Exploring the Memory Access Regularity in Pointer-Intensive Application Programs

Keqiang Wu, Resit Sendag, and David Lilja

Department of Electrical and Computer Engineering, University of Minnesota
200 Union Street S.E., Minneapolis, MN 55455, USA
`{kqwu, rsgt, lilja}@ece.umn.edu`

Abstract. Pointer-intensive and sparse numerical computations typically display irregular memory access behavior. This work presents a mathematical model, called the Self-tuning Adaptive Predictor (SAP), to characterize the behavior of load instructions in procedures with pointer-based data structures by using procedure call boundaries as the fundamental sampling frequency. This model incorporates information about the history of specific load instructions (temporal locality) and their neighboring loads (spatial locality) using a least-squares minimization approach. Simulation results on twelve of the most time-consuming procedures with pointer-based data structures from five of the SPEC2000 integer benchmark programs show that these pointer-based data structures surprisingly demonstrate regular memory access patterns and the prediction error at steady-state is within [-6%, +6%] on average.

1 Introduction

An important characteristic of Pointer-Based Data Structures (PDS) is that they are dynamically allocated and managed with heap allocation. Heap allocation parcels out blocks of contiguous memory as requested by the program at run-time. Memory blocks are deallocated explicitly or via process termination in any order. For example, elements in a linked data structure contain explicit fields that name all adjacent elements by address. This mode of connectivity allows the easy construction and manipulation of data structures of arbitrary shape, such as trees and graphs. Dynamic construction allows PDS to grow arbitrarily large. However, this flexibility makes it challenging to characterize the memory access behavior of these structures. Their behavior was traditionally classified as *irregular* or *arbitrary* [1-2].

The intuitive way for prediction is to track the memory allocation/deallocation behavior by analyzing the program execution path. The cache miss behavior for two specified data structures, i.e. linked list and binary tree, was analyzed by tracking the memory allocation/deallocation sequence in synthetic programs [3]. However, in large and real programs, interactions and branch patterns are difficult to predict. These add complexity in extending their analysis.

In this paper, we avoid the detailed analysis on program execution path and use a mathematical model to extract the path pattern based on the observed paths. The primary contributions of this paper are:

1. The regularity of memory access patterns for procedures with pointer-based data structures is observed using procedure call boundaries as the sampling unit.
2. A mathematical model, Self-tuning Adaptive Predictor (SAP), is proposed that correlates both temporal and spatial locality with the program counter (PC), and optimizes predictions of future memory addresses referenced by the program using a least-squares minimization technique [4].

2 Model Formulation

Consider a general example of the procedure call sequence in Figure 1. The memory access behavior of *main( )* is complex as it jumps to different locations when different procedures are called. Its overall behavior depends on the behavior of all procedures.

1. *Leaf procedure*: a procedure that does not call other procedures.
2. *PC-correlated spatial locality*: the data address referenced by a load instruction at a PC likely depends on memory addresses referenced by loads at nearby PCs.
3. *PC-correlated temporal locality*: the next memory address referenced by a load instruction at a certain PC is likely to depend on the previous memory addresses referenced by the same load instruction.

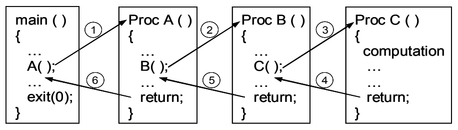

Fig. 1. Schematic of the calling procedure of a simple program.

This paper focuses on the memory access behavior produced by load instructions in leaf procedures using the procedure call as the fundamental sampling unit. The primary assumption is that, within some certain period, the behavior of memory accesses in a procedure depends on the history of both itself and nearby loads. This behavior can be represented as a linear system with constant but unknown parameters. At some point, the behavior changes which causes a consequent change in the specific parameter values. The goal of SAP is to detect such changes and automatically converge on the estimated parameter values (Figure 2).

Consider the leaf procedure C in Figure 1. Suppose that there are r loads within procedure C with program counter (PC) values p_1, p_2, …, p_r, respectively. Within the ith call of the procedure, the corresponding referenced addresses are denoted as $A_{i,1}$, $A_{i,2}$, …, $A_{i,r}$. Within a certain range of consecutive calls, the behavior of memory accesses can be represented with the following equation, which takes PC-correlated temporal and spatial localities into account:

$$A_{n,m} = \sum_{i=1}^{j} a_{i,m} A_{n-i,m} + \sum_{i=0}^{k_1} a_{i,1} A_{n-i,1} + \ldots + \sum_{i=0}^{k_l} a_{i,l} A_{n-i,l} \quad \textbf{(1)}$$

where $1 \le j \le (n\text{-}1)$, $0 \le k_i \le (n\text{-}1)$ $(i=1,2,\ldots,l)$, $0 \le l \le (m\text{-}1)$, and $1 \le m \le r$.

Without loss of generality, we consider the l=1 case in this paper. The prediction of a target load's address is based on the history of itself and one other nearby load. By letting $a_i=a_{i,m}$, $b_i=a_{i,1}$, $y(n)=A_{n,m}$, and $u(n)=A_{n,1}$, Equation (1) can be simplified to the following memory access function:

$$y(n) = a_1 y(n-1) + \ldots + a_j y(n-j) + b_0 u(n) + \ldots + b_k u(n-k) \quad \textbf{(2)}$$

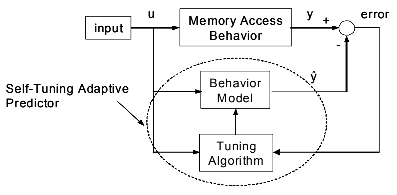

Fig. 2. Block diagram representation of the Self-tuning Adaptive Predictor (SAP).

3 Results and Discussion

The addresses actually accessed by load instructions are collected by modifying the SimpleScalar simulator [5]. Here, we use benchmark *181.mcf* with test input set for illustration. *181.mcf* exhibits the poorest data cache behavior among all benchmarks of the SPEC CINT2000 Benchmarks suite. Profiling results using *gprof* show that the *bea_compute_red_cost* procedure constitutes more than 6.7% of the total running time of this benchmark program. Our simulation results show the total number of misses in this procedure constitutes 9.15% of the total misses in the program. The related code and data structures are schematically shown in Figure 3.

```
typedef long cost_t;
```
(a)

```
typedef struct arc;
{
    node_t *tail, *head;
    …
}arc_t;
```
(b)

```
typedef struct node
{
    …
    cost_t potential;
    …
}node_t;
```
(c)

```
cost_t bea_compute_red_cost (arc_t *arc)
{
    return arc->cost – arc->tail->potential + arc->head->potential
}
```
(d)

```
bea_compute_red_cost:
…
    lw $3,0($4)   #PC=0x406500; load arc->tail
    lw $5,4($4)   #PC=0x406508; load arc->head
    lw $2,16($4)   #PC=0x406510; load arc->cost
    lw $3,44($3)   #PC=0x406518; load arc->tail->potential
    lw $4,44($5)   #PC=0x406520; load arc->head->potential
…
    .end      bea_compute_red_cost
```
(e)

Fig. 3. Related C and assembly code of the procedure *bea_compute_red_cost* from 181*.mcf.*

The five PCs (p_i) (see Figure 3e) are defined as follows and the corresponding access addresses are denoted as A_1, A_2, A_3, A_4, and A_5, respectively. Figures 3 (b), (c) and (e) show that the address relationship among *arc->cost*, *arc->tail*, and *arc-*

>head is determined at compile-time, but the correlation between *arc->tail->potential* and *arc->head->potential* are not known until *arc-> tail* and *arc->head* have been dereferenced at run-time. The memory addresses accessed by p_4 and p_5 depend on values stored in registers \$3 and \$5 respectively, and no correlation can be found from the code. As prediction among A_1, A_2, and A_3 is trivial, our study focuses on A_2, A_4 and A_5. The prediction error is normalized based on the referenced address span of the same load instruction. Figures 4 and 5 show that these two loads display different access patterns at different call ranges. We have sampled several different call ranges and observed similar patterns.

p_1=0x406500; p_2=0x406508; p_3=0x406510; p_4=0x406518; p_5=0x406520.

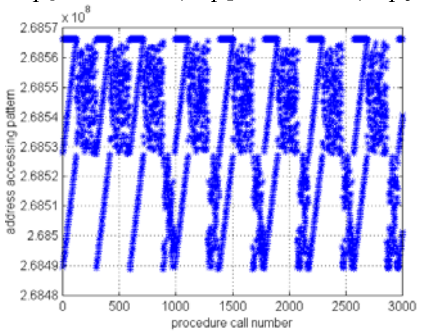

Fig. 4. Memory access pattern for PC4.

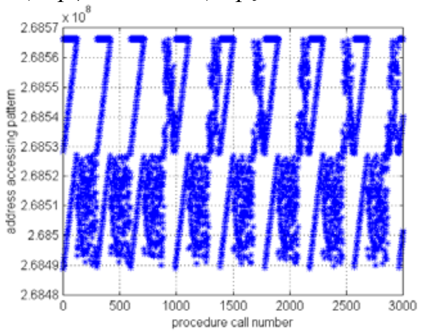

Fig. 5. Memory access pattern for PC5.

Empirical study [6] shows that the regular memory access consists of different patterns at different call ranges and different models can be effective during different phases of the memory access patterns. In this study, multiple versions of the SAP models run concurrently. Selection of a particular model with which to make a prediction is automated by observing the convergence rate of each component model.

Three models with different history depths of the target and reference loads are used as shown in Table 1. Figure 6 show that using A_2 as the reference, SAP gives good prediction in 55% of total execution time for that procedure. Using A_4 as the reference, Figure 7 shows that the steady state error is within the ranges of [-5%, +5%]. We summarize the performance of the prediction in Table 2. The results show that using a reference load that has behavior similar to the target load is better for predicting the behavior of the target than naively selecting a reference load based on simple dependence relationships. The detailed discussion can be found in [6].

Table 1. History depths of the three models used in prediction.

History Depth	Model 1	Model 2	Model 3
Target load	1	3	5
Reference load	2	4	6

4 Conclusions

This paper has proposed a Self-tuning Adaptive Predictor (SAP) model and examined the memory access patterns of leaf procedures with pointer-based data structures.

By taking the procedure as the fundamental sampling unit, SAP incorporates temporal locality and spatial locality to dynamically adapt to the changing behavior of memory accesses. Our evaluations with a subset of the SPEC2000 integer benchmark programs showed that SAP is an accurate model for memory address prediction.

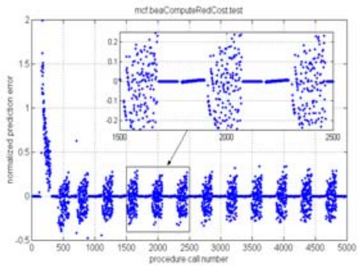

Fig. 6. prediction of A_5 using A_2.

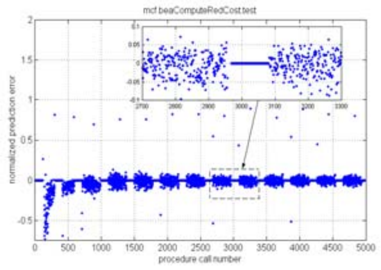

Fig. 7. Prediction of A_5 using A_4.

Table 2. Prediction performance of the procedures at steady state

Benchmark	Procedure	Error	Benchmark	Procedure	Error
164.gzip	pqdown_heap	[-3%, 3%]	175.vpr	alloc_linked_f_pointer	[-4%, 4%]
	gen_bitlen	[-11%, 11%]		net_cost	[-2%, 2%]
181.mcf	bea_compute_red_cost	[-5%, 5%]	255.vortex	chkgetchunk	[-8%, 8%]
	bea_is_dual_infeasible	[-4%, 4%]		memgetword	[-6%, 6%]
	compute_red_cost	[-6%, 6%]	256.bzip2	spec_getc	[-2%, 2%]
	sort_basket	[-14%, 14%]		spec_putc	[-2%, 2%]

Acknowledgments

This work was supported in part by National Science Foundation grants EIA-9971666 and CCR-9900605, IBM Corporation, Compaq's Alpha development group, and the Minnesota Supercomputing Institute

References

1. Chilimbi, T. M., Larus, J. R.: Using generational garbage collection to implement cache-conscious data placement. In Proceedings of the First International Symposium on Memory Management, volume 34(3) of ACM SIGPLAN Notices, October 1998
2. Ding, C, Kennedy, K.: Improving cache performance in dynamic applications through data and computation reorganization at run time. In Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation, Atlanta, GA, May 1999
3. Zhang, H., Martonosi, M.: A Mathematical Cache Miss Analysis for Pointer Data Structures. SIAM Conference on Parallel Processing for Scientific Computing, March, 2001
4. Draper, N. R., Smith, H.: Applied Regression Analysis. 2nd Ed.., John Wiley & Sons, 1981
5. Burger, D. C., Austin, T. M., Bennett, S.: Evaluating future Microprocessors: The SimpleScalar Tool Set. Technical Report CS-TR-96-1308, University of Wisconsin-Madison, July 1996
6. Wu, K., Sendag, R., Lilja, D. J.: Using a Self-tuning Adaptive Predictor to Characterize the Regularity of Memory Accesses in Pointer-Intensive Application Programs. University of Minnesota Technical Report: ARCTiC 2003