $y_{s+1}$ must be in $C_s$. Its eviction would mean that $C_s \not\subseteq C_{s+1}$ and would break the inclusion property. The property is also sufficient. At each eviction, if $y_{s+1} = y_s$, we have $C_{s+1} = C_s + z$; otherwise, we have $y_{s+1} = z$ and $C_{s+1} = C_s + y_s$. In both cases, $C_s \subseteq C_{s+1}$.

The simulation algorithm observes the eviction invariance. The “stack” is embodied in a priority list. Each element has a numerical priority distinct from others. Therefore, the caches it simulates have the inclusion property.

**Lemma 3.4.** Algorithm 3.1 observes the eviction invariance and is therefore a stack algorithm.

**Proof.** Algorithm 3.1 identifies a victim for replacement using one of the two swap loops at lines 11-15 and 25-29. Consider two caches $C_s, C_{s+1}$ of sizes $s, s+1$. Let
z be the element in $C_{s+1}$ but not in $C_s$. Let $y$ be the element in $C_s$ that has the lowest priority. When a cache replacement is needed in $C_{s+1}$, the swap loops would choose as the victim $y$ if $p_y < p_z$ and $z$ otherwise. The eviction invariance is therefore observed.

Intuitively, the simulation is a stack algorithm because the simulated caches of all sizes share a single priority list. It is obvious that sharing a priority list implies eviction invariance. Next we show that Algorithm 3.1 computes the right stack distance. First we have the following lemma. We omit the proof, which is straightforward based on the handling of LRU and MRU accesses.

**Lemma 3.5.** At a miss in LRU-MRU cache, the victim is always the data element with the lowest priority.

**Theorem 3.3.** Given an execution on LRU-MRU cache of size $C$, an access is a cache hit if and only if the stack distance returned by Algorithm 3.1 is no greater than $C$.

*Proof.* The case for infinite distances is easy to verify; we only prove the case when the distance is of a finite value. Specifically, Algorithm 3.1 always stores the data in the priority list such that a cache of size $C$ would contain and only contain the first $C$ elements in the list, $d_1, d_2, \ldots, d_C$. This is equivalent to showing that for each data $d_i$, we have $d_i \in C_i$ and $d_i \notin C_{i-1}$, where $i > 0$ and $C_i, C_{i-1}$ are the sets of data in caches of sizes $i, i - 1$, respectively.

Let the memory trace be $(x_1, x_2, \ldots, x_n)$. We prove by induction on $x_j$.

I. After accessing $x_1$, $x_1$ becomes $d_1$ in the priority list. The base case holds since $d_1 \in C_1$ and $d_1 \notin C_0$.

II. Assume the theorem holds after accessing $x_j$ ($1 \leq j \leq n - 1$). Let the data element at position $i$ be $d_i(j)$ and the data sets of caches of size $i - 1$ and $i$
be $C_{i-1}(j)$ and $C_i(j)$. From the inductive hypothesis, we have $d_i(j) \in C_i(j)$ and $d_i(j) \notin C_{i-1}(j)$. There are two cases after accessing $x_{j+1}$:

(a) $x_{j+1}$ is a (compulsory) miss. Each data element of the priority list is updated from $d_i(j)$ to $d_i(j + 1)$ ($1 \leq i \leq m$ or $1 \leq i \leq m + 1$).

i. $d_1(j + 1) = x_{j+1}$ and satisfies $d_1(j + 1) \in C_1(j + 1)$ and $d_1(j + 1) \notin C_{0}(j + 1)$.

ii. For $d_i(j + 1)$ ($2 \leq i \leq m$), the swap loop (lines 11-15) moves the data element $d_h(j)$ ($1 \leq h \leq i$) with the lowest priority in $C_i(j)$ out of the priority list. According to Lemma 3.5, after evicting $d_h(j)$ from $C_i(j)$, the top $i$ elements in the priority list are still in $C_i(j + 1)$, so $d_i(j + 1) \in C_i(j + 1)$. In the same way, we can show that $d_i(j + 1)$ is either $d_i(j)$ or the victim of $C_{i-1}(j)$, so $d_i(j + 1) \notin C_{i-1}(j + 1)$.

iii. If $d_m(j)$ has a positive priority, $d_{m+1}(j + 1)$ is at the new bottom and must be the victim of $C_i(j)$, so $d_{m+1}(j + 1) \notin C_m(j + 1)$. $d_{m+1}(j + 1) \in C_{m+1}(j + 1)$ follows from Lemma 3.2.

iv. If $d_m(j)$ has a negative priority, the stack distance would be infinite. It is a miss in all finite-size LRU-MRU cache.

(b) $x_{j+1}$ is a hit. Let the hit location be $k$ ($d_k(j) = x_{j+1}$). Each data element of the priority list is updated from $d_i(j)$ to $d_i(j + 1)$ ($1 \leq i \leq m$).

i. Consider $d_i(j + 1)$ ($1 \leq i \leq k - 1$). The access is a miss in caches $C_1(j), \cdots, C_{k-1}(j)$, so the inference of the previous miss case can be reused here. The swap loop in lines 25-29 is identical to the swap loop in lines 11-15.

ii. Consider $d_k(j + 1)$. Because $C_k(j) = C_k(j + 1)$, we have $d_k(j + 1) \in C_k(j + 1)$. From the inference of the miss case, $d_k(j + 1) \notin C_{k-1}(j + 1)$. 

iii. Finally, consider \( d_i(j + 1) \) \((k + 1 \leq i \leq m)\), \( d_i(j + 1) = d_i(j) \) because there is no change made by the algorithm. From \( x_{j+1} = d_k(j) \in C_k(j) \), we have \( x_{j+1} \) is a cache hit in \( C_i(j) \) \((i \geq k + 1)\) and \( C_i(j) = C_i(j + 1) \) \((k + 1 \leq i \leq m)\). From the induction assumption, we have \( d_i(j + 1) \in C_i(j + 1) \) and \( d_i(j + 1) \notin C_{i-1}(j + 1) \) \((k + 1 \leq i \leq m)\).

For all accesses, the cache of size \( C \) would contain and only contain the first \( C \) elements in the priority list, \( d_1, d_2, \cdots, d_C \). Hence the relationship is established between the stack distance and the cache hit/miss as stated in the theorem.

\[\square\]

### 3.4 Trespass LRU Cache

Trespass LRU is very similar to optimal LRU-MRU. In Trespass LRU, an access can be normal LRU or trespass MRU, which are illustrated in Figure 3.1, Figure 3.2, Figure 3.8, and Figure 3.9.

- **Trespass MRU access** uses the most recently used position for placement and the same position for replacement. It differs from all cache replacement policies that we are aware of in that an eviction may happen even for a cache hit.

  - Miss: Evict the data element \( S_1 \) at the MRU position if the MRU is taken and insert \( w \) in the MRU position. See Figure 3.8.

  - Hit: If \( w \) is in the MRU position, then do nothing. Otherwise, evict the data element \( S_1 \) at the MRU position, insert \( w \) there, and shift the elements under the old \( w \) spot up by one position. See Figure 3.9.

Trespass LRU is proved optimal in Section 3.4.1. Section 3.4.2 shows that multi-size optimal Trespass LRU holds the inclusion property.
Figure 3.8: Trespass MRU at a miss: the trespass posits \( w \) at the top of the stack, evicting \( S_1 \)

Figure 3.9: Trespass MRU at a hit: the trespass raises \( S_3(w) \) to the top of the stack, evicting \( S_1 \)

3.4.1 Optimality

The type of each access is determined in the OPT training. The details to obtaining the optimal cache hints are shown in Figure 3.10. 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 \( a_{i+1} \), the next one after the most recent access to \( X \), is selected to use trespass. After training, the unselected accesses still use normal LRU. The training result is specific to the cache size being used.

Two example executions of Trespass LRU execution are shown in Table 3.7 for the same trace used to demonstrate optimal LRU-MRU in Table 3.4 and Table 3.5.
The two examples demonstrate that Trespass LRU can be optimal as the same as LRU-MRU but holds inclusion property.

<table>
<thead>
<tr>
<th>accessed time</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>accessed data</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>e</td>
<td>e</td>
<td>d</td>
</tr>
<tr>
<td>OPT cache</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>d</td>
<td>e</td>
<td>b</td>
<td>b</td>
<td>d</td>
</tr>
<tr>
<td>misses</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>trespasses</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Trespass LRU</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>d</td>
<td>e</td>
<td>b</td>
<td>e</td>
<td>c</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>cache</td>
<td>a</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>b</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>misses</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Table 3.7: Two examples showing Trespass LRU can be optimal and multi-size optimal Trespass LRU holds inclusion property (unlike multi-size optimal LRU-MRU)

We will follow a similar approach to show the optimality of Trespass LRU. The main proof is actually simpler.

The effect of a trespass access is less direct than that of a bypass access. We need four additional lemmata. First, from the way trespass accesses are identified, we have the following:

**Lemma 3.6.** If a data element $w$ is evicted by a trespass access $x$, then $x$ happens immediately after the most recent access of $w$. 
Lemma 3.7. If a data element is in Trespass LRU cache at point \( p \) in the trace, then the element is also in OPT cache at \( p \).

Proof. Assume that a data element \( w \) is in the Trespass LRU cache but is evicted from the OPT cache. Let \( x \) be the most recent access to \( w \). Consider the time of the eviction in both cases. The eviction in Trespass LRU happens right after \( x \). Because the eviction in OPT cannot be earlier, there must be no period of time when \( w \) is in the Trespass LRU cache but not in the OPT cache. \( \square \)

Lemma 3.8. If a data element is evicted by a normal access in Trespass LRU, then the cache is full before the access.

This is obviously true because the normal access cannot evict any element unless the cache is full. Not as obvious, we have the following:

Lemma 3.9. A normal access cannot evict a data element from cache in Trespass LRU-MRU.

Proof. Assume \( y \) is a normal access that evicts data \( w \). Let \( T \) and \( T^* \) be the set of data elements in the Trespass LRU cache and the OPT cache before access \( y \). By Lemma 3.7, \( T \subseteq T^* \). By Lemma 3.8, the Trespass LRU cache is full before \( y \). Then we have \( T = T^* \). In OPT, \( y \) has to evict some element \( d \in T^* \). Let \( x \) be the most recent access to \( d \) before \( y \). Because Trespass LRU evicts \( d \) right after \( x \), the content of the cache, \( T \) and \( T^* \), cannot be the same unless \( y \) is the next access after \( x \), in which case \( d \) is \( w \), and \( y \) must be a trespass access. \( \square \)

Theorem 3.4. Trespass LRU generates no more misses than OPT. In particular, Trespass LRU has a miss only if OPT has a miss.

Proof. We show that there is no access that is a cache hit in OPT but a miss in Trespass LRU. Suppose the contrary were true. Let \( z \) be the first access in the trace that hits in OPT but misses in Trespass LRU. Let \( d \) be the element
accessed by \( z t \), \( z \) be the most recent access to \( d \) before \( z t \), and the reference trace between them be \((z, ..., y, ..., z t)\), where \( y \) is the access that causes the eviction of \( d \) in Trespass LRU-MRU.

By Lemma 3.9, \( y \) is a trespass access. By Lemma 3.6, \( y \) happens immediately after \( z \). Because \( y \) is a trespass after \( z \), then the next access of \( d \), \( z t \) must be a miss in OPT. This contradicts the assumption that \( z t \) is a hit in OPT. Therefore, any access that is a miss in Trespass LRU must also be a miss in OPT. \( \square \)

**Corollary 3.2.** Trespass LRU has the same number of misses as OPT and is therefore optimal.

### 3.4.2 Multi-size Optimality

Given that multi-size optimal LRU-MRU is not a stack algorithm, the next result is interesting and shows an important theoretical difference between multi-size optimal Trespass LRU and multi-size optimal LRU-MRU.

**Theorem 3.5.** Multi-size optimal Trespass LRU is inclusive.

**Proof.** Assume there are two caches \( C_1 \) and \( C_2 \). \( C_2 \) is larger than \( C_1 \), and the access sequence is \( Q = (x_1, x_2, ..., x_n) \). Let \( T_1(t) \) be the set of elements in cache \( C_1 \) after access \( x_t \) and \( T_2(t) \) be the set of elements in cache \( C_2 \) after the same access \( x_t \). The initial sets for \( C_1 \) and \( C_2 \) are \( T_1(0) \) and \( T_2(0) \), which are empty and satisfy the inclusion property. We now prove the theorem by induction on \( t \).

Assume \( T_1(t) \subseteq T_2(t) \) \((1 \leq t \leq n - 1)\). There are four possible cases based on the type of the access \( x_{t+1} \) when visiting either of the two caches. We denote the data element accessed at time \( x_t \) as \( D(x_t) \).
• If $x_{t+1}$ is a trespass access both in $C_1$ and $C_2$, we have:

$$T_1(t + 1) = T_1(t) - D(x_t) + D(x_{t+1})$$

$$\subseteq T_2(t) - D(x_t) + D(x_{t+1})$$

$$= T_2(t + 1)$$

• If $x_{t+1}$ is a trespass access in $C_1$ but a normal access in $C_2$, then by Lemma 3.9, $x_{t+1}$ does not cause any eviction in cache $C_2$ and, therefore:

$$T_1(t + 1) = T_1(t) - D(x_t) + D(x_{t+1})$$

$$\subseteq T_2(t) + D(x_{t+1})$$

$$= T_2(t + 1)$$

• The case that $x_{t+1}$ is a normal access in $C_1$ but a trespass access in $C_2$ is impossible. Since $x_{t+1}$ is a trespass in $C_2$, $D(x_t)$ would be evicted by some access $y$ in $C_2$ using OPT. However, $x_{t+1}$ is a normal access in $C_1$, which means that $D(x_t)$ is in $C_1$ after access $y$ when using OPT. This, in turn, means that at the point of $y$, the inclusion property of OPT no longer holds and contradicts the fact that OPT is a stack algorithm.

• If $x_{t+1}$ is a normal access both in $C_1$ and $C_2$, then by Lemma 3.9, $x_{t+1}$ does not cause an eviction either in $C_1$ or $C_2$, and therefore:

$$T_1(t + 1) = T_1(t) + D(x_{t+1})$$

$$\subseteq T_2(t) + D(x_{t+1})$$

$$= T_2(t + 1)$$
From the induction hypothesis, the inclusion property holds for multi-size optimal Trespass LRU for all $t$.

The next corollary follows from the stack property.

**Corollary 3.3.** Multi-size optimal Trespass MRU as a cache management algorithm does not suffer from Belady anomaly [8].

In Table 3.7, we have revisited the same data trace used to show that multi-size optimal LRU-MRU is not a stack algorithm. It shows that the inclusion property holds when multi-size optimal Trespass LRU is used. The example also shows that Trespass LRU cache can become partially empty after it becomes full. Trespass LRU keeps the data element that is currently being visited and the data elements that will be visited. When the amount of data that have a future reuse is less than the cache size, OPT and optimal LRU-MRU may contain extra data elements that have no future reuse. In OPT the extra data do not destroy the inclusion property, but in optimal LRU-MRU they do.

### 3.5 Priority LRU Cache

The cache hint interfaces used in the previous sections are limited, for example, a binary choice between LRU and MRU. In this section, we generalize the hint interface to priority hint, where a hint is a number encoding a priority.

A new cache replacement policy, Priority LRU, is enabled by priority hints. Priority LRU permits the complete range of choices between MRU and LRU. We prove a new type of inclusion property—non-uniform inclusion—and devise a one-pass algorithm to compute stack distances for Priority LRU.

For this study, the cache is fully associative and organized as a stack. The default scheme is LRU, where the data element is placed at the top of the stack
at position 1 and replaced 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 in which 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 in Section 3.3.

We categorize Priority LRU accesses into six classes, illustrated in Figure 3.11, Figure 3.12 and Figure 3.13. Consider an access to \(w\) with the priority \(i\), that is, \((w, i)\), 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 that are 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 3.11(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 3.12(a) shows that all data elements including \(w\) do not change their positions.
iii) A hit down move (1 \leq j < i \leq m)—Figure 3.11(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 3.12(b) 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 3.13(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 \((j = \infty \text{ and } i > m)\) — We assume that the accessed data elements can be accessed without being stored in the cache. Figure 3.13(b) shows that \(w\) bypasses the cache. The data elements in the cache are unaffected.

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.
Figure 3.13: 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.

In Section 3.5.1, Priority LRU is proved to hold inclusion property even when the stack view is broken. We call this non-uniform inclusion, which is different from other stack algorithms and discussed in Section 3.5.2. Due to the non-uniform inclusion property, a brand new algorithm is devised to calculate stack distances for Priority LRU in Section 3.5.3.

As an interface, priority hints may be used in arbitrary ways, sometimes optimal but probably suboptimal most times and even counter productive. The problem of optimal priority hint insertion will be discussed in Section 3.5.4.

### 3.5.1 Inclusion Property

**Theorem 3.6.** 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 \( \text{loc}() \) returns the location of a data element in the cache stack—\( \text{loc}_i^t(d) = p \) (\( 1 \leq p \leq |C_i| \)) means that the data element \( d \) is at the position \( p \) of \( C_i \) after \( x_t \). In other words, \( \text{loc}(d) \) returns the priority of a data element. We let \( \text{loc}_i^t(d) = \infty \) 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 \( d, d \in C_1(t) \rightarrow \text{loc}_1^t(d) \geq \text{loc}_2^t(d) \). Suppose there is a data element \( d \) satisfying \( d \in C_1(t) \) and \( d \notin C_2(t) \). Then we have \( \text{loc}_1^t(d) < \text{loc}_2^t(d) = \infty \)—a contradiction of the assumption. So we have nine possible cases for the next access \( x_{t+1}(d', p') \) shown in Table 3.8. We prove for any \( d, d \in C_1(t + 1) \rightarrow \text{loc}_1^{t+1}(d) \geq \text{loc}_2^{t+1}(d) \) 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 \( C_2 \).

| \( 1 \leq p' \leq |C_1| \) | \( |C_1| < p' \leq |C_2| \) | \( p' > |C_2| \) | \( x_{t+1} \) 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 \( C_2 \) |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| I               | II              | III             | \( x_{t+1} \) 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 \( C_2 \) |
| IV              | V               | VI              | \( x_{t+1} \) 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 \( C_2 \) |
| VII             | VIII            | IX              | \( x_{t+1} \) 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 \( C_2 \) |

Table 3.8: The nine cases for the next access \( x_{t+1} \) to \( d' \) with a priority \( p' \).

I. From the assumption, we know that \( \text{loc}_1^t(d') \geq \text{loc}_2^t(d') \). There are six subcases of \( x_{t+1} \) as shown in Table 3.9.

<table>
<thead>
<tr>
<th>( x_{t+1} ) hits in ( C_1 ) and ( C_2 )</th>
<th>( x_{t+1} ) misses in ( C_1 ) but hits in ( C_2 )</th>
<th>( x_{t+1} ) misses in ( C_1 ) and ( C_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>up move in ( C_1 )</td>
<td>no move in ( C_2 )</td>
<td>down move in ( C_2 )</td>
</tr>
<tr>
<td>i</td>
<td>ii</td>
<td>iii</td>
</tr>
<tr>
<td>IMPOSSIBLE</td>
<td>iv</td>
<td>v</td>
</tr>
<tr>
<td>IMPOSSIBLE</td>
<td>IMPOSSIBLE</td>
<td>vi</td>
</tr>
</tbody>
</table>

Table 3.9: In the six subcases of Case I in Table 3.8, 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 3.11 and 3.12 except the bypass case.
i) $x_{t+1}$ is a hit up move in both $C_1$ and $C_2$, which means that $p' < \text{loc}_2^t(d') \leq \text{loc}_1^t(d')$. 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 $p' \leq \text{loc}_2^t(d) < \text{loc}_2^t(d')$: ① if $p' \leq \text{loc}_1^t(d) < \text{loc}_1^t(d')$, given $\text{loc}_1^t(d) \geq \text{loc}_2^t(d)$, we have $\text{loc}_1^{t+1}(d) \geq \text{loc}_2^{t+1}(d)$ because $\text{loc}_1^{t+1}(d) = \text{loc}_1^t(d) + 1$ and $\text{loc}_2^{t+1}(d) = \text{loc}_2^t(d) + 1$; ② if $\text{loc}_1^t(d) > \text{loc}_1^t(d')$, given $\text{loc}_1^t(d) \geq \text{loc}_2^t(d)$, we have $\text{loc}_1^{t+1}(d) \geq \text{loc}_2^{t+1}(d)$ because $\text{loc}_2^{t+1}(d) \leq \text{loc}_2^t(d') \leq \text{loc}_1^t(d') < \text{loc}_1^t(d) = \text{loc}_1^{t+1}(d)$. 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 $\text{loc}_2^t(d') \leq \text{loc}_1^t(d') < p'$. A data element $d$ moved up in $C_1$ satisfies $\text{loc}_1^t(d') < \text{loc}_1^t(d) \leq p'$: ① if $\text{loc}_2^t(d') < \text{loc}_2^t(d) \leq p'$, given $\text{loc}_1^t(d) \geq \text{loc}_2^t(d)$, we have $\text{loc}_1^{t+1}(d) \geq \text{loc}_2^{t+1}(d)$ because $\text{loc}_1^{t+1}(d) = \text{loc}_1^t(d) - 1$ and $\text{loc}_2^{t+1}(d) = \text{loc}_2^t(d) - 1$; ② if $\text{loc}_2^t(d) < \text{loc}_2^t(d')$, given $\text{loc}_1^t(d) \geq \text{loc}_2^t(d)$, we have $\text{loc}_2^{t+1}(d) \geq \text{loc}_2^{t+1}(d)$ because $\text{loc}_2^{t+1}(d) = \text{loc}_2^t(d) < \text{loc}_2^t(d') \leq \text{loc}_1^t(d') \leq \text{loc}_1^{t+1}(d)$. 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 subcases about $x_{t+1}$ as shown in Table 3.10.

<table>
<thead>
<tr>
<th></th>
<th>up move in $C_1$</th>
<th>no move in $C_2$</th>
<th>down move in $C_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>a miss insertion in $C_1$</td>
<td>i</td>
<td>ii</td>
<td>iii</td>
</tr>
</tbody>
</table>

Table 3.10: In the three subcases of case II in Table 3.8, the access $x_{t+1}$ misses in $C_1$ but hits in $C_2$. The hit and miss cases are shown in Figures 3.11, 3.12 and 3.13.

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 $C_2$. The data element $d$ that is 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 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 $C_2$. The data element $d$ that is 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 $loc_2^t(d') \leq loc_1^t(d') < p'$. The data element $d$ that is 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 subcases about $x_{t+1}$ as shown in Table 3.11.

<table>
<thead>
<tr>
<th>a hit up move in $C_2$</th>
<th>a hit no-move in $C_2$</th>
<th>a hit down move in $C_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>a miss bypass in $C_1$</td>
<td>i</td>
<td>ii</td>
</tr>
</tbody>
</table>

Table 3.11: The three subcases 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 $C_1$. The data element $d$ that is 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 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.

VII. \( x_{t+1} \) is a hit bypass in both \( C_1 \) and \( C_2 \). The data element \( d \) that is moved up in \( C_1 \) satisfies \( \text{loc}^t_2(d) \geq p' \): 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 \( \text{loc}^t_2(d) \leq \text{loc}^t_1(d) \leq |C_1| < p' \)—a contradiction of the assumption \( \text{loc}^t_2(d) \geq p' \). The induction holds.

VIII. \( x_{t+1} \) is a hit bypass in both \( C_1 \) and \( C_2 \). The data element \( d \) that is moved up in \( C_1 \) satisfies \( \text{loc}^t_1(d) \geq \text{loc}^t_1(d') \): ① if \( \text{loc}^t_2(d) > \text{loc}^t_2(d') \), given \( \text{loc}^t_2(d) \geq \text{loc}^t_2(d) \), we have \( \text{loc}^{t+1}_1(d) \geq \text{loc}^{t+1}_2(d) \) because \( \text{loc}^{t+1}_1(d) = \text{loc}^{t+1}_1(d) - 1 \) and \( \text{loc}^{t+1}_2(d) = \text{loc}^t_2(d) - 1 \); ② if \( \text{loc}^t_2(d) < \text{loc}^t_2(d') \), given \( \text{loc}^t_1(d) \geq \text{loc}^t_2(d) \), we have \( \text{loc}^{t+1}_1(d) \geq \text{loc}^{t+1}_2(d) \) because \( \text{loc}^{t+1}_2(d) = \text{loc}^t_2(d') \leq \text{loc}^t_1(d') \leq \text{loc}^{t+1}_1(d) \). 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.

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. \( \square \)

The inclusion property is shown in the following corollary.
**Corollary 3.4.** 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 3.6. The supposed situation is impossible. Priority LRU preserves the inclusion property.  

3.5.2 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 locate 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, for example, LRU and LRU-MRU, have *uniform inclusion*, in which the same data element has the same position regardless of the cache size.

Figure 3.14 shows an example of non-uniform inclusion. The stack layout at each access is shown in Figure 3.14(a) for cache size 5 and Figure 3.14(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 not in 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 3.6, which allows for the data to locate in a lower position in a smaller cache. Previous inclusive
<table>
<thead>
<tr>
<th>access time</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>data &amp; hint</td>
<td>A:2</td>
<td>B:2</td>
<td>C:5</td>
<td>D:1</td>
<td>B:6</td>
<td>D:6</td>
<td>A:4</td>
<td>C:1</td>
<td>A:4</td>
</tr>
<tr>
<td>cache: 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>B</td>
<td>B</td>
<td></td>
<td></td>
<td>D</td>
<td>D</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>3</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>access time</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>data &amp; hint</td>
<td>A:2</td>
<td>B:2</td>
<td>C:5</td>
<td>D:1</td>
<td>B:6</td>
<td>D:6</td>
<td>A:4</td>
<td>C:1</td>
<td>A:4</td>
</tr>
<tr>
<td>cache: 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>B</td>
<td>B</td>
<td></td>
<td></td>
<td>D</td>
<td>D</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>3</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>A</td>
<td>C</td>
<td>A</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>C</td>
<td>C</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>C</td>
<td>B</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Figure 3.14: An example of non-uniform inclusion. The Priority LRU observes the inclusion principle but permits data to locate 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.

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.
3.5.3 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 [41]. 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.

Priority List ... No Longer Works A priority list is the core data structure in the original stack algorithms [41]. 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 in Section 3.3.

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 3.14(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 3.15. The lowest stack position, 4, is not the right stack distance.
Figure 3.15: For the same trace in Figure 3.14, the access at time 9 is a miss in the size-4 cache.

**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 detail with an example. Figure 3.16 and Figure 3.17 show how spans work on an example trace with nine 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 show 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 3.16(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 are located in the cache with this specific cache size.
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.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>....</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>∞</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(a) After access A-2 at time 1

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>....</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>∞</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>∞</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(b) After access B-2 at time 2

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>....</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>∞</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>∞</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>∞</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(c) After access C-5 at time 3

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>....</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>∞</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>∞</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(d) After access D-1 at time 4

Figure 3.16: An example of Priority LRU stack simulation. The trace has nine
accesses to four 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.

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 3.16(a). For the first
access to a data element, the stack distance is infinity since it is a compulsory
miss [29]. 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
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 \(\text{loc}\), \((A, 1, 1, \infty)\)
and \((A, 2, \infty, \infty)\), 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 3.16(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 3.16(d). First we update the spans for other data elements except for B. There are two spans for A—(A, 1, 3, ∞) and (A, 4, ∞, 4). The former stays the same, and the latter is updated to (A, 4, ∞, 3) because moving B to 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 3.17(a). 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, ∞, 3). So the stack distance is 3. At last, we update the spans of B to (B, 1, 5, ∞) and (B, 6, ∞, 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 3.17(c), A has two spans \((A, 1, 3, \infty)\) and \((A, 4, \infty, 4)\). Moving \(C\) to 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.

**The One-pass 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 \(c_2\) is equals to the \(c_1\) of the next span minus one. For the last span, the \(c_2\) equals to \(\infty\). 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 3.2 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 3.3.

Function `update_one_list()` has three arguments providing sufficient information for span updates. The `while` loop traverses and updates the spans of a
Algorithm 3.2: 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. 
4. if There is no span list for $d$ then
5.   for Each current span list (list_iter) do
6.     Update the span list by calling
7.     update_one_list(list_iter, NULL, $p$)
8.   end
9. Create a new list for $d$
10. Create a new span for the current access with $c_1 = loc = p$ and
11. insert it into the new list
12. Return an infinite stack distance
13. else
14.   Set the_list to the span list for $d$
15.   for Each current span list (list_iter) do
16.     if list_iter $\neq$ the_list then
17.       Update the span list by calling
18.       update_one_list(list_iter, the_list, $p$)
19.     end
20.   end
21. Save the $c_1$ value of the first span in the_list to a temporary
22. Delete all the current spans in the_list
23. Create a new span for the current access with $c_1 = loc = p$ and
24. insert it into the_list
25. Return the saved temporary as the stack distance
26. end
27. end

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 are both 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 3.16 and Figure 3.17.
Algorithm 3.3: 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 $c_1$ and loc values and merging unnecessary adjacent spans
12. | Set updated_span to its predecessor
13. else
14. | if The $c_1$ of updated_span is no less than the $c_1$ of accessed_span then
15. | | Do updates for updated_span including changing $c_1$ and loc values and merging unnecessary adjacent spans
16. | | if The $c_1$ of accessed_span equals to the $c_1$ 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 $c_1$ to $c_1$ 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 $c_1$ 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
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 3.17 when the span \((A, 4, \infty, 4)\) is first split into \((A, 4, 5, 4)\) and \((A, 6, \infty, 4)\). The span splitting aligns the spans to be updated with the spans of the accessed data element. The updating operation becomes simpler because the current access has the same impact for all cache sizes within the span being updated. In this example, \((A, 4, 5, 4)\) is updated to \((A, 4, 5, 5)\) and \((A, 6, \infty, 4)\) 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 \((A, 5, \infty, 5)\) and \((B, 8, \infty, 8)\), and the next access is \(B-1\). The span \((A, 5, \infty, 5)\) is first split into \((A, 5, 7, 5)\) and \((A, 8, \infty, 5)\). Then the new spans are updated to \((A, 5, 7, 6)\) and \((A, 8, \infty, 6)\). The span \((A, 5, 7, 6)\) is split into \((A, 5, 5, \infty)\) and \((A, 6, 7, 6)\). The former span \((A, 5, 5, \infty)\) 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 \((A, 8, \infty, 6)\) into \((A, 6, \infty, 6)\). It is possible to remove unnecessary span splits with a more complex algorithm.

**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 3.3. The number of operations is proportional
to the total number of spans of the data element being updated. If the number of spans is bounded by \( M + D \), the time bound for each access is \( O(D \cdot (M + D)) \).

For LRU cache, there is only one span for each data element, so the time cost is \( O(D) \) per access and matches the cost of the original stack algorithm [41].

**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 3.12 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.\(^4\) 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.

\(^4\)Two updates are needed for a single-span list because of an unnecessary span splitting.
Table 3.12: The measured overhead of Algorithm 3.2 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 the worst cost is still far smaller than the theoretical upper bound.
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 spans per data item is far smaller than the theoretical upper bound: 37 vs. 2048.

### 3.5.4 Optimality of Priority Hint for All Cache Sizes

In Section 3.3.1, an OPT cache simulation is used to decide the access type, LRU or MRU, for each access in optimal LRU-MRU. 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.

Compared with OPT, optimal LRU-MRU encodes the future information by passing hints to the collaborative LRU-MRU cache. In optimal LRU-MRU, the access type, that is, cache hint, 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. However, the change of cache hints is single directionally similar to the inclusion property, as stated in the following Lemma 3.10.

**Lemma 3.10.** In optimal LRU-MRU, 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 3.10 indicates that a minimal cache size \( C \) exists for every access, so an access selected to use MRU should have a cache size no greater than \( C \). Theorem 3.7 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 3.7.** In optimal LRU-MRU, 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.

The last accesses to data, which have infinite forward OPT stack distances, form a special case for Theorem 3.7. We use infinity as the critical cache size to select MRU for these last data accesses because none of them brings a cache reuse in any cache size.

### 3.6 Inclusive Cache Hierarchy

We organize the discussed inclusive caching methods into the following three categories. They form a hierarchy based on the “implemented-by” relationship, as explained below and shown pictorially in Figure 3.18.
Figure 3.18: The Inclusive Cache Hierarchy: Inclusive caches are organized in a hierarchy based on the “implemented-by” relationship. Limited collaborative caching of LRU-MRU in Section 3.3 subsumes non-collaborative schemes of LRU, MRU and OPT [41]. Priority LRU subsumes LRU-MRU and other prior collaborative caches.

- **Level 1, non-collaborative caching**, including LRU, MRU and OPT. The priority is inferred entirely 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 in Section 3.3.3.

- **Level 3, generalized collaborative caching**. Priority LRU 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, Priority LRU subsumes the limited collaborative schemes in Level 2.

We show the generality in a hierarchical relation where collaborative caching subsumes non-collaborative caching, and within collaborative caching, Priority LRU subsumes other collaborative caches because the priority hint subsumes the previous binary hint.
3.7 Summary

In this chapter we have presented three new cache management methods, LRU-MRU, Trespass LRU, and Priority LRU, which are programmable by software, require similar hardware support as LRU, and may produce the same result as optimal cache management.

Both optimal LRU-MRU and optimal Trespass LRU require training analysis, for which we have presented OPT*, an asymptotically faster implementation of optimal cache management than OPT. Multi-size optimal LRU-MRU does not hold inclusion property while multi-size optimal Trespass LRU does.

LRU-MRU is proved that it exhibits the inclusion property under all possible cases in which cache hints (LRU or MRU) can be used in arbitrary ways. A hybrid priority list is designed to calculate LRU-MRU stack distances.

Both LRU-MRU and Trespass LRU use 1-bit cache hint. We extend 1-bit hint to priority hint, where a hint is a number encoding a priority. Priority LRU is based on priority hint. We prove the inclusion property of Priority LRU 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. 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 the limitation of cache size dependence in Section 3.3.3 and Section 3.4.1. Next, we address the remaining difficulties that arise in practice.
4 Pacman: Program-assisted Cache Management

This chapter presents two program-level solutions based on optimal collaborative caching. The first is reference based and generally applicable. The second is loop based and can approximate the optimal solution closely. We call both techniques Pacman for program-assisted cache management.

4.1 Introduction

In practice, we would like to construct a program as easily as possible and run it on a wide range of machines as efficiently as possible. For collaborative caching, we need to determine which collaborative cache to use and how a program should be optimized.

We choose the LRU-MRU collaborative cache as the hardware target. The reasons are many. As shown in Section 3.3, the LRU-MRU cache is a simple variation of LRU and adds just a single bit to the memory interface. It can be made optimal. The inclusion property holds. Most importantly, it facilitates program optimization.

A practical solution should optimize for all cache sizes. In Priority LRU, we have developed an all-size solution where the software specifies the “importance”
of data in a scale, and the hardware chooses which to cache based on the cache size. The solution is not likely practical because it has to add many bits to encode the priority in the instruction code.

There is a conflict between the want for an all-size solution and the desire for simple hardware. Pacman solves this conflict in software.

First, we describe the trace-level solution. Although not practical, it provides the basis for the two program-level solutions. In particular, we focus on the forward OPT distance and show how it enables us to determine the right access hint for every cache size.

Next, we describe Pacman hint insertion. In a program, a memory reference may be executed many times. Based on the relative frequency of LRU versus MRU hints used by the trace-level solution for these accesses, Pacman uses a threshold to determine the program-level hint. Once determined, all accesses of the reference use the Pacman hint.

For loop-based code, we describe a refinement through loop splitting. For each reference in a loop, based on the cache size, Pacman chooses a subset of accesses for LRU and the rest for MRU. In this solution, an optimized program takes an extra parameter before its execution, which is the cache size. To distinguish between them, we call the first reference-based Pacman and the refinement loop-based Pacman.

The common concept in Pacman analysis is the forward OPT distance. Whether an access should be LRU or MRU depends on whether its forward OPT distance exceeds the cache size or not. Using the OPT distance, Pacman changes the hints in a program so it may run optimally on machines with different cache sizes.

It is instructive to compare between the LRU cache and Pacman. The LRU management is characterized by reuse distance. Data with the same reuse distance have the same LRU locality. Consider a simple streaming application traversing a 5MB array repeatedly. If a cache block stores a single element, the array accesses
will all have a reuse distance of 5MB and have the same LRU locality. If the cache size is 4MB, all array accesses will be cache misses. The cache performance does not change with the cache size except at the threshold of 4MB.

Pacman uses the forward OPT distance. For the streaming example, OPT would use the available cache space to store and reuse a part of the array as would Pacman. The cache performance changes continuously, because any additional space is used to store and reuse additional data.

In LRU, every data element loaded between a pair of data reuses would stay in the cache if the reused data element does. The intermediate data all contribute to the reuse distance. Under Pacman, the intermediate data may or may not be kept in the cache, so the Pacman locality can improve over the LRU locality.

Pacman as cache management is essentially a solution to choose which data to store in order to maximize the cache utilization. It makes the choice based on the different OPT distances when different data elements are accessed.

In general, a large volume of data items compete for the limited cache space. OPT is fundamentally about resource management through selective caching. The OPT distance is a synthesis, a joint consideration between the locality of all program data and the available cache resource. It avoids the problem of self feedback. For example, when changing reuse distance by reordering, the shortening of some reuse distances may lengthen other reuse distances. The OPT distance is still optimal when considering all these factors.

Pacman is its program-level implementation of OPT. For each working set in a program, Pacman chooses which sub-part to cache. For multiple working sets, Pacman chooses how much cache to give to each working set. (In loop splitting) Pacman makes adjust these choices based on the cache size of the machine and the input size of the program.

As program-level approximations, Pacman may or may not perform well. We will evaluate the reference hint with and without loop splitting and compare the
performance with LRU and the optimal cache management for a collection of benchmark programs and for a range of cache sizes.

4.2 Reference-based Pacman

We present the reference-based Pacman in this section. Pacman uses an off-line training based on OPT cache simulation to select MRU memory references in a program, which has been described in Section 3.3.3. We first present the design and then use 10 benchmark programs to show that optimal caching may reduce the average miss ratio by 24%, and the reference-based Pacman can utilize LRU-MRU cache to realize 50% optimal improvement.

<table>
<thead>
<tr>
<th>trace</th>
<th>xyaxybxyccxydxyexyfxygxyaxybxyccxydxye</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory reference</td>
<td>RSTRSTRSTRSTRSTRSTRSTRSTRSTRSTRSTRSTR</td>
</tr>
<tr>
<td>optimal hint (c=5)</td>
<td>LLLLLLLLLMLMMLLLMLLLLLLMLMLMLMLMLMLML</td>
</tr>
<tr>
<td>pacman hint (c=5)</td>
<td>LLMLMLMLMLMLMLMLMLMLMLMLMLMLMLMLMLML</td>
</tr>
</tbody>
</table>

Figure 4.1: An example of reference-based pacman hint insertion.

Figure 4.1 is an example of reference-based Pacman hint insertion. The trace is the same as the one in Figure 3.5. We suppose the accesses from three memory references: $x$ from $R$, $y$ from $S$, and $a$ to $g$ from $T$. The optimal hints stay the same, also. Because accesses from the same reference share the same hint, the Pacman hint at program/reference level is decided by the majority type from the optimal hint at access level: LRU for $R$ and $S$ but MRU for $T$. Due to this limitation, some accesses cannot use optimal hints. Here, four accesses by $T$ use non-optimal hints.

4.2.1 The Design

Pacman is a feedback-based compiler optimization. As a study of the performance potential rather than a practical solution, we analyze one execution of a target
program on LRU-MRU cache of one size. The first step is an off-line OPT training that uses an efficient OPT implementation to identify MRU accesses at the trace level and the program instructions that make these accesses. When an eviction happens in the OPT simulation, the most recent access to the victim is selected to use MRU as shown in Figure 3.7.

After training, each reference in program code has a unique indicator values: the MRU ratio. An MRU ratio of $y$ means that $y$ fraction of its accesses are selected as MRU in the optimal solution. We use a simple heuristic to select MRU references: a reference is MRU if at least half of its accesses were MRU in the training run.

Once a reference is selected to use MRU, all its accesses in execution will be MRU. This is most likely suboptimal. For example, if the MRU ratio of a reference is 50%, the reference will be selected and half of the accesses will be issued as MRU while they should be normal (LRU) accesses. Other heuristics may be used. Regardless of the selection method, LRU-MRU cache will always observe the inclusion property as we have shown.

4.2.2 Experiment Setup

The Pacman tool is implemented as follows. We use the gold plugin of LLVM 2.8 [1] with -O4 option to generate executables. To collect memory accesses, a profiling pass is added at the end of the link-time optimization (LTO) passes. The OPT cache simulation uses the OPT* algorithm presented in Section 3.3.5. The same profiling pass is used to measure the performance of LRU and OPT caches using the fastest analyzers available [54, 70].

We examine the floating-point code in three benchmark suites—SciMark 2.0, SPEC 2000, and SPEC 2006 [2-4]—and selected those for which we can reduce the input size so the numbers of accesses are in tens of millions. We increase the
number of time steps in SOR to reduce the effect of its initialization code. As mentioned earlier, as a feasibility study, we use the same input size and cache size in training and in testing.

The 10 test programs are listed in Table 4.3. As the table shows, the programs have between 51 to 37,313 lines of C/Fortran code. There are between 12 to 10,746 static references in the programs. The length of their executions is between 100 and 800 million accesses.

<table>
<thead>
<tr>
<th>workload name</th>
<th>benchmark suite</th>
<th>programming language</th>
<th>#lines of source code</th>
<th>#memory references</th>
<th>#run-time accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOR</td>
<td>SciMark 2.0</td>
<td>C</td>
<td>51</td>
<td>12</td>
<td>1.07E+7</td>
</tr>
<tr>
<td>171.swim</td>
<td>CPU2000</td>
<td>Fortran</td>
<td>435</td>
<td>307</td>
<td>1.02E+7</td>
</tr>
<tr>
<td>172.mgrid</td>
<td>CPU2000</td>
<td>Fortran</td>
<td>489</td>
<td>451</td>
<td>4.13E+7</td>
</tr>
<tr>
<td>173.applu</td>
<td>CPU2000</td>
<td>Fortran</td>
<td>3980</td>
<td>2515</td>
<td>1.50E+7</td>
</tr>
<tr>
<td>183.equake</td>
<td>CPU2000</td>
<td>C</td>
<td>1513</td>
<td>853</td>
<td>8.12E+7</td>
</tr>
<tr>
<td>189.lucas</td>
<td>CPU2000</td>
<td>Fortran</td>
<td>2999</td>
<td>1419</td>
<td>4.26E+7</td>
</tr>
<tr>
<td>410.bwaves</td>
<td>CPU2006</td>
<td>Fortran</td>
<td>918</td>
<td>755</td>
<td>5.30E+7</td>
</tr>
<tr>
<td>433.milc</td>
<td>CPU2006</td>
<td>C</td>
<td>15042</td>
<td>4163</td>
<td>8.43E+7</td>
</tr>
<tr>
<td>434.zeusmp</td>
<td>CPU2006</td>
<td>Fortran</td>
<td>37313</td>
<td>10746</td>
<td>3.75E+7</td>
</tr>
<tr>
<td>437.leslie3d</td>
<td>CPU2006</td>
<td>Fortran</td>
<td>3807</td>
<td>4403</td>
<td>3.80E+7</td>
</tr>
</tbody>
</table>

Table 4.1: The 10 test programs

We simulate fully associative LRU-MRU caches with 8-byte cache blocks. An actual cache is always set associative, but the set associativity on modern systems is high: 4-way L1D, 10-way L2, and 12-way L3 on IBM Power 5; 8-way L1D and L2 and 16-way L3 on Intel Nehalem; and 4-way L1D and 16-way L2 on Niagara II. Hill and Smith showed that for sequential code, 8-way associative cache incurs about 5% more misses than fully associative caches, consistently across cache sizes and cache block sizes [30]. We use fully associative caches, so the results represent the effect of set associative caches without being specific to particular cache parameters. As a limit study, we use 8-byte for the cache line size to exclude the effect of cache spatial reuse, which depends on data layout.
in addition to cache management. The OPT result is the best possible (but possibly not realizable) for all data layouts.

In practice, caches are set associative rather than fully associative. For theoretical analysis, fully associative cache is more interesting (and difficult) because the associativity changes with the cache size. Its properties and results have practical significance. First, the inclusion property holds for each set of set-associative cache and for most real cache hierarchies. Second, modern cache has high associativity, that is, 8-way and up, which means similar performance as fully associative cache [30]. The empirical results in this section show the general effect of LRU-MRU cache of all sizes, regardless of the specific implementation.

4.2.3 The LRU-OPT Gap

Let \( \text{miss}_{\text{LRU}}(C) \), \( \text{miss}_{\text{OPT}}(C) \) be the number of cache misses incurred by LRU and OPT caches of size \( C \). We define the LRU-OPT gap as:

\[
gap(C) = \frac{\text{miss}_{\text{LRU}}(C) - \text{miss}_{\text{OPT}}(C)}{\text{miss}_{\text{LRU}}(C)}
\]

The gap is between 0 and 100%. We have simulated the LRU-OPT gap for the 10 test programs for cache sizes from 1KB up to program data size (before all misses are compulsory misses). The results are summarized in Table 4.2.

The second column of the table shows the average LRU-OPT gap for all measured cache sizes. The highest average gaps are 34% in \( \text{lucas} \) and 31% in \( \text{mgrid} \) and \( \text{milc} \). The first two have hierarchical computations. The least gaps are 12% in \( \text{zeusmp} \) and 17% in \( \text{applu} \). Both are computational fluid dynamics simulation programs. Across all 10 programs, OPT incurs on average 24% fewer misses than LRU does on every cache size.

The improvement from LRU to OPT is not uniform. The gap can be much larger at some cache sizes. The third column of the table shows that the best
improvement is between 50% and 91% in all programs. In other words, for every program there is a cache size for which at least half of the misses in LRU cache can be eliminated by optimal caching. These results show a significant potential for improving cache utilization.

### 4.2.4 The Effect of Reference Hints

Let \( \text{miss}_{\text{Pacman}}(C) \) be the number of cache misses incurred by a program after the Pacman transformation. We define the Pacman improvement as:

\[
\frac{\text{miss}_{\text{LRU}}(C) - \text{miss}_{\text{Pacman}}(C)}{\text{miss}_{\text{LRU}}(C)}
\]

The improvement may be negative if the number of misses is increased by Pacman. We have measured the improvement for the 10 test programs for all cache sizes from 1KB to the program data size. The results are in Table 4.2.

<table>
<thead>
<tr>
<th>Program</th>
<th>the OPT imprv. over LRU</th>
<th>the Pacman imprv. over LRU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>average</td>
<td>largest</td>
</tr>
<tr>
<td>SOR</td>
<td>25%</td>
<td>91%</td>
</tr>
<tr>
<td>171.swim</td>
<td>19%</td>
<td>64%</td>
</tr>
<tr>
<td>172.mgrid</td>
<td>31%</td>
<td>60%</td>
</tr>
<tr>
<td>173.applu</td>
<td>17%</td>
<td>50%</td>
</tr>
<tr>
<td>183.equake</td>
<td>22%</td>
<td>54%</td>
</tr>
<tr>
<td>189.lucas</td>
<td>34%</td>
<td>67%</td>
</tr>
<tr>
<td>410.bwaves</td>
<td>25%</td>
<td>80%</td>
</tr>
<tr>
<td>433.milc</td>
<td>31%</td>
<td>62%</td>
</tr>
<tr>
<td>434.zeusmp</td>
<td>12%</td>
<td>79%</td>
</tr>
<tr>
<td>437.leslie3d</td>
<td>27%</td>
<td>50%</td>
</tr>
<tr>
<td>average</td>
<td>24%</td>
<td>66%</td>
</tr>
</tbody>
</table>

Table 4.2: The LRU-OPT gap and the Pacman improvement. The average improvement is the arithmetic mean of the improvement for all cache sizes between 1KB and data size.
The fourth column of the table shows the average improvement for each program by Pacman. Seven programs—SOR, Lucas, equake, mgrid, swim, bwaves, and leslie3d—show 10% or more average improvements across all cache sizes. Two programs—milc and applu—show near 8% average improvements. The remaining one—zeusmp—does not show a significant improvement (1.4%).

The effect of Pacman can be plotted for all cache sizes using a miss ratio curve. In this section, we show the plots first for Lucas and zeusmp, which have the most and the least improvement in our test set by Pacman, and then for SOR, swim, and applu to show the effects of data size, MRU ratio threshold, and cache line size. The same type of graphs for the other five programs are included in Section 4.2.8.

![Figure 4.2: The miss curves of 189.lucas on fully associative caches](image)

Three miss ratio curves are shown in Figure 4.2 for Lucas when executed with LRU caching, OPT caching, and Pacman caching. The differences between LRU and OPT curves show a large potential for improvement, on average 34% and up to 67%. The Pacman caching realizes over two-thirds of the potential, reducing the miss ratio by 26% on average and up to 64% in the 32KB size cache.
The miss ratio curves of zeusmp are shown in Figure 4.3. There is significant room for improvement over LRU, 12% on average and up to 79%. While Pacman reduces the miss ratio for almost all cache sizes, the reduction is very small (1.4% on average).

The Pacman performance for other programs is somewhere between lucas and zeusmp, as shown by the summary in Table 4.2. On average, Pacman reduces the miss ratio by 12% for each program and each cache size. Optimal caching reduces the miss ratio by 24% on average. Hence, under the idealized conditions used in this study, Pacman realizes one half of the improvement potential of optimal caching.

4.2.5 The Effect of Program Input

So far, we train and test Pacman on the same input. A comprehensive study on the effect of input is outside the scope of this section (our concern here is mainly the theoretical properties and the potential). But we show that for at least one program, Pacman shows similar improvement with different input sizes. In swim,
the matrix size determines the program data size. We compare the results of the matrix sizes $128 \times 128$ and $256 \times 256$.

The miss ratio curves of the two executions of \textit{swim} are shown in Figure 4.4. The Pacman curve has an identical shape in both graphs, showing identical improvements over LRU. But because of the difference in input size, the improvements happen for different cache sizes: 4KB, 8KB, 512KB, and 1MB for the smaller input and 8KB, 16KB, 2MB, and 4MB for the larger input. An LRU curve shows the sizes of working sets in an execution. Comparing the two LRU curves, we can see two working sets in this program. The first working set doubles in size in the larger input, and the second working set quadruples in size. Pacman improves the two working sets by the same degree regardless of the input size.
Figure 4.4: The miss curves of 171.swim on two different inputs. The curves have an identical shape but cover different cache-size ranges: between 1KB and 4MB in the upper graph and between 1KB and 16MB in the lower graph.

4.2.6 The Impact of the MRU Ratio Threshold

Currently, Pacman sets the MRU ratio threshold to 50%. A program reference is designated as MRU if no less than 50% of its run-time accesses are MRU in the optimal solution. The benefit of Pacman depends on the choice of the threshold.

Figure 4.5 shows one example, 173.appu on a 512KB cache, for which different threshold values have a significant effect on performance. When the threshold is
0, all memory references become MRU. The miss ratio jumps to 99%. When the threshold is 100%, only memory references without LRU accesses are selected as MRU. The effect on this program is very close to LRU. When the threshold is 50%, the improvement over LRU is 4.6% (shown for all cache sizes in Figure 4.6). By choosing the threshold 30% or 35%, the miss ratio is further reduced to 20%. This suggests a higher potential if Pacman can use different threshold values for different programs properly.

4.2.7 A Closer Look at SOR

As an example, we show the difference between LRU and OPT cache replacement algorithms using a workload of Jacobi Successive Over-relaxation (SOR) from SciMark 2.0 [2].

Reference Hints at Element Granularity  In collaborative caching, a program designates some of its references to make MRU accesses. Figure 4.7 shows the kernel SOR whose miss rates are just shown in Figure 4.8. It is typical of stencil algorithms. Consider the data access in the loop body. Array $G$ is tra-
Figure 4.6: The miss curves of 173.applu on fully associative caches

```
Require: G is a 2-dimensional double array with the size M*N
1: for p = 1; p < NUM_STEPS; p++ do
2:   for i = 1; i < M-1; i++ do
3:     Gi = G[i];
4:     Gim1 = G[i-1];
5:     Gip1 = G[i+1];
6:       for j = 1; j < N-1; j++ do
7:         Gi[j] = 0.3125*(Gim1[j]+Gip1[j]+Gi[j-1]+Gi[j+1])-0.25*Gi[j];
8:     end for
9:   end for
10: end for
```

Figure 4.7: The SOR kernel computation

versed in each iteration of the outermost loop. If M*N is larger than cache size, array \( G \) cannot fit entirely in the cache. The streaming access of \( G \) would lose all data reuse because LRU evicts the least recently used datum, which is actually the data element that will be reused in the nearest future. OPT, however, would evict the most recently used data element. To obtain the same effect, we can tag the last access to each data element as an MRU access. A bipartite cache of size \( C \) would keep the first \( C \) bytes of \( G \) in cache and reuse them across loop iterations.

We use currently the fastest one-pass analysis methods for LRU [70] and OPT [54]. The LRU and OPT miss rate curves of an execution of SOR are
shown in Figure 4.8 for cache sizes ranging between 1KB and 8MB (twice the size of the program data). One thing to note is that the cache line size is 8 bytes; reference hints get effective at element granularity.

Figure 4.8 shows two interesting aspects of optimal caching compared with LRU.

- **Non-uniform improvement.** OPT is not uniformly better than LRU. The improvement varies greatly between cache sizes.

- **Gradual miss-ratio change.** The miss ratio of OPT decreases gradually as the size of cache increases.

We observe that the curves of OPT and LRU diverge first, converge at size 16KB and then diverge again before both dropping to near zero at 4MB (with only cold-start misses). The difference depends on the cache size. In 16KB or 32KB cache, there is little or no improvement. In 8KB and 2MB cache, the improvement is more than 60% and 90% respectively. It is important to evaluate across all cache sizes.
The OPT miss ratio changes gradually, while the LRU miss ratio either stays the same or drops sharply. The sharp drops mark the size of working sets; each steep descent happens when the cache is large enough to hold the next working set. SOR mainly has two working sets: one at 8KB and one at 2MB. The smooth curvature of OPT shows that it caches a partial working set if the whole set is too large.

**Reference Hints at Block Granularity** All the previous evaluations are based on 8-byte cache line size for limit study. However, real cache systems usually use much larger cache line size such as 64-byte. We change to use 64-byte cache line size to make a more realistic test with SOR.

**Require:** G is a 2-dimensional double array with the size M*N

1: \textbf{for} p = 1; p < \textit{NUM_STEPS}; p++ \textbf{do}
2: \hspace{1em} \textbf{for} i = 1; i < M-1; i++ \textbf{do}
3: \hspace{2em} Gi = G[i];
4: \hspace{2em} Gim1 = G[i-1];
5: \hspace{2em} Gip1 = G[i+1];
6: \hspace{2em} Gijm1 = Gi[0];
7: \hspace{2em} Gij = Gi[1];
8: \hspace{3em} \textbf{for} j = 1; j < N-1; j++ \textbf{do}
9: \hspace{4em} Gim1j = Gim1[j]; \textbf{9.1: if} j\%8 == 7 \textbf{then}
10: \hspace{4.5em} Gip1j = Gip1[j]; 9.2: \text{MRU}\_\text{load}(\text{Gim1}[j]);
11: \hspace{4em} Gijp1 = Gi[j+1]; 9.3: \textbf{else}
12: \hspace{4.5em} tmp1 = Gim1j + Gip1j; 9.4: \text{Gim1}[j] = \text{Gim1}[j];
13: \hspace{4.5em} tmp1 += Gijm1; 9.5: \textbf{end if}
14: \hspace{4em} tmp1 += Gijp1;
15: \hspace{4em} tmp1 *= 0.3125;
16: \hspace{4em} tmp2 = -0.25 * Gij;
17: \hspace{4em} tmp1 += tmp2;
18: \hspace{3em} Gi[j] = tmp1;
19: \hspace{3em} Gijm1 = tmp1;
20: \hspace{3em} Gij = Gijp1;
21: \hspace{2em} \textbf{end for}
22: \hspace{1em} \textbf{end for}
23: \textbf{end for}

Figure 4.9: The SOR kernel loop in SSA form with Pacman transformation. $M = N = 512$ and \textit{NUM\_STEPS} = 10.
Figure 4.9 shows the SSA-form [17] of the SOR loop kernel (for original code see Figure 4.7). The loop indexes into array $G$ to create three virtual arrays $G_i, G_{im1}, G_{ip1}$ for use in the innermost loop.

The innermost loop has four array references. The MRU ratio changes with cache sizes, as shown in Figure 4.10 as a curve for each reference. The ratio for $G_{im1[j]}$ is clearly higher than the other three. For cache size between 8KB and 512KB, the MRU ratio is from over 12.5% to 63% for $G_{im1[j]}$ but near 0 for the other three. Pacman chooses this reference as an MRU reference.

![Figure 4.10: The MRU ratio curves of SOR on fully associative caches with cache line size 64B](image)

The MRU ratio is a factor of 8 lower because of spatial reuse. To separate the last touch of a cache block, we transform line 9 to an if-else block (line 9.1 to 9.5) in Figure 4.9. In actual implementation, we use loop unrolling instead of branching. In LLVM, we adapt the available loop unrolling pass and put it at the end of the LTO passes but before the profiling pass. We also change to use `memalign()` instead of `malloc()` to make array $G$ 64-byte aligned. After loop unrolling, the load in line 9.2 has an MRU ratio of 75% at 512KB.
The miss ratio curves of Figure 4.11 show that Pacman produces almost identical results as OPT for cache sizes over 64KB (up to 2MB). The improvements are significant: 2.3%, 5.2%, 10.8%, 22.2%, 44.9%, and 90.7% respectively between 64KB to 2MB. The average improvement is 15%, as reported in Table 4.2. It is worth mentioning that at cache size 2MB, OPT training found 704 MRU accesses out of more than 10 million accesses. These MRU accesses reduced the miss ratio by an order of magnitude from 3.3% to 0.3%.

### 4.2.8 The Miss Ratio Curves of LRU, OPT, and Pacman

Figure 4.12 to 4.16 shows the miss-ratio curves of the rest of the test programs (in addition to programs already shown).
Figure 4.12: 172.mgrid

Figure 4.13: 183.equake

Figure 4.14: 410.bwaves
4.3 Loop-based Pacman

Section 4.2 has showed a technique called reference-based Pacman using LRU and MRU memory instructions in a mixed way to reduce cache misses. The potential improvement is significant. However, we use MRU ratio to decide whether a memory instruction goes with LRU or MRU. When the value of MRU ratio is 0.5, half of accesses from the memory instruction are selected to use the wrong access type. This fundamental problem has to be solved.
In this section, we present an advanced variant of Pacman for the above fundamental problem. Since accesses to a large amount of data are usually made in loops, in the second solution, loop-based Pacman, we target loop code. Loop-based Pacman has three components to address the above fundamental problem. The first is loop profiling, which builds linear models of the relation between the iteration count and the forward OPT distance. It uses cross-input profiling to identify how the OPT distance changes with the input size. Based on profiled patterns, the second component, LRU/MRU loop splitting, divides a loop at compile time for different types of caching. Finally, at run time, the breakpoints between the split loops are dynamically adjusted based on the current input size and cache size. The section presents results using both simulated and real systems.

<table>
<thead>
<tr>
<th>trace</th>
<th>xyaxybxydxyexyfxygxyaxybxydxye</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory reference</td>
<td>RSTRSTRSTRSTRSTRSTRSTRSTRSTRSTRSTR</td>
</tr>
<tr>
<td>optimal hint (c=5)</td>
<td>LLLLLLLLLMLLMLLLMLLMLLMLLMLLMLL</td>
</tr>
<tr>
<td>pacman hint (c=5)</td>
<td>LMLMLMLMLMLMLMLMLMLMLMLMLMLMLML</td>
</tr>
</tbody>
</table>

Figure 4.17: An example of loop-based Pacman hint insertion.

Figure 4.17 is an example for loop-based Pacman hint insertion, which is the same as Figure 4.1. As mentioned in reference-based Pacman, some accesses have to use non-optimal hints because accesses from the same memory reference share the same hint. Here, reference \( T \) needs multiple hints. Our observation is that optimal hints and non-optimal hints are mixed in a regular way. Loop-based Pacman will do loop splitting to make two copies for \( T \). In this way, every six accesses of \( T \) are divided into two parts: two accesses from an LRU reference and the other four from an MRU reference.

### 4.3.1 Forward OPT Distance Profiling

Pacman profiles three types of program events: memory accesses, loops, and function calls. For memory profiling, we track every load and store. For function
call-site profiling, we insert profiling calls before and after every call site. Loop profiling has three parts: generating the loop entry event, generating the loop exit event, and generating the loop tick event for every iteration. In this way, the instrumented code generates a full memory trace with interleaved loop and function events.

From the execution trace, we first use the OPT stack simulator by Sugumar and Abraham [54] to compute the OPT distance for each memory reference. The distance is *backward* because it comes at the end of the data reuse pair. We convert it into a forward distance by recording it at the start of the data reuse pair. To avoid storing the whole trace in memory, we generate the access time and OPT distance pairs to a file. Because the forward distances are generated out of order, we use the Unix sort command in the second pass to reorder the pairs by access time, after which we have the trace of forward OPT distance needed for pattern analysis.

To prepare for pattern analysis, we split the trace by data references. For each data reference, we record the loop iteration count instead of the logical time. After this step, each reference contains a list of its accesses, each represented by the iteration number and the OPT distance. Since a reference may be inside a nested loop, we generate the iteration count for each of the enclosing loops of the reference. To maintain consistent nesting, we consider only nested loops inside a procedure.

### 4.3.2 Pattern Recognition

An OPT pattern is a formula from which we can calculate the OPT distance that we observe. A reference may make many accesses and exhibit multiple patterns. This section defines the patterns we consider and gives an analysis technique that detects the majority patterns, which will be used in the next step of hint insertion.
**Linear Patterns**  The OPT distance in loops has mainly two types of patterns. The first is short distances due to spatial reuse. For example, as a loop traverses a double precision array, it moves to a different cache block every eighth access. The seven preceding accesses are spatial reuses. The OPT distance of spatial reuses depends on how many other data are being accessed at the same time, but the distance is usually short since the size of the loop body is bounded. The eighth access, however, may have a long OPT distance. The distance increases with the iteration count. Intuitively, this is because it takes a larger cache to store more data. We call the pattern of increasing distance *proportional caching*.

As an example, Figure 4.18 plots the OPT distances for 647 thousand accesses by one of the 197 references in the SPEC 2000 benchmark program *Swim*. Each of the 647 thousand distances (or points) is plotted as a small circle. The spatial reuse pattern is visible as the line at the bottom of the figure. Its distances are between 5 and 10. The pattern contains about 7/8 (88%) of all the points. The five diagonal lines (from lower left to upper right) show the pattern of proportional caching, where the OPT distance increases by 38 cache blocks for every 100 iterations (a slope of 0.38). The diagonal lines contain most of the remaining points. The distances outside these two patterns account for less than 0.001% of the total points (262 out of 646,944).

The previous example is one of the 38 references in a common loop. It is useful to examine another reference to see how much its pattern may differ. The second reference has a spatial reuse pattern with 88% of distances between two numbers, 2 and 3. Most of the remaining distances have the pattern of proportional caching, where the distance increases by 13 cache blocks for every 100 iterations (a slope of 0.13). The ratio of the slopes of the two references, $38/13 = 2.9$, shows the resource allocation by OPT: for every cache block allocated for the first reference, 2.9 blocks are allocated for the second reference.
Figure 4.18: The OPT distances exhibited by a reference in an execution of Swim.

**Grid Regression for Multi-pattern Analysis**  Most OPT distances form linear patterns. Either they stay within a constant range, as in the case of spatial reuse, or they grow at a constant rate, as in the case of proportional caching. However, in real data the patterns are mixed, for example in the one shown in Figure 4.18. Linear regression is ineffective in such cases. The problem is not the absence of patterns but a combination of multiple patterns that makes the analysis difficult. The problem can be solved if we separate the data points into their pattern groups before applying linear regression within each group. Based on this observation, we develop a technique we call *grid regression*.

Given the data as a series of (x,y) coordinates on a two-dimensional plane, grid regression first spatially divides along both x, y dimensions and distributes the data into regions. For each region, we run linear regression. If some regions produce strong linear patterns, they are used as elementary patterns. In the second step, grid regression merges elementary patterns to consolidate them into
overall patterns. An overall pattern includes as many compatible elementary patterns as possible, and different overall patterns represent distinct behavior. For efficiency, we choose large enough overall patterns as major patterns. Finally, the remaining data points, including those in the regions that do not show strong linear patterns in the first step, are fitted into one of the major patterns if possible. The remaining points, if any, are considered to have no major pattern.

The analysis of grid regression is bottom up. The elementary patterns are seeds from which larger patterns “grow”. The remaining data are then absorbed into the established patterns. Consider a region where data points from multiple patterns are mixed. The basic linear regression cannot identify multiple patterns unless the data points are first separated, but it cannot separate the data unless it knows their patterns. Grid regression deals with this “chicken-or-egg” problem by relying on the chance that these patterns have an individual appearance in other regions. The key to the pattern separation is the use of the grid.

To determine whether a group of data points have a strong linear pattern, a traditional method uses some measure of the goodness of fit in particular the coefficient of determination, commonly known as $R^2$. We found that $R^2$ is not a good fit for our purpose. $R^2$ measures strength of linear correlation. Its value is zero when the pattern is constant (for a horizontal line), as in the case of the spatial reuse pattern. Instead, we use the absolute precision and require that the predicted distance be within a constant, for example, 100 in our implementation, of the measured distance. A region has an elementary pattern if 90% of its data points fit into a linear model.

Our algorithm has a linear time and space overhead. The grid partition and the following steps process the data points, at most, a constant number of times, assuming we have bound the number of major patterns by a constant. The algorithm can be parallelized, especially in the first and the last steps, because of its distributive design.
The algorithm records the number of points that fit in each major pattern. It can sort the patterns by size and evaluate each pattern’s significance by comparing its size with the total size. Therefore, multi-pattern analysis is important not only in pattern separation (hence accurate recognition) but also in discerning important patterns.

4.3.3 Loop Splitting and Hint Insertion

In prediction, we look for major patterns that are few but have a high probability. Too many patterns would mean a large overhead and imprecise prediction.

From OPT Pattern to LRU/MRU Decision A reference has a linear pattern $p$ with an intercept and a slope. If the cache size is $c$, the following function finds the break point before which the reference should make LRU accesses and after which it should make MRU accesses for available cache size $c$.

```plaintext
function ref_split(p, c) {
    return (c - p.intercept) / p.slope
}
```

Given the desired cache use at run time, the formula converts a linear pattern into a single number and fixed decision of LRU/MRU with respect to that number. This formula also makes sure that data are decided to retain in cache if and only if their forward OPT distances are less than cache size, which has been showed in Theorem 3.10.

Loop Splitting A loop has $k$ references, each of which has a linear pattern. It produces at most $k$ different breakpoints. The reference is LRU in the loops before its breaking point and MRU in the loops after its breaking point.
/ *loop x*

```c
for i in 1, n {
    body
}
```

is transformed to

```c
s_1, s_2, .., s_k = split_loop(n, p_1, p_2, ..., p_k)
for i in 1, s_1
    body_x_1
end
for i in s_1, s_2
    body_x_2
end
... ...
for i in s_(k-1), s_k
    body_x_k
end
for i in s_k, n
    body_x_(k+1)
end
```

The function `split_loop` calls `ref_split k` times to get all splitting points. After loop splitting, there may be $k + 1$ sub-loops. All references in the first sub-loop `body_x_1` use LRU. In the second sub-loop `body_x_2`, the reference for the smallest splitting point `s_1` uses MRU and other references still use LRU. In this way, one more reference is changed to use MRU for the next sub-loop. Finally, all references in the last sub-loop use MRU.
**Nested Loops** Pattern analysis is designed for generic $x, y$ inputs and used to analyze each loop level. As an example, consider a two-dimensional $i, j$ loop that traverses a 2-dimensional array. The inner $j$ loop instances each have a series of $x, y$ accesses, where $x$ is the iteration count of $j$ and $y$ is the OPT distance. The value $x$ ranges from 0 to $n_j$, the number of $j$ iterations. The outer $i$ loop instances have the same $y$s, but $x$ is the cumulative iteration count, which ranges from 0 to $n_jn_i$, where $n_i$ is the number of $i$ iterations. In this example, grid analysis would identify many linear patterns at the outer loop $i$, one for each $j$ loop instance, but no majority patterns. At the $j$ loop, it would find a single pattern that includes all data and choose $j$ loop to apply loop splitting.

### 4.3.4 Cross-Input Pattern Prediction

Using patterns discerned for two training runs with different input sizes, we can predict the patterns for a third input set at run time. The input size may be taken to be the dominant loop counts or the size of major arrays. For example, a $4 \times 4$ matrix represents an input size of $z = 16$. For linear patterns $p_1$ and $p_2$ with intercepts $\text{intercept}_1$ and $\text{intercept}_2$, and slopes $\text{slope}_1$ and $\text{slope}_2$, the third pattern’s intercept is taken to change in proportion to the input size. In other words, we assume that the first access’s OPT distance scales with the data size. Taking $\Delta z_n \equiv z_n - z_1$, and $\Delta \text{intercept}_n \equiv \text{intercept}_n - \text{intercept}_1$ to represent the changes in the input size and the intercept from $p_1$ to $p_n$,

$$\frac{\Delta \text{intercept}_3}{\Delta \text{intercept}_2} = \left( \frac{\Delta z_3}{\Delta z_2} \right).$$

This leads to

$$\text{intercept}_3 = \text{intercept}_1 + (\text{intercept}_3 - \text{intercept}_1) \left( \frac{z_3 - z_1}{z_2 - z_1} \right).$$
To predict the slope, we make use of two different methods: (1) Conservatively, we only predict a third pattern in the first place if \( \text{slope}_1 = \text{slope}_2 \). (2) More generously, we predict a third pattern from any two training patterns taking the predicted slope to be the average of the two training pattern slopes:

\[
\text{slope}_3 = \frac{\text{slope}_1 + \text{slope}_2}{2}.
\]

The slope represents the increase in OPT distance per iteration. We do not necessarily expect the OPT distance to increase faster for a larger input size. We only expect it to increase at some rate as new data are introduced during program execution, so the slope shouldn’t increase, either.

To find the input parameter for each reference, we examine the loop bounds and use training to select the loop bound that gives the best fit. More complex learning methods may be used to select the input parameter for the whole program instead of for each loop. Mao and Shen gave a three-step solution—feature extraction, incremental modeling, and discriminative prediction—and showed accurate prediction of the loop bounds in a program from a few extracted parameters [40]. Our test programs shown in Table 4.3 are all scientific simulations of some physical space. We will use the size of the data array as the input size.

A compiler can recognize the contiguous access of a single array. It is known as a streaming access or, in compiler terminology, group spatial reuse. However, just knowing the access does not give us the OPT distance information we need. If compiler analysis is not path insensitive or interprocedural, it may not find the last reference to an array. More importantly, the static analysis does not give the relative cache allocation, which depends on all uses of all data, for example, how many arrays are reused, and the amount of cache available. We rely on profiling and grid regression to find last references of arrays and their cache allocation ratio. It is an open question whether static analysis can infer OPT distances.
4.3.5 Evaluation Setup

We use eight workloads in Table 4.3 for the evaluation. Streaming is a synthetic program traversing a large array multiple times. SOR is Jacobi Successive Over-relaxation modified from SciMark 2.0 [2]. The next three—swim, mgrid, and applu—are from SPEC CPU2000 [3]. The last three—bwaves, leslie3d, and zeusmp—are from SPEC CPU2006 [4]. For each workload, we have three different input sizes: small, medium, and large. We chose these programs because their data size and running length can be adjusted by changing the input parameters. The training process uses OPT, which cannot handle overly large traces. In practice, we expect users who have domain knowledge of their programs can effectively reduce the input size for training. The remaining question is whether training on small inputs is effective.

In all testings, we keep the cache associativity 16-way and cache line size 64-byte. The only change is cache size.

We do not evaluate actual loop splitting. Instead we assume each reference can switch its access from LRU to MRU once at some breakpoint in a single execution of a loop. There are ways to reduce the number of cloned loops at the cost of less precise splitting for some references, possibly by weighing on the relative importance of the references. However, our evaluation is mostly done through cache simulation. Since the number of loop splits mainly affects the CPU, which we do not simulate, it is premature to start designing solutions to reduce the looping overhead.

4.3.6 Optimal Caching of Group Spatial Reuse in SOR

McKinley et al. defined the term group spatial reuse in a loop when a group of references all traverse the same array [42]. An example is the second, main loop of SOR:
for (p=0; p<TIME; p++) {
    for (i=1; i<SIZE-1; i++) {
        for (j=1; j<SIZE-1; j++) {
            G[i][j] = CONST1 * (G[i-1][j] + G[i+1][j]
                + G[i][j-1] + G[i][j+1]) + CONST2 * G[i][j];
        }
    }
}

In the j loop, all six array references have group spatial reuse. Pacman profiling shows that the compiler generates three references for the loop. Two of the references have only short OPT distances. Only the third reference shows a linear
pattern. In the smallest input, in which \textit{SIZE} is 512 and \textit{G} contains 512 by 512 double precision numbers, the intercept is 41KB, which is a base allocation of cache for the part of the \textit{G} array reused across the \(i\) loop and scalar variables. The base code reuses two rows of the \textit{G} array (8KB). It appears that the compiler optimization has increased the amount of reuse. This 41KB is the first-level working set observed by profiling.

The rest of the \textit{G} array is the second-level working set. Because it may be larger than cache, we need to choose which part to keep. The slope of the linear pattern gives the answer, which is 8 byte per \((i, j)\) iteration after the first working set. For example, if the cache size is 128KB, the \((i, j)\) loop would be split into two parts. The first has only LRU accesses for 11 thousand iterations \(\left(\frac{128KB - 41KB}{8B}\right)\), which keeps additional 87KB of \textit{G} array data in cache. The second part of the \((i, j)\) loop uses MRU access for the third reference, so to keep the rest of \textit{G} out of cache. In total, we retain 128KB (41KB+87KB) of \textit{G} array data in cache, which is consistent with the intuitive way.

As Pacman profiles the medium-size input, \textit{SIZE} = 1024, it sees the same types of patterns: short OPT distances in all except for the third reference. The linear pattern has the same slope as before, but the intercept has changed from 41KB to 82KB. Pacman does the same calculation for the splitting point. Finally, still only 128KB of \textit{G} array data are retained in cache.

The improvements by Pacman are shown in Figure 4.19 for two inputs of \textit{SOR}. The cache sizes shown are 0.5MB to 4MB for the medium-size input and 2MB to 16MB for the large-size input. The medium-size improvement is based on self training, showing the best case. The large-size improvement is done through the cross-input pattern prediction. The improvements are near identical: 5\%, 11\%, 21\% and 43\% for the four cache sizes. The cross-pattern prediction does not lose performance even though the pattern is applied on an input that has not been used in training. In both runs, the performance is very close to the maximal
The LRU policy cannot effectively cache the array. The miss ratio is 3.4% for 64KB cache and does not change until the 32MB cache, when the whole array, 27MB, fits in cache. Pacman, like OPT, allocates a larger part of the array in cache as the cache size increases. As a result, it increases data reuse gradually. The following table shows this gradual increase, which is the reason for the increasing reduction ratio seen in Figure 4.19.

<table>
<thead>
<tr>
<th>cache size</th>
<th>1MB</th>
<th>2MB</th>
<th>4MB</th>
<th>8MB</th>
<th>16MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRU miss ratio</td>
<td>3.4%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pacman miss ratio</td>
<td>3.3%</td>
<td>3.2%</td>
<td>3.0%</td>
<td>2.6%</td>
<td>1.9%</td>
</tr>
</tbody>
</table>

Table 4.4: Pacman makes full utilization of cache space and gradually reduces the miss ratio as the cache size increases.

Compiler analysis has long been used to identify group reuse in loops for locality optimization [6, 42, 58]. For prefetching, a compiler can identify the reference in the group that begins the access of an element. For optimal caching, in contrast, Pacman finds a reference that uses the data for the last time, and it finds it through profiling; it is the reference with a linear pattern. Pacman profiling analyzes the generated code. It sees the effect of all compiler optimization and code
generation, which gives Pacman a precise view of the aggregate data demand. Its pattern analysis automatically identifies group reuse and the breakpoint of cache allocation. More importantly, when a loop uses more than one array, Pacman may find optimal allocation among the arrays, as we show next.

4.3.7 Program-directed Cache Allocation in Swim

The following is a loop in the swim benchmark. The three statements updates three arrays. Just by looking at the loop, there is no reason to favor one array or the other for caching.

C line 335 in swim.f

DO 215 I=1,M
  UNEW(I+1,N+1) = UNEW(I+1,1)
  VNEW(I,1) = VNEW(I,N+1)
  PNEW(I,N+1) = PNEW(I,1)
215 CONTINUE

Pacman profiling finds six references in the loop, three of which have linear patterns. The slopes are different: 2 bytes per iteration for the 2nd and 6th reference and 2.67 bytes per iteration for the 4th reference. In other words, if the cache size increases by 8KB, the allocation for the 3 arrays are 2KB, 2.7KB and 2KB respectively. The different allocation for the 4th reference indicates a different reuse pattern compared to the other two arrays. The cause is outside this loop: in the outer loop that includes most of the 33 loops of the program.

Figure 4.20 shows the percent reduction of misses for four cache sizes. When training and testing on the same input, Pacman reduces the number of misses by 3%, 6%, 22% and 39%. When training on two small inputs and testing on a large input, the reduction becomes 3%, 5%, 41% and 58%. The larger reduction has to
do with the relationship between the data size and cache size and does not mean
the Pacman pattern is more accurate for the larger input.

**Input Shapes**

We use the array size as the input parameter for *swim*. Two arrays may have
different shapes but the same size. We have tested two other inputs. The first is
200 by 737, which we choose to have the same total size as 384 by 384. Pacman
predicts the same linear patterns for the two executions. Similarly, we choose 300
by 837 to be the same size as the largest input 512 by 512. We test Pacman
by changing just the array shape, just the input size, and both. The reduction numbers are shown in Figure 4.20. There is no significant loss of Pacman benefits as a result of these changes, showing the robustness of Pacman pattern analysis and cross-input pattern prediction for this program.

4.3.8 Six Other Programs

We show the results for the remaining six programs in Figures 4.21 and 4.22. As the commonly used benchmark to measure memory bandwidth, Stream repeatedly traverses a set of large arrays. Our version uses just one array. The pattern is simple. Pacman obtains a performance close to optimal.

The other five programs are physical simulations. Mgrid is a multi-grid solver that computes a three dimensional scalar potential field. Applu, bwaves, zeusmp, and leslie3d are all fluid dynamical simulations. Applu solves five coupled nonlinear PDE’s on a three dimensional grid, bwaves and zeusmp both simulate blast waves (zeusmp does so specifically for astrophysical scenarios), and leslie3d simulates eddy flows in a temporal mixing layer.

Applu and mgrid come from the SPEC CPU2000 floating-point benchmark suite. There are 418 references in 46 loops in mgrid and 1858 references in 125 loops in applu. The types of patterns are more complex than those of SOR.

Figure 4.23 shows the OPT distance plot of the same reference in two of the five enclosing loops. In both graphs, the y coordinate is the OPT distance. The x coordinate is the loop index of one of the two loops. Both are more complex than the pattern we showed in Figure 4.18. Still, the grid regression recognizes the major linear pattern in the loop shown in the bottom graph. We found many references in applu to have this pattern. Mgrid does not have as many strong linear patterns, partly because of its divide-and-conquer type computation. As a result, the reduction by Pacman is higher in applu, from 2% to 34%, than in
Figure 4.21: The improvements by Pacman and OPT over LRU
Figure 4.22: The improvements by Pacman and OPT over LRU
Figure 4.23: One of the representative OPT distance patterns in *applu*. The two graphs show the same reference with the same series of the OPT distances as the *y* coordinate but with different *x* coordinates as the iteration count in two of the five enclosing loops.
mgraid, from -6% to 10%. In both programs, the improvement is reduced when predicting for a different and larger input in most cases.

The last three programs are SPEC CPU2006 floating-point benchmarks. Zeusmp has as many as 10 thousand references in nearly 500 loops. Pacman obtains significant miss-ratio reduction over LRU, 11% to 24% for leslie3d and 0% to 26% for zeusmp.

The third program, bwaves, shows a large increase in the miss ratio as a result of Pacman. The reason is that pattern analysis produces false positives. For large set-associative cache, an over-use of MRU accesses is not immediately harmful because the chance is that the following accesses would visit somewhere else in cache. For small cache sizes, however, the incorrect MRU accesses may cause the eviction of high locality data and hence an increase of the miss ratio. We found that in all cases in which Pacman increases the miss ratio, the cache size is smaller than 1MB. In addition, for bwaves the cross-input analysis finds a few common patterns.

4.3.9 Comparison to Dynamic Insertion Policy

The Dynamic Insertion Policy proposed by Qureshi et al. [46] divides the cache into three sets: (1) a small set of cache blocks dedicated to the LRU policy, (2) another small set of cache blocks dedicated to a mostly-MRU policy called Bimodal Insertion Policy, and (3) the majority of cache blocks that follow whichever of the first two policies is performing best at any point in time.

For workloads that are LRU-averse during any phase in their execution, DIP can outperform LRU by adaptively choosing BIP. For workloads that are LRU-friendly throughout, DIP consistently allocates the majority of cache blocks to LRU, but can in rare instances even be outperformed by LRU because of misses incurred by BIP dedicated blocks.
For our comparisons, we use a DIP implementation of Xiang et al. [63], which follows the description in the original paper and selects the parameters the paper mentioned: a policy selection threshold of 1024 (so that the LRU set must have 1024 more misses than the BIP set in any time window to trigger a policy switch to BIP), and "bimodal throttle parameters" of 1/16, 1/32, and 1/64 (so that each BIP block randomly uses LRU instead of MRU with this probability). The lowest miss rate between these three options is always reported although there is only small variation.

In the SOR workload outlined above, DIP gives nearly the same miss ratio as LRU because, during the traversal of $G$, there are a significant number of immediate reuses, causing misses in the BIP blocks. While there are also low locality accesses, BIP never outperforms LRU strongly enough to trigger a policy switch to BIP. DIP does not reduce the number of misses in our test suite with the exception of zeusmp, for which it makes a significant reduction for two of the cache sizes for each of the inputs shown in Figure 4.22. The changes in all other programs are less than 1%. While DIP does not improve the performance, it does not degrade it, which makes it a safe policy.

Pacman is a program-level technique. It requires profiling analysis and program transformation. The LRU/MRU breakpoint is made sensitive to data size and cache size. As a result, Pacman improves all but one program in the test suite. To be fair, we did not tune DIP beyond the parameters specified in the paper, and DIP was not developed with our tests with the exception of swim. Swim was deemed "inherently LRU" as DIP was not able to improve its performance [46]. At least for this program, program-level assistance is likely to go beyond LRU and approach the level of optimal caching.
4.3.10 Pacman on Real Hardware

The x86 ISA provides non-temporal store instructions which can bypass cache. They can write to memory without loading the corresponding cache line first. By storing data that does not have good temporal locality, non-temporal write can avoid polluting the cache and reduce memory traffic. SSE4.1 adds a non-temporal read instruction which is limited to write-combining memory area. For regular data that resides in main memory, the non-temporal read does not bypass the cache hierarchy [32]. There are also non-temporal prefetch instructions on x86, but they do not provide the full functionality of a non-temporal read.

With the partial support of cache management on x86, we can evaluate our loop splitting and hint insertion technique on real hardware if all MRU accesses inserted by Pacman are writes. We run our tests on a machine with an Intel Xeon E5520 processor. The processor contains four symmetric 2.27GHz cores that share an 8MB L3 cache. With hyper-threading enabled, the processor can support up to eight hardware threads.

Figure 4.24 shows the kernel of our test program. The outer loop advances in time step. In each time step, the inner loop updates each element of array $A$ based on its old value. The inner loop is parallelized using OpenMP. The size of array $A$ is set to 12MB, thus the last level cache is not able to hold all the data. In LRU, the temporal reuses of $A$ in the outer loop are not explored.

```c
for(t=0; t<MAXITER; t++)
#pragma omp parallel for
    for(i=0; i<N; i++)
        A[i] = foo(A[i]);
```

Figure 4.24: An OpenMP example: the inner loop updates the array element by element; the outer loop corresponds to the time step.

To enable the outer loop reuse, Pacman splits the inner loop into two, each of which is still an OpenMP parallel loop. The first loop only processes the first
8MB of array \( A \), and the second loop the last 4MB of \( A \). In the second loop, the store of \( A[i] \) is the non-temporal access via the GCC intrinsic \texttt{_mm_stream_pd}.

The transformed code keeps the first 8MB of \( A \) in the last level cache for reuse throughout the time steps. Because the inner loop also reads array \( A \) before writing to it, the inner loop is unrolled eight times, and the non-temporal stores happen after the eight reads and computations. To exclude the unrolling effect from the performance comparison, we also perform the same loop unrolling on the original program.

Figure 4.25: The performance comparison on Intel Xeon E5520 with hardware prefetching

Figure 4.25 and Figure 4.26 gives the performance comparison between the original program and the optimized version. Another version, which only naively replaces the regular stores with non-temporal ones without splitting the inner loop, is also included for comparison. Figure 4.25 depicts the performance data with hardware prefetching on and in Figure 4.26 without prefetching. In the second
case, we turn off all four kinds of hardware prefetchers by setting the corresponding Model-Specific Register (MSR) on all cores, similar to the approach in [59]. We test our programs for up to eight threads, threads 1-4 are assigned to four physical cores, and threads 5-8 are bound to four hyper-threads.

In Figure 4.25, the Pacman code is 1.19 times slower than the original program when using 1 thread, despite the fact that it utilizes the temporal locality in the outer loop. The reason is the higher overhead of the non-temporal store instruction and possibly the interference with hardware prefetching. At more than two threads, however, Pacman is clearly better. At eight threads, it reduces the execution time by 32.4%. The results show the relationship between cache reuse and prefetching. When there is enough bandwidth, prefetching is effective; when the bandwidth is saturated at high thread counts, cache reuse becomes important.
Loop splitting by Pacman is also important. Without it, the performance is as much as 47% lower than the original program. If we turn off prefetching, Pacman assisted caching is uniformly beneficial, with 17% improvement in one thread and 25% improvement at 8 threads, as shown in Figure 4.26.

4.4 Summary

In this chapter, we have presented an LRU-MRU collaborative caching technique called Pacman based on OPT training. There are two Pacman variants: reference-based Pacman and loop-based Pacman.

The reference-based Pacman can be used for general programs. No program transformation is required other than cache hint insertion. The evaluation on 10 SciMark and SPEC CPU benchmarks show that reference-based Pacman can reduce the miss ratio by 24% on average per program per cache size, and collaborative caching has the potential to realize 50% of the optimal performance improvement.

There are two fundamental problems in reference-based Pacman. The first problem is the repetitive OPT cache simulations for different cache sizes. In Section 3.5.4, we have showed that this limitation can be removed by using priority hint as forward OPT distance. In the refined loop-based Pacman, we use this theoretical finding with OPT distance profiling. We also have showed grid regression and cross-input analysis to recognize and predict forward OPT stack distances.

The second fundamental problem is that a static memory reference may have both LRU and MRU accesses at run time. Because one reference can only have a single cache hint, some accesses may go with wrong access types at run time, which makes collaborative caching suboptimal. Loop-based Pacman uses loop splitting to transform programs to make separate static references for run-time accesses with different access types.
By evaluating loop-based Pacman using a mix of simple and complex benchmark programs, we find out that most programs exhibit strong linear patterns that can be captured by OPT profiling. The reduction over LRU is significant and becomes as much as 40% to 60% when managing the large cache. Real-machine experiments suggest that the special MRU access incurs an overhead. Still, the improved cache reuse can improve performance when prefetching is difficult or when the memory-bandwidth contention is high. From these results, we believe that computer architects should consider adding LRU/MRU hints to enable collaborative caching.
5 Conclusions and Future Work

This dissertation has laid the foundation for optimal collaborative caching and studied ways to realize the theory in practice. The four main conclusions are as follows:

1. **LRU-MRU Cache Can Be Optimal.**
   
   Optimal caching has been studied for a long time. It was deemed impractical and only useful for studying the best cache performance theoretically because it requires the knowledge of the future and complicated hardware support. This dissertation proves that LRU-MRU collaborative caching with a simple hardware interface supporting both LRU and MRU accesses is capable of achieving optimal caching.

2. **LRU-MRU Cache is Inclusive and Admits LRU-MRU Stack Distance.**
   
   Inclusion property is very important for a cache system. It guarantees that there is no degradation when the same program runs with a larger cache. This dissertation proves that LRU-MRU cache preserves the inclusion property. A user can enlarge the LRU-MRU cache for better performance as they can with the LRU cache. The LRU-MRU stack distance exists. We have
devised an efficient algorithm to compute the LRU-MRU stack distance using hybrid priority, which is the current access time for an LRU access but the negative of the current access time for an MRU access.

3. **Priority LRU has Non-Uniform Inclusion.**

Priority LRU is based on generalized priority hints. In LRU-MRU, there are only two choices for a cache hint. A priority hint allows more than two choices. In Priority LRU, depending on the priority, the accessed data may go to any cache location and out of the cache. This dissertation proves that Priority LRU preserves the inclusion property. The proof requires a careful study of all priority cases. More interestingly, the inclusion property in Priority LRU is more generalized because for the first time, it is possible that the stack layout in the existing cache space can differ when the cache is enlarged to include more space. We call it non-uniform inclusion. Due to non-uniform inclusion, the classical algorithms for computing stack distances have to be radically revised to handle Priority LRU. A new one-pass algorithm has been developed.

4. **LRU-MRU Cache is Practical.**

A software technique using the LRU-MRU cache has been developed. The technique is called Pacman. It uses OPT training to decide the optimal cache hint for each access. Then it uses heuristics to insert the LRU-MRU hint for each memory reference. Basic Pacman is too restrictive when the same memory reference needs to mix LRU and MRU accesses for an optimal effect. An improvement to Pacman was developed for loop-based code that uses loop splitting so it allows a memory reference to use both LRU and MRU accesses and dynamically determine the hint based on the program input and the cache size.

There are at least two open questions for the future work:
Performance Improvement on a Real Machine  The study has shown the improvement potential of collaborative caching and measured the potential in the significant reduction of cache misses. However, the study is based on a cache simulator. Section 4.3.10 has found promising improvements for a simple loop kernel on real hardware. It is yet to be verified that more realistic benchmark programs can be similarly improved. On a real system, many practical problems have to be addressed including:

- Hint insertion and instruction scheduling: LRU and MRU could be different in latency. After hint selection, a pass of instruction scheduling is needed, but the new scheduling can change the hint selection, creating a circular problem.

- Code size increase: loop splitting in Pacman may make too many copies of the same loop body, which has a negative impact on the instruction cache.

Performance Improvement for Parallel Programs  In the dissertation, most of the programs evaluated are sequential. The study on parallel programs is very important since parallel programming is becoming ubiquitous with modern multicore processors. Coordinating different threads or processes to achieve optimal caching is an open problem. Cache interference and data sharing make the problem much harder than the original problem in sequential and standalone executions.
Bibliography


