An Inter-Reference Gap Model for Temporal Locality in Program Behavior

Vidyadhar Phalke
Dept. of Computer Science
Hill Center, Rutgers University
New Brunswick, NJ 08903.
Tel: (908) 445–0606
Fax: (908) 445–0981
phalke@aquarius.rutgers.edu

Bhaskarpillai Gopinath
Dept. of Elect. and Computer Engg
Rutgers University
New Brunswick, NJ 08903.
Tel: (908) 445–5394
Fax: (908) 445–0981
gopinath@aquarius.rutgers.edu

Abstract

Locality of reference in program behavior has been studied and modelled extensively because of its application to memory design, code optimization, multiprogramming etc. We propose a k-order Markov chain based scheme to model the sequence of time intervals between successive references to the same address in memory during program execution. Each unique address in a program is modelled separately. To validate our model, which we call the Inter-Reference Gap (IRG) model, we show substantial improvements in three different areas where it is applied. (1) We improve upon the miss ratio for the Least Recently Used (LRU) memory replacement algorithm by up to 37%. (2) We achieve up to 22% space-time product improvement over the Working Set (WS) algorithm for dynamic memory management. (3) A new trace compression technique is proposed which compresses up to 2.5% with zero error in WS simulations and up to 3.7% error in the LRU simulations. All these results are obtained experimentally, via trace driven simulations over a wide range of cache traces, page reference traces, object traces and database traces.

Key Words: Locality of Reference, Markov Chains, Prediction, Memory Replacement, Dynamic Memory Management, Trace compaction, Trace Driven Simulation.

1 Introduction

Locality of reference in almost all kinds of computer program executions has been extensively demonstrated and is a well known behavior [1–5]. Locality is the primary reason why memory hierarchies improve overall performance. An argument usually given against memory hierarchies is that cache memories are becoming bigger and faster, and hence, in the near future, it will be possible to accommodate the entire code and data segments of a program in the cache and do away with hierarchies. There are two main arguments against this theory. The first reason is that even though caches are becoming faster, it has been observed over the past decade that CPU speeds are increasing at a faster rate than those of memory, resulting in a need for an even faster memory device to match the CPU. Hence, on-chip CPU caches are quite common. The second reason is that new applications continually appear that exhaust memory bandwidth, e.g. speech and video applications, which would easily fill up megabyte sized caches. Hence memory hierarchies and the study of locality principles will be needed as long as the "memory bottleneck" problem continues to exist.

There are two broad classifications of locality. *Temporal locality*, which proposes that an address just referred to, has a high probability of getting referred to in the near future; and *spatial locality* which says that an address nearby in memory space to the one just referred to has a high probability of being referenced in the near future. Use of the temporal locality principle is done for deallocating memory, e.g. the least recently used (LRU) cache replacement policy replaces the cache block which hasn't been referred to for the longest duration. This is done using the assumption that chances of the least-recently-used block being referred to again, are very low. Similarly, working-set (WS) principle removes pages in a virtual memory system if they haven't been referred to for a certain predefined amount of time.

Spatial locality, on the other hand, is exploited to transfer larger chunks of data than required between successive levels in a memory hierarchy. For example, when a cache miss occurs, a block (usually much larger than a single word) is brought in from the main memory. The block, in addition to the required memory word, contains addresses which are physically adjacent to the one just referenced. Another example is the sequential prefetching strategy [6], which presumes spatial locality of reference when doing prefetching.

In this paper, we study temporal locality using a wide array of program execution traces. A trace, in general, is a log of all the events that occur during a program run, but in our case we only look at all the memory addresses that get referenced. This is sufficient because temporal locality is concerned only with the addresses. Time is *virtual*, which means that each memory reference is assumed to happen at a clock tick, the real absolute time between consecutive references is immaterial. Without loss of generality, we use the word *address* to imply a unit of data transfer between successive levels in a memory hierarchy. We also assumed that this unit moves in its entirety from one level to the other. Walking through a memory hierarchy from top (CPU) to bottom (disk and network), an address could mean a:

- ☐ cache block: Between an external cache and a main memory system.
- ☐ page: In a virtual memory architecture with paging.
- ☐ sector: In a disk and a main memory environment.
- ☐ data object: In a CAD / database environment. The object could be a database record, a relation or a file depending on the granularity.

We define **IRG** (Inter-Reference Gap) for an address in a trace, as the time interval between successive references to that same address. The IRG stream for an address in a trace, is the sequence of successive IRG values for that address. For example, if an address *a* gets referred to at time t_1, t_2, t_3, t_4 and so on, then the IRG stream for *a* will be t_2–t_1, t_3–t_2, t_4–t_3 and so on. These time values (t_i's) are virtual as explained before, and we are not measuring the absolute time at which the access is made.

Each of the IRG streams is modelled using an order *k* Markov chain. Using the past IRG values, these models are modified on-line and a prediction technique is defined to estimate the future IRG values. The prediction technique, and hence the model is validated in the following three different ways:

First, it is validated by applying it in the memory replacement process. Such prediction based algorithms, although space and time wise expensive, give an idea of how much improvement can be made in the miss ratios by modelling temporal locality. We then explore for a practical solution and propose an explicit predictor based replacement algorithm that works well in practice and does not consume prohibitive amount of space.

Second, we apply the prediction technique for improving variable memory management algorithms. Here both space and time have to be optimized for a process. Using our prediction model, we improve the space-time product over existing techniques like the Working Set (WS) and the Page Fault Frequency (PFF) algorithms.

Finally, we extend our model for trace compaction. Each of the IRG models of a trace is compacted in such a way that information important to the WS simulations is not lost. We get very high compression in program traces at no cost to the WS simulations, i.e. WS simulations get speeded up with zero error.

We present our work in two parts. In the first part, we deal only with the IRG modelling, in the following way. In section 2 we describe some simple properties of the IRG streams and present the motivation for studying them in detail. In section 3 we describe related work on program modelling – both analytical and empirical, and show why it is inadequate for our purposes. In section 4 we formally describe our model and the prediction technique based upon that.

In the second part, we present the three validations of our model. First, in section 5 we apply the prediction techniques to fixed memory replacement algorithms and present the improvements using trace driven simulations. Second, in section 6 we describe a new dynamic memory algorithm based on IRG modelling and show why it is better than the current algorithms. Finally, in section 7 we propose a trace compaction technique and present experimental evidence showing its advantages. In each of these three sections, we also describe the related work pertaining to that area. Finally in section 8 we have the summary and description of some of our ongoing work.

2 Motivation for IRG Modelling

To give a flavor of IRG behavior consider the following figures. Figure 1 shows the IRG value distribution for three addresses in the CC1 trace (See section 5 for trace details). The three addresses are, the most referred, the 10th most referred, and the 100th most referred memory location. The title of each plot tells the memory address value in hexadecimal. The X-axis shows the IRG values and the Y-axis shows their frequencies. In figure 2 we plot the actual IRG streams for the same three addresses. Each point on the Y-axis is a distinct IRG value and the X-axis shows the index of each of the successive IRG values in the IRG stream.

All IRG streams, in all our traces showed similar characteristics, i.e. (a) a multimodal envelope of the distribution, (b) certain IRG values never occur (vertical gaps in the histogram plots), and those that do occur form a small fraction of the possible IRG values, (c) a high degree of skew in the frequencies, and (d) high correlation among successive IRG values. Observations (a), (b) and (c) are derived from figure 1 and (d) from figure 2. We now address the question of what we aim to achieve by studying IRG streams of a program execution.

First, IRG stream modelling isolates temporal locality from spatial locality. This is because it ignores the affect of other addresses and looks only at the past behavior of a particular address. Analysis of all the IRG streams in a trace will give all the information there is, about temporal locality of the whole trace. This has direct impact on memory replacement and deallocation algorithms.

Second, we expect a small fraction of all the IRG streams to capture the temporal behaviour of the entire trace. This is due to the fact that memory references are correlated, and a very small subset of addresses get referenced most of the time. Hence a few IRG streams can approximate the whole trace. This is useful in trace compaction and speeding up of trace driven simulations of memory management algorithms.

IRG stream modelling can provide a way to capture what we call *inter-cluster locality*. Addresses that are spatially far apart show correlation in certain cases. For example, between the code and the data address spaces, which are spatially disjoint, there is a direct correlation between an instruction word and the data memory word that is fetched upon its execution. Neither spatial locality nor temporal locality can capture this behavior, but by finding a correlation between different IRG streams we can model this property automatically. This can be utilized for improving prefetching algorithms, e.g. Chen and Baer [7] improved prefetching by just using the correlation between an instruction and its operand.

Changes in IRG stream behavior can be used to signal phase changes in a program. Intuitively speaking, a visible change in the IRG patterns of the frequently accessed variables, usually implies a global behavioral change. For example, consider the execution of a loop in a program, where a loop index is accessed every time at the top of the loop. While continuously looping, if a switch happens from rapid accesses (small values of IRGs) to infrequent accesses (large values of IRGs) to the loop index, this will imply that either the number of variables accessed inside the loop body has increased, or the same variables are getting accessed in a different pattern, inside the loop. In

either case, it is a shift in the program behavior. If such phase changes are detected early enough via IRG modelling, then they can be applied to prefetching and avoiding *cold* misses at the onset of new working sets.

Lastly, in certain cases IRG streams are the only way to find performance related parameters. For example, in a distributed system, because of lack of knowledge of the global snapshot, we can only monitor each object separately. For example, we can only record the time instants a particular resource is accessed, which is nothing but the IRG stream of that particular resource.

3 Previous Work on Modelling of Program Locality

Most of the work in modelling temporal locality can be classified into two broad categories. First are analytical models which are tractable and yield interesting results, but their precision is questionable. Other program models are more empirical and they try to capture some behavioral characteristics of a program. We discuss both of them, and try to show why they are inadequate for modelling IRGs.

3.1 Analytical Modelling

The simplest mathematical model is the independent reference model (IRM). In this model, each address has a fixed reference probability and references are mutually independent. In other words, the string of references is modeled as a sequence of i.i.d. random variables. King [8], Aven et al [9], Rao [10], among others, have used this model to study performances of replacement algorithms and get closed-form expressions for the miss ratios. In order to use this model for IRG modelling, consider address i. Assuming i is accessed at time t, the probability that it will

Figure 1 IRG distribution of three addresses in the CC1 trace.

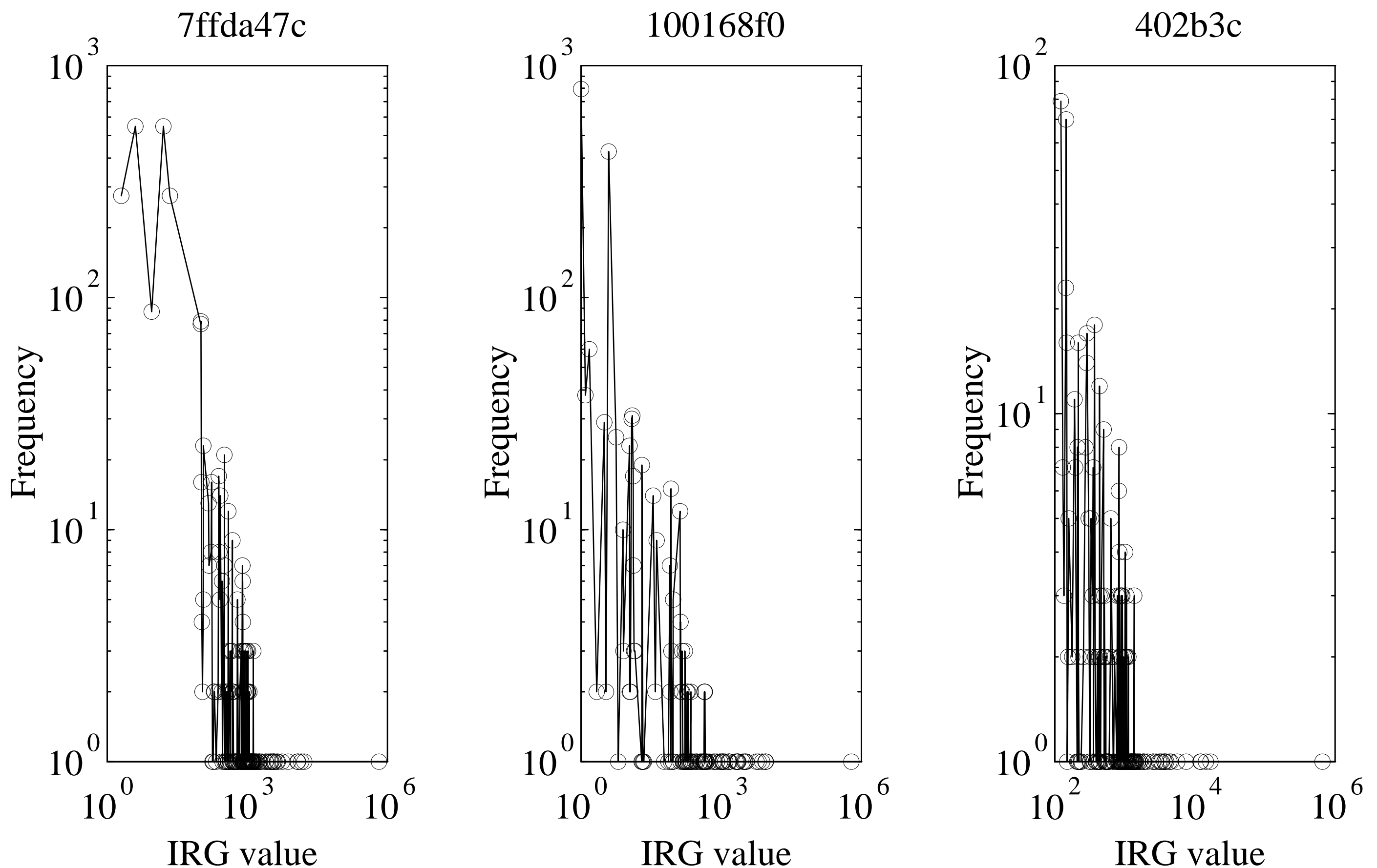
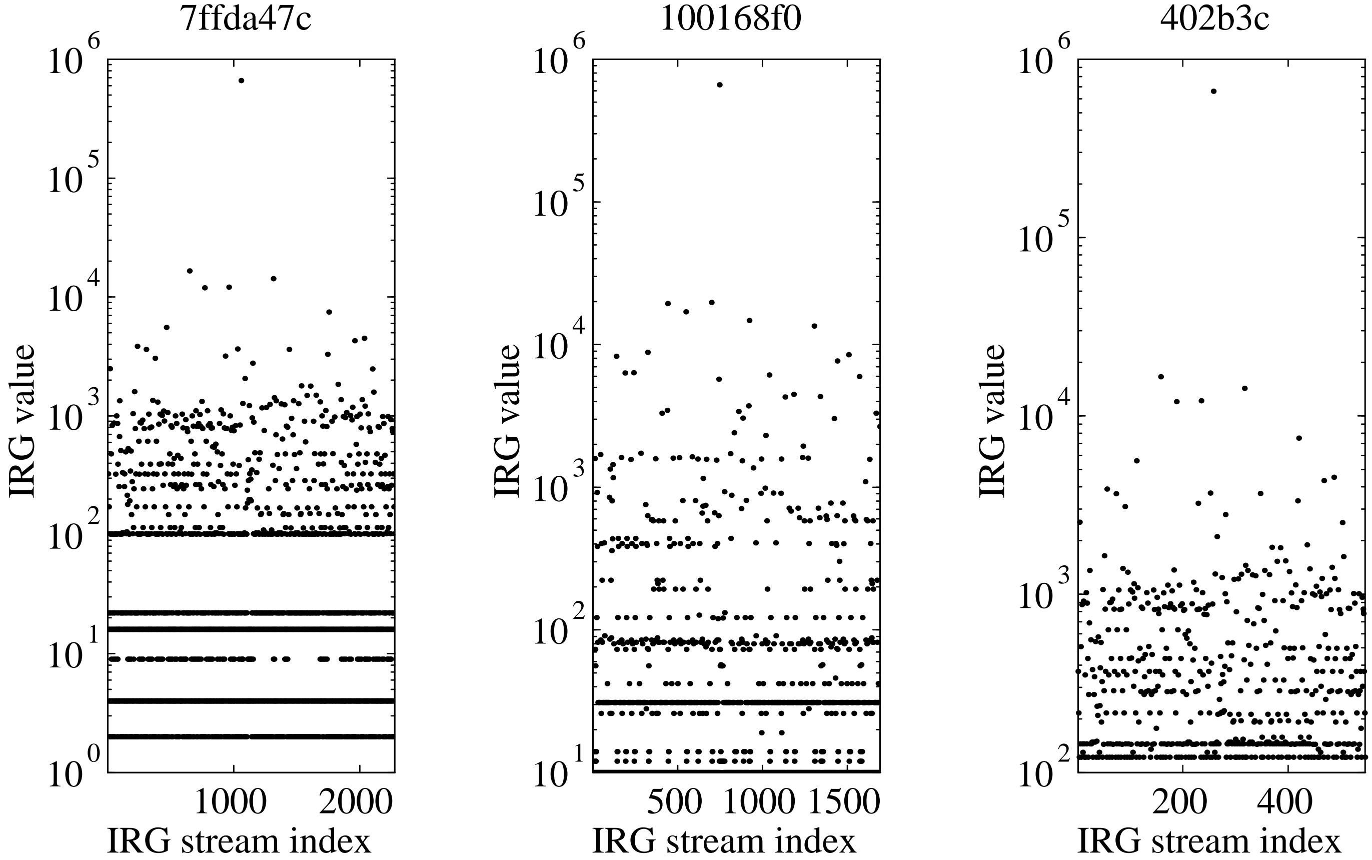


Figure 2 IRG strings of three addresses in the CC1 trace.



be accessed next at time t+k is $Pr(IRG_i = k) = p_i(1-p_i)^{k-1}$. This implies that in all IRG streams, every IRG value has a finite probability of occurrence. In addition, IRG values in a stream are independent of each other and have a unimodal distribution. Spirn's [1] generalized locality model (GLM), also has the same drawbacks because it is made up of locality phases, each of which is an IRM. Thus, IRM based techniques are inadequate for capturing any of the temporal characteristics shown in section 2.

Opderbeck and Chu [11], proposed a renewal model for program behavior. They modelled inter-reference gaps using continuous distributions which decay exponentially with time. In other words, the longer an address remains unreferenced, the smaller its probability of reference becomes. This will give a nonincreasing IRG value distribution, again not agreeing with our observations.

The stack model, introduced by Mattson et al [12] and its derivatives [13, 1, 14] try to capture temporal locality by generating reference strings via a probabilistic access to an LRU stack. If we look at the IRG streams in this model, all of them have the same behavior in the asymptotics. Second, each of the successive IRG values are independent and each of them can possibly take on any value. Finally, if the stack probabilities are nonincreasing, the IRG distribution will also be nonincreasing. None of these properties agree with our observations.

Stochastic models, introduced by Franklin and Gupta [15], model program behavior as a probabilistic transition matrix. As long as there is exactly one node per address in the transition graph, we will get independent successive IRG values. On the other hand, if we have program transition graphs [15], we can get IRG streams which might agree with our observations. But transition graphs are derived from the programs themselves, and not from the traces. So in order to build an IRG model in such a situation, first a transition graph will have to be derived from the trace, which is similar to inferring a Markov chain from its output. This is an open problem in the area of Information Theory [16], hence not applicable for IRG modelling.

3.2 Empirical Modelling

Almost all empirical models which are geared for capturing temporal locality do not focus on each address separately. They see addresses as sets and try to model behavior of those entire sets. Thus those models are at a "macro-level" as compared to ours which is at a "micro-level".

Madison and Batson [17, 2] proposed an LRU stack based model called the bounded-locality-interval (BLI) model. It defines temporal locality as a series of hierarchies S_k using the time periods during which the top k addresses of the LRU stack remain unchanged. Since only the durations of no-change are modelled and address specific information is ignored, IRG modelling can not be extrapolated from this scheme.

Denning's working set [3] models temporal behavior using a threshold T. Temporal locality is represented as a two state model where an address is either in the memory or it is not. The former occurring when there is at least one reference to this address in the last T memory accesses. This is a very simple approximation which "forgets" an address's IRG behavior once it is not referenced in the last T accesses.

Chow's power law [18] and its extension by Thiebaut and others to fractal behavior [19, 20] characterize temporal locality at a macro level. Chow proposed that the miss ratio of a finite cache almost universally obeys the rule $m = A \times c^{\theta}$ where m is the miss ratio, c the cache size, and A and θ are constants. Thiebaut et al extended this idea to model program behavior as a fractal random walk over a one dimensional lattice (the memory), with the jumps having a hyperbolic distribution. Singh et al [21] also model temporal locality using a power law. Although these ideas provide models which can be completely specified by a small set of parameters, they can not describe the behavior of the IRG streams, making them irrelevant in this discussion.

Choi and Ruschitzka [22] model database behavior as a sequence of phases. Each phase is denoted by a set-duration pair (L_i, τ_i) where L_i is a set out of which τ_i references are made in the ith phase. This is similar to Spirn's GLM mentioned above and hence has the same drawbacks for modelling IRGs. In addition, reference behavior within a phase is not modeled, so specific timing information for a particular address is unknown.

A model proposed for databases by Easton [23] models each IRG stream individually. Each IRG stream is modelled as a two mode exponential distribution, i.e. an IRG takes a value from one of the two distributions depending on which mode – "cluster-mode" or "gap-mode", the address is in. Although more powerful than IRM, all it does is split IRM into two modes, and hence has the same modelling drawbacks as the IRM.

4 IRG Model and Prediction

In this section we formally present our IRG model and explain how it is used for future reference estimation. We also present the correlation between data compression algorithms and our prediction techniques

Consider the IRG stream of an address a in a program execution P. Call it $IRG_P(a)$. If address a gets referenced at virtual times t_1, t_2, t_3 and so on, then,

$$IRG_P(a) = X_1\, X_2\, X_3\, \ldots \text{ where } X_i = t_i - t_{i-1},\ t_0 = 0$$

Each of the gap values, X_i, is treated as a symbol generated from an unknown source $IRG_P(a)$. These X_i's take on values in the range $[1,\infty)$, although in a trace of length T, the largest IRG value possible is T. Also, in a finite trace, we ignore the last access of an address because the IRG following that last access is unknown.

We model $IRG_P(a)$ for each a, as a kth order Markov chain. i.e.

$$Pr\{X_t = x_t | X_i = x_i,\ 1 \leq i \leq t-1\} \ = \ Pr\{X_t = x_t | X_i = x_i,\ t-k \leq i \leq t-1\}$$

Thus, X_t is dependent on the last k IRG values, and each distinct k tuple $<X_{i1}X_{i2}...X_{ik}>$ forms a state in the Markov chain. To estimate X_t, given all the past X_i's ($1 \leq i \leq t-1$) we use a frequency count argument over Markov chains of all orders from 0 to k.

Let the current observed $IRG_P(a)$ be $X=X_1X_2...X_{t-1}$. A substring X_p^q is the sequence of symbols occurring in the positions $X_pX_{p+1}...X_q$ ($1 \leq p \leq q \leq t-1$) of X. We say X_p^q *occurs* at position j in X, if X_j^{j+q-p} matches X_p^q symbol by symbol ($1 \leq j \leq t-1-(q-p)$). The level z predictor ($0 \leq z \leq k$) works assuming a zth order Markov chain.

Level *z* predictor: We estimate the probability of the next symbol X_t being x, as the fraction of times symbol x occurred following the substring X_{t-z}^{t-1} in X_1^{t-2}. Let N_{t-1} be the number of occurrences of substring X_{t-z}^{t-1} in

X_1^{t-2}. Let m_x be the number of occurrences of substring $X_{t-z}^{t-1}+x$ ($+$ denotes concatenation) in X_1^{t-1}. Then $Pr\{X_t = x | X_i = x_i,\ 1 \le i \le t-1\}$ is estimated by

$$\widehat{Pr}\{X_t = x | X_i = x_i,\ 1 \le i \le t-1\} = \frac{m_x}{N_{t-1}}$$

where N_{t-1} is assumed to be non zero. Otherwise level z predictor is undefined.

So the level 0 predictor assumes $IRG_P(a)$ to be an i.i.d. source, and the level 1 predictor is a standard Markov chain. The motivation behind these multiple layers of predictors is to have a system which can make a "good" guess even when the kth level predictor fails. Failure of a level k predictor can happen in case X_{t-k}^{t-1} never *occurs* in X (N_{t-1} is zero). It can also happen that we "learn" some information about X_t which does not "agree" with the level k predictions, e.g. we might "learn" that X_t will be none of the symbols with nonzero probability estimates at level k. In such a case, we will switch to level $k-1$ for prediction, and recurse to lower levels if needed.

Readers familiar with PPM data compression techniques [24] will notice a similarity between them, and our estimation method. The difference is that, unlike PPM, at times, we can "learn" that a certain IRG value will not occur even before it is completely known, and hence can switch to a lower level predictor. For example, supposing level k predictor for $IRG_P(a)$ estimates X_t to be one of the values – { 2, 8, 12 } (say), with some finite probabilities. Now, if the time since the last reference to a is already greater than 12, then we "know" that the level k estimator will fail, so we can switch to the level $k-1$ predictor.

Example: We give an example to illustrate our model and the prediction method. Consider the following page reference string "*bcaababbaccacabcabacda*". Page a is referenced at times *3, 4, 6, 9, 12, 14, 17, 19, 22*. The IRG string for a is thus, $X_1^9 =$ *3 1 2 3 3 2 3 2 3*. For the level 2 predictor, we look at the past occurrences of the two most recent IRG values (*2 3*). This gives us the following probability estimates:

Level 2: $\widehat{Pr}\{X_{10} = 2 | X_8 = 2, X_9 = 3\} = 0.5$, $\widehat{Pr}\{X_{10} = 3 | X_8 = 2, X_9 = 3\} = 0.5$

Level 1: $\widehat{Pr}\{X_{10} = 1 | X_9 = 3\} = 0.25$, $\widehat{Pr}\{X_{10} = 2 | X_9 = 3\} = 0.5$, $\widehat{Pr}\{X_{10} = 3 | X_9 = 3\} = 0.25$

Level 0: $\widehat{Pr}\{X_{10} = 1\} = 0.11$, $\widehat{Pr}\{X_{10} = 2\} = 0.33$, $\widehat{Pr}\{X_{10} = 3\} = 0.55$

5 IRG based Memory Replacement Algorithm

In this section, we present the first application of our IRG model which is to improve memory replacement algorithms. We first describe the related work in this area, then our algorithm, followed by simulation results. At the end of this section we describe a page replacement algorithm which uses an approximation of the IRG model and is also practical.

5.1 Introduction

In the steady state of process execution, the higher level of memory is full, and a miss implies not only a fetch but also a replacement; an address must be removed from the higher level. The address to be replaced is decided by what is called the *replacement algorithm*. Various studies of memory reference models and simulations of program traces have been done to determine a good replacement algorithm. Belady [25] proposed a forward distance based optimal algorithm, called OPT or MIN, for replacement in a fixed memory scenario. It works under the assumption that all the future references are known beforehand. Whenever an address needs to be replaced, the algorithm finds out the one that is referenced farthest in the future (out of those in the memory), and replaces that one. If an address won't be referenced ever in the future then its future reference time is assumed to be at ∞. So the forward distance of an address x in reference string $r_1, r_2 \ldots r_t \ldots$, at time t is defined as:

$$d_t(x) = \begin{cases} k \text{ if } t_{t+k} \text{ is the first occurrence of } x \text{ in } r_{t+1}, r_{t+2}, \ldots \\ \infty \text{ if } x \text{ does not appear in } r_{t+1}, r_{t+2}, \ldots \end{cases}$$

Thus, the address with the largest d_t value is replaced. Previous prediction based techniques for replacement used heuristics, in a loose way, to pinpoint addresses that need to be retained, and those that can be replaced. We use our temporal locality models to predict forward distances more precisely and apply them to memory replacement algorithms. We validate our model using a variety of samples from cache traces, page reference traces, and CAD / database traces. The principles of predictability, which we propose, in general, hold at all these levels of memory hierarchy.

5.2 Related Work

All classic replacement algorithms try to estimate the address with the longest forward distance, using some information from their past behavior. Forward distance of an address is the number of time units, from the current time, when that address will be referred to next. This is done because Belady's MIN algorithm (also called OPT in the literature), which is off-line optimal for the number of misses for a fixed size memory, replaces the address with the largest forward distance.

LRU estimates that the address with the longest backward distance (analogously defined like the forward distance) has the largest forward distance. LRU-K [26], estimates the address with the kth earliest reference to be the one with the largest forward distance. (Note – LRU-1 is the same as LRU). Least frequently used (LFU) replaces the address with the smallest number of references. This is the same as estimating the forward distance by averaging all the IRGs of the past. First in first out (FIFO), uses the time since the arrival as an estimate for the forward distance. Other replacement algorithms like A_1^m, CLIMB [9] and frequency based replacement (FBR) [27] use an underlying stack, which implies an LRU kind of forward distance estimation. Only random replacement (RR) does not try to estimate the forward distance. It works on the principle that a random replacement will rarely throw out a frequently used address because they are in a very small number.

5.3 IRG Replacement Algorithm

Assume that the memory can hold only `M` addresses (an address, as mentioned before, could be a cache block, a page or a data object depending on the context) at a time. For each address, we maintain IRG stream information as will be needed by the underlying predictor. Upon reference to an address *x* at time t_{now}, assuming *x* was referred to last at time t_{prev}, we get the new IRG symbol $t_{now}-t_{prev}$ for *x*'s IRG stream. Procedure `access()` is invoked every time a memory access is made. If the requested address *a* is found in memory, a *hit* occurs, otherwise it is a *miss*. When a miss occurs, procedure `access()` invokes another routine `estimate_farthest()` to find the address with the highest forward predicted distance. If the process of estimation does not succeed (there are various reasons for that, which we explain later), the least recently used address is replaced. Otherwise, the address with the largest predicted forward distance is replaced. In addition, upon access to *a*, the latest IRG symbol of *a*'s IRG stream is generated, which is taken care of by the `update_irg_stream()` procedure. Figure 3 has the pseudo code.

Figure 3 Pseudo code for the IRG replacement algorithm.

```
PROC access(address a, memory M)
     update_irg_stream(a);
     IF(a not in M)THEN
          x = estimate_farthest(M);
          replace x by a;
     ENDIF
     bring a to TopOfStack of M;
     RETURN a;
ENDPROC
```

```
PROC estimate_farthest(memory M)
     max = 0; pmax = NULL;
     FOR each x in M DO
         y = estimate_forward(x);
         IF(y == FAIL)THEN
             RETURN LRU(M);
         ENDIF
         IF(y > max)THEN
             max  = y;
             pmax = x;
         ENDIF
     ENDFOR
     RETURN pmax;
ENDPROC
```

The procedures `update_irg_stream()` and `estimate_forward()` are dependent upon the order *k* of the underlying model. When `update_irg_stream(x)` is invoked, a new IRG symbol is added and it updates frequency counts for all the level *z* predictors ($0 \le z \le k$). Figure 4 has the pseudo code for these subroutines.

Figure 4 Pseudo code for the IRG model update and the prediction subroutines.

```
PROC update_irg_stream(address a)
/*S1S2...Sv-1 be a's current IRG stream.
 *Sv be the new IRG symbol added.
 */
     FOR u=k to 0 DO
         Count[Sv-u...Sv-1,Sv]++;
     ENDFOR
ENDPROC
```

```
PROC estimate_forward(address a)
/*S1S2...Sv be a's current IRG stream.
 *G be the current gap i.e. the time
 *since last reference to a.
 */
     FOR u=k to 0 DO
        find d,(d > G) which has the
        highest frequency count among
        Count[Sv-u...Sv-1,D] ;
        IF(such d is found)THEN
            RETURN d-G;
        ENDIF
     ENDFOR
     RETURN FAIL;
ENDPROC
```

Array `Count[C,s]` maintains frequencies of symbols occurring after substring `C`. It takes two parameters, a *context* (`C`) and a symbol (`s`). `C` is a sequence of symbols, following which `s` occurred. `C` is NULL when `u` is 0 in the above procedure. Procedure `estimate_forward()` uses level z predictors of all orders from $z=k$ to $z=0$, till it finds an IRG symbol with value greater than the current gap. If nothing appropriate is found, it returns a FAIL.

This technique requires frequency counts for all possible context-symbol pairs, for all contexts of length 0 to k. A context tree, as defined in [28] is used to keep these counts. The tree has k levels and the number of children per node is at most i, where i is the number of distinct symbols in the IRG stream. At each node a frequency table of size at most i is maintained, making the space requirement $O(i^{k+1})$. At each `update_irg_stream()` k frequency counts are incremented and a pointer set at the appropriate leaf at level k. Hence the process of `estimate_forward()` involves only a search in the frequency tables along a path from a leaf to the root. We only deal with models of order smaller than three in our simulations, in which case space is not prohibitive.

5.4 Simulation Traces

We obtained traces at different levels of the memory hierarchy to demonstrate the validity of our IRG modelling. ATUM traces were obtained from the DINERO suite [29]. Each trace is approximately a half-second snapshot execution on a machine the speed of a VAX-11/780 and contains virtual memory references. KENS is a trace of a SPEC benchmark, collected on a SPARC 1+ from the BACH-BYU suite [30]. This is also a virtual memory trace. Page reference traces were obtained from a CAD tool written at Digital's CAD/CAM Technology Center. These contained page numbers of successive references. Object reference traces (sequence of accessed UIDs) were obtained from the Object Operations Benchmark (OO1), OO7 benchmark of University of Wisconsin and the same CAD tool as the one for the page traces. Table 1 gives a detailed description of all the traces.

Table 1: Trace Description.

Name	Description	Trace Length (in thousands)	Total unique references	
			Number (in thousands)	Normalized by trace length (%)
CC1	Gnu C compilation	1000	43.1	4.3
DEC0	DECSIM, a behavioral simulator at DEC, simulating some cache hardware	362	18.8	5.2
FORA	FORTRAN compilation	388	20.8	5.4

Table 1: (Continued) Trace Description.

FORF	Another FORTRAN compilation	368	30.1	8.2
IVEX	DEC Interconnect Verify, checking net lists in a VLSI chip	342	37.0	10.8
LISP	LISP runs of BOYER (a theorem prover)	291	5.95	2.0
MUL8	VMS multiprogramming at level 8 : spice, alloc, a Fortran compile, a Pascal compile, an assembler, a string search in a file, jacobi and an octal dump	429	33.1	7.7
PASC	Pascal compilation of a microcode parser program	422	14.2	3.4
SPIC	SPICE simulating a 2-input tri-state NAND buffer	447	9.2	2.1
SPICE	Another SPICE simulation	1000	15.3	1.5
TEX	Text formatting utility	817	38.2	4.7
UE02	Simulation of interactive users running under Ultrix	358	31.6	8.8
BACH-BYU				
KENS	Kenbus1 SPEC benchmark simulating 20 users	4372	160.8	3.7
CAD page references				
CAD1P	Graphical display of a DEC CAD tool doing circuit design using ICs	74	1.67	2.3
CAD2P	A longer session of CAD1P	147	1.67	1.1
SALEMP	A CAD tool trace	50	0.16	0.3
Object references				
OO1F	OO1 database benchmark running on DEC Object/DB system with forward traversal of relations	11.7	0.52	4.4
OO1R	OO1 database benchmark with reverse traversal of relations	11.7	0.53	4.5
OO7T1	OO7 benchmark running on DEC Object/DB product doing query traversals	28.1	6.0	21.4
OO7T3A	Another traversal trace like OO7T1	30.1	6.3	20.9
CAD2O	UID reference trace in CAD2P above	147	15.4	10.5

5.5 Description of Experiments

We did our simulations with the 0th and the 1st order predictors, labelled as IRG0 and IRG1 in the plots. For comparison purposes we also simulated least recently used (LRU) and the off-line optimal algorithm (OPT).

With the ATUM traces and the KENS trace, which were main memory references, we simulated a fully associative cache with block size of 4 words. The IRG modelling was done with respect to the block references rather than each memory word having its own IRG model.

For the DEC0 trace we also simulated the 2nd order predictor (IRG2). In addition, we compared the performance of IRG algorithms with the LRU-K algorithms [26], for K equal to 2 and 3. We present these results in a chart (table 2) for the sake of clarity.

Table 2: Miss ratios for DEC0 trace under a fully associative cache.

Algorithm	Cache Size (number of words)						
	512	1024	2048	4096	8192	16384	32768
LRU	0.4290	0.3434	0.2861	0.2161	0.1415	0.0638	0.0453
LRU-2	0.4532	0.3752	0.3093	0.2358	0.1392	0.0839	0.0537
LRU-3	0.4626	0.3839	0.3088	0.2226	0.1465	0.0964	0.0509
IRG0	0.3860	0.3199	0.2653	0.2042	0.1415	0.0638	0.0453
IRG1	0.3804	0.3152	0.2619	0.2032	0.1415	0.0638	0.0453
IRG2	0.3780	0.3148	0.2612	0.1943	0.1348	0.0638	0.0453
OPT	0.3125	0.2455	0.1881	0.1302	0.0752	0.0484	0.0397

For rest of the ATUM and the KENS traces figure 5 has the miss ratio plots for the OPT, LRU, IRG0 and IRG1 algorithms. The cache size (in number of memory words) is on the X-axis and the Y-axis has the miss ratio.

Two important features stand out in these plots. First IRG1 is only marginally superior to IRG0. In fact, in some cases it performs worse than IRG0. The main reason for this is that it adapts at a slower rate to a drastic change in an IRG stream than does IRG0. Thus, when some IRG stream changes drastically, IRG1 makes more incorrect predictions than IRG0.

Second, for larger cache sizes, IRG0 and IRG1 tend away from OPT towards LRU. The main reason for this is the inability of IRG0 and IRG1 to predict, due to lack of information. When the cache becomes larger, more and more blocks with very few references (one or two) are present, so the predictors return a FAIL, most of the time. In this case we replace the least recently used block. On the other hand, in a smaller cache, all the blocks present have a long IRG history, making a good prediction possible.

On a side note, the reason why LRU-K performs poorly is that it assumes an Independent Reference Model as the underlying program model. In practice this is not true since our algorithms, which assume a discrete and predictable IRG stream, perform better.

Figure 5 Miss ratio comparison in a fully associative cache.

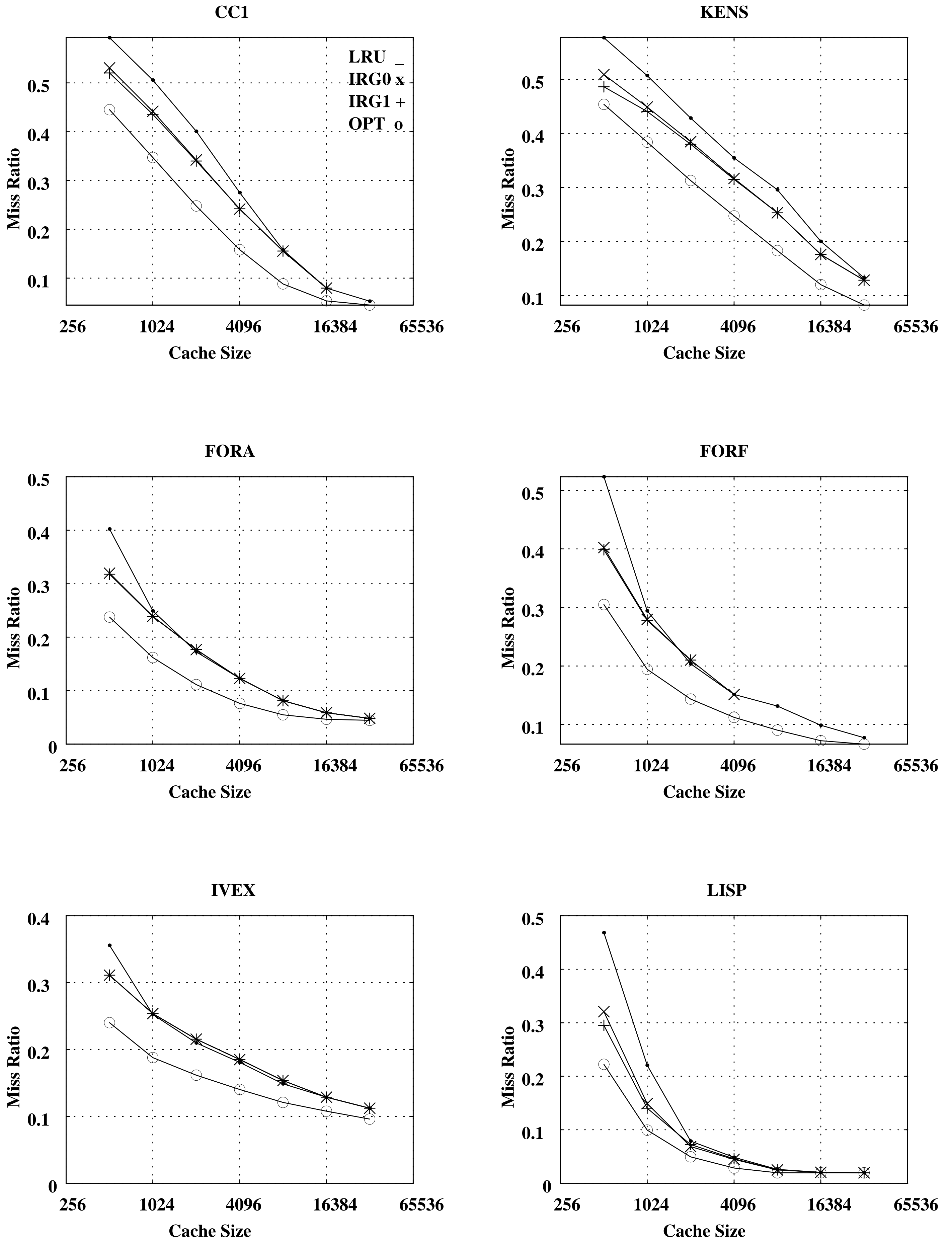
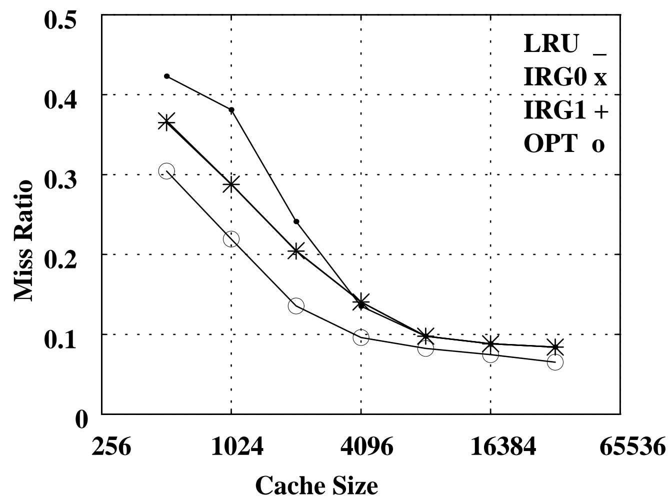
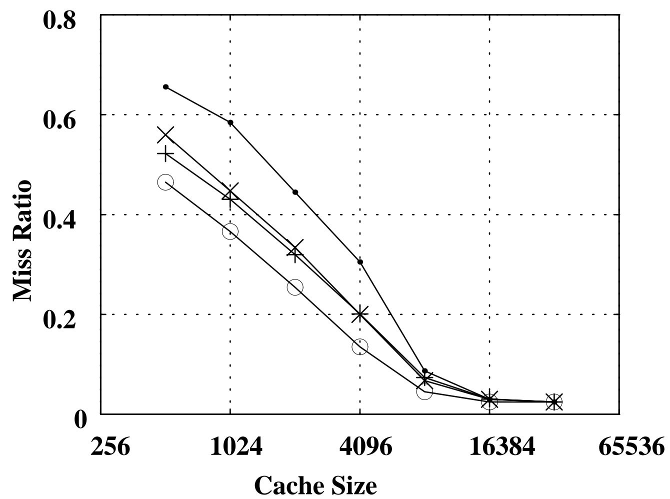
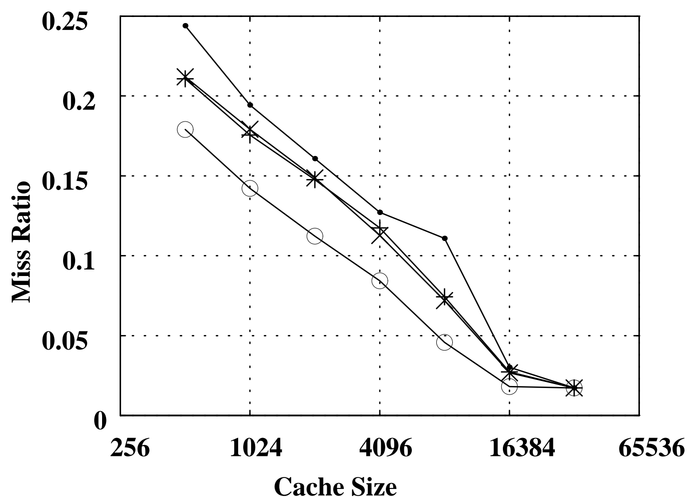
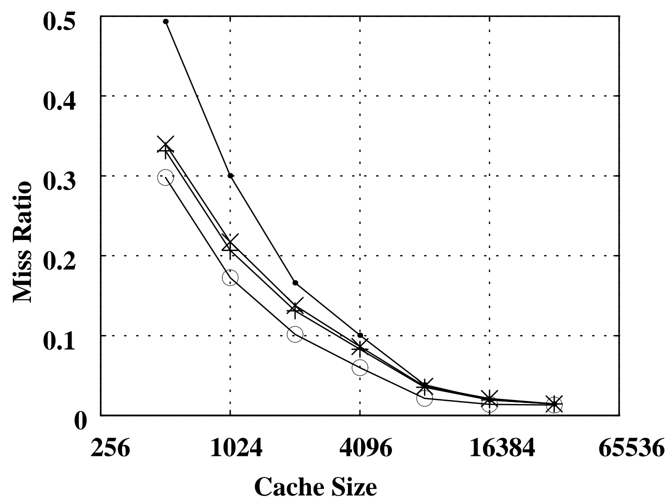


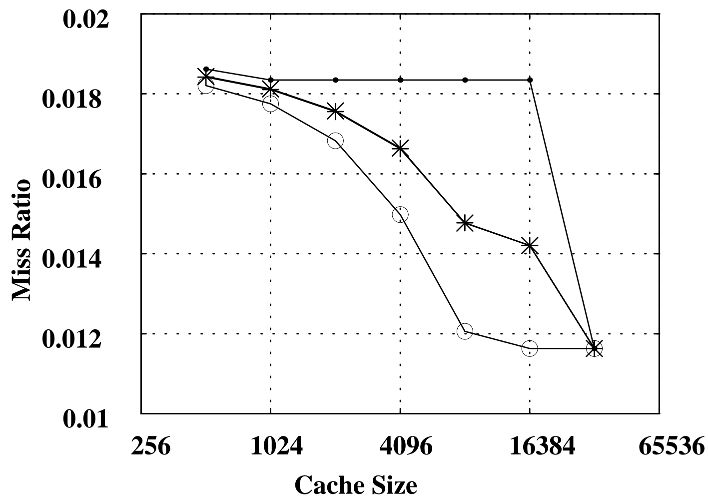
Figure 5 (Contd) Miss ratio comparison in a fully associative cache.

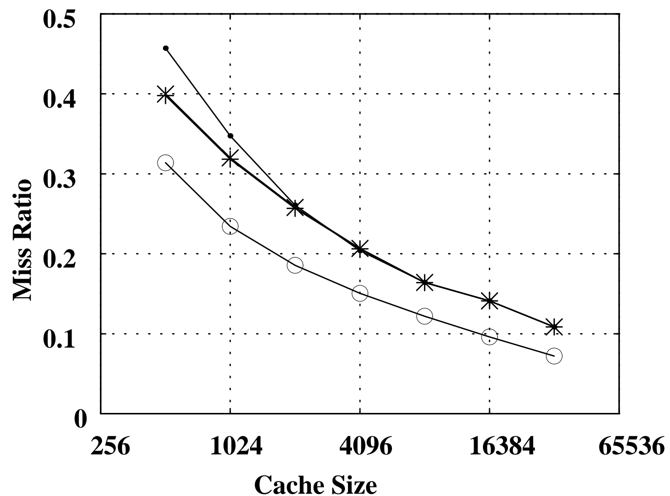












We also simulated a 4-way set-associative cache 4 word block size, for CC1, DEC0 and FORF t Here also, the IRG model was per block. In add time was virtual with respect to each set, i.e. the reference gap for a particular block was measured only the references made to the set to which this belonged to. We only compare LRU and IRG0 i case. Notice the miss ratios are very similar to the associative cases. Figure 6 has these plots.

We simulated a paged memory environment for the page reference traces and applied our IRG algo for replacement. Figure 7 shows the comparison LRU, IRG0 and OPT algorithms. The X-axis h size of the memory in number of pages. Notice although LRU does not have a "smooth" curve, does, because it "mimics" OPT more accuratel LRU.

Finally, in figure 8 we show the miss ratio plo Object reference traces. The size of the mem expressed in the number of objects it can hold. also, the slant of IRG0 follows OPT much more c than LRU.

Figure 6 Miss ratio in a set associative cache.

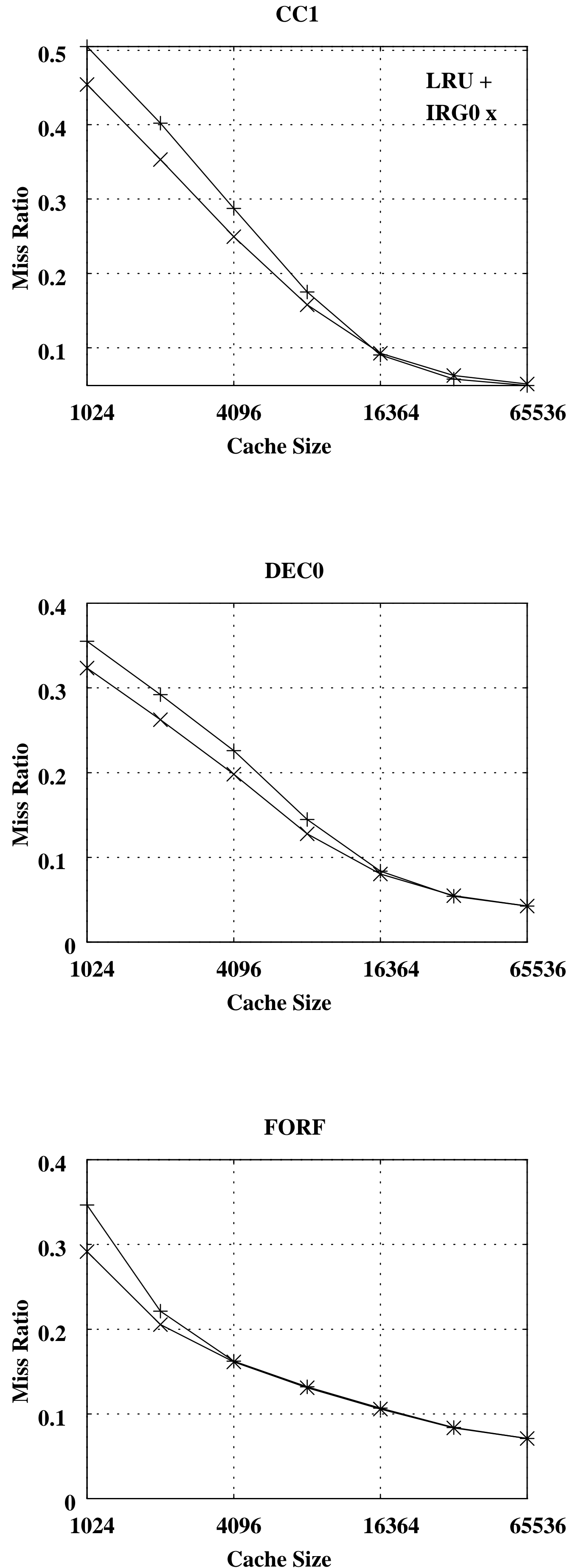
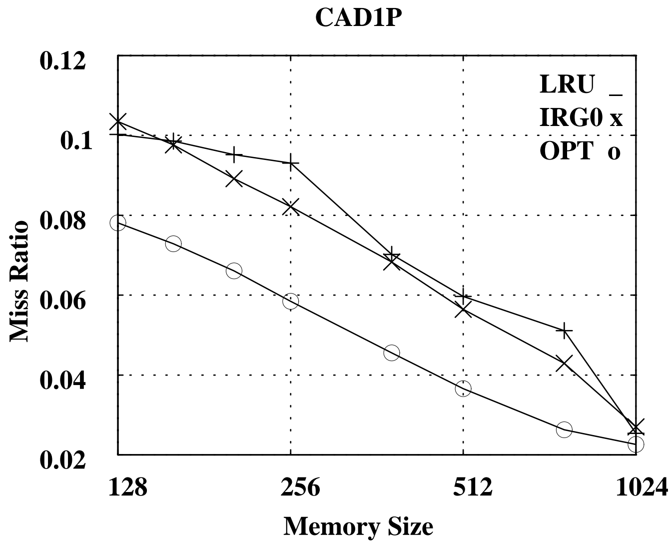
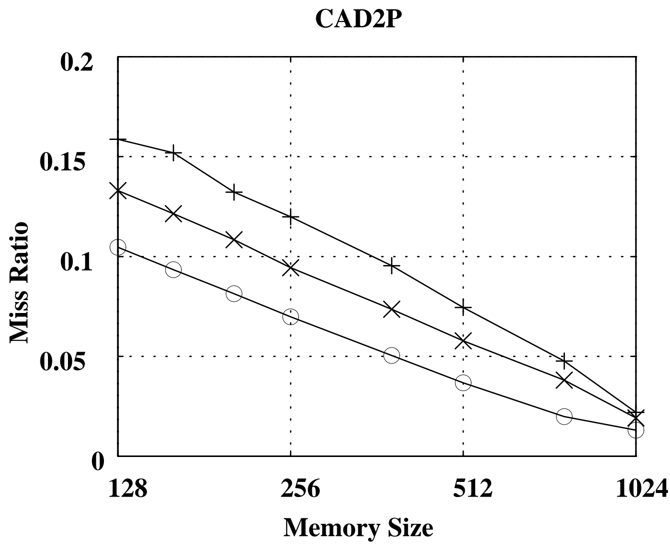


Figure 7 Miss ratio in CAD page reference traces.





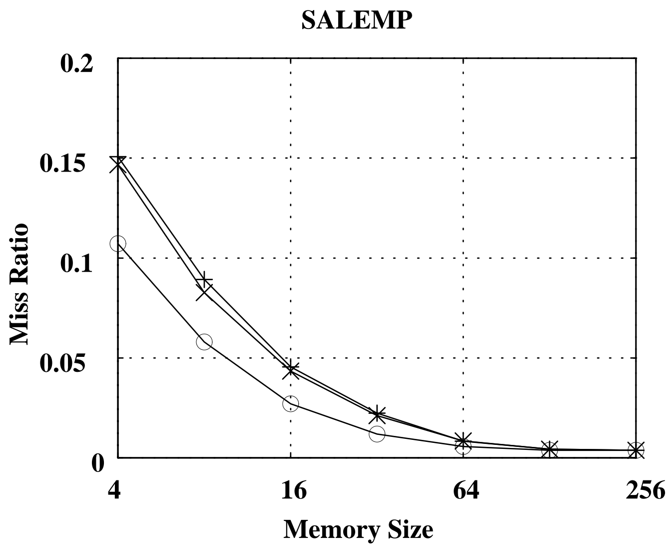
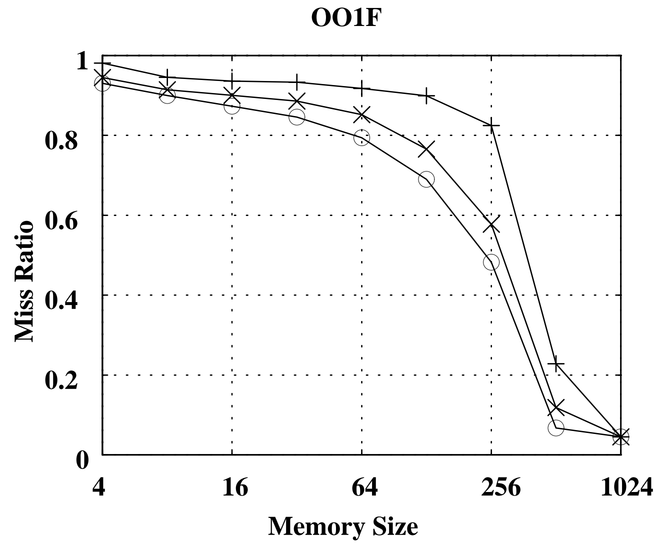
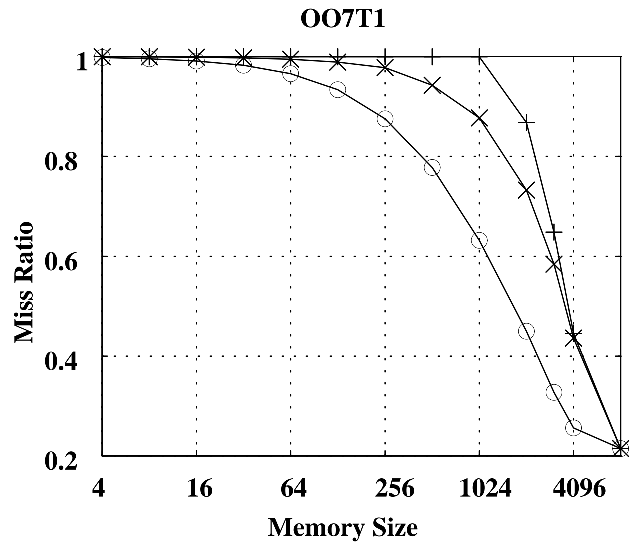
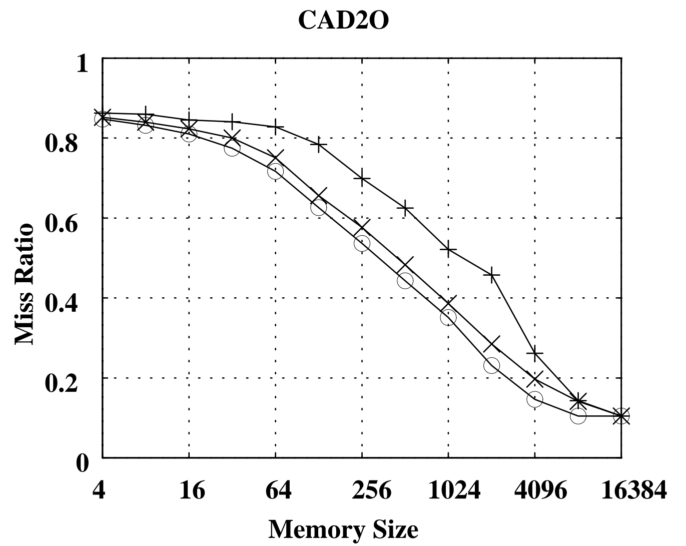


Figure 8 Miss ratio in Object reference traces.







5.6 Implementation Overheads

The replacement decisions using the IRG strategy have large time and space overheads. An IRG model has to be maintained for each one of the referenced addresses. In addition, at every access the IRG model of the referenced address has to be updated. On the prediction side, at each miss, each of the IRG models have to be queried to predict the address with the farthest expected reference.

Table 3 describes the space-time overheads for some of our *simulations* from figure 5 for PASC and SPIC traces. We simulated a fully associative cache with 4 word blocks. We normalize IRG time with the time taken for the LRU simulations. Absolute time taken by the IRG methods decreases with cache size, because a larger cache implies a smaller number of misses and hence a fewer number of replacement decisions. The time here is the simulation time and should not be mistaken for the cache access time. These numbers merely depict the overheads of IRG methods over LRU. The space shown is the average number of words needed per IRG model. This space is not always needed because once an address is replaced, its IRG model can also be removed from the memory.

Table 3: IRG simulation overheads for PASC and SPIC traces.

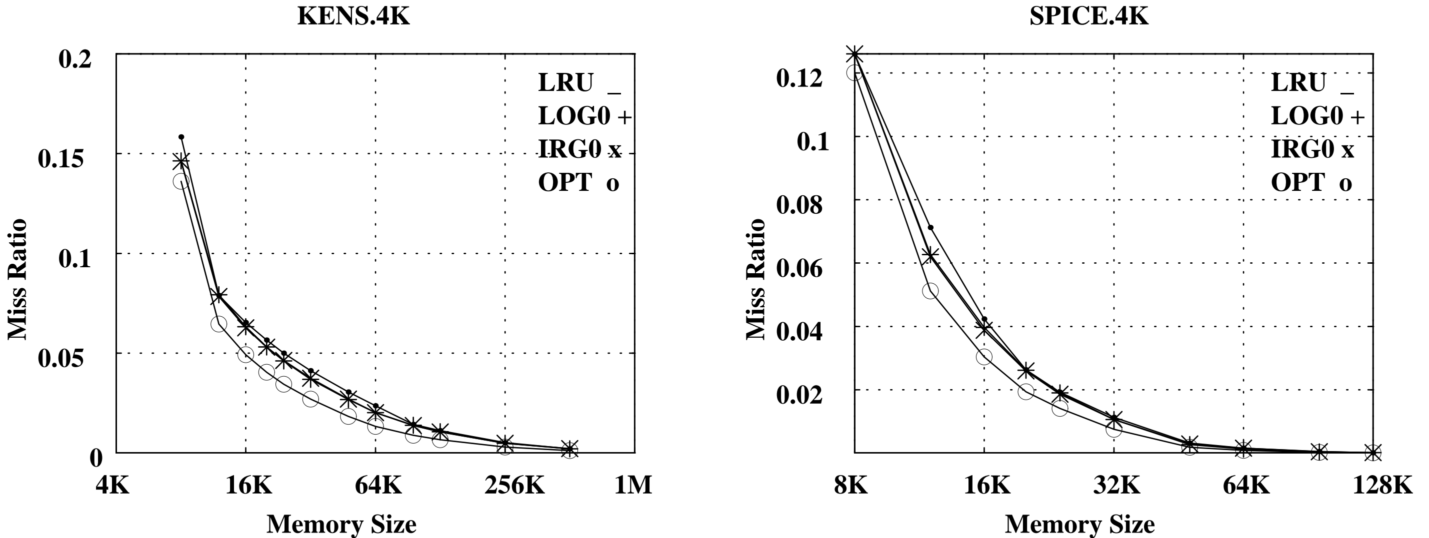
	IRG0		IRG1	
Cache Size	Space per IRG model (in words)	Simulation time relative to LRU	Space per IRG model (in words)	Simulation time relative to LRU
PASC simulation				
2 K	38.8 Independent of cache size	5.15	1967.3 Independent of cache size	127.63
4 K		4.21		134.34
8 K		2.55		85.66
16 K		1.13		35.83
SPIC simulation				
2 K	29.9 Independent of cache size	3.45	873.1 Independent of cache size	44.65
4 K		3.62		29.59
8 K		2.50		14.48
16 K		1.13		3.03

5.7 A Practical Implementation

As observed in our experiments, order 0 model achieves improvements up to 35% over the LRU miss ratio. In order to implement a replacement algorithm with the order 0 predictor, we need to keep frequency counts of all possible IRG values that occurred in the past for each of the addresses. In addition, at each replacement decision, prediction needs to be done for each of the resident addresses. Both of these tasks make it impossible to have a practical solution using 0th order IRG models. We consider some approximations that can be done and describe their behavior using trace driven simulations.

Space reduction: First we address the storage issue. If counters for each IRG value are kept, we will need space proportional to the number of different IRG symbols that occur. This will imply a very low space requirement for the rarely referenced addresses. But this argument will not hold when memory is small and most of the addresses in the memory are the highly referenced ones, implying a large overall space requirement. To circumvent this problem, we can approximate IRG values. We cannot do a simple divide operation to approximate the IRGs because small IRG values are important in modelling loop behavior etc. On the other hand, a large enough IRG value will usually make an address, a candidate for replacement, so two large IRG values can be approximated by one. A simple strategy will be to approximate an IRG value by its logarithm, i.e. approximate IRG g by $2^{\lceil log(g) \rceil}$. Figure 9 shows the effect of approximating IRG using the logarithmic scheme. Replacement decisions are made for a paged memory system. Each page is 4 kbytes in size. The X-axis shows the main memory size in bytes. LOG0 is the log (base 2) IRG0 approximation. The two traces were originally virtual memory references which were converted to page reference traces.

Figure 9 Miss ratio comparison of $\log_2$ IRG approximations in page reference traces.



For the SPICE trace there were 64 unique pages referenced in the trace. IRG0 used 166.3 words on the average per IRG0 model. On the other hand LOG0 used only 23.2 words per model. Similarly the KENS trace had 870 unique page references, with 191.3 words per IRG0 model. The logarithmic model used 21.2 words per model in this case. Moreover, the number of bits needed to code logarithmic IRG values are even smaller. Another observation is that IRG0 performs only marginally better than LOG0. So, for these kinds of numbers, a simple implementation is to keep about 100 bytes reserved in each page (each page being 4 kbytes) for the LOG0 model.

Other methods that we tried for saving space were:

1. Keeping an address's IRG model only for the duration that address is in the memory. Whenever an address is replaced, its IRG model is reset. This method did not work well (tended away from OPT towards LRU) because deleting the entire IRG model of the replaced address implies less information for the predictor. This results in more number of no predictions (FAILs) and hence more LRU replacements.
2. Keeping only a few of the frequent IRG values and approximating the rest. This method did improve upon LRU but did not work better than the logarithmic approximations.
3. Keeping only the IRG values of the last k (a predefined threshold) IRG symbols. This saved on space for a small enough k, but did not work better than logarithmic approximation for too small a k. This also had a larger overhead of recomputing the IRG frequencies every time a new IRG symbol is encountered.

Time reduction: Extra time is spent both on a hit, as well as on a miss. Upon a hit on address a, a new IRG value gets generated for $IRG_P(a)$. The frequency count corresponding to this value needs to be incremented. Also, a pointer keeping track of a's last reference needs to be updated. Upon a miss, in addition to the above steps, predictions need to be carried out for all the addresses in the memory. The overhead in a hit is very small so we only consider ways to save time whenever a replacement decision has to be made.

We know that LRU is a good replacement algorithm, in general. So, we keep our memory as an LRU stack. At the time of replacement, we choose one of the m lowest addresses in the LRU stack for replacement. We query only these m IRG models for the farthest. We simulated a fully associative cache with 4 byte block size for the PASC and SPIC traces. Figure 10 describes the miss ratio as a function of the fraction of IRG models queried. 0% is the same as LRU and 100% is the original IRG0. Consequently, 20% querying for a cache size of 4K words (1024 blocks) implies that 205 least recently used IRG models are queried, instead of all the 1024, when a replacement decision has to be made.

Figure 10 Miss ratio variation with % of resident IRG models queried for replacement.

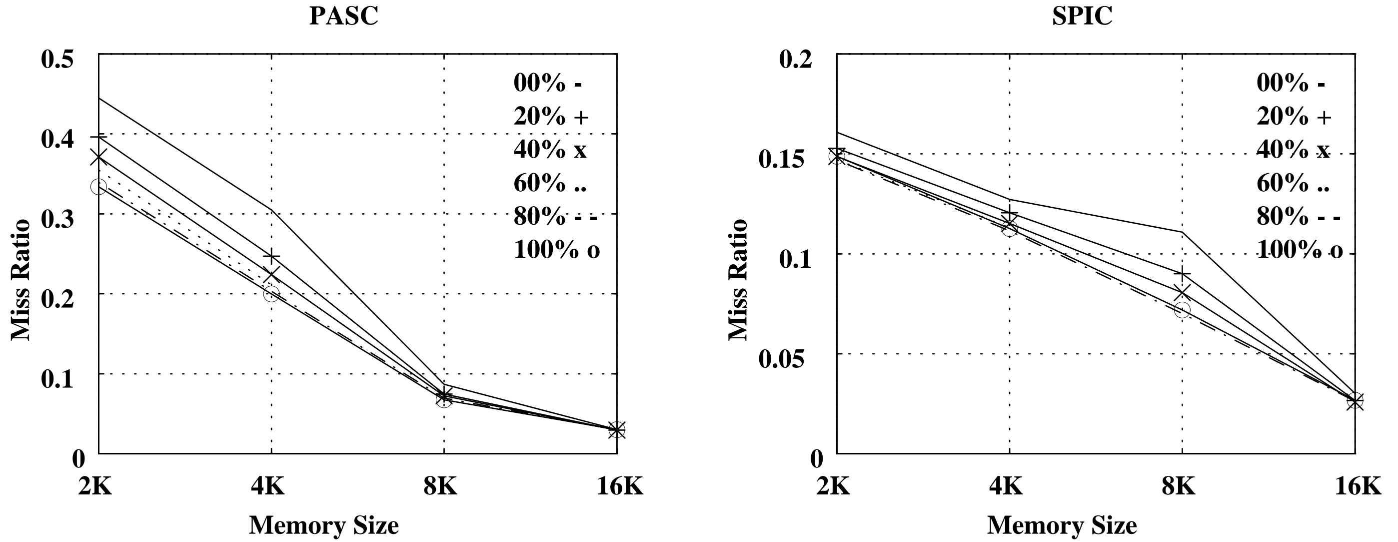


Table 4 describes the speed up in *simulation* time achieved using this selective querying process. Again, time is relative with respect to the LRU simulation. These numbers are merely for quantizing the overheads of prediction and are not to be mistaken for the real cache access time. As the size of query becomes larger, the time taken also increases. On the other hand, with increase of cache size, the time taken usually decreases because there are fewer misses and hence, fewer replacement decisions. For a 16K cache, in both PASC and SPIC, the time is largest in the 80% column, this is because they have the least miss ratio, even better than the 100% case. This happens because in large caches there are blocks with IRG models having less information. In such case, it is better to use a combination of the least recently used ranking and the IRG model.

Table 4: IRG simulation time overheads with selective querying.

	Simulation time relative to LRU				
Cache Size	20 % query	40 % query	60 % query	80 % query	100 % query (IRG0)
PASC simulation					
2 K	2.11	3.04	3.72	4.45	5.15
4 K	1.79	2.56	3.24	3.85	4.21
8 K	1.45	1.77	2.01	2.28	2.55
16 K	1.08	1.08	1.10	1.15	1.13
SPIC simulation					
2 K	1.36	2.04	2.55	3.07	3.45
4 K	1.57	2.17	2.70	3.30	3.62
8 K	1.44	1.73	1.82	2.20	2.5
16 K	1.02	1.06	1.06	1.19	1.13

6 IRG Model based Variable Space Management

In this section we propose the second application of our IRG model – a variable memory management algorithm. A variable (or dynamic) memory management algorithm's task is to allocate and deallocate pages to a process in such a way so as to keep the *space-time product* as low as possible. This is applicable in multiprogramming environments where miss ratio as well as space has to be minimized for each of the processes. We use our IRG model to predict pages which will be accessed "far" in the future and remove them from memory. We first briefly describe the problem and the significant algorithms that have tried to solve it. Then we describe our IRG based algorithm and present simulation results for the same.

6.1 Introduction

In a multiprogrammed paged environment, the two most important criterions on which the overall system performance depends are, memory usage, and the fault rate of each process. Memory is a shared resource among multiple processes which makes it a critical parameter – unlike the fixed space uniprogrammed scenario where reducing the fault rate is the only concern. *Space-Time Product (ST)* as defined by Denning [3] is a standard measure for evaluating the performance of a process. It is defined as the integral of the memory used over the time the process is running or waiting for a missing page to be swapped into the main memory:

$$ST = \sum_{t=1}^{T} s(t) \, + \, \tau \times \sum_{i=1}^{M} s(t_i)$$

where T is the total time a process lasts, $s(t)$ is the memory (in number of pages), occupied by it at time t, τ is the fault penalty or the swapping delay, t_i (i = 1, 2 ... M) is the time at which the ith fault took place and M are the total number of faults. Prieve and Fabry [31] defined a simpler *Space-Time Product (C)* which makes a simplifying assumption that all faults have the same cost τ, thus:

$$C = \sum_{t=1}^{T} s(t) \, + \, \tau \times M$$

Under both these measures, the smaller the space-time product, the better is the performance of the system. All the standard algorithms try to minimize this product by estimating pages which need not be kept in the memory. These are the pages which either will never be accessed in the future, or they will be accessed so far away in the future that keeping them in the memory for that long is not cost effective. IRG modelling gives us a direct method for estimating how far in future a page will be referenced. Our algorithm is validated via trace driven simulations by showing space-time improvements over the current best known algorithms.

6.2 Related Work

To achieve a lower space-time product, numerous algorithms have been proposed. We will only sketch the important ones. Denning proposed the Working Set (WS) algorithm [3] which keeps the pages referenced in the last τ memory accesses, in the memory. Upon a fault it fetches the faulted page, and after each memory reference it removes the page that has not been referenced in the last τ memory accesses, if any. The Page Fault Frequency (PFF) algorithm [32], on the other hand, does swapping of pages only at fault times. At a fault it swaps in the faulting page, and if the time since the last fault is less than θ (some predefined constant) then it keeps the pages as such, otherwise it removes the pages that were not referenced since the last fault. Thus it can be viewed as an algorithm which tries to keep the fault rate less than $1/\theta$. Experimental and analytical studies have shown WS to perform better than PFF and to be more stable [3, 33]. Smith's Damped Working Set [34] has less than 5% space-time product improvements over WS and its main purpose was to remove temporary memory overflows and not to improve the space-time product. Fixed space algorithms, e.g. LRU, in general have been shown to have worse space-time product than WS and PFF [35, 36], so we won't discuss them here.

Prieve and Fabry [31] proposed VMIN - an optimal variable space algorithm for the C (see above) space-time product measure, i.e. an algorithm that produces the minimal fault rate for a given average memory usage. But their algorithm is off-line in the sense that it needs to know the next τ references beforehand. After each fault it brings in the faulting page, and after each reference it swaps out the referred page if it will not be accessed in the next τ memory accesses. DMIN was proposed by Budzinski et al [37]. This off-line algorithm is optimal for the space-time cost criteria *ST*. They need to know the entire trace beforehand and map the *ST* minimization problem to the maxflow problem in graphs.

6.3 Drawbacks of the WS Algorithm

We analyze why the WS algorithm does not perform as well as the VMIN algorithm. These observations along with our IRG model are used to improve on the WS algorithm.

1. VMIN and WS have identical faults for a given τ (fault penalty) and a given reference string. This is because the only difference between VMIN and WS is that VMIN removes those pages early which WS removes after they leave its window. Consider a page referred at time t and next at time $t+x$. If $x \leq \tau$ then a hit will happen at time $t+x$ for both VMIN and WS. On the other hand if $x > \tau$ then VMIN will remove that page immediately at time t whereas WS will remove it at time $t+\tau$, and in both cases a fault will occur at time $t+x$. But VMIN saves one page of space for an entire duration τ.
2. Consider a page which is accessed at time t and then again at time $t+\tau+x$, where $0 < x \leq \tau$. At time $t+\tau$, WS will remove this page. On the other hand if we keep this page for x more units of time then we will avoid a fault and get a better C space-time product. WS assumes that a page not referenced for τ time units, will not be accessed in the next τ references. This gives bad performance when IRG values are in between τ and 2τ.
3. The WS algorithm can be looked at as a crude IRG predictor. Immediately after a page is referenced, it "predicts" its next IRG value to be $\leq \tau$ and keeps it in the memory. If the page stays unreferenced for τ time units, it "predicts" the next IRG to be greater than 2τ and removes it. A better knowledge of the past IRG behavior of a page, and a flexibility to "predict" at more time instances (instead of just two) can improve this prediction technique.

6.4 WIRG Dynamic Memory Algorithm

We propose a dynamic space management algorithm WIRG-k, that uses an underlying level k IRG prediction technique . This prediction techniques is similar to the one used in the fixed space scenario in section 5.

At each reference to a page p, we predict the next IRG value of p, using its past IRG history. If the predicted value is $\leq \tau$ then we keep that page, else we remove it. There are two scenarios when we can make an error. First, when due to overestimation we remove the page, when in fact, it is referred to within the next τ references. In this case we will cause an extra fault, which we call an R(emove) error. Second, we might underestimate and keep a page when it is actually referred to at a time beyond the next τ references (or not referenced at all in the future). To alleviate this problem, which we call the K(eep) error, we again use IRG prediction for a resident page that has not been referenced for more than τ time units. If the predicted next IRG value is smaller than τ then we keep the page else we remove it. Note that IRG predictions in the case of the K errors will use the added information about the current non reference interval for that page, i.e. if a page hasn't been referred to for the last m time units then its next IRG value has to be larger than m. In figure 11 we give the pseudo code of the algorithm.

In the algorithm, when `estimate_forward()` returns a FAIL because the current duration of non reference is greater than any of the IRGs seen so far, we remove that page. We did this because such an event usually implies a change in access pattern of that page, making its IRG history obsolete.

6.5 Simulation Experiments

We used the same set of traces as used in section 5 for our WIRG simulations. Simulation was done for a paged virtual memory environment using 512 words per page. The page level traces were obtained from the virtual address traces by dividing the address value by 2^9. One IRG model was built for each unique page in a trace.

We compared our WIRG-i algorithms that use an i level IRG predictor as defined in section 4, with the Working Set (WS) and the VMIN algorithms. Here we only present results for eight traces – CC1, DEC0, FORF, IVEX, KENS, PASC, SPIC and UE02. The results were essentially similar for the rest of the traces. Figure 12 depicts the average memory used (in pages) versus the fault rate for these traces. The experiments were carried out by varying the value of τ. We also simulated the PFF algorithm for the CC1 and the SPIC traces. We do not present the results because PFF performed worse than WS in both the simulations.

In table 5 we present the space-time product under the C measure for the SPIC trace simulations. The values are normalized with respect to the length of the trace. Similar results were obtained for other traces.

Finally, in table 6 we present the R and K errors in our WIRG algorithms for the SPIC trace simulations. We normalized them with respect to the length of the trace.

Figure 11 Pseudo code for the WIRG algorithm. *T* is the fault penalty.

```
PROC access(address a, memory M)
     update_irg_stream(a); /*Same as in the fixed memory algorithms*/
     IF(a not in M)THEN
         Fetch (a);
     ENDIF
     Access (a); /*Use page a*/
     FOR each x in M DO
       IF(x was just accessed OR
          x was accessed more than T units ago)THEN
           y = estimate_forward(x); /*Same as in the IRG0, IRG1 algorithms*/
           IF(y > T)THEN
               remove(x);
           ELSEIF(y==FAIL AND x has been accessed more than once)THEN
               remove(x);
           ENDIF
       ENDIF
     ENDFOR
ENDPROC
```

Figure 12 Fault rate as a function of average memory used (in number of pages).

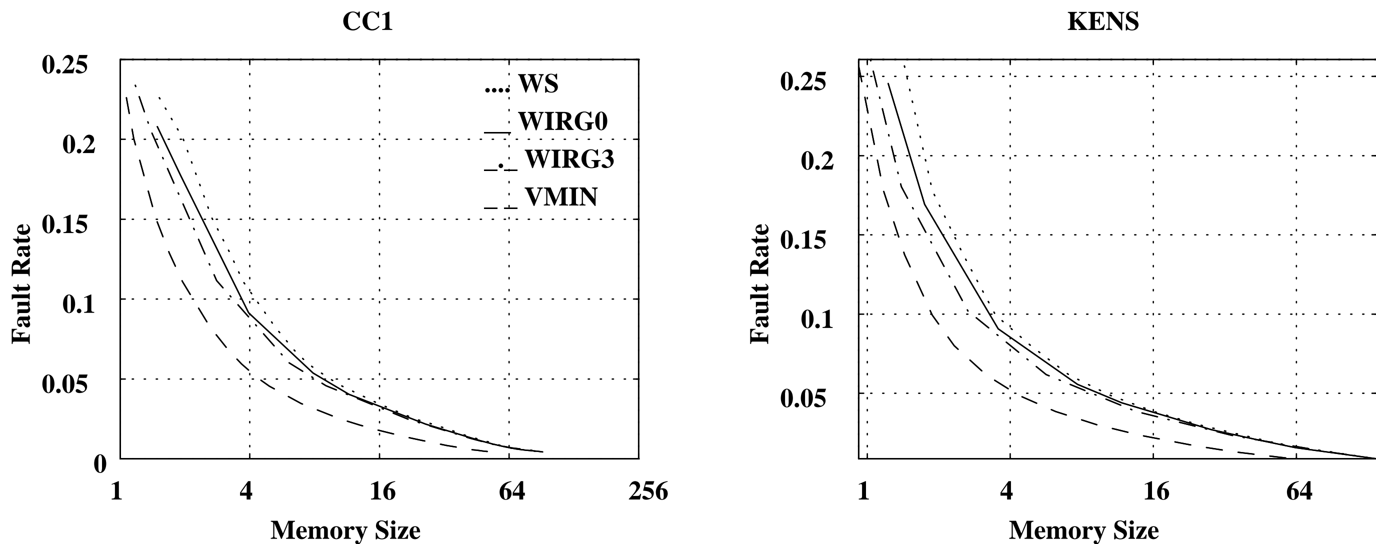
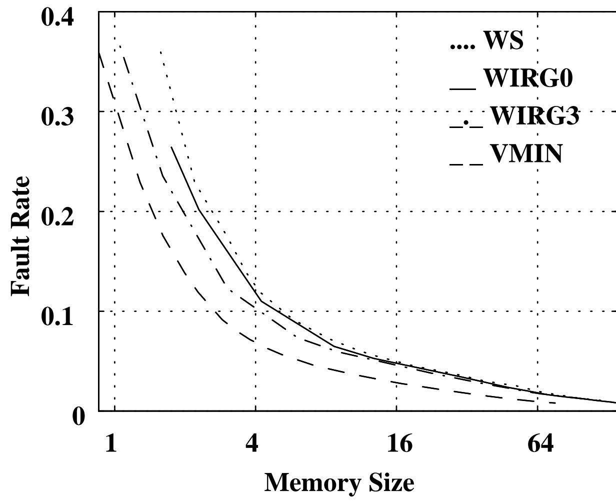
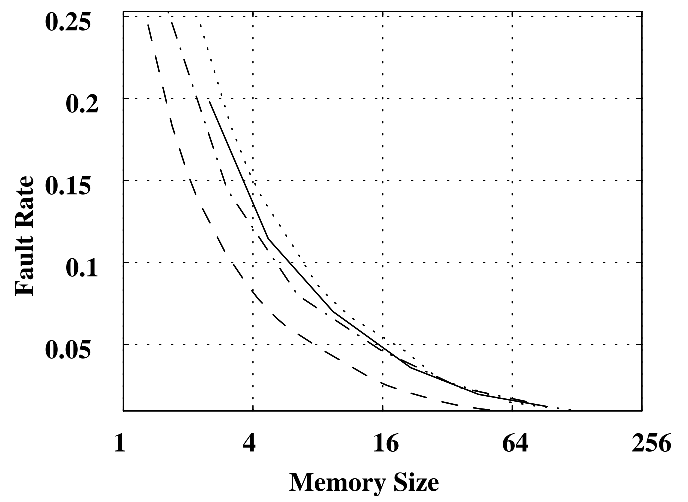
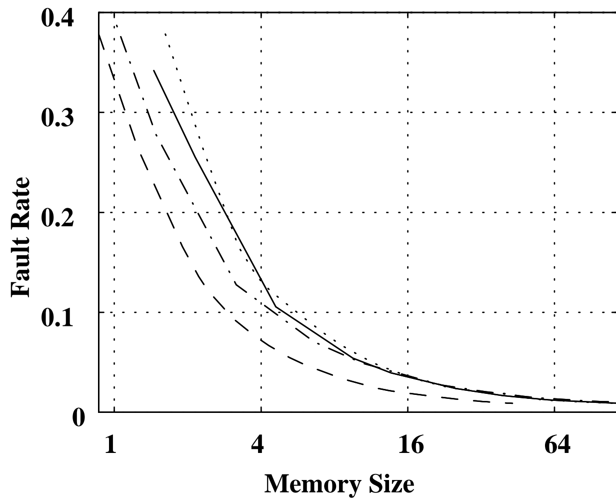
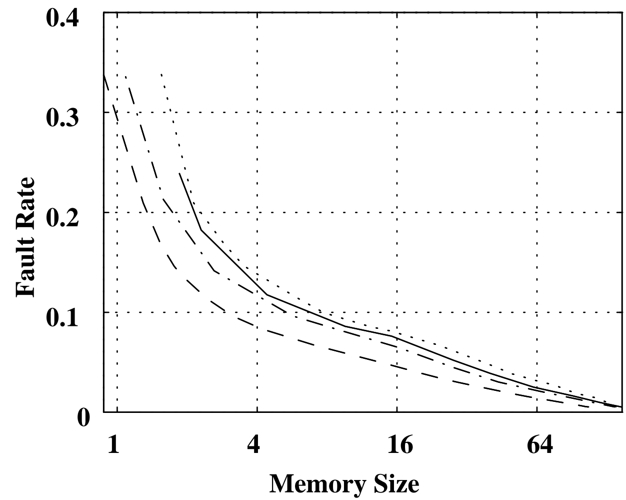


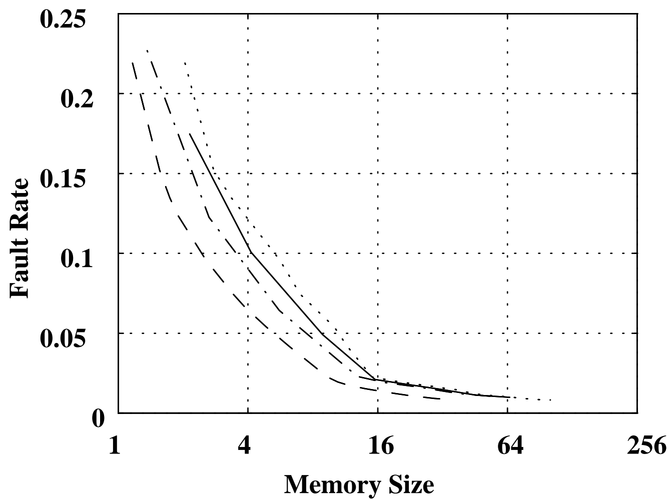
Figure 12 (Contd) Fault rate as a function of average memory used (in number of pages).











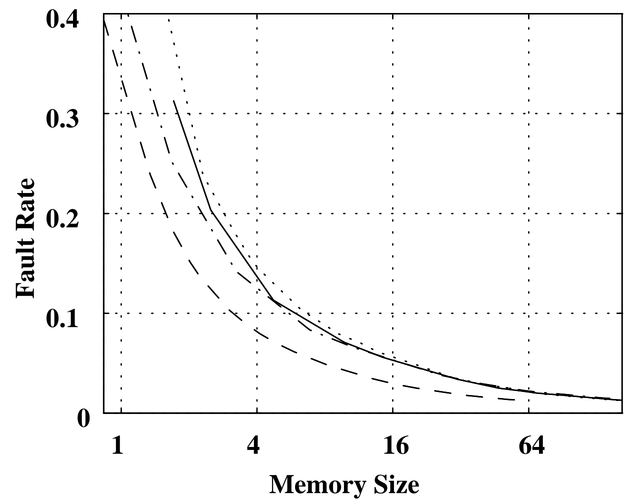


Table 5: C Space-Time Product for the SPIC simulations.

τ (miss penalty)	Normalized Space-Time product C = (AvgMem + $\tau \times$ MissRatio)					
	WS	WIRG0	WIRG1	WIRG2	WIRG3 (& improvement over WS)	VMIN
4	2.92	2.84	2.51	2.35	2.27 (22%)	2.04
16	5.85	5.74	5.21	4.84	4.84 (21%)	3.87
64	12.34	12.04	11.07	10.09	9.70 (21%)	8.11
256	20.85	20.96	19.99	19.29	18.73 (10%)	15.11
1024	46.47	46.65	44.58	43.12	41.75 (10%)	29.56
4096	107.05	106.19	102.71	100.49	97.21 (9%)	66.74

Table 6: R and K errors for the SPIC simulations.

τ (miss penalty)	Normalized R and K errors							
	WIRG0		WIRG1		WIRG2		WIRG3	
	R	K	R	K	R	K	R	K
4	0.09	0.76	0.09	0.43	0.09	0.25	0.07	0.19
16	0.17	1.77	0.18	1.22	0.22	0.81	0.18	0.60
64	0.66	3.57	0.61	2.59	0.53	1.66	0.51	1.28
256	0.51	5.53	0.59	4.51	0.65	3.77	0.66	3.19
1024	3.30	15.49	3.15	13.38	3.19	11.81	3.29	10.38
4096	7.43	34.49	7.47	30.74	7.51	28.41	7.69	24.98

Error Analysis: (1) The number of K errors was always an order of magnitude larger than R errors. The main reason is that, the decision to remove a page is only made either right after an access, or after an interval of τ non-references to that page. This reduces the number of places where an R error could be made. (2) The number of K errors went down with increase in the order of the underlying predictor. This is mainly because a higher order predictor implies more accurate predictions. (3) The R errors remained neutral with respect to the order of the underlying predictor. This is due to the fact that most of the R errors occur during the initial references to a page when the IRG history is too small to benefit from the higher order predictors.

6.6 Variations in WIRG

As explained in section 5, IRG models consume a large amount of space and time. So we tried the following variations in our WIRG algorithm in order to find a practical improvement over WS:

1. Doing prediction for removal at every instant of time. In this case the number of R errors went up, although the K errors did not go down substantially, resulting in worse performance than WS for large values of τ.
2. Approximating the IRG stream to 0's and 1's, when the IRG value is $\leq \tau$ and $> \tau$, respectively. Although this resulted in simpler prediction overheads, the R and K errors went up considerably for high values of τ. The performance was better than WS for small values of τ and worse for larger ones.
3. Averaging for prediction. Instead of using the IRG value with the highest probability we took the mean of the likely IRG values weighted by their probabilities. This degraded performance considerably due to the fact that IRGs do not have a continuous distribution. Averaging them could predict an IRG value that has a zero probability of occurrence in reality.
4. Approximating the prediction by looking only at the last k (some predefined constant) IRG values in each of the IRG streams. Although storage gets reduced, prediction becomes difficult as the statistics need to be

recomputed at the occurrence of each new IRG value. A better solution was to maintain frequency counts in a fixed buffer and use it as a cyclic queue. This improved performance over WS but not considerably.

7 Trace Compaction using IRG

Finally in this section, we propose a scheme for compressing traces in a lossy manner so as to reduce the time taken for trace driven simulations. We store each IRG string for each page referred to in a trace, separately. These separate IRG strings are then interleaved to generate the original trace. The key idea being that if the WS algorithm with window size τ is to be simulated on a trace, then all IRGs with values smaller than τ can be ignored because they do not cause a fault. We first describe the previous work on trace compaction for speeding up simulations, followed by the description of our algorithm. Finally we present our compression results.

7.1 Previous Work and Motivation

Previous work on trace compaction has been motivated by two reasons. First, the lossless techniques meant for better storage utilization. These include techniques which utilize temporal characteristics [38] and those which utilize spatial locality [39] to store traces in a way that no information is lost. These techniques do not reduce simulation time.

On the other hand many lossy techniques for trace compression have been proposed which not only save storage but also help in speeding up trace driven simulations of cache and memory management algorithms.

Smith's *stack deletion* [40], simulates an LRU stack of size D (a parameter for the algorithm) and removes all the hits from the trace. In this way, all temporally "nearby" references get removed. His other method called the *snapshot method* records the set of memory references at regular time intervals. The rationale being that working sets do not change very rapidly with time. Puzak [41] proposed *trace stripping* applicable for cache simulations. Here a direct-mapped cache (called the cache filter) is simulated and the hits are removed from the trace. This scheme introduces no errors when a cache with larger number of sets than in the filter is simulated on the stripped trace. Agarwal and Huffman [42] combined the cache filter technique with a blocking technique. They replace runs in a trace by single representative memory addresses.

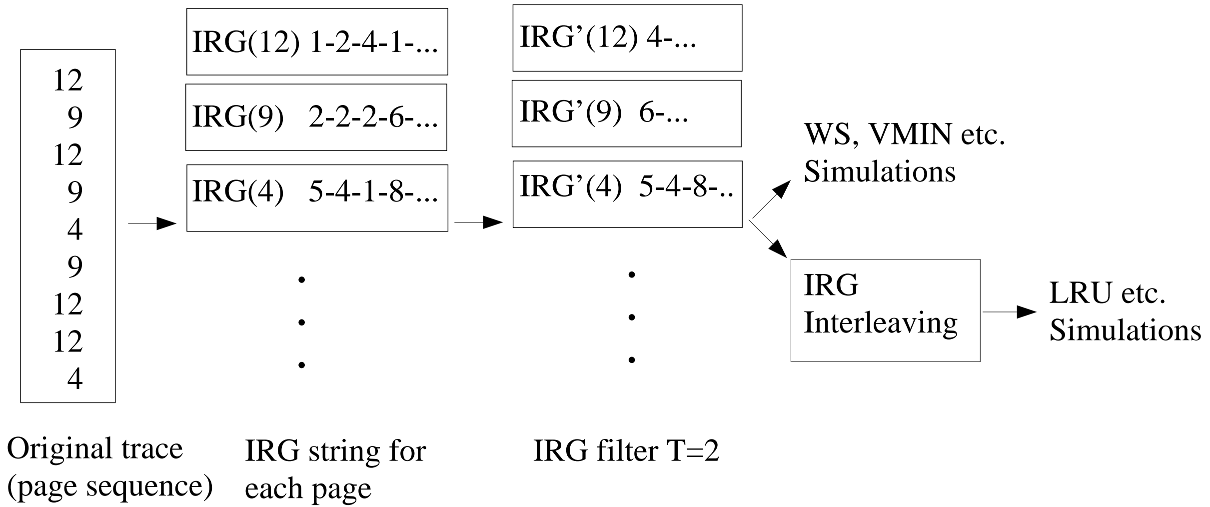
We are not interested in speeding up of cache memory simulations, so we compare our method only with the stack deletion algorithm. The important point regarding the stack deletion and almost all of the other compaction methods is that they are designed assuming that the compacted traces would be used for simulating LRU, and LRU based memory policies. These compacted traces are not useful for simulating algorithms like WS, which need precise timing information to eject a page. For example, consider a memory trace (1, 2, 3, 4, 2, 2, 1, 3, 1, 2, 2, 3). Under stack deletion with D = 2 this trace will be compacted to (1, 2, 3, 4, 2, 1, 3, 2, 3). Running LRU with buffer size 3 will generate 6 faults in both the traces. But for the WS algorithm with $\tau = 3$, the original trace yields 8 faults while the compacted one gives only 6. The error is due to the removal of repeated references during the compaction process. Pages which would have left the WS window a long time ago, still remain in there, resulting in lesser number of faults. This also results in an increased average memory value.

7.2 IRG Filter

We propose a scheme based on the IRG model introduced in section 4 for trace compaction. The main application is the speeding up of dynamic memory management algorithms like the WS, although LRU simulations can also be performed after doing some preprocessing.

Consider a page p having an IRG stream $g_1\ g_2\ g_3$ .... If g_i is smaller than the WS window size τ, then the reference following the ${g_i}^{th}$ IRG will not cause a fault on page p, otherwise it will. Also the faults in WS with a larger window form a subset of those in WS with a smaller window. In our IRG filter scheme with parameter T, we simply remove IRG values smaller than T in each of the IRG streams of a trace and store them in separate files. The WS algorithm with a window size greater than T, will give same number of faults on the compacted trace as in the original trace, resulting in zero error in the fault rate.

Figure 13 Schematic of the IRG filter process. IRG'() are actually stored on the disk.



To simulate WS with window size τ in our scheme, we simply walk from one IRG stream to another, counting the number of gaps that are larger than τ. The sum of such gaps is the total number of faults. To simulate LRU and LRU-like algorithms, first we have to reconstruct a single trace from the IRG streams. We do this by simply interleaving the compacted IRG streams. The reason why we expect this to work is because most of the cache and memory algorithms fault when a reference is made to the same address or page after a long interval of time – which we do preserve in our compacted IRG models. The interleaving method does involve extra work in comparison to the stack deletion method. But then it is done only once, following which multiple simulations can be done. We leave out the details of interleaving in this presentation.

Average Memory Usage: The other important parameter in a dynamic memory simulation is the average memory usage. Stack deletion and other stack based compacting methods drop the timing information and hence they give erroneous memory usage statistics when used for WS simulations. For example, simulation of WS on a stack deleted trace with D=4 gave an error of up to 240% for the SPIC trace.

The IRG filter with parameter T, will underestimate average memory usage if used directly, because all the gaps smaller than T are removed during compression. These small gaps represent intervals during which the corresponding page is memory resident. To solve this problem, all we need to maintain is the sum of all the gaps with value $\leq T$, over all the IRG strings. This is just one extra integer and therefore the compression remains the same and we get zero error for the average memory usage in WS simulations.

7.3 Compression results

We compared the IRG filter with Smith's stack deletion method. The parameters for the two compression techniques were chosen such that nearly the same compression was obtained using both the techniques. We then simulated the WS, Page Fault Frequency (PFF) and the LRU algorithms on the compacted traces. Here we present results for the SPIC page reference trace with 512 lines per page and the CC1 page reference trace with 1024 lines per page. These are the same traces as used in section 6. Similar results were obtained for other page reference traces. In table 7, the τ in the WS rows is the window size of the WS algorithm. The θ in the PFF rows is the inter-fault duration threshold of the PFF algorithm. The M in the LRU rows is the size of the main memory in number of pages. Error is calculated as

$$\left(\frac{\text{Miss Ratio(Compressed Trace)}}{\text{Miss Ratio(Original Trace)}} - 1\right) \times 100\ \%$$

Positive error implies an overestimation and a negative error implies an underestimation. We define compression as the ratio of the number of references in the output trace and those in the original trace.

Table 7: Error in fault rate while simulating WS, PFF and LRU on the compacted traces. Comparison of IRG filter (highlighted) and stack deletion methods.

SPIC trace (512 lines per page)		≈12.5% Compression		≈2.5% Compression	
		IRG Filter T=16 Comp= 12.4%	Stack Deletion D=4 Comp= 12.6%	IRG Filter T=256 Comp= 2.2%	Stack Deletion D=16 Comp= 2.7%
WS VMIN	τ = 512	**0**	-49.7%	**0**	-73.4%
	τ = 1024	**0**	-52.5%	**0**	-92.0%
	τ = 2048	**0**	-53.2%	**0**	-91.7%
	τ = 4096	**0**	-79.5%	**0**	-90.5%
PFF	θ = 128	**6%**	-45.8%	**7.2%**	-76.5%
	θ = 256	**10.2%**	-36.8%	**6.3%**	-80.6%
	θ = 512	**1.5%**	-55.7%	**-5.2%**	-88.1%
	θ = 1024	**-4.6%**	-67.5%	**-19.8%**	-91.6%
LRU	M = 32	**-1.5%**	0.6%	**-13.6%**	0.2%
	M = 64	**4.0%**	-0.1%	**0.5%**	1.3%
	M = 128	**1.4%**	-0.1%	**0.06%**	0.13%
	M = 256	**-1.2%**	0.1%	**-1.9%**	1.2%

CC1 trace (1024 lines per page)		≈11.5% Compression		≈2.7% Compression	
		IRG Filter T=12 Comp= 11.5%	Stack Deletion D=3 Comp= 11.6%	IRG Filter T=256 Comp= 2.8%	Stack Deletion D=16 Comp= 2.6%
WS VMIN	τ = 512	**0**	-77.4%	**0**	-89.7%
	τ = 1024	**0**	-76.5%	**0**	-90.6%
	τ = 2048	**0**	-75.8%	**0**	-88.2%
	τ = 4096	**0**	-72.8%	**0**	-82.9%
PFF	θ = 128	**5.2%**	-73.9%	**4.7%**	-89.4%
	θ = 256	**1.1%**	-74.2%	**3.2%**	-88.2
	θ = 512	**-4.3%**	-74.7%	**-3.6%**	-86.6
	θ = 1024	**11.1%**	-67.4%	**12.3%**	-83.7%
LRU	M = 64	**-2.7%**	0.1%	**-12.8%**	1%
	M = 128	**-2.2%**	0.1%	**-9%**	0
	M = 256	**0**	0	**0**	0
	M = 512	**0**	0	**0**	0

Table 7 shows results for two different compression values – one is of the order of 10%, and the other is of the order of 1%. The stack deletion method performed poorly for WS and PFF simulations in both the cases, for all values of τ and θ respectively, while IRG filter performed very well. On the other hand, LRU simulations after doing IRG filtering gave errors up to 13.6%, and sometimes outperformed the LRU simulations done on the stack deleted traces.

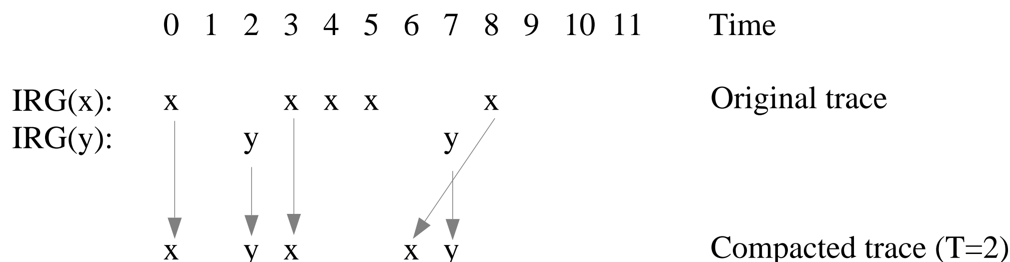
7.4 Error Analysis and Improvement

Stack deletion performed poorly for WS, VMIN and PFF simulations because the precise timing information was lost during compression. We remedied this by storing the original clock-tick information in the compacted trace. The miss ratio errors in the WS simulations for the CC1 trace dropped down to 6.8%, 14.8%, 11.3%, and 6.4% for τ equal to 512, 1024, 2048, and 4096 respectively (stack size D=16). Although this did improve the WS miss ratio performance, it has the following disadvantages: (1) One more set of data (time stamps), as big as the compacted trace itself, needs to be maintained, (2) WS and VMIN miss ratio and average memory errors will still be nonzero, and (3) WS simulations will be slowed down because the sliding window algorithm will have to take into account the original clock-ticks.

IRG filtering, gave errors in LRU simulations because gap-removal followed by interleaving, can result in wrong

ordering of references. Consider figure 14.

Figure 14 Wrong ordering due to interleaving.



After doing IRG filtering with T=2, the two original references y(7) and x(8) become x(6) and y(7) respectively. We remedied this problem by adding precise timing information as in the stack deletion improvement above. This worsened compression (doubled it) but the LRU error became less than **3.7%** for all the simulations described in table 7.

8 Conclusions and Future Work

We have presented a program modelling technique for capturing the temporal locality behavior of memory references made by a program, on a per address basis. We observed that the sequence of gaps between consecutive accesses to the same location in memory (called IRG) was, in general, a repetitive and hence a predictable string. Consequently, a *k*–order Markov chain was used to model the IRG strings and predict an address's next reference in the future. We validated this model by applying it in two memory management scenarios – replacement algorithms and dynamic memory management. We also validated the predictive property of the IRGs by implementing a new IRG based trace compaction technique.

The IRG model was shown to be effective in improving replacement algorithms at various levels of the memory hierarchy. Using trace driven simulations, improvements up to 37% in the miss ratio over LRU, were obtained. A practical implementation (LOG0) was proposed for a paged memory environment that only required 3% more space than LRU and improved the miss ratio up to 25% over LRU. Although a practical implementation was not possible for IRG1 and higher order models, the results from these simulations established the authenticity of our model.

This model was further applied to the variable (dynamic) memory environment where the average space and the miss ratio have to be minimized. The working set (WS) algorithm was enhanced using our IRG model and space-time product improvements up to 22% over WS were obtained for a paged memory scenario.

Finally, we applied our IRG model to trace compaction. We showed that by using IRG filters we can do lossy compression of traces up to 2.5%. The main advantage being that WS simulations are speeded up with zero error in the fault rate and the average memory usage. Up to 13.6% error was obtained in the LRU simulations on small memory sizes, which was subsequently reduced to 3.7% by adding timing information and increasing the compression to 5%.

Presently, we are trying to capture the IRG behavior using simple approximations of a *k*–order Markov model, so as to make IRG based cache replacement and page replacement algorithms more practical. This will also make our WIRG algorithms more practical.

Another direction for investigation is the relationship between data compression and prediction of program behavior. Vitter and Krishnan [43] proved that compression can be used for optimal prefetching. We are investigating if a similar relationship exits between trace compression and the best possible on-line replacement algorithm.

Bibliography

[1] J.R. Spirn. *Program Locality and Dynamic Memory Management*. PhD thesis, Princeton University, 1973.

[2] A.W. Madison and A. Batson. Characteristics of program localities. *Communications of the ACM*, 19, May 1976.

[3] P. Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, SE–6, January 1980.

[4] A.I. Verkamo. Empirical results on locality in database referencing. In *Proceedings of 1985 ACM SIGMETRICS Conference on Measurement & Modeling of Computer Systems*, August 1985.

[5] S. Majumdar and R.B. Bunt. Measurement and analysis of locality phases in file referencing behaviour. In *Proceedings of Performance '86 and ACM SIGMETRICS 1986 Joint Conference on Computer Performance Modeling, Measurement and Evaluation*, pages 180–192, May 1986.

[6] Juan Rodriguez-Rosell. Empirical data reference behavior in data base systems. *Computer*, pages 9–13, November 1976.

[7] T.F. Chen and J.L. Baer. Reducing memory latency via non-blocking and prefetching caches. *ASPLOS-V*, October 1992.

[8] W.F. King. Analysis of paging algorithms. In *Proceedings of IFIP Congress*, Ljublanjana, Yugoslavia, August 1971.

[9] O.I. Aven, L.B. Boguslavsky, and Y.A. Kogan. Some results on distribution-free analysis of paging algorithms. *IEEE Transactions on Computers*, 25(7), July 1976.

[10] G.S. Rao. Performance analysis of cache memories. *Journal of the ACM*, 25(3), July 1978.

[11] H. Opderbeck and W.W. Chu. The renewal model for program behavior. *SIAM Journal of Computing*, 4:356–374, 1975.

[12] R.L. Mattson, J. Gecsei, D.R. Slutz, and I.L. Traiger. Evaluation techniques and storage hierarchies. *IBM Systems Journal*, 9:78–117, 1970.

[13] G.S. Shedler and C. Tung. Locality in page reference strings. *SIAM Journal of Computing*, 1(3), September 1972.

[14] Dominique Thiebaut. Synthetic traces for trace-driven simulation of cache memories. *IEEE Transactions on Computers*, 41(4), April 1992.

[15] M.A. Franklin and R.K. Gupta. Computation of pf probabilities from program transition diagrams. *Communications of the ACM*, 17:186–191, 1974.

[16] S. Rudich. Inferring the structure of a markov chain from its output. In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science*, 1985.

[17] Alan Batson. Program behavior at the symbolic level. *Computer*, pages 21–26, November 1976.

[18] C.K. Chow. On optimization of storage hierarchy. *IBM Journal of Research and Development*, 18:194–203, May 1974.

[19] Dominique Thiebaut. On the fractal dimension of computer programs and it s application to the prediction of the cache miss ratio. *IEEE Transactions on Computers*, 38(7), July 1989.

[20] Abraham Mendelson, Dominique Thiebaut, and Dhiraj L. Pradhan. Modeling live and dead lines in cache memory systems. *IEEE Transactions on Computers*, 42(1), January 1993.

[21] Jaswinder Pal Singh, Harold S. Stone, and Dominique F. Thiebaut. A model of workloads and its use in miss-rate prediction for fully associative caches. *IEEE Transactions on Computers*, 41(7), July 1992.

[22] Andrew Choi and Manfred Ruschitzka. Managing locality sets: The model and fixed-size buffers. *IEEE Transactions on Computers*, 42(2), February 1993.

[23] M.C. Easton. Cold-start vs. warm-start miss ratio. *Communications of the ACM*, 21, October 1978.

[24] J.A. Storer. *Data Compression methods and theory*. Computer Science Press, MD, 1988.

[25] Edward G. Coffman and Peter J. Denning. *Operating Systems Theory*. Prentice-Hall, 1973.

[26] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proceedings of 1993 ACM SIGMOD*, June 1993.

[27] John T. Robinson and Murthy V. Devarakonda. Data cache management using frequency-based replacement. In *Proceedings of 1990 ACM SIGMETRICS Conference on Measurement & Modeling of Computer Systems*, May 1990.

[28] Jorma Rissanen. A universal data compression system. *IEEE Transactions on Information Theory*, IT-29(5), September 1983.

[29] John L. Hennessy and David A. Patterson. *Computer Architecture : A Quantitative Approach*. Morgan Kaufman, 1990.

[30] J.K. Flanagan, B. Nelson, J. Archibald, and K. Grimsrud. BACH: BYU address collection hardware; the collection of complete traces. In *International Workshop on Modeling Techniques and Tools for Computer Performance Evaluation*, September 1992.

[31] B.G. Prieve and R.S. Fabry. VMIN — an optimal variable-space page replacement algorithm. *Communications of the ACM*, 19(5), May 1976.

[32] Wesley W. Chu and Holger Opderbeck. Program behavior and the page-fault-frequency replacement algorithm. *Computer*, pages 29–38, November 1976.

[33] Ram K. Gupta and Mark A. Franklin. Working set and page fault frequency algorithms: A performance comparison. *IEEE Transactions on Computers*, C-27(8), August 1978.

[34] A.J. Smith. A modified working set paging algorithm. *IEEE Transactions on Computers*, 25(9), September 1976.

[35] Peter J. Denning and G. Scott Graham. Multiprogrammed memory management. In *Proceedings of the IEEE*, June 1975.

[36] Alan Jay Smith. Analysis of optimal look-ahead demand paging algorithms. *SIAM Journal of Computing*, 5(4), December 1976.

[37] R.I. Budzinski, E.S. Davidson, W. Mayeda, and H.S. Stone. DMIN: an algorithm for computing the optimal dynamic allocation in a virtual memory computer. *IEEE Transactions on Software Engineering*, SE-7(1), January 1981.

[38] A. Dain Samples. Mache: No-loss trace compaction. In *Proceedings of ACM SIGMETRICS 1989 Conference on Measurement & Modeling of Computer Systems*, May 1989.

[39] Vidyadhar Phalke and B. Gopinath. Using spatial locality for trace compression. In *Proceedings of the Data Compression Conference*, 1994.

[40] Alan Jay Smith. Two methods for the efficient analysis of memory address trace data. *IEEE Transactions on Software Engineering*, SE-3(1), January 1977.

[41] Thomas R. Puzak. *Analysis of Cache Replacement Algorithms*. PhD thesis, University of Massachusetts Department of Electrical and Computer Engineering, February 1985.

[42] Anant Agarwal and Minor Huffman. Blocking: Exploiting spatial locality for trace compaction. In *Proceedings of ACM SIGMETRICS 1990 Conference on Measurement & Modeling of Computer Systems*, May 1990.

[43] J.S. Vitter and P. Krishnan. Optimal prefetching via data compression. In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science*, October 1991.