Pacman: Program-Assisted Cache Management *

Jacob Brock
Department of Computer Science
University of Rochester
Rochester, NY, USA
jbrock@cs.rochester.edu

Xiaoming Gu †
Azul Systems, Inc.
Sunnyvale, CA, USA
xiaoming@azulsystems.com

Bin Bao †
Adobe Systems Incorporated
bbao@adobe.com

Chen Ding
Department of Computer Science
University of Rochester
Rochester, NY, USA
cding@cs.rochester.edu

Abstract

As caches become larger and shared by an increasing number of cores, cache management is becoming more important. This paper explores collaborative caching, which uses software hints to influence hardware caching. Recent studies have shown that such collaboration between software and hardware can theoretically achieve optimal cache replacement on LRU-like cache.

This paper presents Pacman, a practical solution for collaborative caching in loop-based code. Pacman uses profiling to analyze patterns in an optimal caching policy in order to determine which data to cache and at what time. It then splits each loop into different parts at compile time. At run time, the loop boundary is adjusted to selectively store data that would be stored in an optimal policy. In this way, Pacman emulates the optimal policy wherever it can. Pacman requires a single bit at the load and store instructions. Some of the current hardware has partial support. This paper presents results using both simulated and real systems, and compares simulated results to related caching policies.

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

General Terms Algorithms, Performance, Theory

Keywords cache replacement policy, collaborative caching, optimal caching, priority cache hint

* The research is supported in part by the National Science Foundation (Contract No. CCF-1116104, CCF-0963759, CNS-0834566), IBM CAS Faculty Fellowship and a grant from Huawei.

† The work was done when Xiaoming Gu and Bin Bao were graduate students at the University of Rochester.

ISMM'13, June 20–21, 2013, Seattle, Washington, USA.

1. Introduction

There are two basic strategies to reduce the number of cache misses: locality optimization and cache management. In cases where locality optimization falls short (e.g. when loop tiling and loop fusion cannot reduce the working set size to fit in the cache), improved cache management can pick up the slack by storing as much of the active data as possible.

In this paper, we present program-assisted cache management (Pacman), a practical solution to approximate optimal cache management. It solves mainly two problems: at compile time, deciding how to best cache data, and at run time, communicating the decision to hardware.

To decide whether or not to cache data, we employ a comparison with OPT, the optimal caching policy. Under this policy, the stack distance of a block at any access (which we will call the OPT distance) represents the smallest cache size for which the access will be a cache hit [19].

We present profiling techniques that collect and identify patterns in the OPT distance for individual references over the program. Two training runs with different program inputs show linear patterns for some references that allow the inference of patterns for future runs with any input. The decision of whether to cache the data is then simple: If the OPT distance is less than the cache size, the data is cached. Otherwise, it is not.

In order to communicate this decision to the hardware, Pacman divides each loop into two parts at compile time: a high-locality part with short OPT distances, and a low-locality part with long OPT distances. At run time, the loop boundaries are adjusted based on the input and cache size, and high-locality accesses are cached, while low-locality accesses are not.

Pacman requires hardware support, in particular, a single bit at each instruction to control whether cache should store the accessed data. A limited form of such an interface is the non-temporal stores on Intel machines, which have recently been used to reduce string processing and memory zeroing time [21, 30]. A number of other systems have been built or proposed for software hints to influence hardware caching. Earlier examples include the placement hints on Intel Itanium [7], bypassing access on IBM Power series [23], the evict-me bit of [26]. Wang et al. called a combined software-hardware solution *collaborative caching* [26].

Past collaborative caching solutions have used the dependence distance [4] and the reuse distance [7, 26] to distinguish between

high and low-locality data and to cache the former more securely over the latter. However, in an optimal cache policy, low-locality data is not always thrown out (consider a program, e.g. streaming, that has *only* low locality data). A reuse distance based hint would mark all data as low locality and does not exploit the fact that a portion of it can fit in cache and benefit from cache reuse. Pacman conducts partial data caching by mimicking OPT management (as illustrated in Figure 4). In addition, unlike the previous solutions, the Pacman hints change with the cache and the input size (which are the two parameters to loop splitting).

Pacman optimizes loop code by performing different types of cache accesses based on the loop index. In this paper, we analyze and evaluate scientific loop kernels and benchmark programs. For loops in other types of programs, the Pacman framework can be applied without change, but the effect depends on whether Pacman can identify exploitable patterns for cache management. Another possible use of Pacman is to augment the implementation of the array abstraction in higher level languages (Section 4.5).

The rest of the paper first characterizes OPT caching and then describes the Pacman system that allocates cache among data within an array and across multiple arrays.

2. Pacman System Overview

Cache management is traditionally done online in hardware. Unfortunately, the optimal caching solution is impossible to compute online, because it requires knowledge of future memory accesses. Pacman circumvents this problem by computing the optimal solution at profiling time and then applying the solution at run time. Figure 1 shows an overview of the system, and the three phases are outlined as follows:

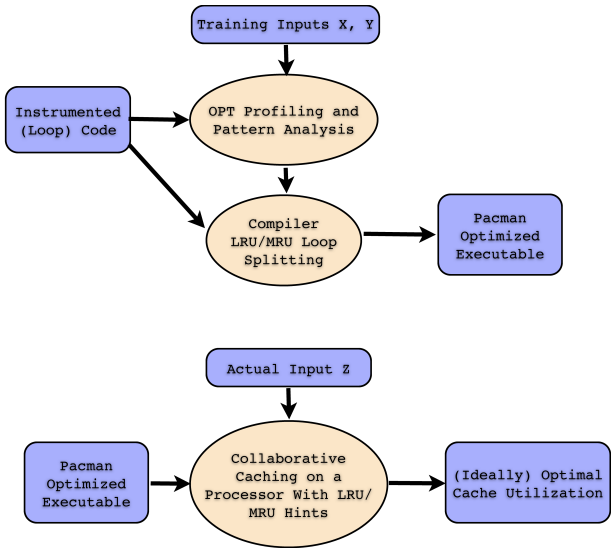


Figure 1. Overview of the Pacman system with OPT pattern profiling, LRU/MRU loop splitting, and collaborative caching

1. **Profiling Time** At a training run, the memory access trace is profiled to determine the OPT distance at each access. Patterns are then identified in these OPT distances. With multiple training runs using different input array sizes, patterns for other inputs can be extrapolated.
2. **Compile Time** Based on the patterns, a compiler performs loop splitting so that at run time, the loop iterations can be divided into groups where each static memory reference in the loop is tagged with a hint (for either LRU or MRU eviction).
3. **Run Time** At run time, based on the cache size and the input, a Pacman-optimized loop determines the actual division of loop splitting to guide the cache with the hints to approach optimal cache management.

3. Optimal Caching and the OPT Distance

The optimal (OPT) algorithm replaces the block that is accessed furthest in the future whenever a replacement was necessary. Another way to put it is that OPT evicts the *least imminently used* block, instead of the least recently used one that the *least recently used* (LRU) policy would. This is the basis of the optimal caching algorithm of [5].

The idea was expanded by [19] with the concept of the OPT stack, a priority list for what blocks should be the cache at any given time; A cache of size (1, 2, 3, ...) will contain the first (1, 2, 3, ...) blocks in the stack, thus an access to a data block is a miss when its stack position (OPT distance, or OPTD) is greater than the size of the cache. Figure 2 shows a memory access trace for a simple streaming program and demonstrates the application of three rules for managing the OPT cache stack:

- **Upward Movement:** *A block can only move up when accessed, and then it moves to the top.* Moving upward implies entering a cache (of size less than its current stack position), and this can only happen if the block is accessed. The block goes to the top of the OPT stack because it needs to be accessible by a cache of size 1.
- **Downward Movement:** *Vacancies are filled by whichever block above it will not be used for the longest time in the future.* A new block is treated as having vacated a new spot at the bottom (in order to usurp the top position). Moving a block downward to fill a vacancy represents its eviction from all caches smaller than the position of the vacancy (and the block with the most remote future use is evicted).
- **Tiebreaking:** *When there is a tie due to two infinite forward distances, it may be broken arbitrarily.*

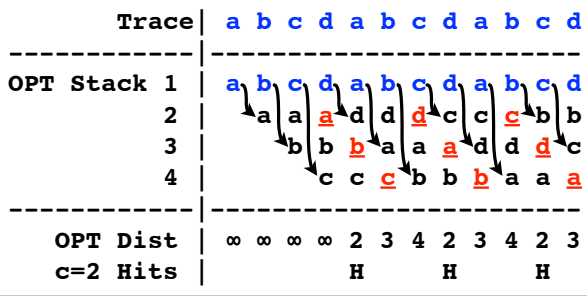


Figure 2. An example memory access trace for a simple streaming application. Because a block must always be loaded into the cache for the processor to use it, the top stack position is always given to the current memory block. When they are necessary, demotions are given to the block in the stack which will not be needed for the longest time in the future. For a cache size of 2, this program would cause cache thrashing under an LRU policy, but under the OPT policy, every third access is a hit once the cache is warm. This is a result of the $(2, 3, 4)$ pattern in the OPT distances which arises for streaming accesses, as explained the appendix. Demotions are shown with arrows, and the next access is shown in red with underline.

To demonstrate the practical difference between the optimal policy and the LRU policy, consider a simple streaming application which repeatedly traverses a 5MB array. Figure 3 compares

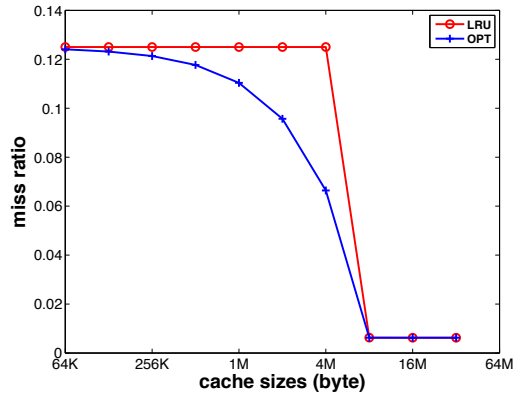


Figure 3. LRU & OPT miss ratios of a streaming application on power of 2 cache sizes from 64KB to 32MB (64-byte blocks, 16-way set associative). LRU does not benefit from a larger cache unless/until the working set fits in completely. OPT fully utilizes the available space to cache the working set as much as possible.

the cache performance of LRU and OPT for 16-way set associative cache with 64-byte cache blocks (8 data elements per block). The figure shows the miss ratio for all power-of-two cache sizes between 64KB and 32MB. When the cache size is less than 5MB, the LRU cache has a constant miss ratio of $\frac{1}{8} = 12.5\%$, but there is a sharp drop in the miss ratio at 5MB because the whole array then fits in the cache.

The OPT distances in this example vary from 2 cache blocks to roughly the data size. To show why this is the case, we demonstrate in Figure 2 how the OPT distances vary from 2 to roughly the data size in a smaller access trace. As the OPT distance varies between 128 bytes and 5MB in the streaming application, those accesses whose OPT distance is below the cache size will be cache hits. As the cache size grows, more accesses will be hits, so there is a more gradual reduction in the miss ratio.

Another way to view the performance discrepancy is that at the end of the array, LRU is acting on the assumption that the array will now be traversed backwards, while OPT knows that it will be traversed forwards again from the beginning.

4. OPT-based Hint Insertion

This section describes the theory, analysis, transformation, and two extensions of Pacman.

4.1 Optimal Collaborative Caching

Collaborative caching may obtain the performance of OPT on LRU like caches, as proved for several papers including the bypass and trespass LRU caches [10] and the LRU/MRU cache [11]. Instead of clairvoyance and complex cache replacement as required by OPT (described in Section 3), the collaborative cache uses two alternatives more amenable to hardware implementation:

- *1-bit hint*. A memory access is marked by a single bit to be either a normal (LRU) access or a special (MRU, e.g.) access.
- *LRU-like hardware*. The deviation from LRU involves only actions affecting the boundary positions of the stack (while OPT requires reordering in the middle of the stack) [11]. In fact, limited implementations are already available on existing hardware (see Section 5.6).

Pacman uses the LRU/MRU cache and the OPT distance for hint insertion. As an example, consider Pacman being given the data access trace in Figure 4. To make it interesting, the trace has mixed locality: 2 blocks `xy` are reused frequently, having high locality; while the other 7 blocks `abcdefg` (highlighted in red) have a streaming pattern and low locality. Pacman first computes the forward OPT distances, which are shown below the trace.

trace	xyaxybxycxydxyexyfxygxyaxybxycxydxye ...
fwd. optd	2342352362372382392342352362372382 39 ...
hint (c=5)	LLLLLLLLMLLMLLMLLMLLLLLLLMLLMLLMLLM ...

Figure 4. Since the non-xy accesses (highlighted in red) all have reuse distances greater than the cache size, a reuse distance based hint would cache none of them. A better solution is to cache some, but not all of them.

Suppose that the cache size is 5. For each access, Pacman inserts either an LRU or an MRU hint, shown in the table below the distances, by checking whether the distance is over 5. Being so hinted, the cache keeps `xy` and 2 of the streaming data in cache while leaving the others out, despite the fact that each of the streaming data have reuse distances larger than the cache size. In this way, the cache stores as much data in the cache as can benefit from it.

To quantify cache performance, the following table gives the stack distances for LRU (i.e. the reuse distance), MRU, OPT [19], and Pacman (i.e. the LRU/MRU distance) [11]. Capacity misses happen on accesses whose distance exceeds the cache size (miss iff $dis > c = 5$). The miss counts are 5 and 10 for LRU and MRU but just 3 for OPT and Pacman. Where Pacman accurately predicts the OPT distance, it gives the optimal cache hint. [1] OPT and Pacman distances differ, because the Pacman cache has LRU-like logic, while OPT does not.

trace	xyaxybxycxydxyexyfxygxyaxybxycxydxye	miss (c=5)
lru	---33-33-33-33-33-33-11911911911911919	5
mru	---23-34-45-56-67-78-89292434545656 7	10
opt	---23-23-23-23-23-23-23423523623723 8	3
pac.	---33-33-32-32-32-32-32533433632732 8	3

Figure 5. Comparison of caching policies, demonstrating that, with correct OPTD predictions, Pacman can match the optimal policy.

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 Pacman treat the two working sets separately and always have low distances for high-locality data and linearly increasing distances for low-locality data. The varying distances are effectively priorities through which these policies select program data to cache.

4.2 OPT Pattern Recognition

In an execution, a reference in a loop may make many accesses, each with a forward OPT distance. Pattern analysis uses training runs to find the correlation between the loop index and the OPT distance so the OPT distance can be predicted when the program is run for real.

OPT Forward Distance Profiling The profiling step records the forward OPT distance and the iteration count at all enclosing loops. Each reference-loop pair provides one sequence of `<index, distance>` pairs for the next step of pattern analysis. The profiling mechanism and cost are described in Section 4.4.

[1] In fact, it can be proved that the LRU/MRU cache will miss if and only if the OPT cache does (for the chosen cache size, 5 in this case). [11]

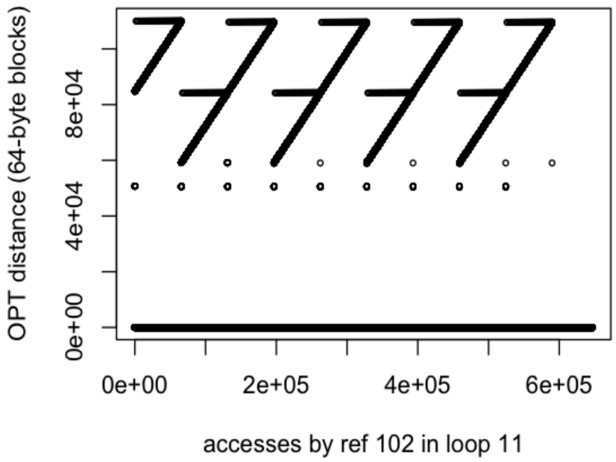


Figure 6. The OPT distances exhibited by a reference in *Swim*. The bounded pattern is between 5 and 10. The linear pattern has a slope of 38 cache blocks for every 100 iterations.

Linear and Bounded Patterns The OPT distances of a reference have mainly two types of patterns. The first is bounded length. For example, as a loop traverses a (8-byte) double precision array, it moves to a different (64-byte) cache block at every eighth access. The seven preceding accesses are spatial reuses. The length of these OPT distances are bounded. The eighth access, however, has an OPT distance that varies and can be as large as the size of the array. The first case is a *bounded pattern*, and the latter a *linear pattern*. We define the bounded pattern by the upper bound of the distances and the linear pattern by an intercept α and a slope β. An `<index, distance>` pair $< i, d >$ belongs to a linear pattern $< \alpha, \beta >$ if $d \approx \alpha + \beta i$.

As an example, Figure 6 plots the OPT distances for 647 thousand accesses by one of the 197 references in the SPEC 2000 benchmark program *Swim*. The 647 thousand distances are drawn as circles on a 2D plot, with the distance on the y-axis and the iteration count on the x-axis.

The bounded pattern is visible as the line at the bottom of the figure. The distances are between 5 and 10. The pattern contains about 7/8 (88%) of all points. The 5 diagonal lines (from lower left to upper right) show five linear patterns. The common slope (0.38) shows that the distance increases by 38 cache blocks for every 100 iterations. Although it is not visually apparent, the two patterns include most of the points on the plot. The points outside these two patterns account for less than 0.001% of the total (262 out of 646,944).

Grid Regression Most OPT distances are part of some linear pattern: 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 an array traversal. However, in real data the intermingling of multiple patterns (as in Figure 6) makes linear regression difficult. The problem can be solved by separating the data points into their pattern groups and then applying linear regression within each group. Based on this observation, we developed a technique we call *grid regression*.

```
algorithm grid_regression(set)

  tile_partition(set) #3 rows x 40 cols

  for each tile
    slope = regression(tile).slope
    intercept = regression(tile).intercept
    y_pred(x) = slope*x + intercept
    for each point
      if abs(y - y_pred) < 100
        add point to pattern

  combine_tile_patterns

end algorithm
```

Given a set of (x,y) coordinates for iteration index and OPT distance of the reference at that access, the data set is divided into 40 rows and 3 columns. Each tile is assigned a slope and intercept based on standard linear regression. Each point in each tile is then accepted as "in the pattern" if the OPT distance is within 100 of the regression line. If more than 90% of the points in a region are in the pattern, the tile is considered to have a pattern. Next, each pattern merges with each neighbor if the neighbor's smallest and largest points are within 100 of the first pattern's regression line. After as many patterns have merged as possible, each pattern's *accuracy* is defined as the percentage of points in the set that lie in the pattern.

An Example in Swim *Swim* is a SPEC 2000 benchmark, adapted from vectorized code for shallow water simulation. It has 197 references to 14 arrays in 22 loops. As an example profiling analysis, we run the program on the input 256 by 256. Each array is 524KB in size. The following table shows OPT distance patterns for a 2-level nested loop, with 18 references at the inner loop (Loop 3) and 6 more at the outer loop (Loop 4).

```
Loop 4: r257 to r262
  Loop 3: r251 to r256

ref loop  alloc intercept  accuracy/size
251   4   48.6%  534kb     96.2%
252   4   24.4%  1620kb    99.6%
253   4   48.6%  534kb     97.0%
254   4   24.4%  1613kb    99.2%
255   4   77.3%  186kb     22.2%
256   4   24.4%  1620kb    99.2%

257   4   48.9%  545kb     100.0%
258   4   32.3%  3789kb    100.0%
259   4   48.9%  540kb     100.0%
260   4   24.5%  1620kb    100.0%
261   4   48.8%  540kb     100.0%
262   4   24.5%  1628kb    100.0%
```

The multi-level grid regression found patterns (third to fifth columns) for these references (first column). The patterns show an elaborate size-dependent cache allocation. If the cache size is smaller than 186KB, all these references are MRU. Between 186KB and 540KB, *r255* in the inner loop is allocated in cache. From 540KB to 1.6MB, *r251* and *r253* in the inner loop and *r257, r259* and *r261* in the outer loop all take an equal share, 49%. These six references access only two arrays, with the inner-loop ones accessing the body and outer-loop ones the boundary of the arrays. As the cache size increases beyond 1.6MB, more referenced data is allocated into cache. Using *Swim* as a non-trivial example, we next discuss the issues in analyzing programs with more complex loop structures.

Nested Loops For each reference, all enclosing loops are analyzed for patterns. The loop whose pattern has the highest accuracy is considered *pattern carrying*. Its loop index and those of the inner loops are used for the OPT distance prediction. As an example, Figure 7 shows a reference in a two-level i, j loop that traverses a matrix. In the inner j loop, the iteration count ranges from 0 to n_j,

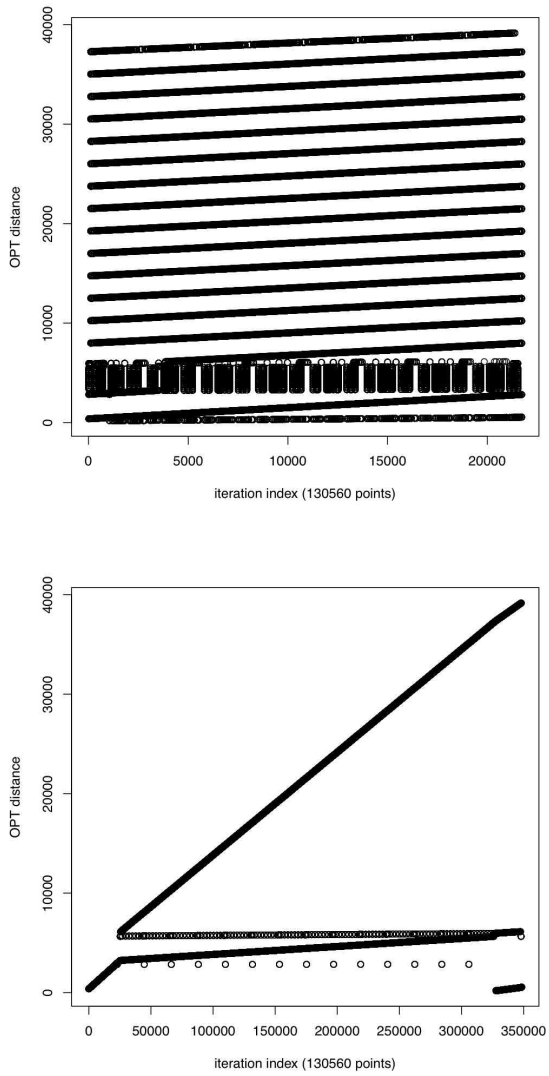


Figure 7. The OPT distance patterns of a reference in a nested loop. The x-axis shows the inner-loop iteration count in the upper graph and the cumulative iteration count in the lower graph. The outer loop (cumulative iteration count) is selected for Pacman to predict the OPT distance.

the number of j iterations. In the outer i loop, the iteration count is cumulative and ranges from 0 to $n_j n_i$. Grid regression identifies many linear patterns at the inner loop, one for each loop instance, but no majority pattern. At the j loop, it finds a single (linear) pattern. The patterns in *Swim* are all carried by the outer loop as in this example, as stated by the second column in the previous table.

Non-contiguous access, Non-perfect Nesting, Branches, and Indirection Linear patterns are fairly robust against code complexity. The previous *Swim* loop nest processes the boundary elements differently than the body elements outside the inner loop (`r257-r262`). These accesses form a regular linear pattern since their processing recurs at regular intervals. Similarly, if there is conditional processing, an array is accessed by multiple references in branches or switch cases. If some of those references access each element, then they will all have the same linear pattern. Still, the linear pattern may fail in two cases. The first is when an array is accessed only partially *and* at irregular intervals, e.g. through indirection. Linear regression would show an accuracy proportional to the degree of regularity. The second is when the same reference, through indirection, accesses two different arrays at different regular intervals. The analysis can detect multiple linear patterns, but each pattern is small in size and unsuitable for prediction.

Conservative Pattern Selection Pacman uses a pattern only if the accuracy is above a threshold, e.g. 90%. When the pattern accuracy is low, we may not know which access should be labeled MRU. To be conservative, we make them LRU, so at least not to perform worse than the original program. A compiler solution may remedy this. Beyls and D'Hollander developed conditional hints for loop references that have mixed reuse-distance patterns [7]. To use it here, we need compiler analysis of the OPT distance, which we leave as future work.

Whole-array and Priority-array Patterns Two characteristics in the *Swim* example are common in programs we have tested. First, the linear patterns occur at the whole-array level, e.g. the outer loop in *Swim*. The OPT distance grows with the array *size*, independent of the array *shape*. We call it the *whole-array pattern*. Second, the intercept represents a kind of stratification — lower priority arrays (because of the greater intercepts) are cached only after all higher priority arrays have been. We call it the *priority-array* pattern. These two properties are important in cross-input pattern analysis.

Cross-Input Pattern Prediction When the input changes, a reference may access the same array but with a different shape and size. The shape does not affect whole-array patterns. The size affects the intercept of priority-array patterns. The intercept changes proportionally to the size of the high priority arrays.

To predict the change in the intercept, Pacman uses multiple training runs. Each provides an `<intercept, input size>` pair. Then the pattern is established by standard linear regression. The minimum number of tests needed is two. Our test programs in Section 5 are mostly scientific simulations on some dense physical grid. We use the total grid size as the input size in the experiment. It can be automated by examining the loop bounds in training and using them to calculate the input size before executing the loop.

The predictability of loop bounds has been carefully studied by Mao and Shen. They gave a three-step solution — feature extraction, incremental modeling and discriminative prediction — and showed accurate prediction of the loop bounds in a set of Java program from just a few extracted parameters [17].

4.3 Loop Splitting and Hint Insertion

After Pacman finds a linear pattern for a reference, it uses a compiler to split the pattern-carrying loop. The following function finds the break point before which a reference should make LRU accesses and after which it should make MRU accesses. In the formula, p is a pattern with $intercept$ and $slope$. The two splitting parameters are the cache size c and the input size n.

```
function lru-mru-split( p, c, n )
   return (c - p.intercept * n) / p.slope
end
```

A loop may have k references that require loop splitting. Pacman produces at most k different breakpoints. The reference is LRU in the loops before its breakpoint and MRU in the loops after its breakpoint. Of the $k + 1$ loops, the first will have all LRU accesses, and the last will have MRU accesses.

```
for i in 1, n  // loop x
  body
end
```

is transformed to

```
b1,..,bk = lru-mru-split( p1,...,pk, n, c )
sort(b1,...,bk)
for i in 1, b1
   body-1
end
...
for i in bk-1, bk
  body-k
end
for i in bk, n
  body-k+1
end
```

Loop splitting increases the code size as the loop body is duplicated for every breakpoint. However, the degree of splitting is limited. First, not all references require splitting. It is required only for references that have a linear OPT distance pattern. Second, references with similar patterns can use the same breakpoint. For the example shown in Section 4.2, the loop nest from *Swim* has 18 references in the inner loop. Twelve have linear patterns and require loop splitting. Five of them have a similar pattern, so they can use the same split. The other 7 have 2 additional patterns, so this example would take 4 replicated loops. Third, loop code is compact and has good spatial locality. As the execution moves from one to the next, the active code size does not change. Forth, some of the loops may not be executed. In the *Swim* example, the largest pattern (of the 3) has an intercept of 3.8MB. At a smaller input size or a larger data size, the last replicated loop would have zero iteration. Finally, if a loop has a large body and requires a large number of splits, we may choose a subset of splits by weighing on the relative importance of the references. In evaluation, we will show that loop code tends not to incur misses because the instruction cache on modern machines is large.

4.4 OPT Forward Distance Profiling

OPT profiling (i.e. stack simulation [19]) takes a data access trace as the input and computes the OPT distance for each access. The distance is *backward* because it is computed at the end of each data reuse pair. Pacman converts it to a forward distance by recording it at the start of the data reuse pair. Since a program trace is too large to fit in main memory, Pacman dumps the access time and OPT distance pairs to a file. While the backward distance is ordered, the forward distance is not. Pacman uses the Unix sort command in a second pass to order the pairs by the access time in preparation for pattern analysis.

We can reduce the cost of storing the OPT distance traces by ignoring short distances if needed but by far the greatest cost in Pacman is the OPT simulation. For fastest OPT profiling, we use the tool by Sugumar and Abraham [25]. It implements the OPT stack using a splay tree for cache efficiency and a novel technique to update stack layout quickly. Profiling times for each workload are shown in Table 1.

Accompanying the OPT distance, Pacman records three types of program information: memory reference, loop, and function call. For memory profiling, we instrument every load and store. For function profiling, we insert profiling calls before and after every call site. Loop profiling captures 3 events: loop entry, loop exit, and a loop tick for every iteration.

The purpose of function profiling is to detect context-sensitive patterns in loops. In the following we discuss only loop pattern analysis, which is the same once the context is fixed. A function may be called inside a loop. There are two effects. First, the same function may be called by multiple loops. Pacman treats it in each loop context as a different function so not to mix their patterns. Second, a function may be executed multiple times in the same loop iteration. Pacman records these OPT distances as happening at the same time (measured by the iteration count).

4.5 Non-loop Pacman

We discuss an extension that takes advantage of the flexibility of program control: the support of non-loop code. In modern languages, most array operations are implemented in libraries such as the Vector class in Java and Array class in Ruby. Some of the functions traverse the entire array. These functions can take the LRU/MRU breakpoint as an optional parameter. The implementation would access the array using LRU access up to the breakpoint and MRU afterwards. A recent paper discusses the modern implementation of dynamically sized array [22]. Pacman may be added as another optimization technique to enable programmer control over the cache allocation for the array.

5. Evaluation

5.1 Experimental Setup

Workloads To evaluate Pacman, we simulate its use with 8 workloads. As the commonly used benchmark to measure memory bandwidth, *stream* repeatedly traverses a set of large arrays. Our version uses just one array. *SOR* implements Jacobi Successive Over-relaxation modified from SciMark 2.0 [1]. *Swim, mgrid*, and *applu* are from SPEC CPU 2000 [2] and *bwaves, leslie3d* and *zeusmp* from SPEC CPU 2006 [3]. For each workload, we have 3 different input sizes: small, medium, and large. Table 1 shows the number of references and loops and the size of their executions.

We chose these programs because their data size and running length can be adjusted by changing the input parameters. The training uses OPT, so Pacman cannot profile programs on overly large inputs. The testing of Pacman can be done on any size input. In order to compare with OPT, however, we choose small enough inputs in testing as well.

Pacman Implementation We implemented the profiling support a version of the LLVM compiler [15]. We did not implement loop splitting. Instead, all loop accesses are processed during simulation, and the splitting is done by the simulator. The simulated cache has 64-byte blocks, 16- way set associativity for cache sizes between 2MB and 16MB. The block size and associativity are based on the x86 machine we use for the simulation. Since we simulate only (data) cache, not the CPU, we cannot evaluate the looping overhead. We do not expect a significant overhead for the list of reasons given in Section 4.3. In Section 5.6, we will use a simple test to measure the complete effect of Pacman on a real system.

The OPT profiling times shown in Table 1 were measured on a 3 GHz AMD Opteron Processor 4284 with a 2 MB L3 cache.

Dynamic Insertion Policy (DIP) We compare Pacman with a hardware solution called Dynamic Insertion Policy (DIP) by Qureshi et al. [20]. DIP 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 (BIP), and (3) the majority of cache blocks which will 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

Workload	Mem. Ref's (Num.)	Loops (Num.)	Input Size	Trace Length (M)	Data Set Size (MB)	OPT Profile Time (s)
streaming (C)	1	2	small	5.2	2.1	11.7
			med.	10.5	4.2	18.1
			large	21.0	8.4	35.0
SOR (C)	7	5	small	10.7	2.1	10.8
			med.	42.8	8.4	35.3
			large	171.7	33.6	149.7
swim (F77)	341	33	small	42.5	7.6	50.8
			med.	95.5	16.9	151.7
			large	169.6	29.9	306.6
mgrid (F77)	418	46	small	5.1	0.2	5.7
			med.	39.5	1.1	37.9
			large	312.2	7.6	323.5
applu (F77)	1858	125	small	42.2	2.8	50.9
			med.	62.9	4.1	74.9
			large	163.1	10.3	213.2
bwaves (F77)	630	96	small	25.6	1.1	33.4
			med.	52.9	2.2	69.0
			large	389.6	16.6	631.4
leslie3d (F90)	3718	295	small	31.1	1.9	32.7
			med.	64.9	3.7	72.9
			large	147.6	6.9	171.4
zeusmp (F77)	10428	446	small	13.9	2.1	19.4
			med.	39.5	2.9	53.8
			large	85.5	4.2	117.6

Table 1. The statistics of the 8 workloads

instances even be outperformed by LRU because of misses incurred by BIP dedicated blocks.

For our comparisons, we used a DIP implementation of Xiang et al. [28], 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.

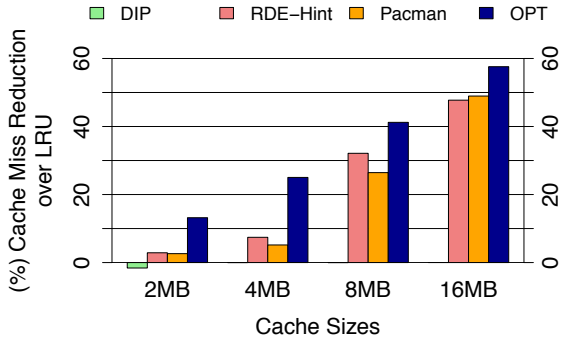
Estimated Reuse Distance Hint Policy (RDE-Hint) We compare Pacman with a software solution by Beyls and D'Hollander [7]. They proposed a reuse distance based hint. If over 90% of memory accesses generated by an instruction have reuse distances greater than the cache size, the instruction is given a cache hint of "MRU". Pacman works similarly, but using the OPT distance metric described in Section 3 instead of reuse distance.

The estimated reuse distance hint (*RDE-Hint*) policy is a modification of Pacman and an approximation of the Beyls and D'Hollander method. Figure 4 shows an example where the reuse distance for certain accesses (`abcdefg`) remains constant (and large), but the OPT distance forms a linear pattern peaking above the reuse distance. When an OPTD pattern is identified in the Pacman policy, these references will likely have large reuse distances, as demonstrated above, so they are tagged by the RDE-Hint policy for MRU eviction.

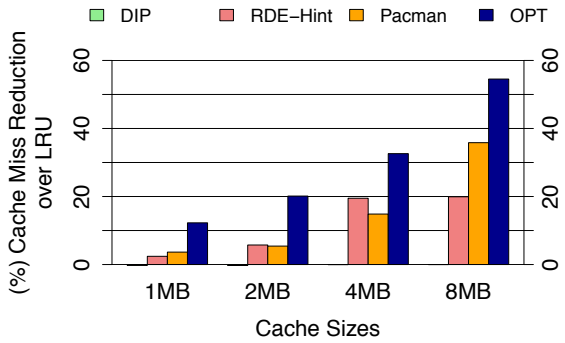
5.2 Pacman for Swim

Figure 8 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 relation between the data size and cache size and does not mean the predicted OPT distance is more accurate for the larger input.

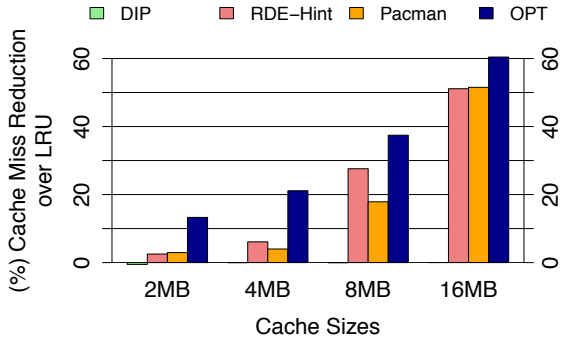
We use the total grid size as the data size for *swim*. Two grids may have different shapes but the same size. We have tested two



(a) Swim, training on input 256 by 256 and 384 by 384 and testing on 512 by 512



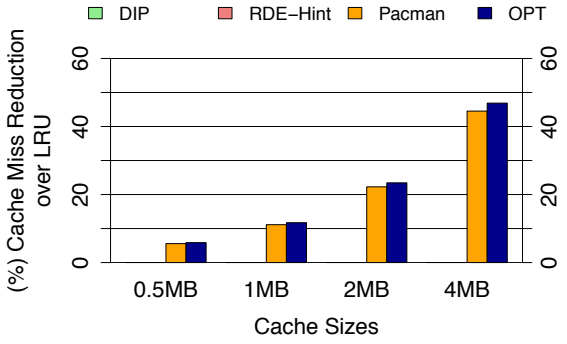
(b) Swim, training on input 384 by 384 and testing on 200 by 737



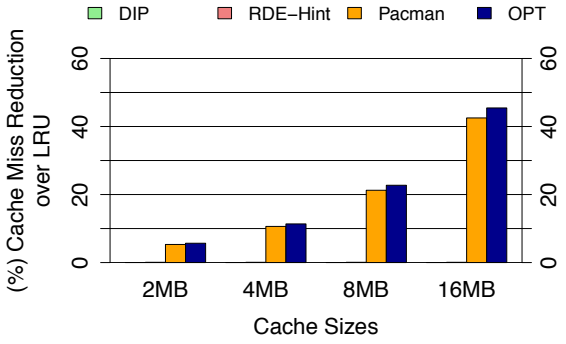
(c) Swim, training on input 256 by 256 and 384 by 384 and testing on 300 by 873

Figure 8. DIP, RDE-Hint, Pacman and OPT tested on *swim* when the input size (a), array shape (b) and both (c) change from training to testing.

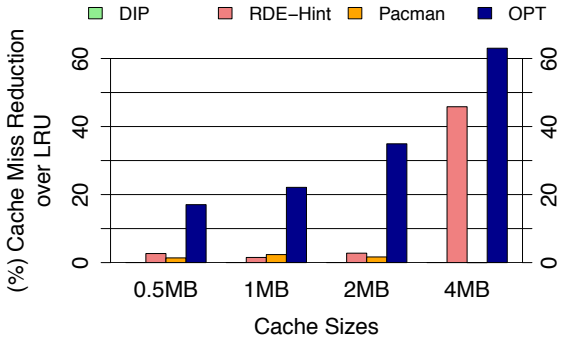
other inputs. The first is 200 by 737, which we choose to have the same total size as 384 by 384. Pacman predicts the same linear patterns for the two executions. Similarly, we choose 300 by 837 to have the same size of 512 by 512. We test Pacman by changing just the grid shape, just the input size, and both. The reduction numbers are shown in Figure 8. There is no significant loss of Pacman benefits as a result of these changes. It demonstrates the robustness of the Pacman pattern analysis and the cross-input prediction for this program.



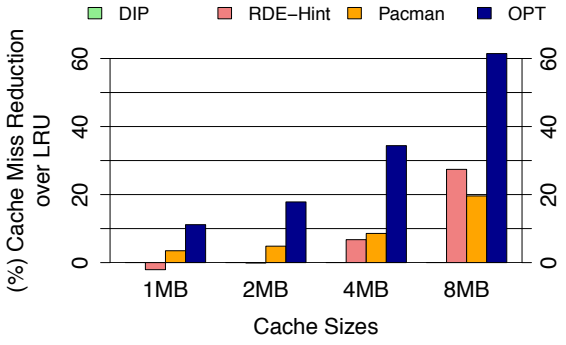
(a) Streaming, training on *SIZE*=256 and *SIZE*=512 and testing on *SIZE*=1024



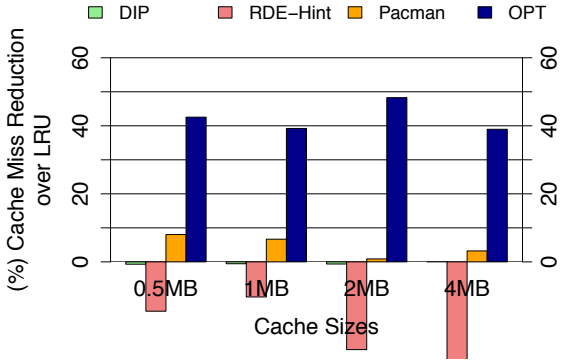
(b) SOR, training on *SIZE*=512 and 1024 and testing on *SIZE*=2048



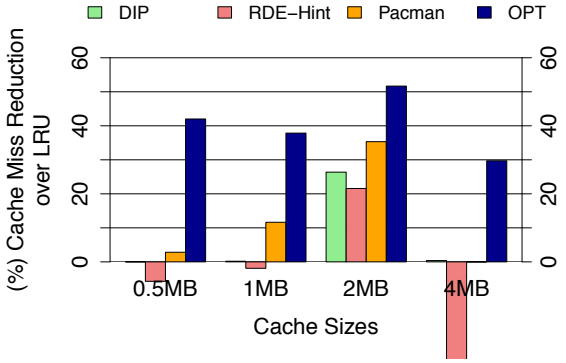
(c) Mgrid, training on *SIZE*=2^4 and *SIZE*=2^5 and testing on *SIZE*=2^6



(d) Applu, training on inputs 12 by 12 by 12 and 18 by 18 by 18 and testing on input 24 by 24 by 24



(e) Leslie3d, training on inputs 21 by 21 by 2 and 31 by 31 by 2 and testing on 41 by 41 by 3



(f) Zeusmp, training on inputs 8 by 8 by 8 and 12 by 12 by 12 and testing on 16 by 16 by 16

Figure 10. The improvements by DIP, RDE-Hint, Pacman and OPT over LRU. The input size is given by grid dimensions. The total size is used by the run-time predictor.

5.3 Other Programs

We show the results for 6 programs in Figure 10. Stream has the simplest pattern. Pacman obtains a performance close to optimal. In *SOR*, most computation happens in a 3-nested loop in Figure 11.

In the j loop, all 6 array references have group spatial reuse. Pacman profiling shows that the LLVM compiler generates 3 references for the loop. Two of the references have only short OPT distances. Only the third reference shows a linear pattern. Pacman recognizes it and reduces the miss ratios by 5%, 11%, 21% and 43% for the four cache sizes from 2MB to 16MB.

A more powerful compiler can apply loop tiling and reduce the size of the working set to fit entirely in cache. Pacman would find no linear pattern. On the other hand, there are iterative computations not amenable to tiling (e.g. k-means) and programs not amenable to compiler analysis. For those, Pacman provides the benefit of better cache management as in this example.

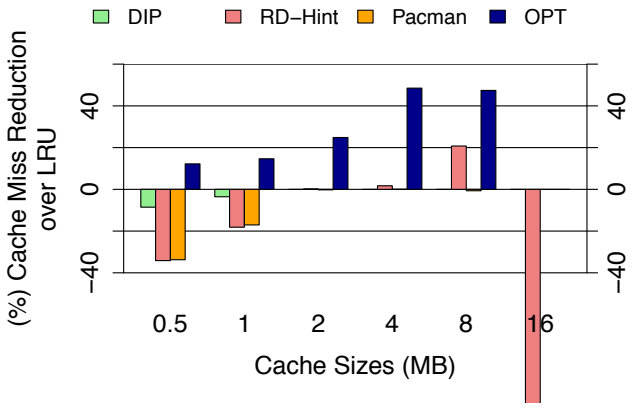


Figure 9. DIP, RDE-Hint, Pacman and OPT tested on *bwaves* (training on input 9 by 9 by 4 and 9 by 9 by 8 and testing on 17 by 17 by 16).

```
for (p=0; p<TIME; p++) {
  for (i=1; i<SIZE-1; i++) {
    for (j=1; j<SIZE-1; j++)
      G[i][j] = omega_over_four * (G[i-1][j] +
                G[i+1][j] + G[i][j-1] +
                G[i][j+1]) + one_minus_omega
                * G[i][j];
  }
}
```

Figure 11. The main loop in SOR, where Pacman inserts MRU hints for 3 of the references

The other four 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 PDEs 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.

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*. Many references in *applu* have a similar look as shown by an example one in Figure 7. *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 *mgraid*, from -6% to 10%.

There are a few cases of negative impacts on cache whose size is small. The reason is an error in cross-input analysis which predicts MRU for LRU access. For large set-associative cache, an overuse of MRU is not immediately harmful. The chance is that the following accesses would visit somewhere else in cache. For small cache, however, the incorrect MRU accesses may cause the eviction of high locality data and hence increase the miss rate.

Pacman obtains significant miss-ratio reductions over LRU, 11% to 24% for *leslie3d* and 0% to 26% for *zeusmp*. As members of SPEC2006, their code is larger. *Zeusmp* has as many as 10 thousand (static) references in nearly 500 loops.

Pacman does not improve the last program, *bwaves*, as shown in Figure 9. *Bwaves* has 260 references in 68 loops; most of the loops are nested 4 to 6 levels deep, and the input grid has just 3 dimensions. Although the performance is worse than LRU for caches of 0.5 MB and 1 MB, for larger caches, Pacman does no worse than LRU. Figure 12 shows the OPT distances for one of the 14 references in a 5-nested loop indexed by the combined iteration count from the outermost loop. In this and most other references, Pacman could not find a usable pattern because of the low accuracy (as low as 20% in the largest pattern).

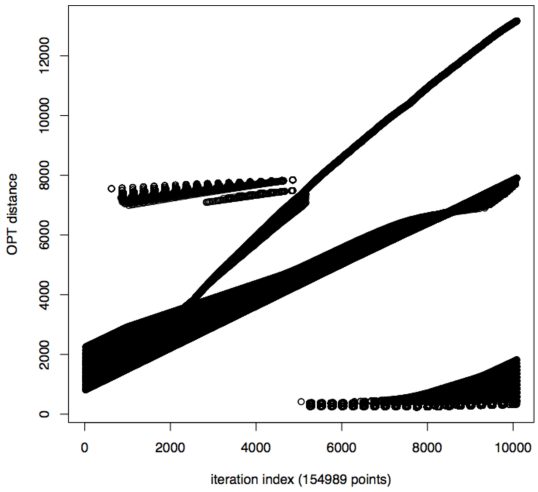


Figure 12. The OPT distances of a reference-loop pair in *bwaves*. Pacman finds multiple linear patterns but none is usable because of low accuracy.

5.4 Comparison to Dynamic Insertion Policy

In the SOR workload outlined above, DIP gives nearly the same miss rate 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 shown in Figure 10. 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 outperforms DIP in most of our trials. It has the advantage of gathering program information through profiling (where DIP uses on-line execution history). On the other hand, the mechanism differs. DIP applies a single policy across the board at any one point in program execution. The goal is to find the better solution of two policies. Pacman assigns an eviction policy to each access. The goal is the best solution of all policies.

To be fair, we limited our tuning of DIP to the parameters specified in that paper. 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 [20]. Nonetheless, this does demonstrate that, due to its fine-grained policy selection (as shown for the specific *Swim* loop in Section 4.2), Pacman has the edge for "inherently LRU" programs.

5.5 Comparison to Estimated Reuse Distance Hint Policy

RDE-Hint performed at or above the level of Pacman for some trials. There is only one trial (Mgrid, 4MB) where Beyls significantly outperforms Pacman.

The difference between RDE-Hint and Pacman is that whenever Pacman *might* provide an MRU hint (when there is a linear OPTD pattern), RDE-Hint *does*. In the example in Figure 4, RDE-Hint will provide MRU hints to the high reuse distance data `abcdefg`, just like the LRU policy. This similarity is shown in the results for the Streaming and SOR benchmarks, where RDE-Hint performs on

par with LRU. Like the OPT policy of Mattson [19], Pacman places a portion of the high reuse distance data in the cache. While RDE-Hint beats Pacman in some tests, the possibility of throwing out too much data makes it more volatile, as seen in the results for Leslie3d and Zeusmp

RDE-Hint can do worse than LRU because LRU evicts unused data, whereas RDE-Hint does not cache data in the first place if it is predicted to be unused. Overclassification of data as low locality then results in throwing out data that could have been used. For example, when there is both low and high-locality data as in Figure 5, LRU eviction is better than MRU eviction, but RDE-Hint will flag the low-locality data for MRU eviction. In contrast, Pacman will almost never do worse than LRU; since only a portion of the low-locality data is flagged for MRU, it is unlikely that too many accesses will be uncached.

5.6 Performance on Real Hardware

The x86 ISA provides non-temporal store instructions which can bypass cache. They write to memory without loading the corresponding cache line first. 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 [13]. There are also non-temporal prefetch instructions on x86, but they do not provide the full functionality of a non-temporal read. Still, we can evaluate the effect of Pacman using just the non-temporal store.

We run our tests on a machine with an Intel Xeon E5520 processor. The processor contains 4 symmetric 2.27GHz cores which share an 8MB L3 cache. With Hyper-Threading enabled, the processor can support up to 8 hardware threads.

Figure 13 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 with OpenMP. The size of array A is set to 12MB, which is too large for cache reuse under LRU.

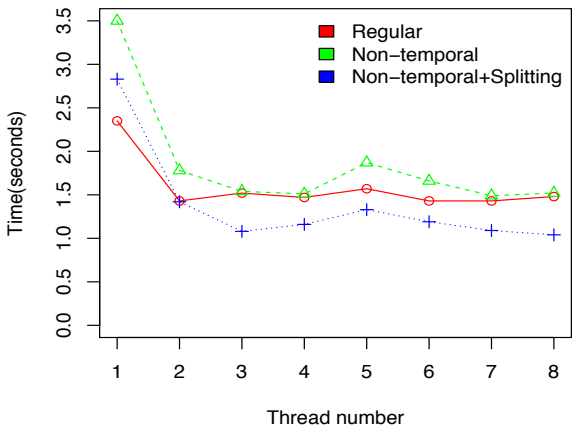
$$
\begin{array}{l}
for(t = 0; t < MAXITER; t++) \\
\#pragma\ omp\ parallel\ for \\
\quad for(i = 0; i < N; i++) \\
\quad\quad A[i] = foo(A[i]);
\end{array}
$$

Figure 13. An OpenMP loop nest: the inner loop updates the array element by element; the outer loop repeats each the inner loop at each time step.

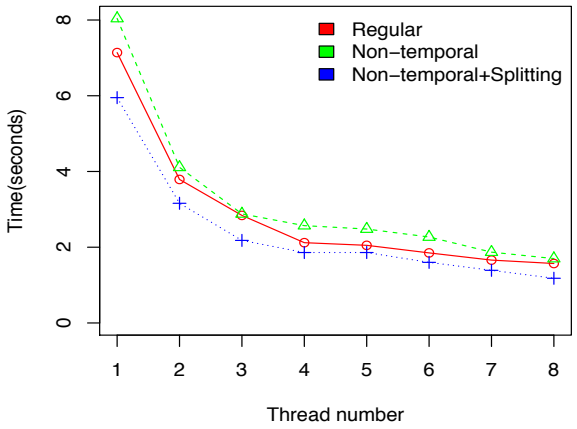
To enable the outer loop reuse, Pacman splits the inner loop into two, each of which is still an OpenMP parallel loop. The first loop writes to the first 8MB of array A using the normal (LRU) access, and the second loop the last 4MB of A using the non-temporal access. In the second loop, the non-temporal store is via the GCC intrinsic *_mm_stream_pd*. We unroll the inner loop 8 times, and the non-temporal stores happen once for each cache block rather than each element. To exclude the unrolling effect from the performance comparison, we perform the same loop unrolling on the original program. After loop splitting, Pacman keeps the first 8MB of A in the last level cache for reuse throughout the time steps.

Figure 14 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 in Figure 14 for comparison.

Figure 14(a) depicts the performance data with hardware prefetching on and in Figure 14(b) 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 [27]. We test our programs for up to 8 threads,



(a) Performance with hardware prefetching



(b) Performance without hardware prefetching

Figure 14. The performance comparison on Intel Xeon E5520.

threads 1-4 are assigned to four physical cores, and threads 5-8 are bound to four hyper-threads.

First let's consider single-thread performance. For the experiment, we turn on and off prefetching to measure its effect. With prefetching, the Pacman code is 19% slower than the original program when using 1 thread. Without prefetching, Pacman is 17% faster. The difference shows that for single-thread execution, prefetching is more effective than Pacman despite the fact that it improves the locality. The 19% slowdown suggests that non-temporal accesses either have a higher overhead or interfere with prefetching (which must be designed to maximize performance for sequential streaming access).

Prefetching requires sufficient memory bandwidth to be effective. In the single-thread test, one CPU core has the memory bandwidth of the entire system. Once the memory bandwidth is shared by more cores, cache management becomes important because better caching reduces the bandwidth demand and hence the contention on memory bandwidth. At more than 2 threads, Pacman is clearly better, as shown in Figure 14(a). At 3 and 8 threads, Pacman reduces the parallel execution time by over 30%. The results show the relation between cache management and prefetching. When there is enough bandwidth, prefetching is sufficient, at least for contiguous memory traversals; when the bandwidth is under con-

tention, caching is effective in further improving performance (by over 30% in this test).

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, the better cache management by Pacman is uniformly beneficial, from 17% improvement in one thread to 25% improvement at 8 threads, as shown in Figure 14(b).

We have tested the overhead of loop splitting by running the sequential version with one loop doing all the work and with eight replica loops each doing one eighth of the work. In addition, each loop is unrolled 8 times, so each loop body has 8 statements. The binary size is 7583 bytes before loop splitting and 8159 bytes after splitting. The running time is completely identical between the two versions. There is no visible run-time overhead from loop splitting. We also tested on an AMD Opteron machine and found no overhead by loop splitting. The reason is that the total code size, despite unrolling and splitting, is still small compared to the size of L1 instruction cache (32KB on Intel Xeon and 64KB on AMD Opteron). Furthermore, loop code has good instruction locality. As long as the size of a single loop does not exceed 32KB or 64KB, we do not expect repeated misses in the instruction cache.

On multicore machines, the memory bandwidth is shared among all co-run programs. By reducing the memory bandwidth demand of one program, Pacman improves not only the cache performance for the program but possibly its co-run peers because they can now use a greater share of the available memory bandwidth.

6. Related Work

Several previous solutions used reuse distance analysis, either by a compiler [7, 26] or by profiling [6, 7, 16], to distinguish between high locality and low locality accesses. The different caching mechanisms include Itanium placement hints [6, 7], evict-me tags [26] and cache partitioning through page coloring [16].

The reuse distance shows the program locality independent of cache management and size. When data larger than cache size have the same locality (reuse distance), it is unclear how to select a subset for caching. If we choose one group of accesses for LRU and the others for MRU, the cache locality changes for affected data but also for other data. This common problem when optimizing a tightly coupled system is described in a Chinese adage: "pulling one hair moves the whole body".

Reuse distance has been used to predict cross-input locality from the whole program to individual references (e.g. [8, 18]. Pacman uses the same training strategy but predicts the change in a linear pattern rather than in a histogram.

Optimal collaborative caching has been studied as a theoretical possibility [10–12]. Gu et al. proved the theoretical optimality in bypass and trespass LRU cache [10], the LRU-MRU cache [11], which we use in Pacman, and the priority LRU cache [12]. They designated each reference as LRU or MRU based on whether the majority of its OPT distances is greater than a threshold [11], following the technique of Beyls and D'Hollander [7]. A reference is always MRU or always LRU. The drawback is the same with the reuse distance: there is no partial caching of a working set. The optimality requires re-inserting hints for each different input and cache size. A recent solution allowed the same hints to optimize for caches of an arbitrary size but required passing integer-size priority hints rather than a single bit [12]. Pacman uses linear-pattern analysis and loop splitting to adapt cache hints across input and cache sizes. It addresses practical problems such as non-unit size cache blocks, nested loops, and cross-array and cross-program cache allocation.

Cache can implement adaptive solutions entirely in hardware with no visible overhead and with transparency to the user program. Indeed, perhaps no modern cache is implemented strictly as LRU. Techniques such as DIP [20] (compared in Section 5.4), recently reuse-time predictor [9] and many previous techniques improve over LRU by revising the LRU strategy or switching among multiple strategies. For memory management, elaborate strategies have been developed for paging, including EELRU [24], MRC [31], LIRS [14], and CRAMM [29]. While these previous policies are based on heuristics, Pacman is based on the optimal strategy, which it computes at the profiling time (to obtain program information and tolerate the OPT overhead). It inserts program hints so the hardware can obtain optimal management without needing program analysis or computing the optimal strategy. The empirical comparison shows the need for program-level control to allocate cache among differently for different groups of arrays and use different LRU/MRU policies within the same array. Finally, collaborative caching permits direct software control.

7. Summary

In this paper, we have presented the Pacman system for program-assisted cache management. It uses profiling to obtain the forward OPT distances, grid regression and cross-input analysis to identify linear patterns, and loop splitting to enable the dynamic designation of LRU/MRU accesses for each original data reference. Pacman needs the hardware to support LRU/MRU access interface. The interface requires at most one bit for each memory access. Most of the analysis and all program transformation are done off-line before a program executes.

By evaluating the system using simple and complex benchmark programs, we found that most programs exhibit strong linear patterns that can be captured by profiling. The reduction over LRU is significant and becomes as much as 40% to 60% when managing a large cache. Real-machine experiments suggest that MRU access incurs an overhead. Still, the improved cache reuse can improve performance when prefetching is difficult or when the contention on memory bandwidth is high. While Pacman is not the first caching policy to make use of program profiling, its unique contribution is in its use of the OPT distance for providing cache hints.

From these results, we believe that computer architects should consider supporting LRU/MRU hints to enable collaborative caching. The interface enables new types of cache memory management by a program or a compiler.

Acknowledgments

The authors would like to thank Yaoqing Gao, Xipeng Shen the anonymous ISMM reviewers, and our colleagues and visitors at the University of Rochester: Lingxiang Xiang, Xiaoya Xiang, Sandhya Dwarkadas, Engin Ipek, Jorge Albericio, and Li Shen for their insightful ideas, questions, and comments, and the use of code.

References

[1] SciMark2.0. `http://math.nist.gov/scimark2/`.

[2] SPEC CPU2000. `http://www.spec.org/cpu2000`.

[3] SPEC CPU2006. `http://www.spec.org/cpu2006`.

[4] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers, Oct. 2001.

[5] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

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

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

[8] C. Fang, S. Carr, S. Önder, and Z. Wang. Instruction based memory distance analysis and its application. In *Proceedings of PACT*, pages 27–37, 2005.

[9] M. Feng, C. Tian, C. Lin, and R. Gupta. Dynamic access distance driven cache replacement. *ACM Trans. on Arch. and Code Opt.*, 8(3):14, 2011.

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

[11] X. Gu and C. Ding. On the theory and potential of LRU-MRU collaborative cache management. In *Proceedings of ISMM*, pages 43–54, 2011.

[12] X. Gu and C. Ding. A generalized theory of collaborative caching. In *Proceedings of ISMM*, pages 109–120, 2012.

[13] A. Jha and D. Yee. Increasing memory throughput with intel streaming simd extensions 4 (intel sse4) streaming load, 2007. Intel Developer Zone.

[14] S. Jiang and X. Zhang. Making lru friendly to weak locality workloads: A novel replacement algorithm to improve buffer cache performance. *IEEE Trans. Computers*, 54(8):939–952, 2005.

[15] C. Lattner and V. S. Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *Proceedings of PLDI*, pages 129–142, 2005.

[16] Q. Lu, J. Lin, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Soft-OLP: Improving hardware cache performance through software-controlled object-level partitioning. In *Proceedings of PACT*, pages 246–257, 2009.

[17] F. Mao and X. Shen. Cross-input learning and discriminative prediction in evolvable virtual machines. In *Proceedings of CGO*, pages 92–101, 2009.

[18] G. Marin and J. Mellor-Crummey. Cross architecture performance predictions for scientific applications using parameterized models. In *Proceedings of SIGMETRICS*, pages 2–13, 2004.

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

[20] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. S. Jr., and J. S. Emer. Adaptive insertion policies for high performance caching. In *Proceedings of ISCA*, pages 381–391, 2007.

[21] S. Rus, R. Ashok, and D. X. Li. Automated locality optimization based on the reuse distance of string operations. In *Proceedings of CGO*, pages 181–190, 2011.

[22] J. B. Sartor, S. M. Blackburn, D. Frampton, M. Hirzel, and K. S. McKinley. Z-rays: divide arrays and conquer speed and flexibility. In *Proceedings of PLDI*, pages 471–482, 2010.

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

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

[25] R. A. Sugumar and S. G. Abraham. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *Proceedings of SIGMETRICS*, Santa Clara, CA, May 1993.

[26] Z. Wang, K. S. McKinley, A. L.Rosenberg, and C. C. Weems. Using the compiler to improve cache replacement decisions. In *Proceedings of PACT*, Charlottesville, Virginia, 2002.

[27] C.-J. Wu and M. Martonosi. Characterization and dynamic mitigation of intra-application cache interference. In *Proceedings of ISPASS*, pages 2–11, 2011.

[28] L. Xiang, T. Chen, Q. Shi, and W. Hu. Less reused filter: improving L2 cache performance via filtering less reused lines. In *Proceedings of ICS*, pages 68–79, New York, NY, USA, 2009. ACM.

[29] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. CRAMM: Virtual memory support for garbage-collected applications. In *Proceedings of OSDI*, pages 103–116, 2006.

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

[31] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. In *Proceedings of ASPLOS*, pages 177–188, 2004.

A. Two Properties of OPT Distance

OPT distance is a foundational concept in the paper. We present two of its theoretical properties to aid its understanding. First, we show a general relation with the reuse distance.

PROPOSITION 1. *At each access, the OPT distance is no more than the reuse distance.*

Proof Without loss of generality, consider an access of the data element x and the reuse window from the last access to the current access of x. At the start of the window, x is just accessed, so it is at the top position in both the LRU and the OPT stacks. Next we compare the movement of x in these two stacks.

The movement is based on priority. LRU ranks data by the last access time. OPT ranks by the next access time. As the other data are accessed, they come to the top of the stack. In LRU, they always gain a higher priority than x, so they stay over x in the LRU stack. In OPT, depending on the next access time, they may rank lower than x and drop below x. As a result, x always stays at the same or a higher stack position in OPT than in LRU until the end of the reuse window. At the end, the OPT distance is smaller than or equal to the reuse distance.

The next proposition states the exact OPT distance formula for repeated streaming accesses. We have seen this pattern in a specific example in Section 3. Here we prove a general result.

PROPOSITION 2. *When repeatedly traversing n data $a_1 \dots a_n$, the OPT distance is ∞ for the first n accesses and then repeats from $2 \dots n$ until the end of the trace.*

Proof It is trivial to show the pattern if $n = 2$. Next we assume $n > 2$. At the nth position, a_n is accessed. All n blocks are in cache. The OPT stack, from top to bottom, is $a_n, a_1 \dots a_{n-1}$. Except for a_n, the elements are ordered by the next access time. Now we examine the OPT distance and the stack layout in the following $n-1$ accesses: $a_1, \dots, a_{n-1}$. At a_1, the OPT distance is 2, and the top two stack entries are changed to a_1, a_n. At a_2, the OPT distance is 3, and the top 3 spots are changed to a_2, a_n, a_1. The pattern continues. At a_{n-1}, the OPT distance is n, and the stack is $a_{n-1}, a_n, a_1, \dots, a_{n-2}$. Now, the next $n-1$ accesses in the trace are $a_n, a_1, \dots, a_{n-2}$. Comparing to the last $n-1$ accesses, we see the identical configuration of the stack and the upcoming $n-1$ accesses, if we re-number data blocks from $a_n, a_1, \dots, a_{n-1}$ to $a_1, \dots, a_n$. The same reasoning applies, and hence the pattern of OPT distances repeats.

We note that the periodicity of the OPT distances is $n-1$. As a result, the same datum does not have the same OPT distance over time. Its OPT distance increases by 1 each time the datum is reused in cache. For any cache size, every datum will alternate to stay in and out of the OPT cache. The optimal caching is not by a preference of a data subset but a rotation of all data over time. In fact, it can be shown that choosing a preferred subset is an inferior solution. If we phrase this theoretical result in political terms, we have a proof that democracy outperforms aristocracy in data caching.