Inter-Reference Gap Distribution Replacement: An Improved Replacement Algorithm for Set-Associative Caches

Masamichi Takagi
Dept. of Computer Science, University of Tokyo
takagi-m@is.s.u-tokyo.ac.jp

Kei Hiraki
Dept. of Computer Science, University of Tokyo
hiraki@is.s.u-tokyo.ac.jp

ABSTRACT

We propose a novel replacement algorithm, called Inter-Reference Gap Distribution Replacement (IGDR), for set-associative secondary caches of processors. IGDR attaches a weight to each memory-block, and on a replacement request it selects the memory-block with the smallest weight for eviction. The time difference between successive references of a memory-block is called its Inter-Reference Gap (IRG). IGDR estimates the ideal weight of a memory-block by using the reciprocal of its IRG. To estimate this reciprocal, it is assumed that each memory-block has its own probability distribution of IRGs; from which IGDR calculates the expected value of the reciprocal of the IRG to use as the weight of the memory-block. For implementation, IGDR does not have the probability distribution; instead it records the IRG distribution statistics at run-time. IGDR classifies memory-blocks and records statistics for each class. It is shown that the IRG distributions of memory-blocks correlate their reference counts, this enables classifying memory-blocks by their reference counts. IGDR is evaluated through an execution-driven simulation. For ten of the SPEC CPU2000 programs, IGDR achieves up to 46.1% (on average 19.8%) miss reduction and up to 48.9% (on average 12.9%) speedup, over the LRU algorithm.

Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Design Styles—*Cache memories*

General Terms

Algorithms, Design, Management, Performance

Keywords

Cache memory, replacement algorithm, set-associative cache

ICS'04, June 26–July 1, 2004, Malo, France.

1. INTRODUCTION

Recently, the speed-gap between processor and DRAM has widened [19]. As a cache-miss can cause a stall time of hundreds of processor cycles, it has become imperative to reduce cache misses for speeding up computer systems. A high performance cache replacement algorithm is one method to reduce cache-misses. Many studies have proposed replacement algorithms for virtual memory paging, database buffer caches [11, 6], system buffer caches [8, 3, 9, 20, 5, 10, 13], and processor caches [12]. A replacement algorithm attaches a weight to each memory-block; on a replacement request, it selects the memory-block with the minimum weight. Assume that the time at which a memory-block will next be referenced is known; then the optimal algorithm with regard to miss count is to replace the memory-block that has the largest time difference between the next reference time and the current time. This time difference is called forward distance (FD). The ideal weight of a memory-block is a monotonically decreasing function of its FD. Practical replacement algorithms estimate this ideal weight of a memory-block to use as its weight. The accuracy of the weight estimation method determines the performance of a replacement algorithm.

The LRU algorithm [4] attaches a greater weight to a memory-block that has smaller elapsed time since its last reference. The LRU algorithm utilizes only a single history of a memory-block, that is the elapsed time; and estimates the ideal weight using this history under the assumption that this history reflects future behavior. Often this assumption does not hold in practice; therefore, the weight estimation accuracy of the LRU algorithm is poor [11]. The LFU algorithm [1, 4] attaches a greater weight to a memory-block that has a larger number of references. The accuracy of this method is poor for memory-blocks whose behavior varies significantly over time. For example, a poor estimate is generated for the weight of a memory-block which was referenced many times in a short period of time initially but will not be referenced again [13]. This case occurs frequently in practice; therefore the weight estimation accuracy of the LFU algorithm is poor.

In this paper, we propose a novel replacement algorithm based on an estimation method which is more accurate than the LRU and LFU algorithms. We call our method Inter-Reference Gap Distribution Replacement (IGDR). The time difference between successive references to a memory-block is called its Inter-Reference Gap (IRG). **Figure 1** gives an overview of the IGDR. IGDR estimates the ideal weight of

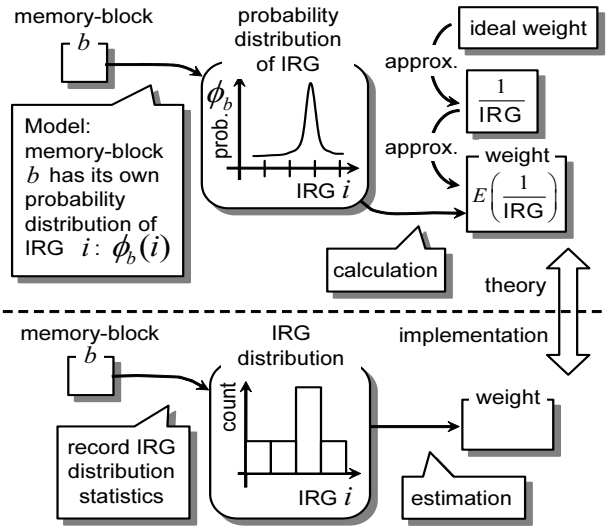

Figure 1: Overview of IGDR weight estimation

a memory-block using the reciprocal of its IRG. To estimate this reciprocal, IGDR assumes that each memory-block has its own probability distribution of IRGs. Using this probability distribution, the expected value of the reciprocal of the IRG is calculated. IGDR uses this expected value as the weight of the memory-block. In the implementation, IGDR estimates the weight using the IRG distribution statistics instead of the probability distribution of the IRG. IGDR classifies memory-blocks into several classes, and takes statistics for each class.

The organization of this paper is as follows: Section 2 presents the definitions of terms and related work; in section 3, our replacement algorithm is explained; Section 4 details the implementation; In Section 5 the evaluation is presented; and Section 6 summarizes this paper.

2. DEFINITIONS AND RELATED WORK

2.1 Definitions of Terms

A segment of data in main memory that is as large as a cache-block is called a **memory-block**. The set of all memory-blocks that a program references is expressed as $B = \{b_1, b_2, \cdots, b_n\}$. A counter is used and called the **virtual time**, which is zero at the beginning of the program and increases by one on each reference. The units of the virtual time are called **ticks**. $r_i = b$ indicates that a memory-block b is referenced at the virtual time i.

DEFINITION 1 (**Forward Distance (FD)** $f_b(t)$). *The forward distance $f_b(t)$ of memory-block b at virtual time t is the time difference between the next reference time and t:*

$$f_b(t) = \begin{cases} x - t, & \text{if } r_x = b \text{ for } {}^{\exists}x > t \\ & \text{and } r_y \neq b \text{ for } t < {}^{\forall}y < x \\ \infty, & \text{if } r_x \neq b \text{ for } {}^{\forall}x > t \end{cases}$$

DEFINITION 2 (**Backward Distance (BD)** $d_b(t)$). *The backward distance $d_b(t)$ of memory-block b at virtual time t is the time difference between the last reference time and t:*

$$d_b(t) = \begin{cases} 0, & \text{if } r_t = b \\ t - x, & \text{if } r_x = b \text{ for } {}^{\exists}x < t \\ & \text{and } r_y \neq b \text{ for } x < {}^{\forall}y \leq t \\ \infty, & \text{if } r_x \neq b \text{ for } {}^{\forall}x \leq t \end{cases}$$

Inter-Reference Gap (IRG) of a memory-block is defined as the virtual time difference between successive references to the memory-block. Assume a memory-block b is referenced n times at virtual times $t_1, t_2, \cdots, t_n$. From a viewpoint of a certain virtual time t, (1) **the last IRG** is defined as the last observed IRG and (2) **the next IRG** is defined as $d_b(t) + f_b(t)$; and is defined as a plus infinity when $t \geq t_n$.

When we know $f_b(t)$, replacing the memory-block with the largest $f_b(t)$ is optimal with regard to the miss count [2]; and this algorithm is called MIN. Therefore the ideal weight of a memory-block is a monotonically decreasing function of $f_b(t)$; however, evaluation of $f_b(t)$ requires knowledge of future events; thereby replacement algorithms estimate the ideal weight of a memory-block, and use this estimated value as its weight.

2.2 Related Work

Utilizing History Information Phalke et al. proposed a method that predicts the IRG of a memory-block using the Markov model [12]. It predicts the FD using the predicted IRG, and replaces the memory-block with the largest predicted FD. Their method records a long IRG history for each memory-block that is referenced. O'Neil et al. proposed the LRU-2 method [11] that evicts the memory-block with the minimum timestamp of the second to last reference. This timestamp includes the single history of IRG and the single history of BD for the memory-block. Jiang et al. proposed the LIRS method [5], which classifies memory-blocks referenced recently into two classes using a single history of IRG. LIRS classifies memory-blocks with reference counts of one or large IRGs into the HIR class; and memory-blocks with small IRGs into the LIR class. The LIRS method gives priority for eviction to the HIR class. Megiddo et al. proposed the ARC method [10], which classifies and puts memory-blocks referenced recently into the L_1 and L_2 stacks using reference counts. ARC automatically tunes the number of memory-blocks that are allowed to be cached for each stack. Basically, the replacement is performed using the LRU algorithm. These methods record reference history of memory-blocks to predict the future behavior. Phalke's method records too long histories to implement. The histories of other methods are often too short and thereby their weight estimation accuracy is poor. In contrast, IGDR manages longer but implementable reference histories, providing improved accuracy.

Combination of the LRU and LFU Replacement Algorithms Robinson et al. proposed the FBR method [13], which manages memory-blocks in the cache using a stack, and divides the stack into three regions, which are *new*, *middle*, and *old*, according to reference recency. FBR records the reference counts and replaces the memory-block with the smallest reference count and in the *old* region. FBR does not increase reference counts of memory-blocks in the *new* region, so that multiple references to these memory-blocks in a short period of time do not promote them more than necessary. Zhou et al. proposed the MQ method [20], which manages memory-blocks in the cache using a chain of queues. These queues hold the memory-blocks in accord with their reference counts. Memory-blocks in the queue chain move periodically toward the queue for smaller reference counts. MQ replaces the memory-block at the head of the queue chain. Lee et al. proposed the LRFU method

[9], which increases the weight of a memory-block by one when it is referenced, and decays the weights of all memory-blocks according to their BDs. These methods all combine the LRU and LFU algorithms using heuristics and are not based on reference models. In contrast, IGDR is based on a reference model and adapts to reference patterns that prefer LRU or LFU.

Detection and Adaptation of Reference Patterns Johnson et al. proposed the 2Q method [6], which puts memory-blocks with reference counts of one into the A_1 queue and puts memory-blocks with reference counts of greater than one into the A_m stack. 2Q exploits the empirical rule that the memory-blocks with a reference count of one often have extremely large IRGs; and gives a priority for eviction to memory-blocks in A_1. Kim et al. proposed the UBM method [8], which detects (1) references that scan a contiguous region and cause no reuse and (2) references that scan a contiguous region periodically. UBM applies the appropriate replacement algorithms to memory-blocks according to the detected patterns. Choi et al. proposed the AFC method [3], which is similar to UBM and detects four patterns. These methods detect and adapt to reference patterns in a case-by-case manner; this solution complicates implementation and cannot adapt to unexpected reference patterns. In contrast, IGDR is a unified and robust method because of its underlying model.

3. INTER-REFERENCE GAP DISTRIBUTION REPLACEMENT

IGDR approximates the ideal weight of a memory-block using the reciprocal of its IRG. This estimated value is used as the weight of the memory-block. On a replacement request, it selects the memory-block with the smallest weight for eviction.

3.1 Weight of a Memory-Block

The ideal weight of a memory-block is a monotonically decreasing function of its FD; IGDR estimates it using the reciprocal of its IRG. IRG is utilized because the IRG distribution of a memory-block is highly skewed and the IRGs of a memory-block take a small number of distinct values [12, 17]; these characteristics facilitate IRG estimation. The reciprocal is utilized because the calculation is simple and an extremely large IRG can be treated as zero. To estimate the reciprocal of the IRG, we propose the following reference model.

Independent Inter-Reference Gap Distribution (IIGD) Each memory-block b has its own probability distribution of IRG. This probability distribution is independent of other memory-blocks and the virtual time. It is modeled as a function of the IRG i and called $\phi_b(i)$.

Assume that each memory-block follows this model. The expected value of the reciprocal of a memory-block's IRG can be calculated from the IRG probability distribution of the memory-block. This expected value is used as the weight of the memory-block.

$\psi_b(i|\delta)$ is defined as follows, which is expected to represent the probability of $f_b(t) = i - \delta$ at the virtual time t with $d_b(t) = \delta$.

$$\psi_b(i|\delta) = \frac{\phi_b(i)}{\sum_{j=\delta+1}^{\infty} \phi_b(j)} \tag{1}$$

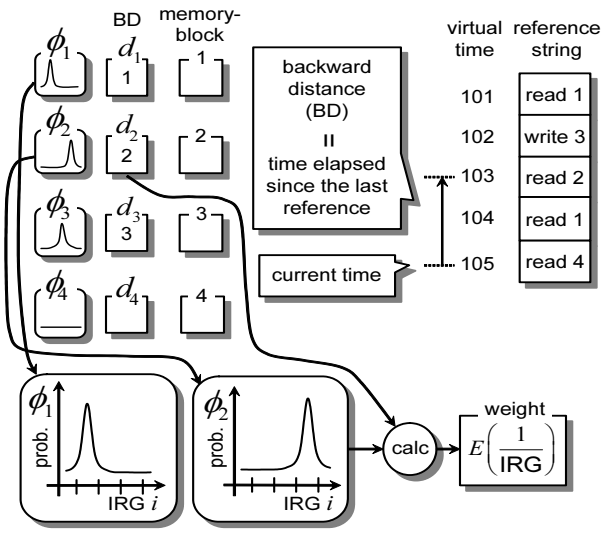

Figure 2: Weight calculation of a memory-block

The weight of b at virtual time t is defined as the expected value of the reciprocal of the next IRG, which is expressed as a function of $\delta = d_b(t)$ as:

$$w_b(\delta) = \sum_{i=\delta+1}^{\infty} \psi_b(i|\delta)\frac{1}{i} \tag{2}$$

$i = \delta$ is excluded from the summation of Equation (2) because it is assumed to be used for unreferenced memory-blocks at that time. From Equations (1) and (2), $w_b(\delta)$ is expressed as follows.

$$w_b(\delta) = \frac{\sum_{i=\delta+1}^{\infty} \phi_b(i)\frac{1}{i}}{\sum_{j=\delta+1}^{\infty} \phi_b(j)} \tag{3}$$

Figure 2 illustrates the memory-block weight calculation. Four memory-blocks are shown and they have their own probability distribution of IRG, i.e. $\phi_1(i)$, $\phi_2(i)$, $\phi_3(i)$, and $\phi_4(i)$. The current virtual time t is 105, and backward distance is associated to each memory-block; e.g. memory-block 2 was last referenced at virtual time 103; thereby $d_2(t)$ is 2 ticks. The weight of a memory-block b at the virtual time t is calculated using $\phi_b(i)$ and $d_b(t)$. For example, the weight of memory-block 2 is calculated using $\phi_2(i)$ and $d_2(t)$.

Equation (3) requires the probability distribution of the IRGs of a memory-block. Because the probability distribution is not given, the implementation records the IRG distribution statistics and estimates the weight of a memory-block using statistics. Recording the distribution statistics per memory-block (1) requires an enormous storage and (2) has the problem that the speed of the recording is slow [17]; therefore IGDR classifies memory-blocks into several classes and records statistics per class. Then the weight of a memory-block is estimated per class. Thus, the concept of memory-block weight per class is introduced.

3.2 Weight of a Memory-Block Class

The set of memory-blocks B is divided into m disjoint subsets, namely $B = B_1 \oplus B_2 \oplus \cdots B_m$; each subset corresponds to one class. $\phi_{B_k}(i)$ is defined as the probability that a reference of a memory-block in the class B_k occurs with the

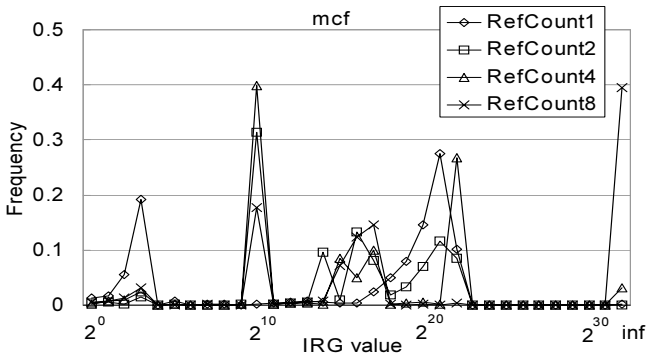

Figure 3: IRG distribution of program `mcf`

IRG i. $w_{B_k}(\delta)$ is defined as the weight of a memory-block in the class B_k, in accord with Equation (3).

$$w_{B_k}(\delta) = \frac{\sum_{i=\delta+1}^{\infty} \phi_{B_k}(i)\frac{1}{i}}{\sum_{j=\delta+1}^{\infty} \phi_{B_k}(j)} \quad (4)$$

3.3 Weight Estimation using IRG Distribution Statistics

IGDR estimates the value of Equation (4) using IRG distribution statistics per class. For memory-blocks belonging to B_k, IGDR records the number of references where their IRGs are i ticks in $\mathcal{D}_{B_k}(i)$. IGDR estimates $\phi_{B_k}(i)$ as follows.

$$\phi_{B_k}(i) \approx \frac{\mathcal{D}_{B_k}(i)}{\sum_{j=1}^{\infty} \mathcal{D}_{B_k}(j)} \quad (5)$$

w_{B_k} is approximated using Equation (5) and (4) as follows.

$$w_{B_k}(\delta) \approx \frac{\sum_{i=\delta+1}^{\infty} \mathcal{D}_{B_k}(i)\frac{1}{i}}{\sum_{j=\delta+1}^{\infty} \mathcal{D}_{B_k}(j)} \quad (6)$$

3.4 Classification of Memory-Blocks

IGDR classifies memory-blocks by exploiting the characteristics of IRG. First, the IRG distributions of memory-blocks correlate their reference counts [17]. As an example, **Figure 3** shows a linear-log plot of the IRG value distribution for a program `mcf` (See Section 5 for details). The X-axis shows IRG values; the rightmost position is plus infinity; and the Y-axis shows frequency. Points labeled "RefCount$n$" show the next IRGs of memory-blocks with reference counts n. It is observed that (a) memory-blocks have frequency peaks at certain IRG values, (b) memory-blocks with different reference counts have different IRG values for their peaks. Therefore, memory-blocks can be classified according to their reference counts. Second, memory-blocks which are referenced in loops often have stable IRGs; therefore memory-blocks can be classified using their IRG history.

Integrating these two classifying ideas, (1) IGDR classifies memory-blocks into those having stable IRGs and those which do not, and (2) classifies the memory-blocks in the latter group according to their reference counts. Memory-blocks are classified into the following five classes as a result. Listed first are the names of the classes. T_{RC}, T_{irg}, and T_{SC} are constant parameters.

OT Memory-blocks with a reference count of one.

TT Memory-blocks with a reference count of two.

ST Memory-blocks with a reference count greater than two.

MT Memory-blocks with a reference count greater than T_{RC}.

PS The IRG of a memory-block is considered to be stable when the difference between two consecutive IRGs is less than T_{irg}. When this stable situation occurs consecutively T_{SC} times, the memory-block is classified into PS.

The classes are listed in increasing order of priority, in the sense that if a memory-block meets multiple class conditions then it is classified into the class which is listed lower. A memory-block might move from one class to another when it is referenced and its reference count and its last IRG change. In this case, the IRG is recorded using its old class.

3.5 Adaptation to Various Reference Patterns

This subsection describes several reference patterns generated by programs and how IGDR adapts to each of them.

One Touch One Touch is defined as a pattern where a memory-block has an extremely large IRG after it is referenced. IGDR classifies this memory-block into the OT class. The $\phi_b(\delta)$ of One Touch memory-blocks have a peak near $\delta{=}\infty$; thereby $w_{\mathrm{OT}}(\delta)$ is small. As a result, IGDR gives priority for eviction to One Touch memory-blocks. Every memory-block is classified into OT the first time it is referenced; however, One Touch memory-blocks often occupy a large fraction in OT when they appear.

LRU Stack Depth Distribution LRU Stack Depth $s_b(t)$ of memory-block b at virtual time t is defined as the number of other memory-blocks referenced since its last reference. LRU Stack Depth Distribution (SDD) is a reference model where memory-block b with smaller $s_b(t)$ has larger reference probability [15]; the LRU algorithm is optimal for SDD [4]. Consider memory-block b following SDD. IGDR estimates that $\phi_b(\delta)$ takes its maximum at $\delta{=}0$ and monotonically decreases after that, and so does $w_b(t)$. Thereby IGDR gives a larger weight to the memory-block with a smaller BD, behaves similarly to the LRU algorithm.

Independent Reference Model Independent Reference Model (IRM) is a reference model where each memory-block b has its own reference probability β_b; the LFU algorithm is optimal for IRM [4, 1]. Memory-block b following this model also follows IIGD. $\phi_b(\delta)$ has a peak at $\delta{=}1/\beta_b$. $w_b(\delta)$ is almost flat with a value around β_b until $\delta{=}1/\beta_b$ and decreases rapidly after that. Therefore, IGDR gives larger weights to memory-blocks with larger β_b, behaves similarly to the LFU algorithm.

Periodic Stable Periodic Stable is defined as a pattern where a memory-block has a stable IRG. IGDR classifies this kind of memory-blocks into the PS class. Their weights can be obtained using their stable IRG; thus the weight estimation is accurate.

Correlated Reference Memory-blocks are often referenced several times in a short period of time and have extremely large IRGs after that [11]. Memory-block b with this behavior belongs to either TT or ST when its next IRG

is extremely large. Assume its maximum IRG in the initial frequent references is $\delta_{\max}$; $w_b(\delta)$ decreases rapidly from $\delta = \delta_{\max}$. In this way, IGDR can detect that the next IRG of b is extremely large.

4. IMPLEMENTATION

In the following, the discussion is restricted to the case of a two level cache hierarchy; as only a cache-miss in the secondary cache causes access to main memory, only the secondary cache is considered. The virtual time counter is incremented every time a reference to the secondary cache occurs.

IGDR implementation performs the following steps: (1) it classifies memory-blocks, (2) records the IRG distribution statistics per class, (3) estimates weights of memory-blocks using the distribution and stores the result in the weight table, and (4) on a replacement request, it retrieves the weights of the replacement candidate memory-blocks from the weight table and selects the replacement victim.

IGDR attaches information used for memory-block classification and replacement decision to each memory-block; this information is called **block information**. IGDR records the IRG distribution and the estimated weights per class; these statistics and weight table are called **class information**. This section describes the implementation, centering on block and class information.

4.1 Block Information and Memory-Block Classification

4.1.1 Contents of Block Information

Block information contains the following information for the BD calculation and memory-block classification. T_{irg} is a constant parameter.

CL Current memory-block class.
LA Virtual time of the last reference.
RC Reference count.
LG Last IRG.
SC Number of consecutive references where IRGs are stable; it is considered that the IRGs are stable when the difference between two consecutive IRGs is below the threshold T_{irg}.

The backward distance $d_b(t)$ of memory-block b is calculated as $t - \mathrm{LA}$ using its block information. A table which stores the block information is called a **directory**.

4.1.2 Information Update and Memory-Block Classification

Figure 4 shows the algorithm for the block information update including the memory-block classification. T_{RC}, T_{irg}, and T_{SC} are constant parameters. Assume the directory entry of a referenced memory-block b is d, and the current virtual time is t. When b is referenced, the reference count d.RC is incremented. The new IRG is calculated as $t-d$.LA. The difference between the last IRG and the new IRG is calculated as $|d.\mathrm{LG} - (t-d.\mathrm{LA})|$. This difference is called D_{irg}. The consecutive count of the stable IRG d.SC is incremented when $D_{\mathrm{irg}}<T_{\mathrm{irg}}$. d.SC is set to zero when $D_{\mathrm{irg}}\geq T_{\mathrm{irg}}$. Then it is determined whether or not to set the next class to PS. The next class is set to PS when $d.\mathrm{SC}\geq T_{\mathrm{SC}}$. If this is not the case, the next class is set to TT when the reference count is two, to ST when the reference count is from three to T_{RC}, and to MT when the reference count is greater than T_{RC}.

```
UPDATE_BLOCK_INFO(d)
d: directory entry of referenced memory-block

  t: current virtual time
  D_irg ← |d.LG − (t − d.LA)|
      /* difference between the last IRG and new one */
  d.RC ← d.RC + 1
  if d.RC ≥ 3 and D_irg < T_irg then d.SC ← d.SC + 1
  else d.SC ← 0 end if
  switch d.CL do
    case OT: d.CL ← TT
    case TT: d.CL ← ST
    case ST or MT or PS:
      if d.SC ≥ T_SC then d.CL ← PS
      else if d.RC > T_RC then d.CL ← MT
      else d.CL ← ST
      end if
  end switch
  d.LG ← t − d.LA,   d.LA ← t
```

Figure 4: Block information update and memory-block classification algorithm

T_{SC} is used to exclude memory-blocks with small loop counts. T_{irg} is the tolerance level of the temporal fluctuation of IRG. The optimal values for both depend on programs. T_{RC} is the criterion for classifying a memory-block into ST or MT. The optimal value also depends on programs because the correlation between the IRG distribution and the reference count depends on programs.

4.1.3 Storage of Block Information

Block information should be recorded for every memory-block, but this would incur an enormous storage cost. Therefore IGDR keeps the block information in a table for the memory-blocks currently in the cache; this table is called the **main directory**. However, this method has the problem that the block information of some memory-blocks are stored for too short a period of time [11]. To alleviate this problem, IGDR keeps the block information also for memory-blocks recently evicted from the cache in another table called the **ghost directory** [11]. The main directory can be integrated into the secondary cache tag, while the ghost directory takes the structure of a set-associative cache.

4.1.4 Allocation of Block Information

Figure 5 shows the allocation algorithm for entries in the main and ghost directories. One of the following cases occurs on a reference to a memory-block b.

Directory entry of b is not found in the main or ghost directory A replacement occurs in both the main and ghost directories. The replacement victim in the ghost directory is deleted. The replacement victim in the main directory moves to the vacant slot in the ghost directory. The directory entry of b is put in the vacant slot of the main directory, and is initialized. Its class is set to OT, the reference time is recorded, and the reference count is set to one.

Directory entry of b is found in the ghost directory The IRG of b is recorded to the class information. After that, the block information of b is initialized. Then a replacement occurs in the main directory. The replacement victim in the main directory and the directory entry found in the ghost directory swap their positions. The initializa-

ON_REFRENCE(b)
b: referenced memory-block

d: directory entry for b
t: current virtual time
S_{MD} (S_{GD}): the cache-set of d in MD (GD)

if d is not in MD or GD **then** /* cache-miss */
$v_{\mathrm{md}} \leftarrow$REPLACE_MD($S_{\mathrm{MD}}$)
$v_{\mathrm{gd}} \leftarrow$REPLACE_GD($S_{\mathrm{GD}}$)
d.CL $\leftarrow$ OT, d.LA $\leftarrow$ t, d.RC $\leftarrow$ 1, d.SC $\leftarrow$ 0
delete v_{gd}, put v_{md} in GD, put d in MD

else if d is in GD **then** /* cache-miss */
d.CL $\leftarrow$ OT, d.LA $\leftarrow$ t, d.RC $\leftarrow$ 1, d.SC $\leftarrow$ 0
$v_{\mathrm{md}} \leftarrow$REPLACE_MD($S_{\mathrm{MD}}$)
put v_{md} in GD, put d in MD

else if d is in MD **then** /* cache-hit */
UPDATE_BLOCK_INFO(d)
end if

Figure 5: Allocation algorithm for entries in the main directory (MD) and ghost directory (GD)

tion is a heuristic method to improve the performance, and is detailed elsewhere [17].

Directory entry of b ***is found in the main directory*** The IRG of b is recorded to its class information; then its block information is updated.

4.1.5 Replacement of Block Information

Figure 6 shows the replacement algorithm for entries in the main and ghost directories. $\mathcal{W}_A(G(\delta+1))$ is an estimated weight whose details are described in Section 4.2.3. The replacement algorithm for the main directory corresponds to the cache replacement algorithm. Replacement candidates have corresponding memory-blocks; the replacement algorithm retrieves the weights of these memory-blocks from the weight table, and then selects for eviction the directory entry whose corresponding memory-block has the minimum weight. The algorithm uses a different weight for a memory-block of PS; (1) the reciprocal of its last IRG when its BD is within its last IRG; (2) the value in the weight table when its BD is outside its last IRG. IGDR replaces ghost directory entries so that the number of memory-blocks in each class is equalized; this is a heuristic method to improve the performance, and is detailed elsewhere [17].

4.2 Class Information and Weight Calculation

$\mathcal{D}_{B_k}$ in Equation (6) is a table allocating an entry for every IRG value; thus it requires an enormous storage area. Therefore, a method recording the distribution using a table with a practical number of entries, the number being a constant parameter U, is introduced: (1) The IRG space is divided into regions with a span of g ticks and the occurrence count per region is recorded; g is a constant parameter, and (2) occurrence counts of IRGs that are equal to or greater than $g\times(U-1)$ are recorded into one entry. A function $G(\delta)$, which gives the table index from a IRG δ, is defined as follows.

$$G(\delta) = \begin{cases} \lfloor \delta/g \rfloor, & \delta/g < U-1 \\ U-1, & \delta/g \geq U-1 \end{cases} \quad (7)$$

A table $\mathcal{D}'$, which corresponds the $\mathcal{D}$ table and records the IRG distribution using this index, is defined as follows.

REPLACE_MD(s)
s: cache-set where a replacement occurs
Output: directory entry to be replaced

$d_1, d_2, \cdots, d_n$: directory entries of replacement candidates
$b_1, b_2, \cdots, b_n$: memory-blocks corresponding to d_i
t: current virtual time
w_i: weight of b_i

for $i = 1$ **to** n **do**
$A_i \leftarrow d_i$.CL /* class of b_i */
$\delta_i \leftarrow t - d_i$.LA /* backward distance of b_i */
switch d_i.CL **do**
case OT or TT or ST or MT: $w_i \leftarrow \mathcal{W}_{A_i}(G(\delta_i+1))$
case PS:
if $\delta_i \leq d_i$.LG **then** $w_i \leftarrow 1/d_i$.LG
else $w_i \leftarrow \mathcal{W}_{A_i}(G(\delta_i+1))$ **end if**
end switch
end for

if there exist multiple d_i with the smallest w_i **then**
return LRU entry using d_i.LA
else return d_i with the smallest w_i
end if

REPLACE_GD(s)
s: cache-set where a replacement occurs
Output: directory entry to be replaced

$d_1, d_2, \cdots, d_n$: directory entries of replacement candidates
$b_1, b_2, \cdots, b_n$: memory-blocks corresponding to d_i
$H(X)$: total number of memory-blocks whose class is X

if there exist multiple d_i with the largest $H(d_i.\mathrm{CL})$ **then**
return LRU entry using d_i.LA
else return d_i with the largest $H(d_i.\mathrm{CL})$
end if

Note 1: n is the number of ways in the cache.

Figure 6: Replacement algorithm for entries in the main directory (MD) and ghost directory (GD)

$$\mathcal{D'}_{B_k}(i) = \begin{cases} \sum_{j=gi}^{g(i+1)-1} \mathcal{D}_{B_k}(j), & 0 \leq i < U-1 \\ \sum_{j=gi}^{\infty} \mathcal{D}_{B_k}(j), & i = U-1 \end{cases}$$

IGDR approximates w_{B_k} using $\mathcal{D}'$ as follows.

$$w_{B_k}(\delta) \approx \frac{\sum_{i=G(\delta+1)}^{U-1} \mathcal{D'}_{B_k}(i) \frac{1}{g(i+1)}}{\sum_{j=G(\delta+1)}^{U-1} \mathcal{D'}_{B_k}(j)} \quad (8)$$

For $0 \leq i < U$, IRGs δ where $g\,i \leq \delta < g\,(i+1)$ are treated as the same. This degrades the weight estimation accuracy; therefore small g is preferred. Occurrence numbers of IRGs equal to or greater than $g\,U$ are recorded in the same table entry as those with $g\,(U-1)$. This also degrades the accuracy; thus it is desirable that $g\,U$ is greater than the maximum IRG that a program exhibits. On the other hand, the storage cost increases as U increases. Therefore parameter g and U should be chosen considering a trade-off between storage cost and performance.

IGDR attaches the IRG distribution statistics required to calculate Equation (8) to each class as class information, which comprises; (1) the $\mathcal{N}$ queue and the Q register that

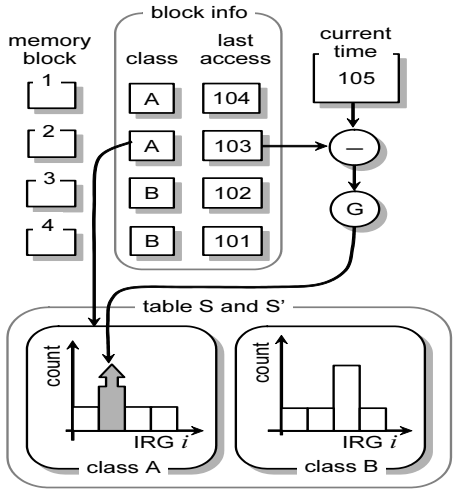

Figure 7: Recording the IRG distribution statistics per class

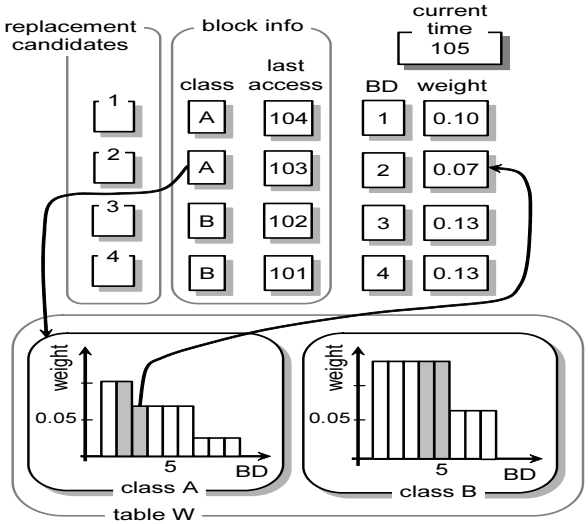

Figure 8: Replacement decision

record the number of memory-blocks with a certain range of BD at the current time to assist the IRG distribution recording, (2) the $\mathcal{S}$ and $\mathcal{S}'$ tables corresponding to $\mathcal{D}'$ table, and (3) the $\mathcal{W}$ table storing the estimated weights. These structures are prepared for each class and a subscript is used to denote the class, e.g. $\mathcal{W}$ for class A is expressed as $\mathcal{W}_A$.

4.2.1 Queue $\mathcal{N}$ and Register Q

$\mathcal{N}$ is an integer queue with U entries. An entry with a value of zero is inserted into its tail, denoted as $\mathcal{N}(0)$, every g ticks. Assume memory-block b in class A is referenced at virtual time t_c; and the last reference time of b is t_l. The reference might change the class of b from A to A'. $\mathcal{N}$ is updated as follows in this case.

$i \leftarrow \lfloor t_c/g \rfloor - \lfloor t_l/g \rfloor$
if $i < U$ **then** $\mathcal{N}_A(i) \leftarrow \mathcal{N}_A(i) - 1$ **end if**
$\mathcal{N}_{A'}(0) \leftarrow \mathcal{N}_{A'}(0) + 1$

$\mathcal{N}$ is updated in a similar way when the block information of b is evicted from the ghost directory. $\mathcal{N}(i)$ records the number of memory-blocks where $\lfloor t_c/g \rfloor - \lfloor t_l/g \rfloor = i$. That is, $\mathcal{N}(i)$ records the approximate number of memory-blocks whose BDs are from $g\,i$ to $g\,(i+1)$. The Q register accumulates the value in the head entry which is pushed out every g ticks. F is a constant parameter and Q is initialized every F ticks. Q approximates the number of memory-blocks that are not referenced over $g\,U$ ticks, observed for F ticks.

4.2.2 Table $\mathcal{S}$ and $\mathcal{S}'$

These tables record the IRG distributions. $\mathcal{S}$ and $\mathcal{S}'$ is used to calculate the numerator and denominator in Equation (8), respectively. Note that the numerator allows $\mathcal{S}$ to ignore extremely large IRGs. These tables are reset every F ticks to record recent distribution. The following events are utilized for recording the IRG distribution.

When memory-blocks are referenced Assume a memory-block in class A is referenced and its BD is δ. One is added to $\mathcal{S}_A(G(\delta))$ and $\mathcal{S}'_A(G(\delta))$. **Figure 7** illustrates the recording of statistics. Assume memory-block 2 in class A is referenced and the current virtual time is 105. The BD of memory-block 2 is 2 ticks; then the $\mathcal{S}_A(G(2))$ and $\mathcal{S}'_A(G(2))$ is incremented.

When memory-blocks are evicted from the ghost directory Assume the block information of a memory-block is evicted from the ghost directory and its BD is δ. Its BD information is being lost; and there exist two options for $\mathcal{S}'$; (a) one is added to $\mathcal{S}'(G(\delta))$ reflecting the fact that its BD is at least δ or (b) one is added to $\mathcal{S}'(U-1)$ assuming its next IRG is extremely large. There exist two corresponding options for $\mathcal{S}$; (c) one is added to $\mathcal{S}(G(\delta))$ or (d) nothing is added to $\mathcal{S}$. The combination of (a) and (d) is used because it exhibits the best performance over the other combinations, though not treating $\mathcal{S}'$ and $\mathcal{S}$ in the same manner.

When memory-blocks are not referenced for a long time It is considered that a memory-block which is not referenced over $g\,U$ ticks will either not be referenced again or its next IRG is extremely large. The number of these memory-blocks is estimated using Q and added to $\mathcal{S}'(U-1)$.

4.2.3 Table $\mathcal{W}$, Weight Calculation, and Replacement Decision

Equation (8) is calculated using $\mathcal{S}$ and $\mathcal{S}'$; and the result is stored to the $\mathcal{W}$ table. This calculation takes a long time; thus the following operation is performed every F ticks. v_1 and v_2 are temporal registers.

$v_1 \leftarrow \mathcal{S}(U-1)/(g * U)$
$v_2 \leftarrow \mathcal{S}'(U-1) + Q$
$\mathcal{W}(U-1) \leftarrow v_1/v_2$
for $i = U-2$ **downto** 0 **do**
  $v_1 \leftarrow v_1 + \mathcal{S}(i)/(g * (i+1))$
  $v_2 \leftarrow v_2 + \mathcal{S}'(i)$
  $\mathcal{W}(i) \leftarrow v_1/v_2$
end for

Assume a memory-block b in class A and its BD is δ at a certain virtual time; $\mathcal{W}_A(G(\delta+1))$ is used as its weight. On a replacement request, the weights of the replacement candidate memory-blocks are retrieved from the $\mathcal{W}$ table; and the weights are compared; and then the memory-block with the smallest weight is selected. **Figure 8** illustrates the replacement decision. Memory-blocks 1, 2, 3 and 4 are the replacement candidates. Their weights are retrieved from the $\mathcal{W}$ table. The weight of memory-block 2 is 0.07; as this is the minimum weight memory-block 2 is evicted.

The $\mathcal{S}$ and $\mathcal{S}'$ tables record IRGs for F references. A small value of F facilitates adaptation to the reference behavior when that behavior changes frequently. On the other hand,

Table 1: Simulation parameters

Fetch	fetch up to 8 instrs, 32-entry instr queue
Decode Issue	decode/issue up to 8 instrs, 128-entry instr window
Exec Unit	4 INT, 4 LD/ST, 2 other INT, 2 FADD, 2 FMUL, 2 other FLOAT, 64-entry load/store queue, 16-entry write buffer, 16 MSHRs, oracle resolution of ld-st addr. dependency
Retire	retire up to 8 instrs, 256-entry reorder buf
L1 cache	instr: 32 KB, 2-way, 128 B block size data: 32 KB, 2-way, 128 B block size, 3-cycle load-to-use latency
L2 cache	512 KB, 8-way, 128 B block size, 13-cycle load-to-use latency
Main memory	200-cycle load-to-use latency, 128-bit address/data muxed bus to L2, clocked at 1/4 speed of processor core

Table 2: IGDR parameters

Parameter	Description	Value
U	number of IRG regions	2^8
g	span of IRG region	2^9
F	number of references for recording IRG distribution	2^{18}
T_{RC}	threshold of reference count for class MT	8
T_{irg}	threshold of IRG fluctuation for stable IRG	2^{10}
T_{SC}	threshold of consecutive occurrences of stable IRG for class PS	4

the IRGs are not recorded while the calculation of Equation (8) is under way, which begins every F ticks, because the calculation uses the $\mathcal{S}$ and $\mathcal{S}'$ tables; thus, a large value of F is preferred from this aspect.

5. EVALUATION

5.1 Methodology

An execution-driven, cycle-accurate software simulator is newly developed for the evaluation of IGDR. The simulated processor is a superscalar, 64-bit processor and uses Alpha ISA. A 40-bit physical address space is used. **Table 1** shows the simulation parameters. The latencies and the issue intervals of the instructions are set to the same as Alpha 21264. The secondary cache is 512 KB, 8-way set-associative; and its baseline replacement algorithm is the LRU algorithm. The main memory has a 200-cycle load-to-use latency.

Ten programs from SPEC CPU2000 are used, namely `ammp`, `gcc`, `art`, `facerec`, `vpr`, `mcf`, `mgrid`, `applu`, `apsi`, and `vortex`. Our simulator skips the first 2^{30} instructions, and then simulates the following 2^{30} instructions. All programs use the 'reference' dataset. A Linux Alpha system was used for the compilation. All C programs were compiled using Compaq C Compiler version 6.2.9-504 using option '-arch ev6 -fast -O4'. All FORTRAN programs were compiled using Compaq Fortran Compiler version 1.0.1 using option '-arch ev6 -fast -O5'.

Table 2 shows the IGDR parameters. The g, U, and F parameters are explained in Section 4.2. In the programs used, many IRGs are within the range from 2^9 to 2^{17}; therefore g=2^9 and U=2^8 are used. F is set to 2^{18}; the size of the ghost directory is set to a fourth of that of the main directory. Sensitivity analysis for these parameters is set out elsewhere [17], that shows in the programs used, the number of cache-misses varies little when varying these parameters around these settings. The T_{RC}, T_{irg}, and T_{SC} parameters are explained in Section 4.1.2. In the programs used, it is rare that memory-blocks with reference counts of greater than eight have extremely long IRGs; thus T_{RC} is set to eight. When a program exhibits the stable IRGs, the fluctuation of the IRGs are often less than 2^{10}; therefore T_{irg} is set to 2^{10}. T_{SC} is set to four; T_{SC} has little effects on the number of cache-misses when $T_{\mathrm{SC}} \geq 2$.

The LFU [4], LRU-2 [11], and LRFU [9] methods are chosen for comparison because they can be applied directly to set-associative caches. The results of MIN [2] are also shown, but only for cache-miss number.The parameter called Correlated Reference Period for LRU-2 was set to 2^7 as this value maximizes the average cache-miss reduction. The parameter called γ for LRFU is set to 2×10^{-5} for the same reason as for LRU-2.

5.2 Results and Discussion

Cache-Miss Reduction

Not all of the classes are always required; the notation OS is used to represent IGDR using the memory-block classes of OT and ST; OSM represents IGDR using OT, ST, and MT; and so forth. **Figure 9** shows the cache-miss reductions of LFU, LRU-2, LRFU, IGDR, and MIN. The results are compared with the baseline LRU algorithm. "average" shows the averages over all programs. IGDR reduces cache-misses by up to 46.1%; and by 19.8% on average, performing better than LFU, LRU-2, and LRFU, that reduce them by 2.8%, 9.4%, and 13.7%, respectively, on average. Note that IGDR never performs worse than LRU, while LFU, LRU-2, and LRFU does in some programs. Excluding the MIN algorithm, IGDR achieves the best cache-miss reduction in `ammp`, `gcc`, `mgrid`, `applu`, `apsi`; and achieves cache-miss reductions comparable to the best algorithm for the other programs except for `vpr`. By contrast, other algorithms exhibit cache-miss reductions far behind that of the best algorithm in some programs.

In `ammp`, `gcc`, and `art`, many memory-blocks follow the IRM pattern. LFU, LRU-2, and LRFU can adapt to this pattern and can reduce cache-misses. IGDR also can adapt and reduce cache-misses as much as LFU and LRFU. In `vpr`, many memory-blocks follow the IRM pattern; LFU, LRFU, and LRU-2 can reduce cache-misses. However, memory-blocks have many different reference probabilities, and thus have a wide variety of IRG distributions. IGDR classifies many memory-blocks of `vpr` into MT, so that memory-blocks with different IRG distributions are mixed, degrading the accuracy of the weight estimation; as a result, IGDR cannot reduce cache-misses as much as LFU and LRFU. In `mcf`, `applu`, and `vortex`, many memory-blocks follow the correlated reference pattern. IGDR can adapt to this pattern and reduce cache-misses; while LRU-2 can adapt to this pattern, it fails to reduce cache-misses in `vortex`. In `facerec` and `mgrid`, many memory-blocks have large IRGs. IGDR can adapt to this pattern and reduce cache-misses; whereas other algorithms cannot exhibit stable performance. In `apsi`, many memory-blocks have stable IRGs while they have different reference counts. LFU cannot reduce cache-misses because it utilizes only reference counts; whereas IGDR can adapt to this pattern and reduce cache-misses because it utilizes IRG.

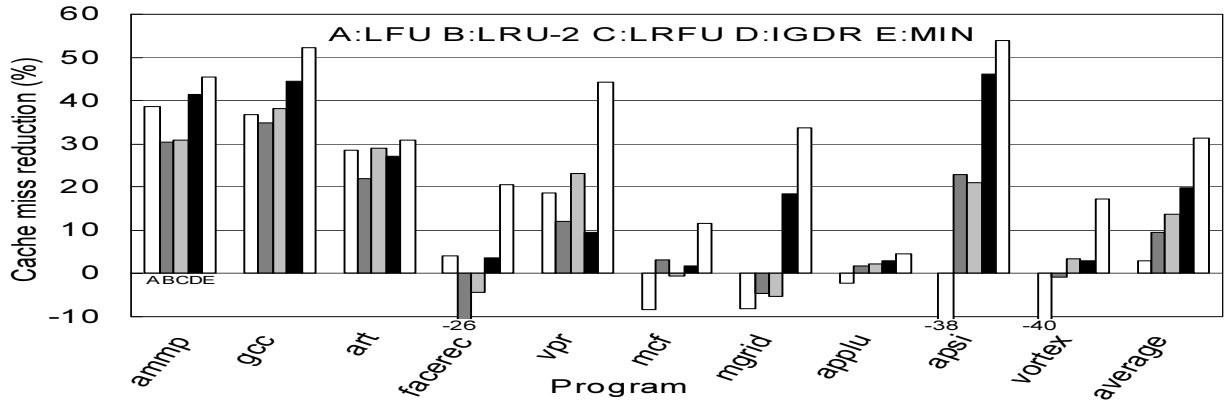

Figure 9: Cache-miss reduction over the LRU algorithm, comparing IGDR with existing methods

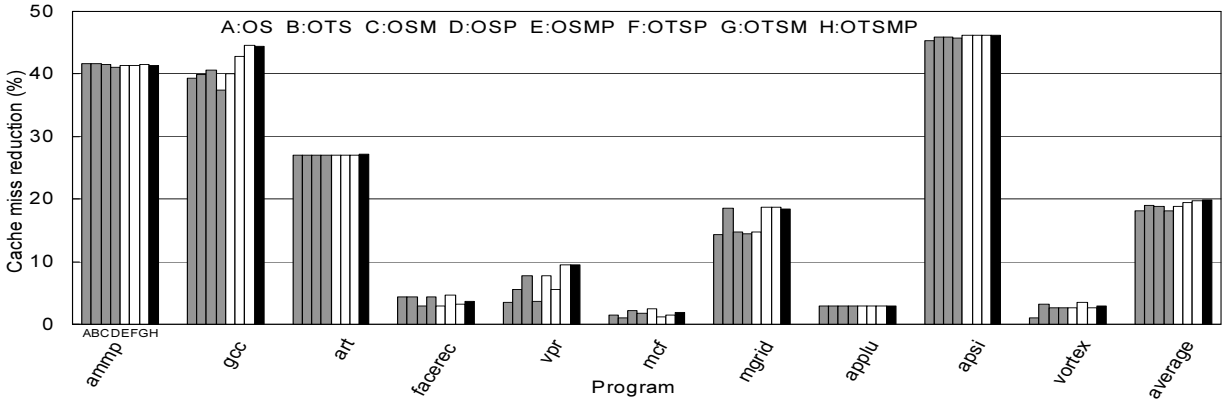

Figure 10: Cache-miss reduction over the LRU algorithm, comparing IGDR variants

Figure 10 shows cache-miss reductions by IGDR varying the class set used, compared with the LRU algorithm. "average" shows the averages over all programs. Using OT and ST contributes to cache-miss reductions over the LRU algorithm for all programs. The primary reason is that IRG distribution difference is large between memory-blocks with reference count of one and the others in all programs. The subsumed secondary reason is that many memory-blocks follow One Touch pattern, explained in Section 3.5. Comparing OSMP and OTSMP in `gcc`, `vpr`, and `mgrid` indicates that TT further contributes to cache-miss reductions for these programs. The primary reason is the IRG distribution difference. The subsumed secondary reason is that `gcc` and `vpr` have many memory-blocks with correlated reference property, explained in Section 3.5. Comparing OTSP and OTSMP in `gcc` and `vpr` indicates that a combination of ST and MT further contributes to cache-miss reductions for these programs; the reason is similar as in the TT case. Comparing OTSM and OTSMP in `facerec` and `mcf` indicates that PS further contributes to cache-miss reductions for these programs; this is because there are many memory-blocks with stable IRGs.

Speedup

Figure 11 shows the speedup of LFU, LRU-2, LRFU, and IGDR over the baseline LRU algorithm. "average" shows the harmonic mean over all programs. IGDR improves the speed of the programs by up to 48.9%; and by 12.9% on average, performing better than LFU, LRU-2, and LRFU, that improve the speed on average by 6.3%, 7.8%, and 10.2%, respectively.

Table 3: Additional storage of IGDR

Name	Bits per entry	Num of entries	Size (KB)
Main directory (MD)	CL:3, LA:14, RC:3, LG:14, SC:2	4096	18 (9)
Ghost directory	same as MD	1024	4.5 (2.25)
Ghost directory tag	$40-\log_2(2^{17}/8)$ =26	1024	3.25
$\mathcal{S}$ table	19	$256\times m$	$0.59\times m$
$\mathcal{S}'$ table	19	$256\times m$	$0.59\times m$
$\mathcal{W}$ table	mantissa:15 exponent:6	$256\times m$	$0.65\times m$
$\mathcal{N}$ queue	10	$256\times m$	$0.31\times m$
Div table	12	4096	6
OS ($m=2$)			24.8
OTSM (m=4)			29.1
OTSMP (m=5)			42.5

Note 1: m denotes the number of classes utilized.
Note 2: sizes in parenthesis are those when class PS is not used.

5.3 Area Issue

This subsection discusses die area overhead compared to the LRU algorithm.

Additional Storage The main and ghost directories, the $\mathcal{S}$, $\mathcal{S}'$, $\mathcal{W}$, $\mathcal{N}$ structures, and the division table are additional storages for IGDR compared to the LRU algorithm. Each entry in the main and ghost directories is block information. The LA component uses a 23-bit integer expression because empirically few memory-blocks have IRGs of more than 2^{23}. The lower $\log_2 g$ bits are omitted and the upper 14 bits are stored. The LG component is expressed and stored in the

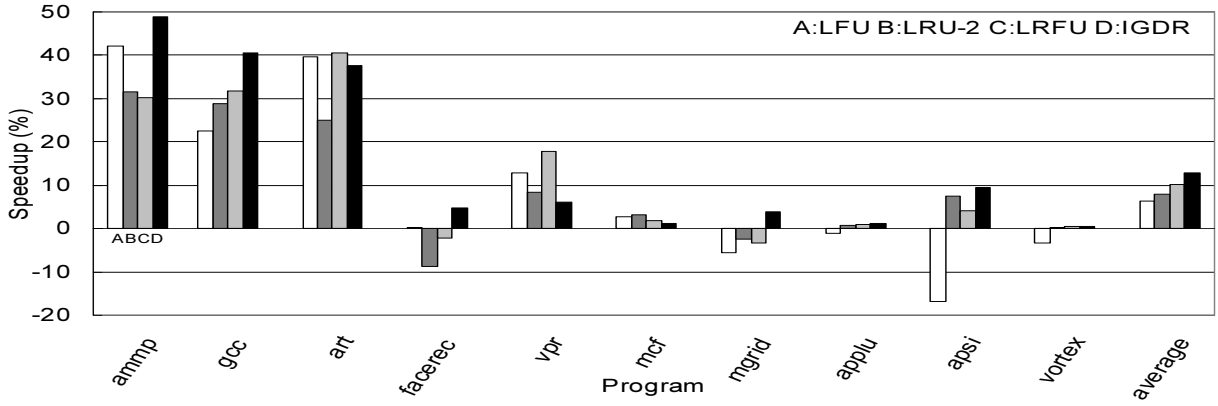

Figure 11: Speedup of the programs over the LRU algorithm, comparing IGDR with existing methods

same way as for LA. IRG distribution statistics are recorded for 2^{18} references in the $\mathcal{S}$ and $\mathcal{S}'$ tables; thus a 19-bit integer expression is used for their entries. An entry of the $\mathcal{W}$ table uses a floating-point expression; it is assumed that a 16-bit mantissa and a 6-bit exponent suffice. An entry of the $\mathcal{N}$ queue records the number of memory-blocks within a time span of g ticks; thus uses a 10-bit integer expression. A 2^{12}-entry division table with 12-bit values is used because such precision is sufficient for the weight calculation.

Table 3 summarizes the storage costs; m denotes the number of classes utilized. Not all of the classes are always required. The same notations as in Section 5.2 are used, e.g. OS for IGDR using OT and ST classes, OTSM for IGDR using OT, TT, ST, and MT. The SC and LG components of the block information are omitted when class PS is not used. OTSM is recommended considering cost versus performance. The storage costs of the directories are proportional to the number of cache-blocks in the secondary cache.

Additional Logic The major components of the additional logics are one floating-point addition and two floating-point multiplication logics. A detailed estimation is set out elsewhere [17]. Using the implementation for IEEE single precision values [14, 18], the area for the addition and multiplication logic is estimated as $0.14\,\mathrm{mm}^2$ and $0.26\,\mathrm{mm}^2$, respectively, in $0.18\,\mu$m technology. The total additional area is $0.66\,\mathrm{mm}^2$.

IGDR versus Larger Cache The die area overhead of IGDR can be used to increase the cache size. The cache area is estimated using the CACTI model [16]; 512 KB cache requires $59.00\,\mathrm{mm}^2$ for $0.18\,\mu$m technology; 554.5 KB cache incorporating 42.5 KB storage overhead requires $57.77\,\mathrm{mm}^2$; and 576 KB cache requires $62.52\,\mathrm{mm}^2$, which is comparable with the 554.5 KB cache including $0.66\,\mathrm{mm}^2$ logic overhead. Therefore IGDR is compared with the 9-way set-associative 576 KB cache using the LRU algorithm; the former and latter reduce cache-misses on average by 19.8% and 4.6%, respectively, justifying the die area overhead of IGDR.

5.4 Latency Issue

This subsection discusses computational overhead compared to the LRU algorithm. The critical path of the weight calculation for the $\mathcal{W}$ table consists of a $\mathcal{S}$ table read, a floating-point multiplication, a floating-point addition, a floating-point multiplication, and a write to the $\mathcal{W}$ table. These operations can be pipelined with per-stage latency of 4 processor cycles. The $\mathcal{S}$ and $\mathcal{S}'$ table cannot record IRG distribution while they are used by this operation; this is simulated and increases the cache-misses by no more than 1% for all the programs used. The critical path of the IRG recording to the $\mathcal{S}$ and $\mathcal{S}'$ table consists of block information retrieval, calculation of the IRG, integer addition, and a write to these tables. These operations can be pipelined with per-stage latency of 4 processor cycles; this per-stage latency is within the access time of the secondary cache [7]. The critical path of a replacement decision consists of block information retrieval, BD calculation, reading the $\mathcal{W}$ table, and a floating-point comparison. These operations can be pipelined with 4-cycle per-stage latency; the total latency is 16 cycles, which is hidden by the access time of main memory.

5.5 Sensitivity Analysis

Figure 12 compares the cache-miss numbers with various cache sizes, associativity, and block sizes, respectively. The cache-miss number of a program is normalized to that of the LRU algorithm of the leftmost configuration; then an average over all programs is calculated. As shown in the figure, IGDR significantly outperforms the LRU algorithm over the various configurations.

6. SUMMARY

This paper has proposed a novel replacement algorithm for set-associative secondary caches of processors, called Inter-Reference Gap Distribution Replacement (IGDR). IGDR attaches a weight to each memory-block and replaces the memory-block with the minimum weight. The time difference between successive references to a memory-block is called its Inter-Reference Gap (IRG). IGDR estimates the ideal weight using the reciprocal of the IRG. We have proposed a reference model where each memory-block has its own probability distribution of IRGs; from which IGDR calculates the expected value of the reciprocal of the IRG to use as the weight of a memory-block. The IRG probability distribution of each memory-block has its own peaks; thus its weight reflects its behavior over time.

For implementation, IGDR does not have the probability distribution; instead it records the IRG distribution statistics at run-time. IGDR classifies memory-blocks into several classes; and records the distribution statistics per class; then estimates the weight of a memory-block. It has been shown that the IRG distributions of memory-blocks correlate their reference counts; this enables classifying memory-blocks by their reference counts; this classification produces only a small number of classes, saving storage overhead for record-

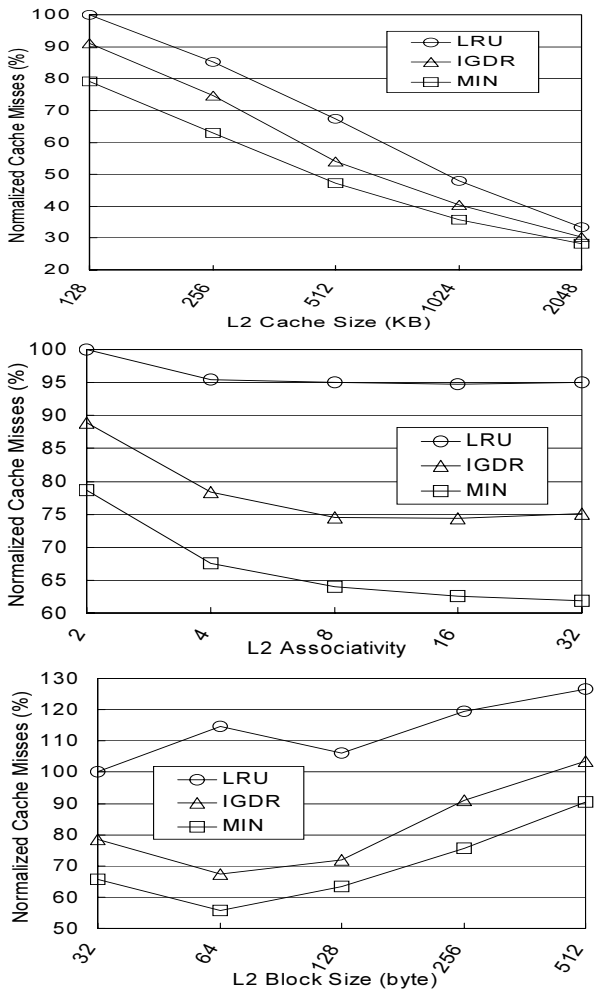

Figure 12: Cache-miss number comparisons with various secondary cache configurations

ing the distribution statistics. The effectiveness of IGDR has been evaluated through an execution-driven simulation. For ten of the SPEC CPU2000 programs, IGDR achieves up to 46.1% (on average 19.8%) cache-miss reduction and up to 48.9% (on average 12.9%) speedup over the LRU algorithm; it achieves a greater amount of average cache-miss reduction than the existing methods of LFU, LRU-2, and LRFU; IGDR never performs worse than the LRU algorithm; and the die area overhead of IGDR is justified by comparing IGDR to a cache with a larger configuration.

7. REFERENCES

[1] A. V. Aho, P. J. Denning, and J. D. Ullman. Principles of optimal page replacement. *J. of ACM*, 18(1):80–93, 1971.

[2] L. A. Belady. A study of replacement algorithms for virtual storage computers. *IBM Sys. J.*, 5(2):78–101, 1966.

[3] J. Choi, S. H. Noh, S. L. Min, and Y. Cho. Towards application/file-level characterization of block references: A case for fine-grained buffer management. In *Proc. ACM SIGMETRICS Conf.*, pages 286–295, June 2000.

[4] J. E. G. Coffman and P. J. Denning. *Operating Systems Theory*. Prentice-Hall, Englewood Cliffs, 1973.

[5] S. Jiang and X. Zhang. Lirs: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proc. ACM SIGMETRICS Conf.*, 2002.

[6] T. Johnson and D. Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In *Proc. the 20th Int'l Conf. on VLDB*, pages 439–450, 1994.

[7] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proc. ASPLOS*, pages 211–222, Oct. 2002.

[8] J. M. Kim, J. Choi, J. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. A low-overhead, high-performance unified buffer management scheme that exploits sequential and looping references. In *Proc. the 4th USENIX Symp. on Operating System Design and Implementation*, pages 119–134, Oct. 2000.

[9] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Trans. on Computers*, 50(12):1352–1360, 2001.

[10] N. Megiddo and D. S. Modha. Arc: A self-tuning, low overhead replacement cache. In *Proc. the 2nd USENIX Conf. on File and Storage Technologies (FAST 03)*, Mar. 2003.

[11] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The lru-k page replacement algorithm for database disk buffering. In *Proc. ACM SIGMOD Conf.*, pages 297–306, 1993.

[12] V. Phalke and B. Gopinath. An inter-reference gap model for temporal locality in program behavior. In *Proc. SIGMETRICS Conf.*, pages 291–300, 1995.

[13] J. T. Robinson and M. V. Devarakonda. Data cache management using frequency-based replacement. In *Proc. SIGMETRICS Conf.*, pages 134–142, 1990.

[14] J. Schauer. Opensource floating point adder, May 2002. http://www.hmc.edu/chips.

[15] G. S. Shedler and C. Tung. Locality in page reference strings. *SIAM J. on Computing*, 1:218–241, Sept. 1972.

[16] P. Shivakumar and N. Jouppi. Cacti 3.0: an integrated cache timing, power, and area model. Technical Report 2001/2, Compaq WRL, Aug. 2001.

[17] M. Takagi. *Improving Cache Performance by Exploiting Redundancy, Temporal Affinity, and Reuse of Data*. PhD thesis, Univ. of Tokyo, Dec. 2003.

[18] M. Uya, K. Kaneko, and J. Yasui. A cmos floating point multiplier. *IEEE J. of Solid-Sate Circuits*, 19(5):697–702, Oct. 1984.

[19] W. A. Wulf and S. A. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1):20–24, Mar. 1995.

[20] Y. Zhou, J. F. Philbin, and K. Li. The multi-queue replacement algorithm for secondary level buffer caches. In *Proc. USENIX Technical Conf.*, June 2001.