

Dynamically-Partitioned Level-1 Instruction/Data Caches

Casey Morrison

Adam Barnett

Department of Electrical and Computer Engineering

University of Florida

{caseym82,kamochan}@ufl.edu

Abstract– Two designs for a dynamically-partitioned level-1 instruction/data cache are proposed and evaluated through simulation. Each of the proposed architectures combine Instruction Cache (I-cache) and Data Cache (D-cache) into a single level-1 memory while maintaining a physical separation between instructions and data. The first proposed architecture is a *Way-partitioned* cache (WP-cache) that is designed to partition each set such that some ways are dedicated to instructions and the remainder are dedicated to data. The second proposed architecture is a *Set-partitioned* cache (SP-cache) that is designed to allocate a certain number of sets for instructions and the remainder of the sets for data. The distribution of instructions and data in each of the proposed caches can be dynamically adjusted over time according to I-cache and D-cache miss rates. After simulating both architectures with several benchmark cache traces, the WP-cache is shown to have marginal but inconsistent performance benefits while the SP-cache is shown to have consistent performance degradation.

I. INTRODUCTION

WHEN it comes to modern memory hierarchy designs, the Harvard Architecture has become the predominant structure for high-performance, low-latency memory systems. A machine that uses a Harvard Architecture has a single main memory and separate instruction and data caches. The benefits of having separate caches for instructions and data are twofold: (1) a higher bandwidth can be achieved by accessing both instruction cache and data cache at the same time; and (2) memory is protected from unintentional corruption by keeping instructions and data separated.

Typical implementations of level-1 cache provide for equally-sized Instruction Cache (I-cache) and Data Cache (D-cache). In addition, it is common for each of these caches to implement the same degree of associativity and similar replacement strategies. Fig. 1(a) and Fig. 1(b) illustrate a typical Harvard Architecture implementation of separate I-cache and D-cache.

| Set | Way | Word 0 | Word 1 | Word 2 | Word 3 | Valid | Tag |
|-----|-----|--------|--------|--------|--------|-------|-----|
| 0 | 0 | | | | | | |
| | 1 | | | | | | |
| | 2 | | | | | | |
| | 3 | | | | | | |
| 1 | 0 | | | | | | |
| | 1 | | | | | | |
| | 2 | | | | | | |
| | 3 | | | | | | |
| ... | | | | | | | |
| Ni | 0 | | | | | | |
| | 1 | | | | | | |
| | 2 | | | | | | |
| | 3 | | | | | | |

Fig. 1(a). 2-way set associative I-cache

| Set | Way | Word 0 | Word 1 | Word 2 | Word 3 | Valid | Tag |
|-----|-----|--------|--------|--------|--------|-------|-----|
| 0 | 0 | | | | | | |
| | 1 | | | | | | |
| | 2 | | | | | | |
| | 3 | | | | | | |
| 1 | 0 | | | | | | |
| | 1 | | | | | | |
| | 2 | | | | | | |
| | 3 | | | | | | |
| ... | | | | | | | |
| Nd | 0 | | | | | | |
| | 1 | | | | | | |
| | 2 | | | | | | |
| | 3 | | | | | | |

Fig. 1(b). 2-way set associative D-cache

This paper explores the potential performance benefits that can be achieved by implementing a single level-1 cache that is dynamically partitioned into separate, but not necessarily equally-sized, instruction and data caches. Performance metrics gathered for each cache in a real-time fashion are used to periodically adjust the distribution.

Similar architectures have been proposed in the past [1], but not for the explicit purpose of partitioning between I-cache and D-cache in a real-time, hardware-implemented sense. The authors of [1] discuss the benefits of partitioning a cache into several user-defined “columns,” whereas this paper focuses on a single partition for the purposes of separating instructions from data. Furthermore, the proposed methods of partitioning will be implemented strictly in hardware and will allow for dynamic repartitioning. There may, however, be some benefits to creating additional partitions, but that is outside the scope of this paper.

II. DESIGN

Two methods of dynamic cache partitioning are proposed: (1) *Way Partitioning* and (2) *Set Partitioning*. Each of these architectures preserves some key Harvard Architecture benefits, including simultaneous access and data integrity preservation.

A. *Way-Partitioned Cache*

One method of dynamically partitioning a cache is to allocate some ways within each cache set to instruction and allocate the rest of the ways to data. With *Way Partitioning*, each set in the cache is distributed the same, as determined by a Global Partition Register (GPR). The GPR serves as a global pointer to the first data way within each set. In addition, minimum and maximum distributions are defined so that, for example, no more than 75% of the cache can be dedicated to instruction or data. Fig. 2 illustrates a way-partitioned cache in a particular state such that 37.5% of the cache is dedicated to instruction (ways zero through two), and 62.5% is dedicated to data (ways three through seven).

As cache accesses are made, the WP-cache will periodically examine the difference between the I-cache miss rate and the D-cache miss rate. If this difference exceeds some pre-defined threshold (e.g. 1% difference), then the cache will do three things: (1) modify its GPR in favor of the worst performing cache; (2) invalidate any cache lines that changed hands as a result of repartitioning; and (3) reset the cache access counter. The cache access counter simply counts cache accesses up to a certain number—the repartitioning period—and triggers a repartitioning event.

This architecture does require some additional hardware beyond what a standard set associative cache would require. Because instructions can occupy, in this example, ways zero through five at any given moment, a set of comparators must operate in parallel to determine if there is an instruction hit (the same is true for determining a data hit). As shown in Fig. 3, the outputs of these comparators are multiplexed by the GPR, which contains the number of the first data way. This multiplexer represents the only additional latency incurred by this design as compared to the standard set associative cache.

Besides the additional multiplexer delay, this architecture also incurs some significant design costs. The additional comparators required to implement this design is not trivial by any means. In the example structure shown in Fig. 3, the hardware cost would be a total of five comparators as opposed to the standard one used for each instruction and data set.

B. *Set-Partitioned Cache*

Another method of dynamically partitioning a cache is to allocate a certain number of cache sets for instructions and the remainder for data. With *Set Partitioning*, individual cache sets are claimed for instruction or data as the cache is first populated. These assignments may change hands several times until the pre-defined cache distribution is achieved. After this occurs, set assignments cannot change until the cache determines it is time to repartition.

| Set | Way | Word 0 | Word 1 | Word 2 | Word 3 | Valid | Tag |
|-----|-----|--------|--------|--------|--------|-------|-----|
| 0 | 0 | | | | | | |
| | 1 | | | | | | |
| | 2 | | | | | | |
| | 3 | | | | | | |
| | 4 | | | | | | |
| | 5 | | | | | | |
| | 6 | | | | | | |
| | 7 | | | | | | |
| 1 | 0 | | | | | | |
| | 1 | | | | | | |
| | 2 | | | | | | |
| | 3 | | | | | | |
| | 4 | | | | | | |
| | 5 | | | | | | |
| | 6 | | | | | | |
| | 7 | | | | | | |
| ... | | | | | | | |
| Ns | 0 | | | | | | |
| | 1 | | | | | | |
| | 2 | | | | | | |
| | 3 | | | | | | |
| | 4 | | | | | | |
| | 5 | | | | | | |
| | 6 | | | | | | |
| | 7 | | | | | | |

Fig. 2. 8-way set associative way-partitioned cache

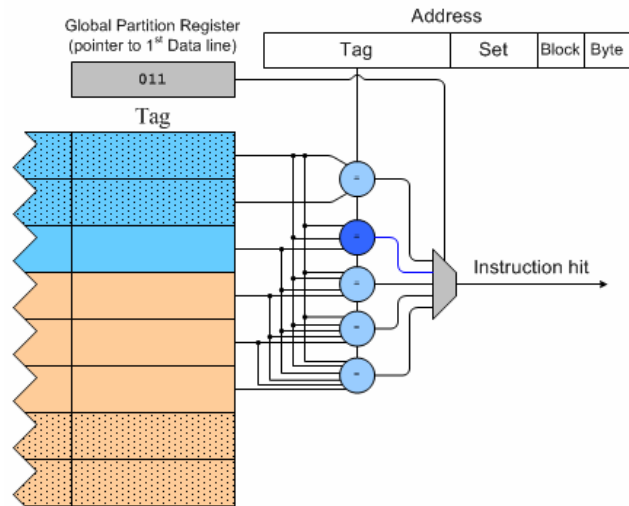


Fig. 3. Way partitioning comparator logic for one set

If a data cache access maps to a set that has been claimed by instruction, and if the maximum number of data sets has already been reached, then the microprocessor will be forced to fetch the data from Level-2 cache, and the instruction set will *not* be evicted. The instruction set may be evicted, however, if the maximum number of data sets has not yet been reached.

Fig. 4 illustrates a set-partitioned cache that has already been populated with memory words. The potential benefits of this architecture might not be readily apparent. If, for example, the particular program being executed consisted of many iterations of a relatively small code loop, then this architecture would allow I-cache to claim only the few sets that are needed to execute the loop of instructions. The remainder of the cache could be left for data to claim. This is, in theory, a much more efficient use of cache space as compared to having fixed-size I-cache and D-cache.

| Set | Type | Way | Word 0 | Word 1 | Word 2 | Word 3 | Valid | Tag |
|-----|------|-----|--------|--------|--------|--------|-------|-----|
| 0 | I | 0 | | | | | | |
| | | 1 | | | | | | |
| 1 | D | 0 | | | | | | |
| | | 1 | | | | | | |
| 2 | D | 0 | | | | | | |
| | | 1 | | | | | | |
| 3 | I | 0 | | | | | | |
| | | 1 | | | | | | |
| 4 | D | 0 | | | | | | |
| | | 1 | | | | | | |
| 5 | I | 0 | | | | | | |
| | | 1 | | | | | | |
| 6 | I | 0 | | | | | | |
| | | 1 | | | | | | |
| 7 | I | 0 | | | | | | |
| | | 1 | | | | | | |
| ... | | | | | | | | |
| Ns | D | 0 | | | | | | |
| | | 1 | | | | | | |

Fig. 4. 2-Way set associative set-partitioned cache

The additional costs associated with the SP-cache are not as significant. In addition to the standard hardware required for a Harvard Architecture implementation, a type comparison must be made during every cache access. This is a two-bit comparison (e.g. 2'b00 denotes "not claimed," 2'b01 denotes "claimed by instruction," and 2'b10 denotes "claimed by data"). This additional cost is insignificant both in terms of latency and transistor count.

III. EVALUATION METHODOLOGY

To evaluate the effectiveness of the two proposed cache architectures, a custom cache simulation environment was constructed. Although the quantity of evaluations performed was not exhaustive, the quality of the methodology used was enough to reinforce the results that follow.

A. Simulation Environment

A three-part simulation environment was created for the purpose of evaluating the Way-partitioned Set-partitioned cache architectures. Fig. 5 illustrates the simulation flow discussed in the following sections.

1) *Cache Trace Extraction*: Using the SimpleScalar tool set for the Alpha processor architecture, cache traces for various benchmark programs were extracted. SimpleScalar's cache.c file was modified to dump all level-1 cache accesses to a file. With this modification in place, benchmark program binaries such as cc1.alpha, perl.alpha, anagram.alpha, and go.alpha were simulated for the purpose of extracting their unique cache traces.

2) *Cache Simulation*: A custom cache simulation environment was developed in the C language. Data structures were created to model three different types of caches: (1) a "control" cache, (2) a Way-partitioned cache, and (3) a Set-partitioned cache. The control cache was a standard Harvard Architecture level-1 cache with physically separate I-cache and D-cache. The two proposed architectures were implemented as a single physical memory with partitioned I-cache and D-cache. Each of the three cache structures were simulated with the benchmark cache traces, and the simulator would periodically output cache performance statistics to a text file. See Appendix III for the complete code listings for this simulator.

3) *Cache Performance Comparison*: The cache performance statistics produced by the custom cache simulator were fed into a data processing environment created in Microsoft Excel. Comparisons were made between each cache implementation in terms of I-cache miss rate, D-cache miss rate, and total cache miss rate. Performance statistics were plotted as a function of cache access.

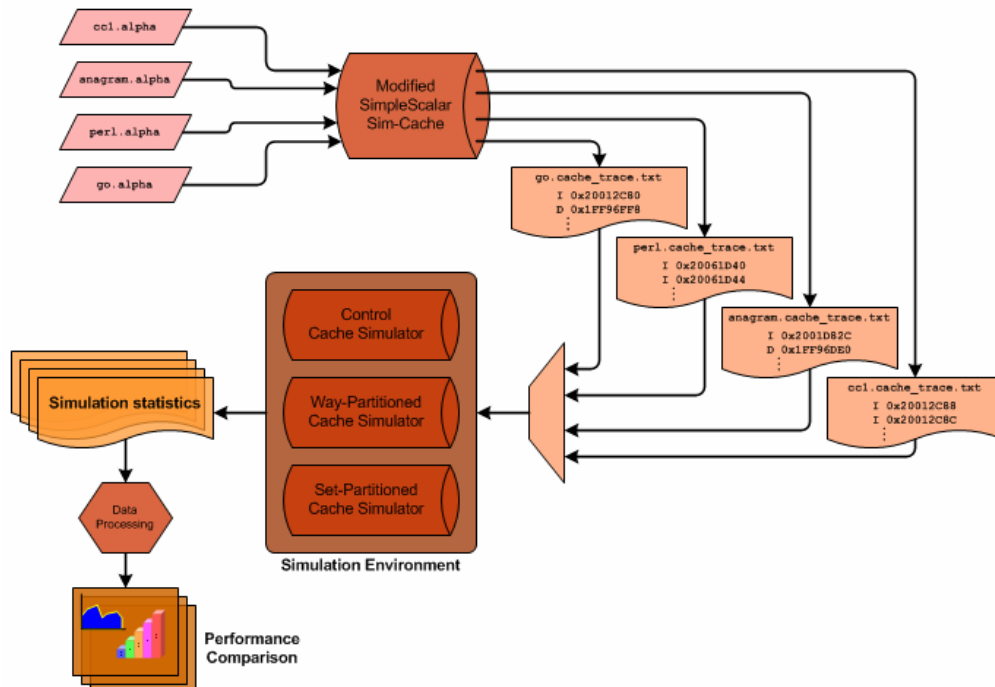


Fig. 5. Cache simulation flow

B. Simulation Parameters

Numerous simulations were run on each of the proposed cache architectures, and the configurations that showed to be the most promising are presented below.

1) *Control Cache for Way-Partitioning Comparison*: The control cache used to evaluate the WP-cache was configured to provide a good size and associativity comparison for the WP-cache. The exact configuration is described in Table I.

2) *Control Cache for Set-Partitioning Comparison*: The control cache used to evaluate the SP-cache was slightly different from the former control cache. It was configured with less associativity to provide for a more accurate comparison with the low associativity of the SP-cache. The exact configuration is described in Table II.

3) *Way-Partitioned Cache*: The WP-cache configuration chosen for evaluation allows for the dynamic partitioning of eight cache ways per cache set. The size of the WP-cache is the same as that of its control cache. The exact configuration of the WP-cache is described in Table III.

4) *Set-Partitioned Cache*: The SP-cache chosen for evaluation has a lower associativity than its Way-partitioned counterpart. This allows for a greater granularity when it comes to distribution between instructions and data. The exact configuration of the SP-cache is described in Table IV.

TABLE I
CONTROL CACHE CONFIGURATION FOR WP-CACHE

| Parameter | Value |
|----------------------|-------|
| I-cache size | 8 KB |
| D-cache size | 8 KB |
| Associativity | 4-way |
| Replacement strategy | LRU |

TABLE II
CONTROL CACHE CONFIGURATION FOR SP-CACHE

| Parameter | Value |
|----------------------|-------|
| I-cache size | 8 KB |
| D-cache size | 8 KB |
| Associativity | 2-way |
| Replacement strategy | LRU |

TABLE III
WAY-PARTITIONED CACHE CONFIGURATION

| Parameter | Value |
|--------------------------|---------------------------|
| Cache size | 16 KB |
| Associativity | 8-way |
| Repartitioning period | 100,000 cache accesses |
| Repartitioning threshold | 0.01 miss rate difference |
| Repartitioning step size | 1 way |
| Replacement strategy | LRU |

TABLE IV
SET-PARTITIONED CACHE CONFIGURATION

| Parameter | Value |
|--------------------------|---------------------------|
| Cache size | 16 KB |
| Associativity | 2-way |
| Repartitioning period | 100,000 cache accesses |
| Repartitioning threshold | 0.02 miss rate difference |
| Repartitioning step size | 16 sets |
| Replacement strategy | LRU |

IV. EXPERIMENTAL RESULTS

Simulations were performed using the aforementioned cache configurations, and the results that follow are mixed.

A. Way-Partitioned Cache

The proposed Way-partitioned cache exhibited unique performance characteristics and marginal improvements. As a result of the periodic redistribution of cache resources, the miss rates for I-cache and D-cache tend to be oscillatory. Fig. 6 and Fig. 7 depict the I-cache and D-cache miss rates over time for the GCC benchmark. In contrast to the behavior of the control cache in Fig. 6, the WP-cache exhibits oscillatory behavior as illustrated in Fig. 7.

When comparing the total cache performance of the WP-cache versus the control cache, the WP-cache demonstrated marginal but inconsistent improvements over the control cache. Fig. 8 and Fig. 9 show at most a half-percent improvement over the control cache in the latter half of program execution.

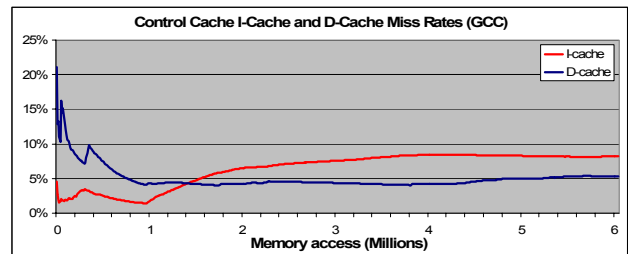


Fig. 6. Control cache instruction/data miss rate (GCC)

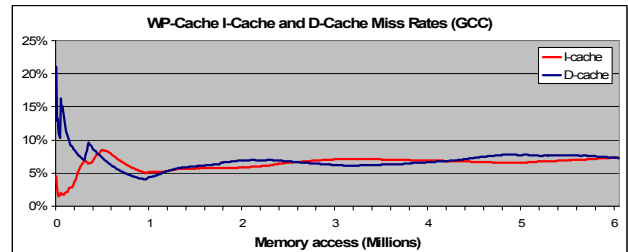


Fig. 7. WP-cache instruction/data miss rate (GCC)

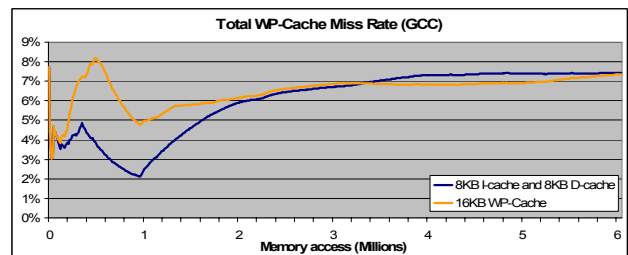


Fig. 8. Total WP-cache performance (GCC)

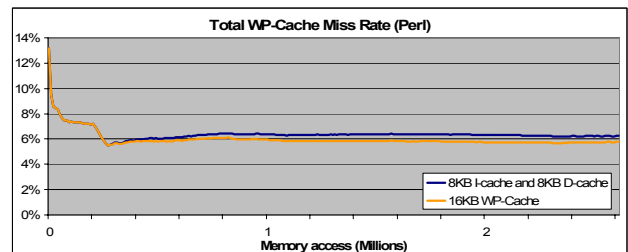


Fig. 9. Total WP-cache performance (Perl)

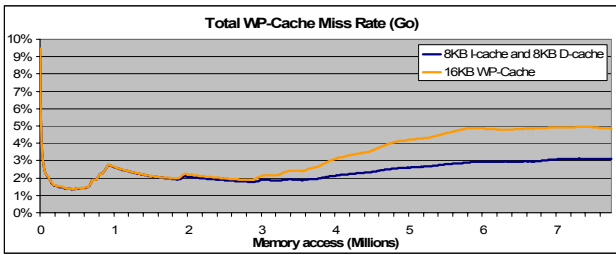


Fig. 10. Total WP-cache performance (Go)

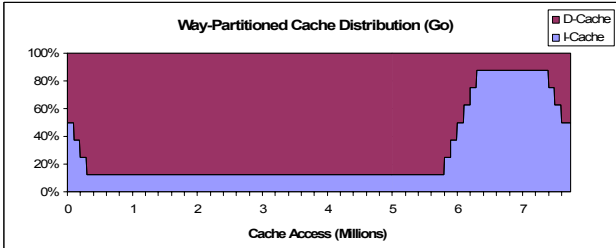


Fig. 11. WP-cache distribution over time (Go)

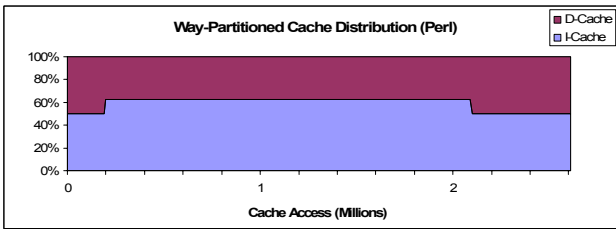


Fig. 12. WP-cache distribution over time (Perl)

Fig. 10 shows a performance degradation as compared to the control cache. This performance loss gets as large as two percent towards the end of the Go benchmark. This performance loss may be attributed to the extreme repartitioning that occurs during the simulation of the Go benchmark as illustrated by Fig. 11. Other benchmarks, like Perl shown in Fig. 12, exhibit less extreme repartitioning.

As a result of the disparity between I-cache and D-cache performance inherent in the Go benchmark, the WP-cache makes an extreme effort to mitigate the miss rate difference—an effort that results in a loss of total cache performance.

In total, the WP-cache outperformed the control cache, in the long run, for two of the four benchmarks (GCC and Perl). The control cache, however, outperformed the WP-cache in the other two benchmarks (Anagram and Go).

B. Set-Partitioned Cache

The Set-partitioned cache showed less promising results than the Way-partitioned cache. In every measure, the total cache performance for the SP-cache was worse than that of the control cache to which it was being compared. Like the WP-cache, the SP-cache demonstrated oscillatory behavior for its I-cache and D-cache components. Fig. 13 illustrates this behavior for the Go benchmark.

As shown in Fig. 14, the SP-cache has a significantly higher total miss rate than the control cache. This may be attributed to the restrictive effects of set claiming that are part of the proposed SP-cache architecture. When instruction or data makes a claim to a particular set, every subsequent

access of the opposite type will be a miss. This negates the fundamental advantages of a set associative cache.

Fig. 15 shows the distribution of the SP-cache over time (cache accesses) for the Go benchmark. When the difference between the I-cache miss rate and the D-cache miss rate exceeds the threshold (in this case, 0.02), the cache is repartitioned such that 16 sets are “given” to the worst performing sub-cache (I-cache or D-cache). For example, if I-cache is performing significantly worse than D-cache, then I-cache will be granted the right to evict up to 16 sets that are currently claimed for D-cache. Fig. 16 illustrates the initial population of the SP-cache during the first 25,000 accesses of the Go benchmark.

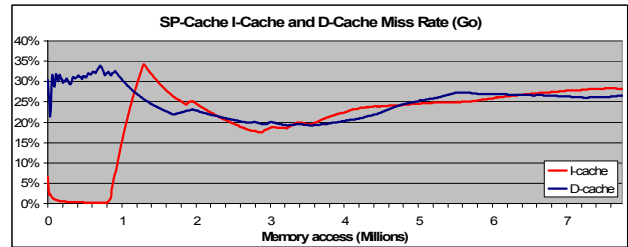


Fig. 13. SP-cache instruction/data miss rate (Go)

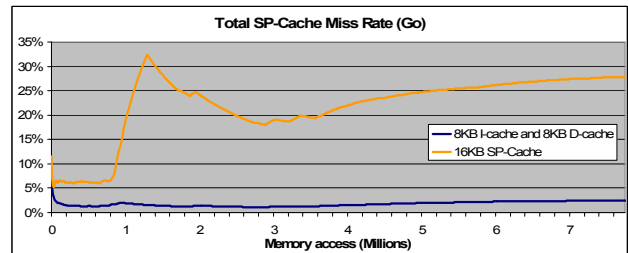


Fig. 14. Total SP-cache performance (Go)

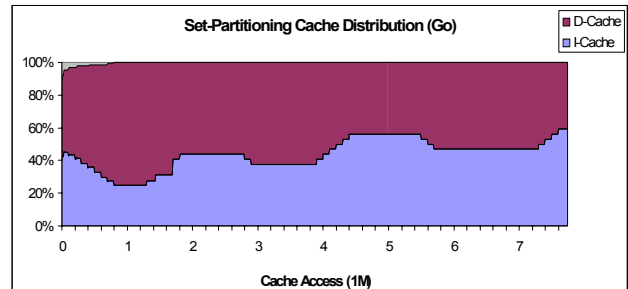


Fig. 15. SP-cache distribution over time (Go)

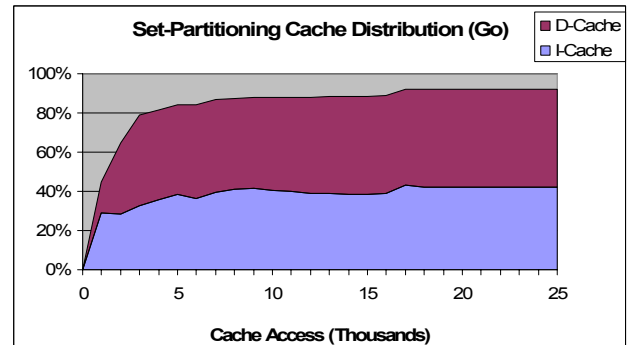


Fig. 16. SP-cache population over time (Go)

V. CONCLUDING REMARKS

Although the goal of dynamic cache partitioning is to reduce the overall miss rate of level-1 cache, it appears that the actual effect was to minimize the difference between the I-cache miss rate and the D-cache miss rate. This, in turn, had little effect on reducing the overall cache miss rate.

The Way-partitioned cache showed some modest performance improvements over the traditional Harvard Architecture, but these improvements were relatively small and inconsistent. Conversely, the Set-partitioned cache showed consistent and significant performance degradation, making it an impractical approach to dynamic cache partitioning.

The repartitioning threshold and frequency has much to do with the overall performance of the two proposed caches. This fact was established during the initial parameter-adjustment trials. It is clear that the optimum values for these parameters depend heavily on the benchmark being simulated, and a program-independent set of optimal parameters has yet to be found.

It is possible that, with moderate adjustments to the cache parameters and architecture, these two techniques could prove to have consistent and worthwhile performance benefits. However, given the data collected throughout these experiments, it is conclusive that the costs associated with implementing either of these cache architectures significantly outweigh any foreseeable performance improvements.

REFERENCES

- [1] B. Ang, D. Chiou, S. Devadas, L. Rudolph, "Dynamic Cache Partitioning via Columnization," *Computation Structures Group Memo 430*, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, November 1999.
- [2] D. Bacon. (1994, Aug.) Cache Advantage. [Online]. Available: <http://www.byte.com/art/9408/sec6/art4.htm>
- [3] B. Juurlink, "Unified Dual Data Caches," *Proc. of the Euromicro Symposium on Digital System Design*, 2003.
- [4] D. Kaeli. Profile-Guided Instruction and Data Memory Layout. [Online]. Available: <http://www.ece.neu.edu/info/architecture/publications/Tufts.pdf>
- [5] P. Srivatstan, P. B. Sudarshan, P. P. Bhaskaran. Dynora: A New Caching Technique. [Online]. Available: <http://ieeexplore.ieee.org/iel5/8715/27588/01231902.pdf>
- [6] G. E. Suh, L. Rudolph, S. Devadas, "Dynamic Partitioning of Shared Cache Memory," *The Journal of Supercomputing*, vol. 28, pp. 7-26, 2004.
- [7] S. Yang, M. Powell, B. Falsafi, T. N. VijayKumar. Exploiting Choice in Resizable Cache Design to Optimize Deep Submicron Processor Energy Delay. [Online]. <http://ieeexplore.ieee.org/iel5/7816/21483/00995706.pdf?arnumber=995706>

APPENDIX I
CACHE PERFORMANCE PLOTS

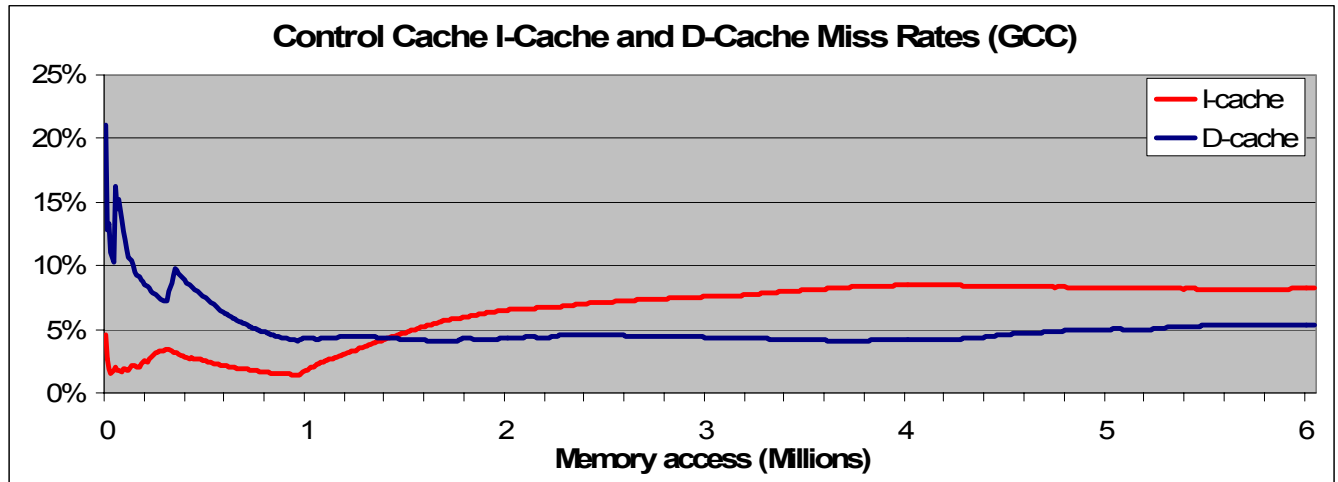


Fig. 17. Way-partitioning control cache instruction/data miss rate (GCC)

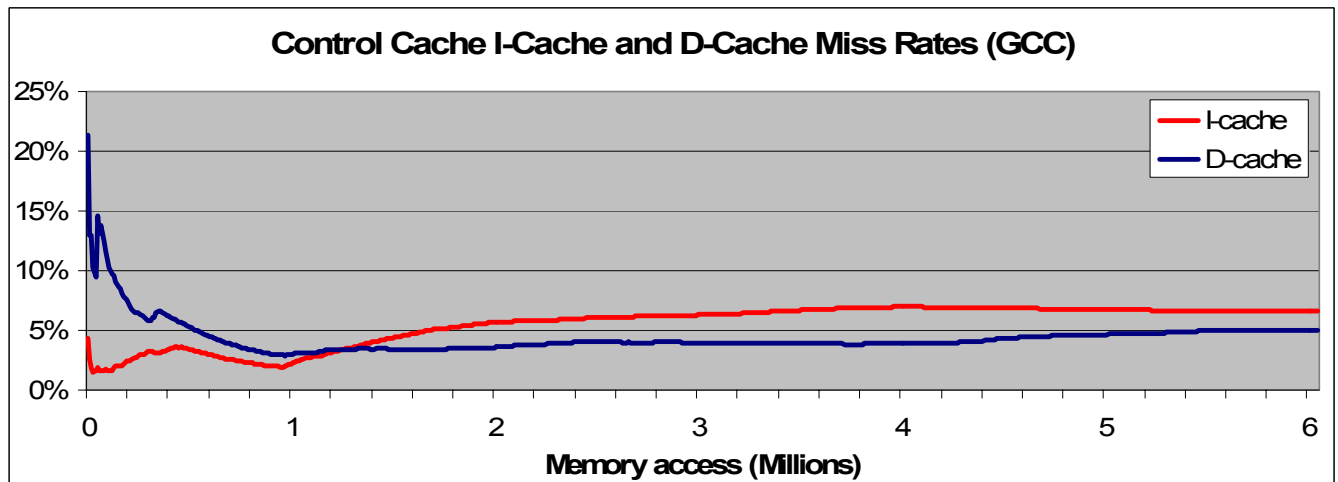


Fig. 18. Set-partitioning control cache instruction/data miss rate (GCC)

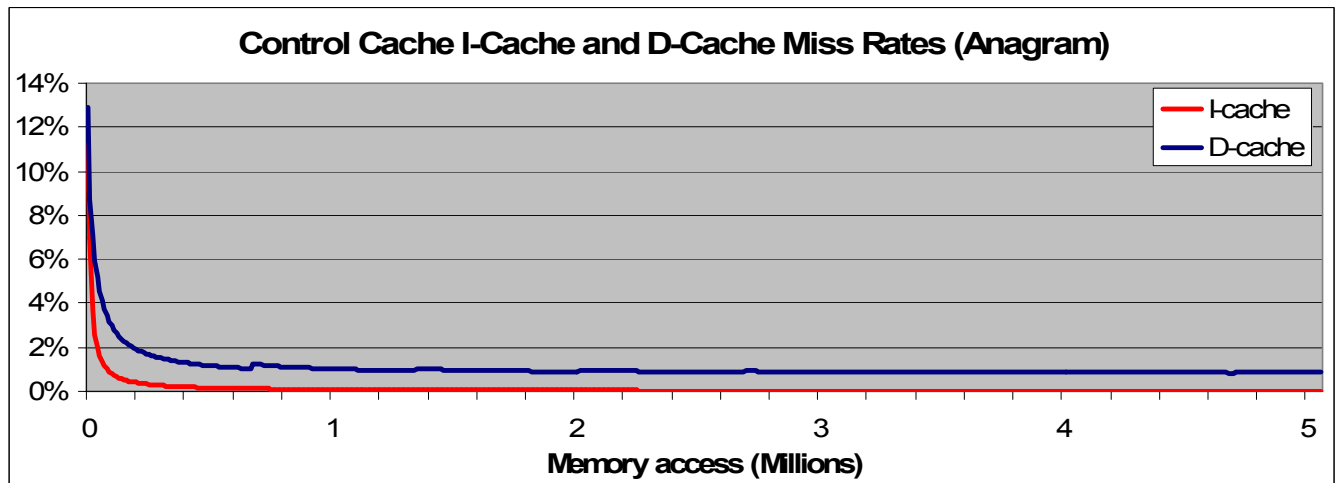


Fig. 19. Way-partitioning control cache instruction/data miss rate (Anagram)

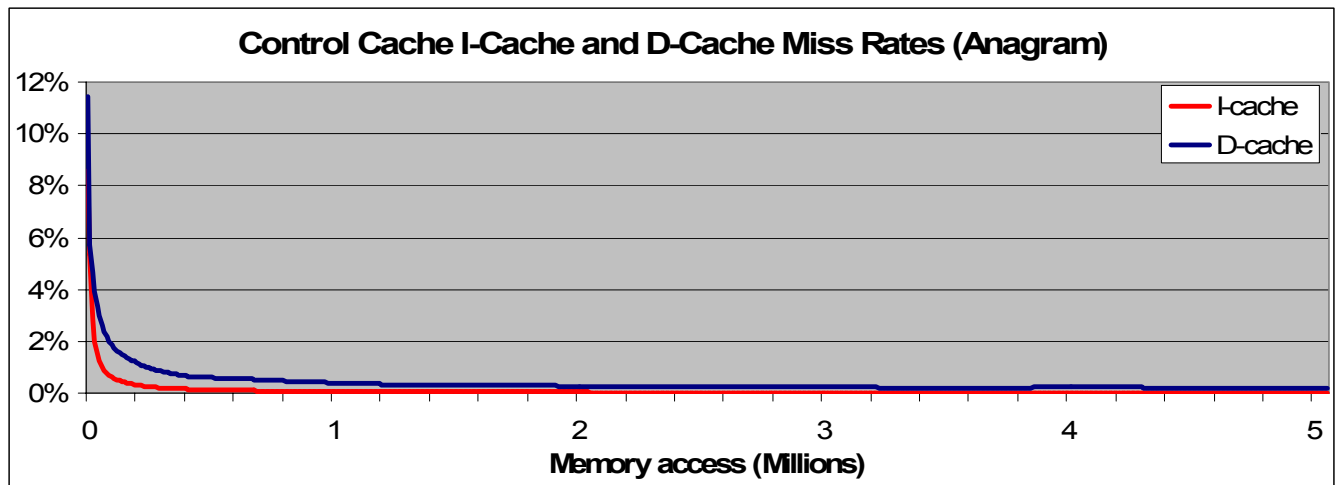


Fig. 20. Set-partitioning control cache instruction/data miss rate (Anagram)

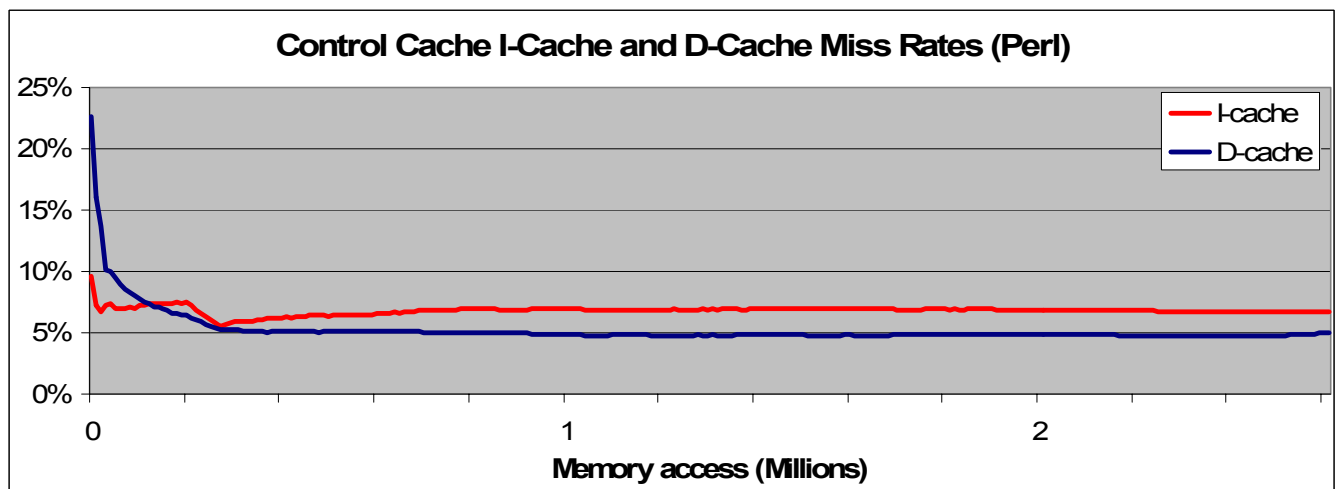


Fig. 21. Way-partitioning control cache instruction/data miss rate (Perl)

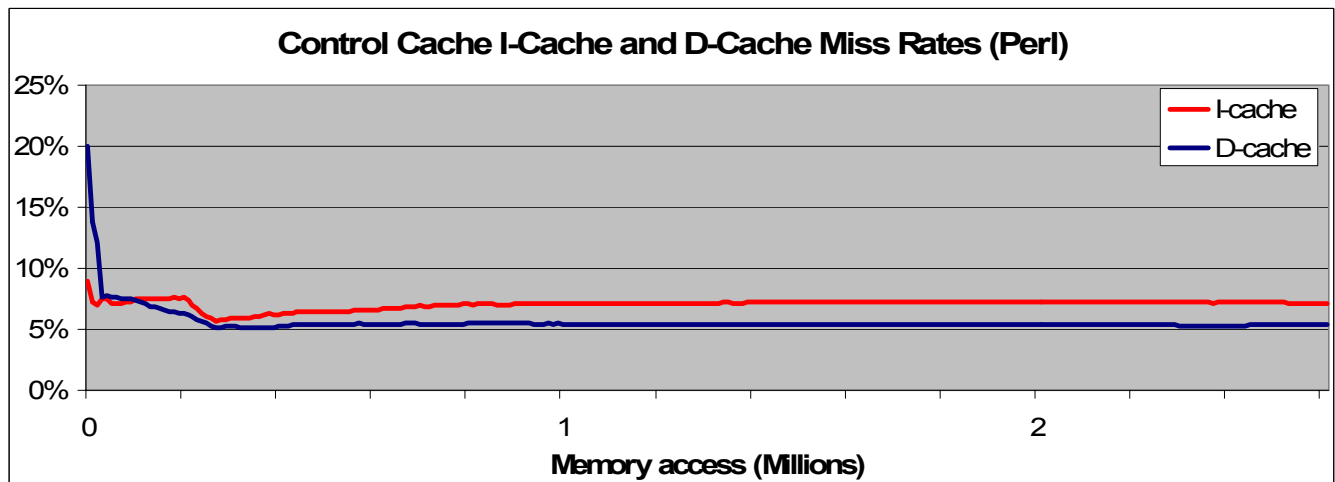


Fig. 22. Set-partitioning control cache instruction/data miss rate (Perl)

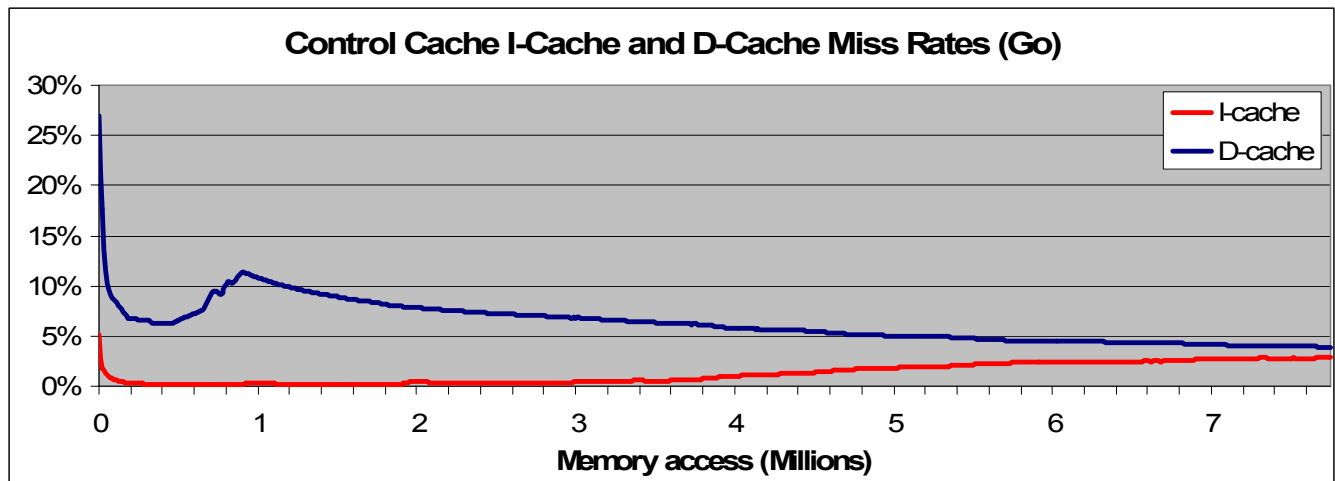


Fig. 23. Way-partitioning control cache instruction/data miss rate (Go)

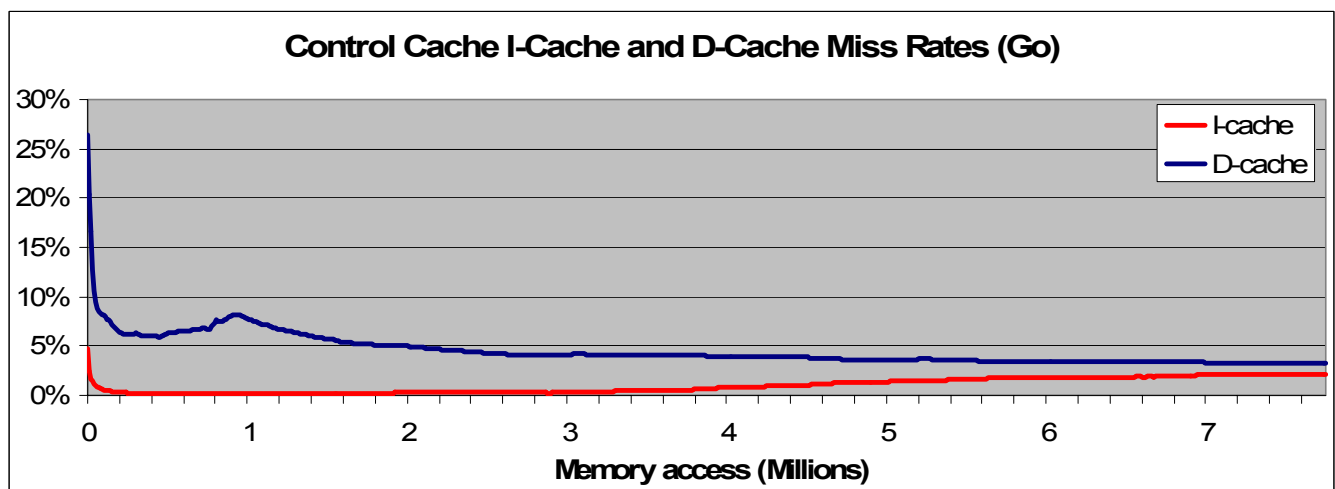


Fig. 24. Set-partitioning control cache instruction/data miss rate (Go)

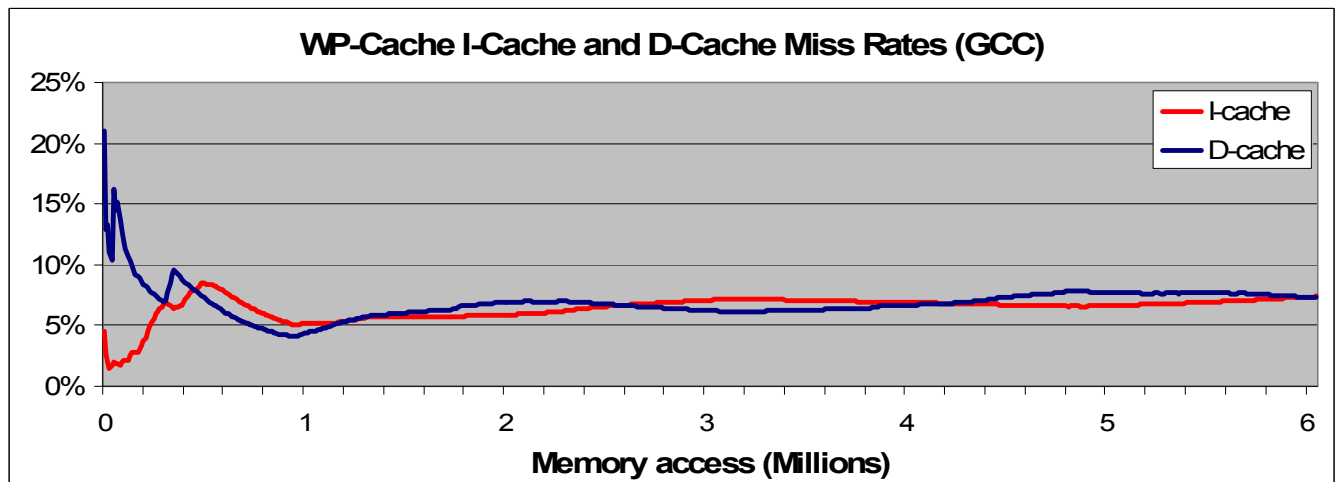


Fig. 25. WP-cache instruction/data miss rate (GCC)

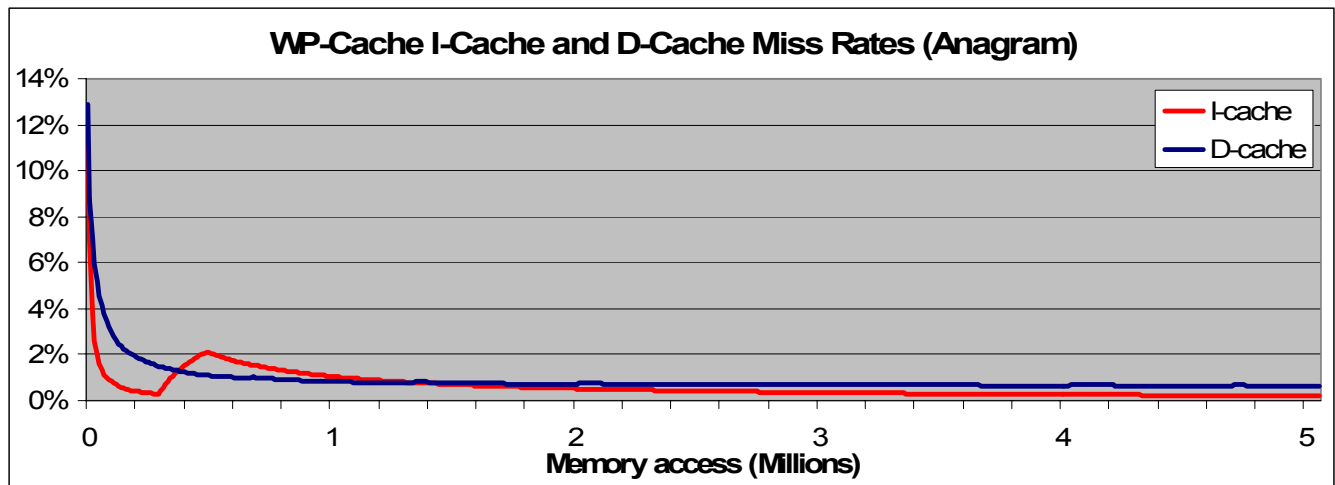


Fig. 26. WP-cache instruction/data miss rate (Anagram)

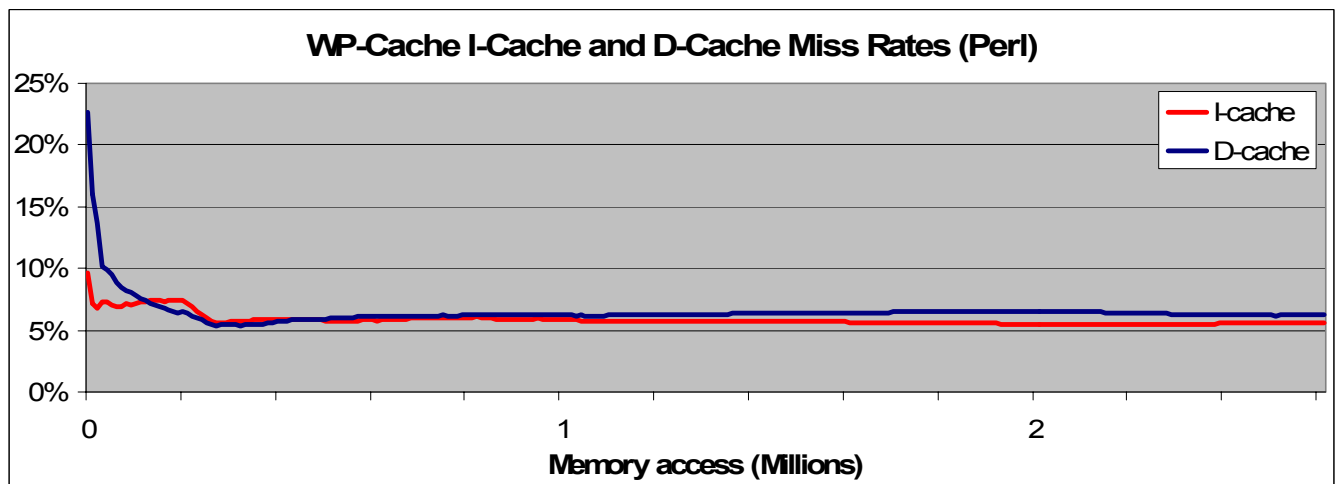


Fig. 27. WP-cache instruction/data miss rate (Perl)

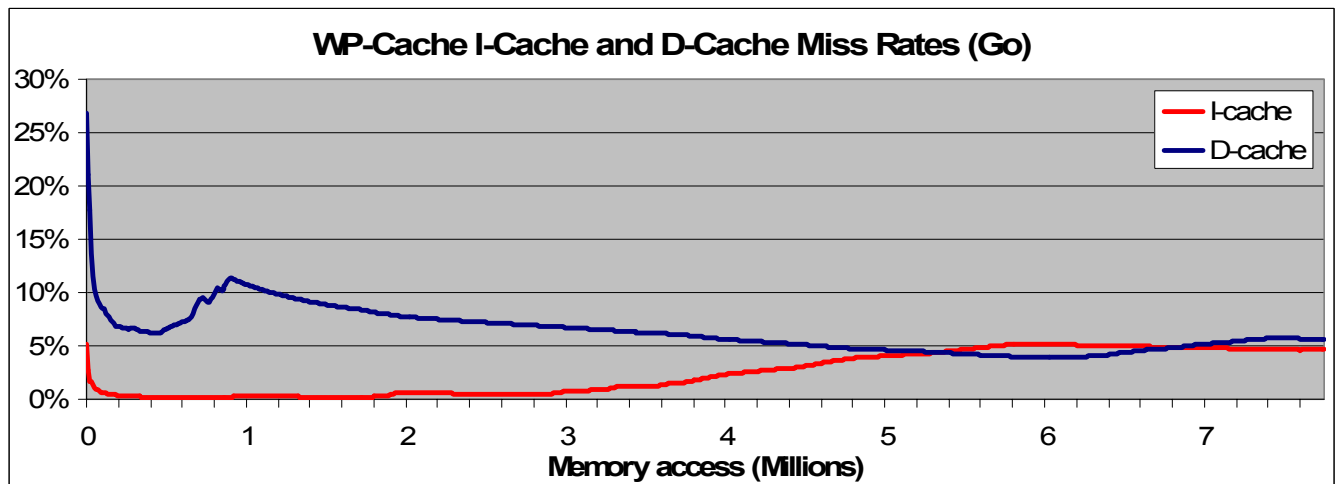


Fig. 28. WP-cache instruction/data miss rate (Go)

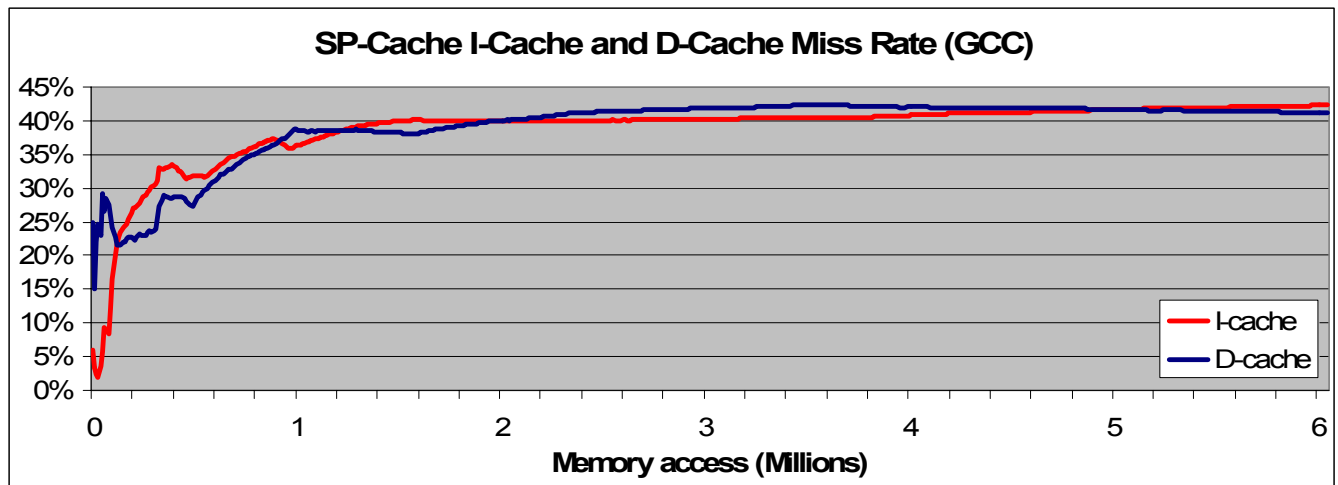


Fig. 29. SP-cache instruction/data miss rate (GCC)

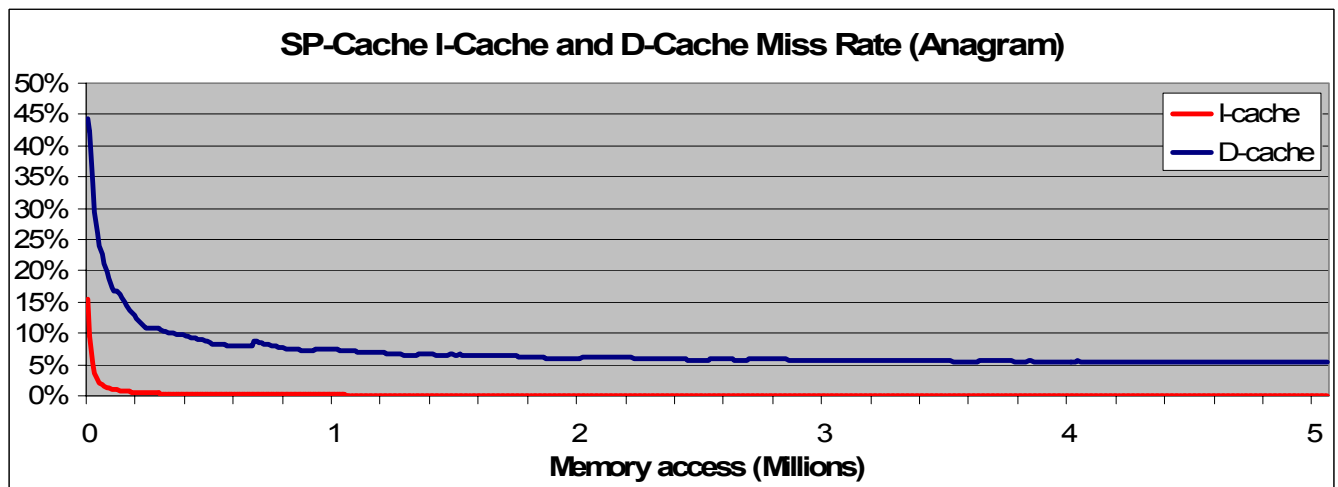


Fig. 30. SP-cache instruction/data miss rate (Anagram)

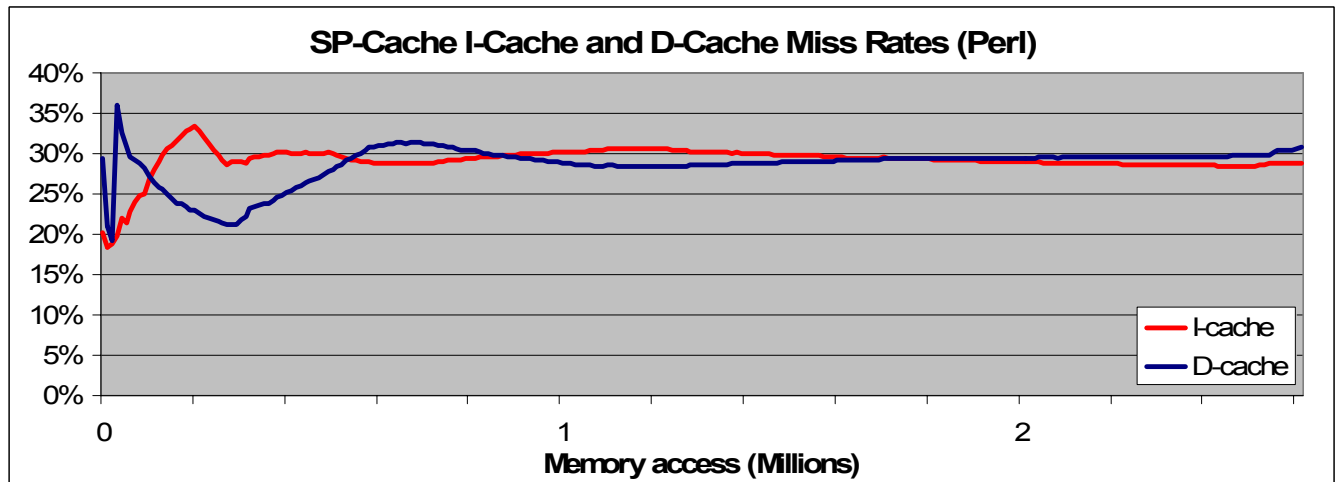


Fig. 31. SP-cache instruction/data miss rate (Perl)

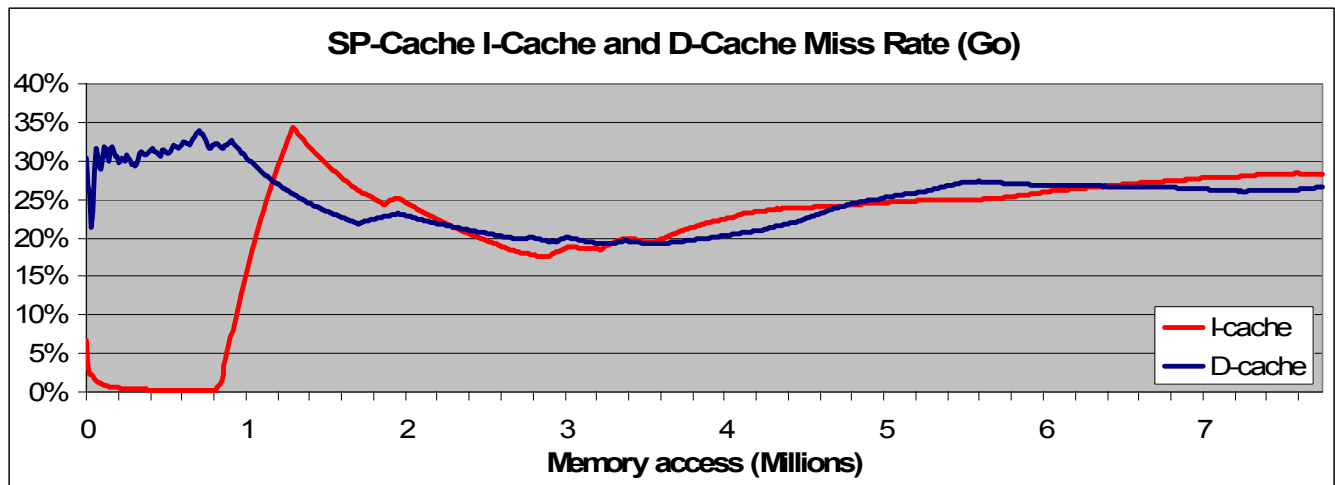


Fig. 32. SP-cache instruction/data miss rate (Go)

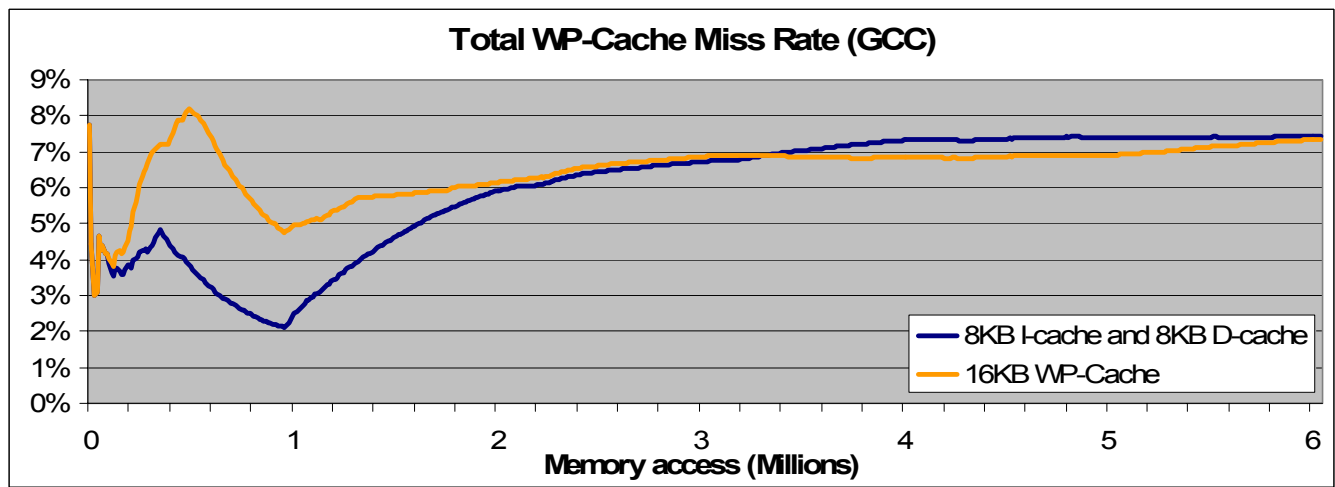


Fig. 33. Total WP-cache performance (GCC)

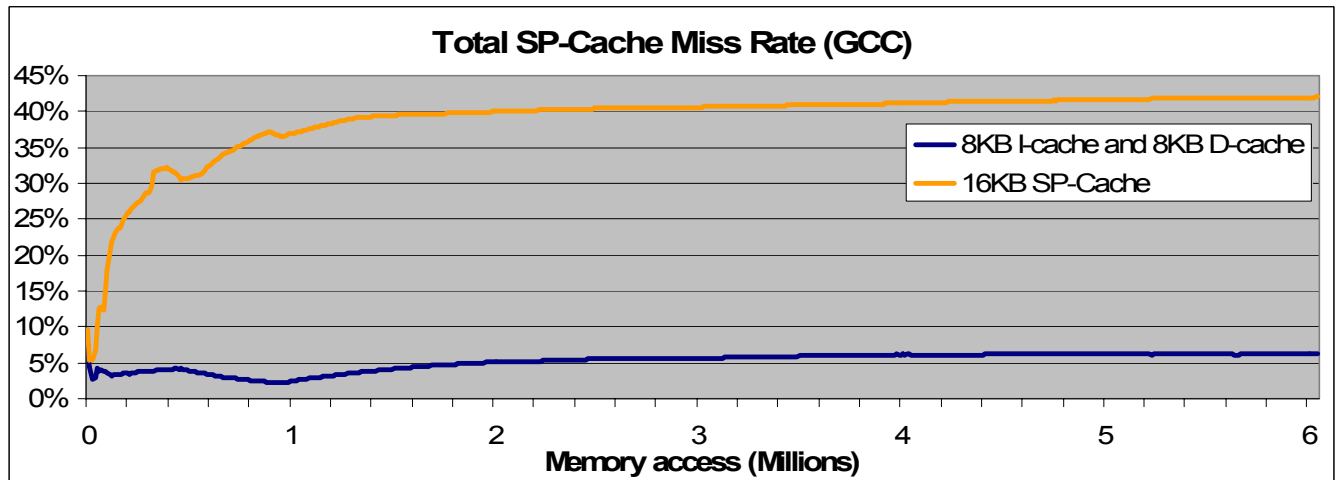


Fig. 34. Total SP-cache performance (GCC)

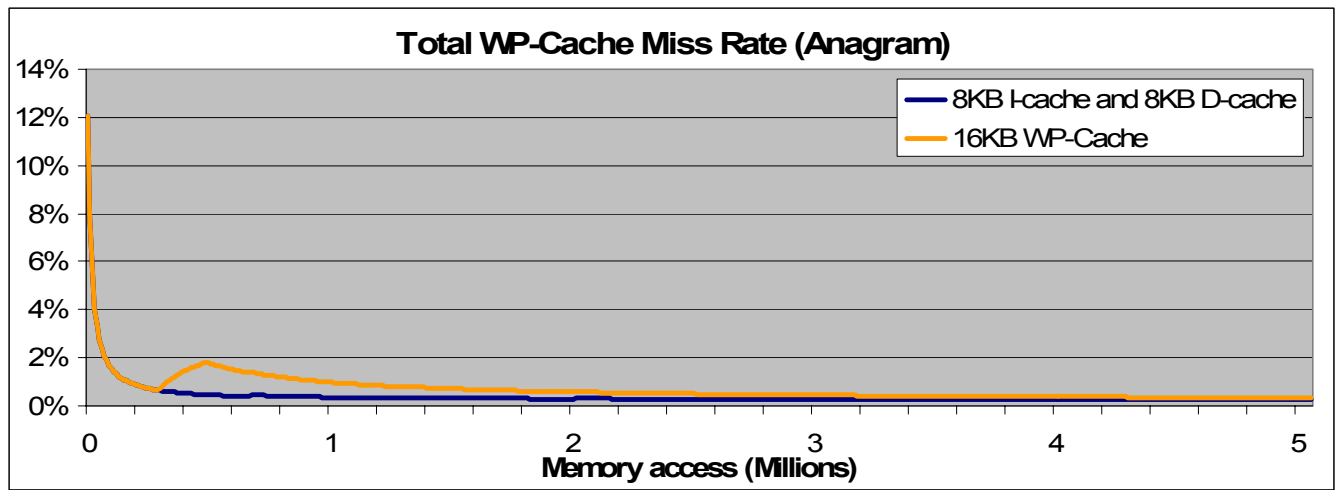


Fig. 35. Total WP-cache performance (Anagram)

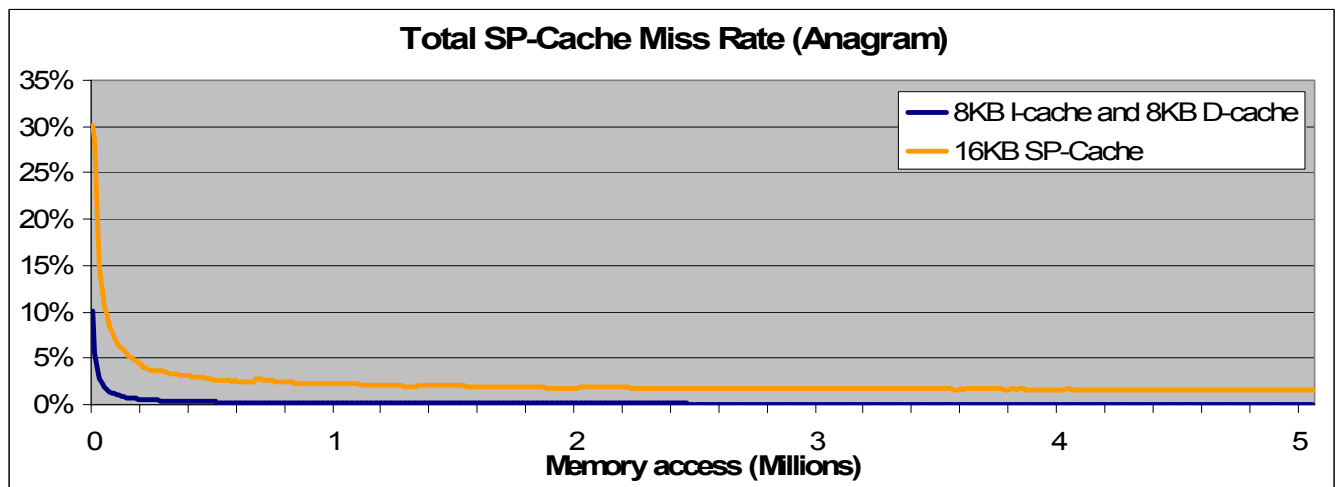


Fig. 36. Total SP-cache performance (Anagram)

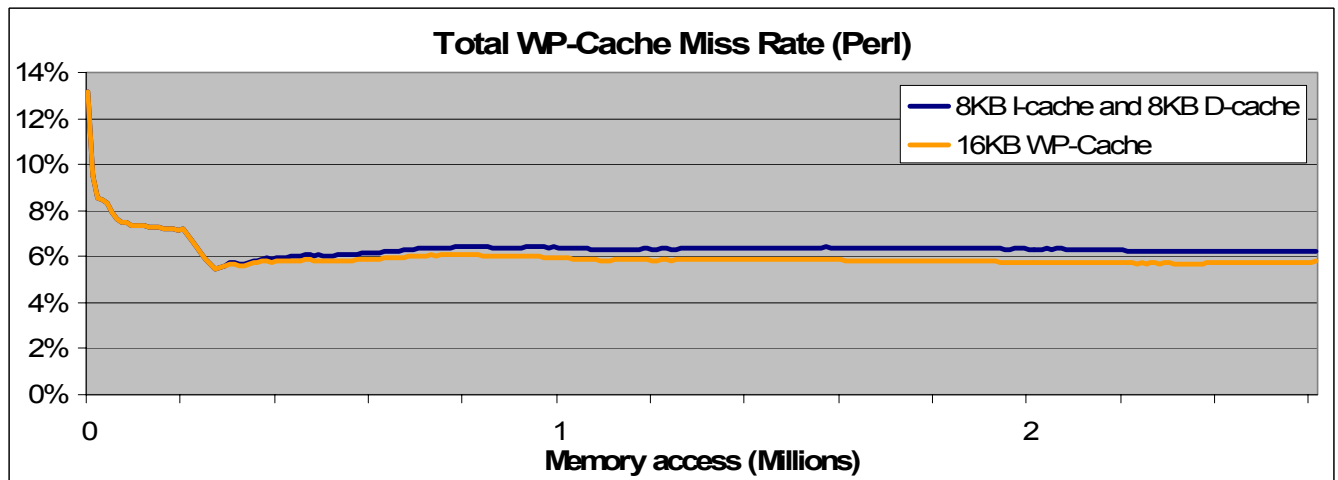


Fig. 37. Total WP-cache performance (Perl)

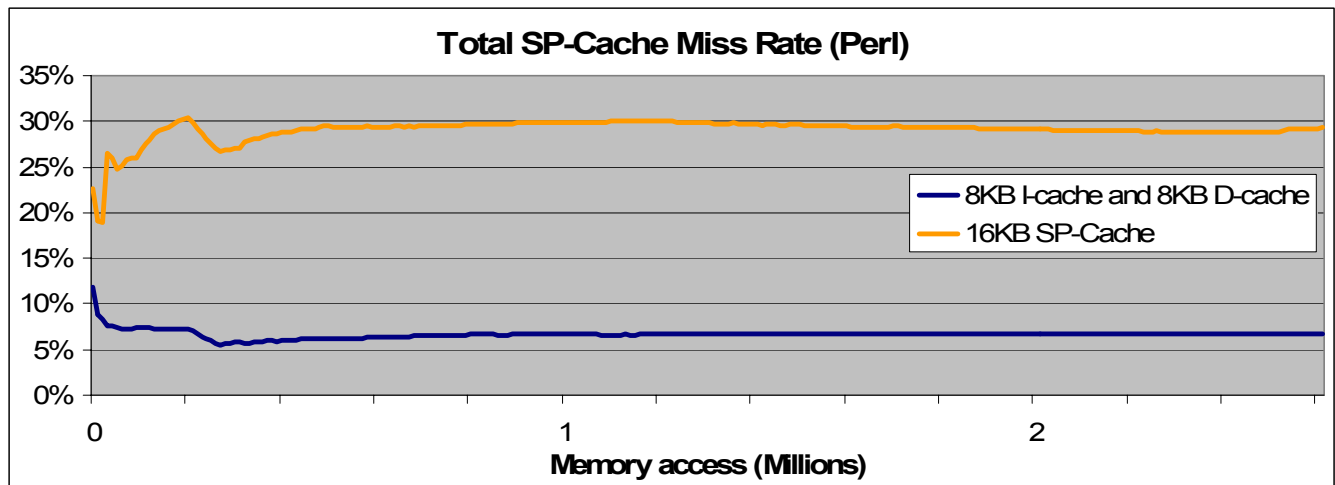


Fig. 38. Total SP-cache performance (Perl)

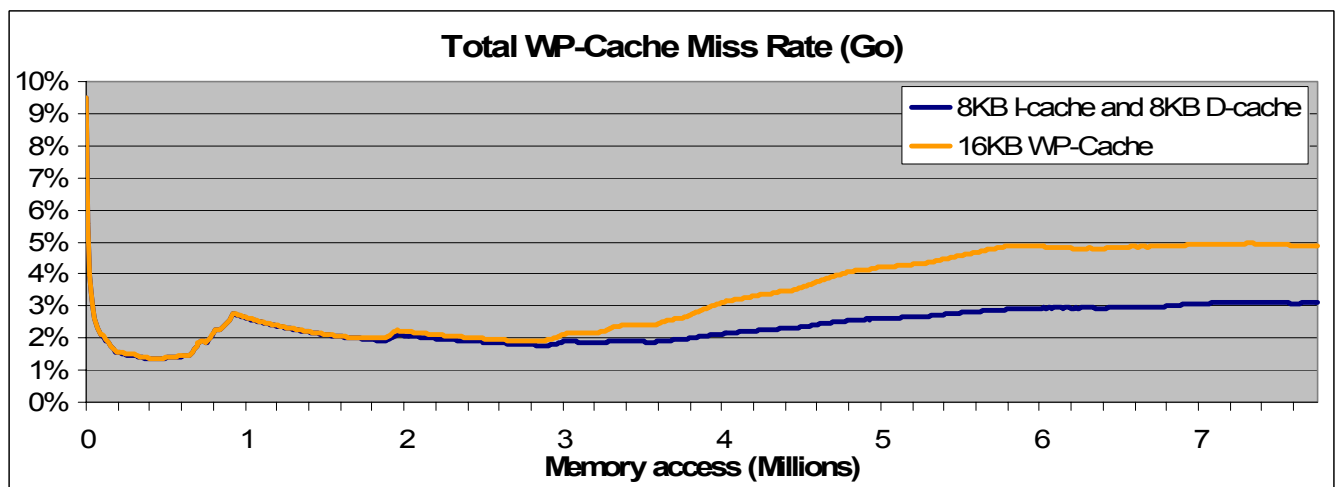


Fig. 39. Total WP-cache performance (Go)

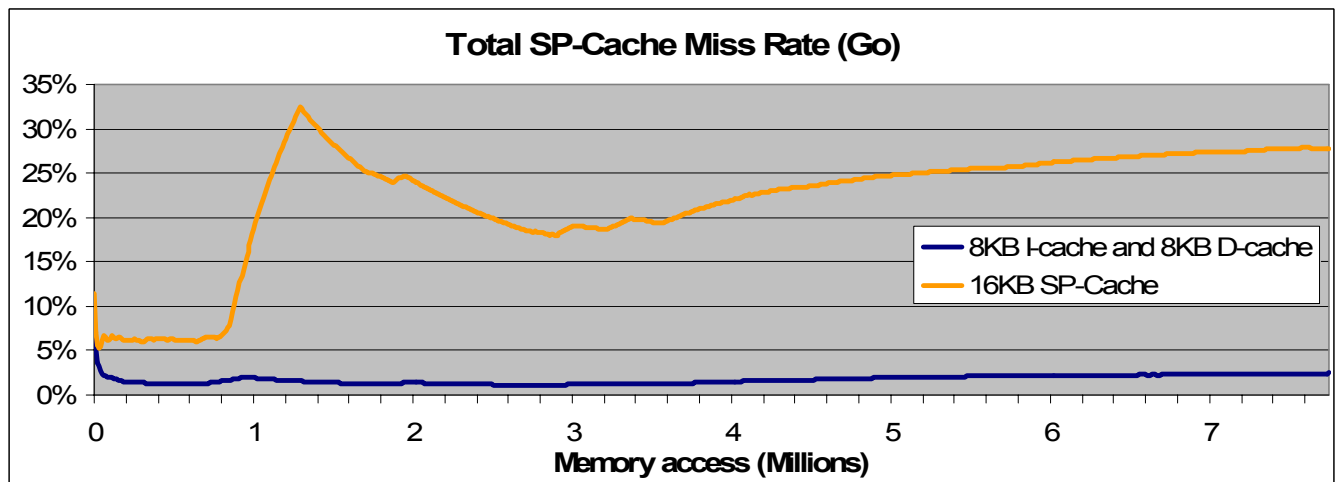


Fig. 40. Total SP-cache performance (Go)

APPENDIX II
CACHE DISTRIBUTION PLOTS

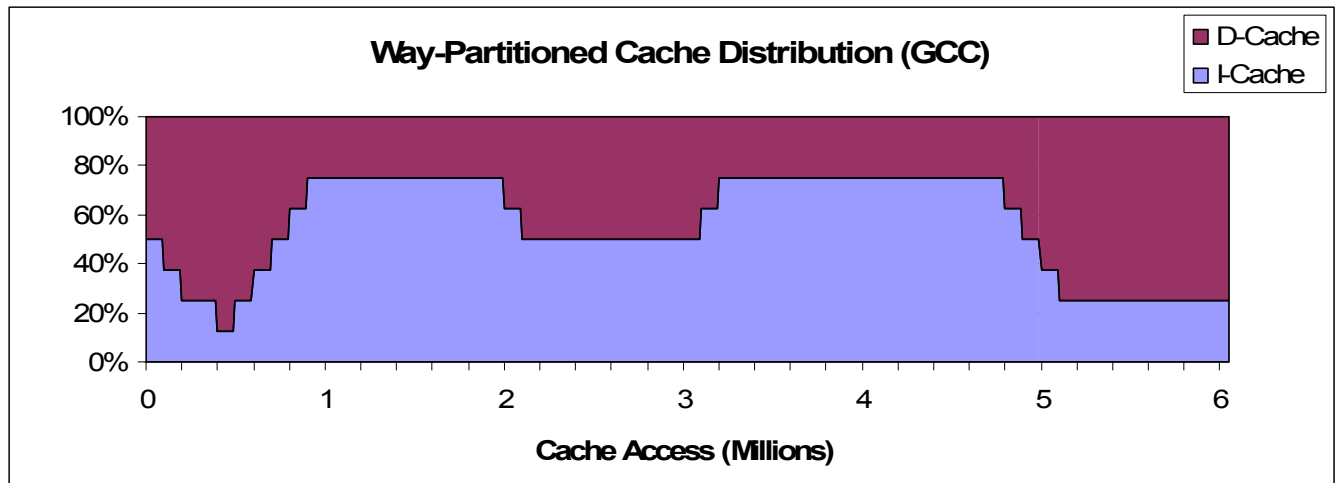


Fig. 41. WP-cache distribution over time (GCC)

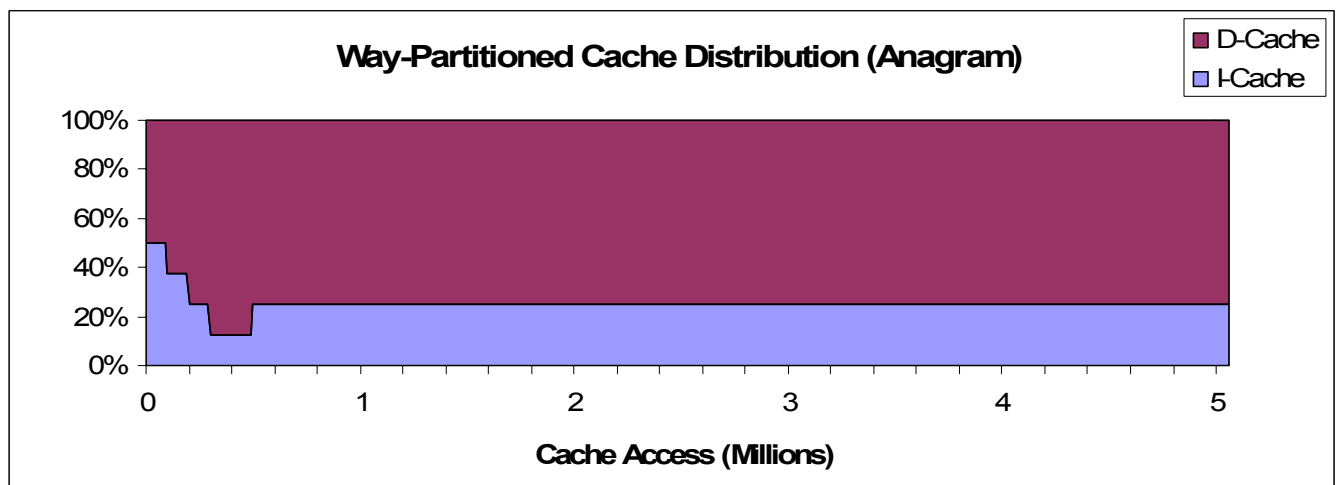


Fig. 42. WP-cache distribution over time (Anagram)

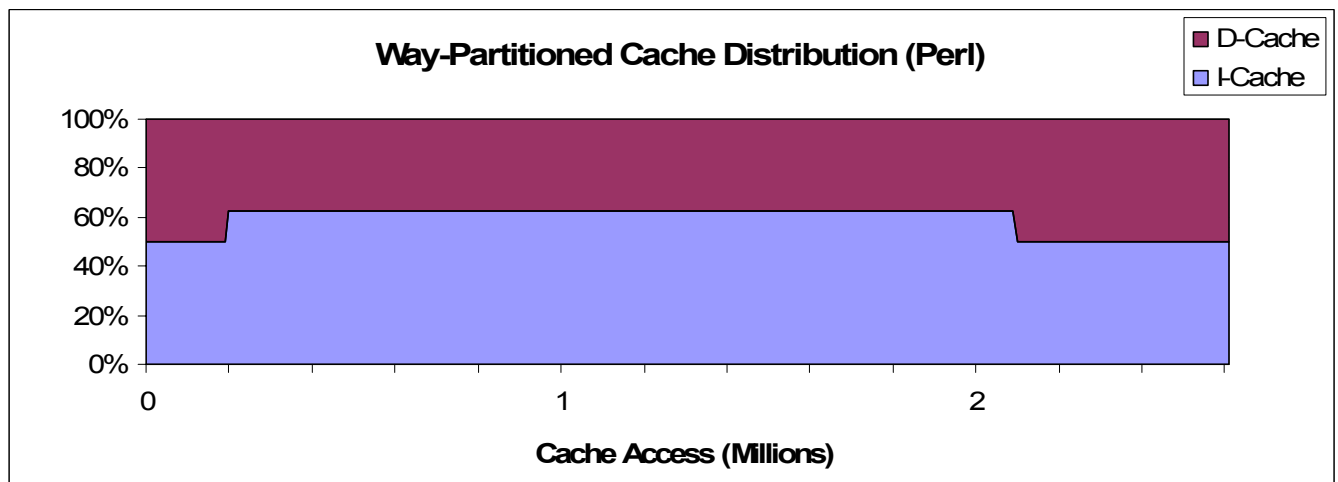


Fig. 43. WP-cache distribution over time (Perl)

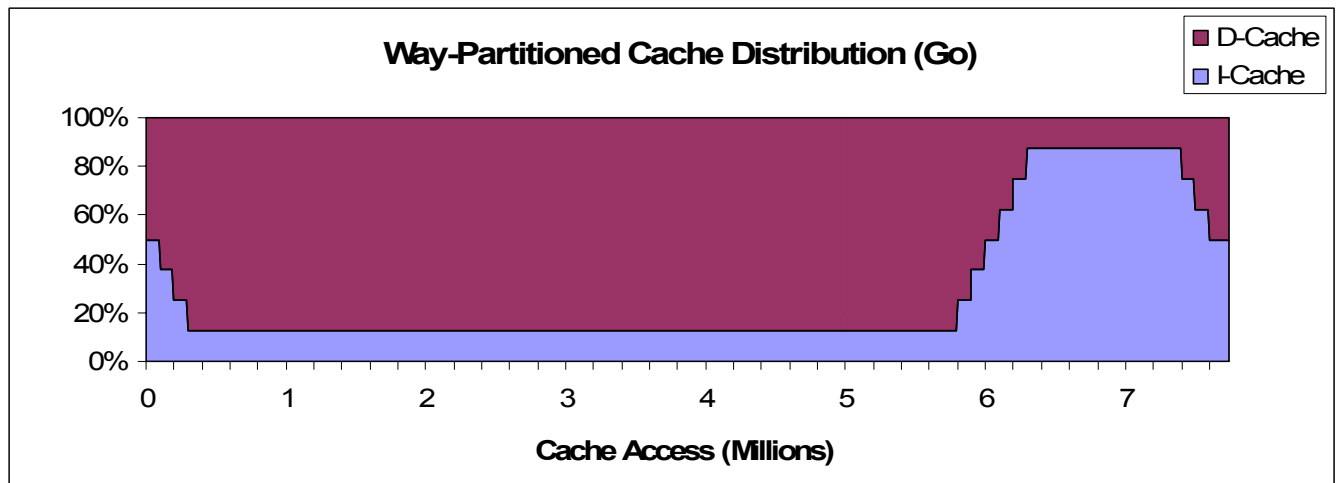


Fig. 44. WP-cache distribution over time (Go)

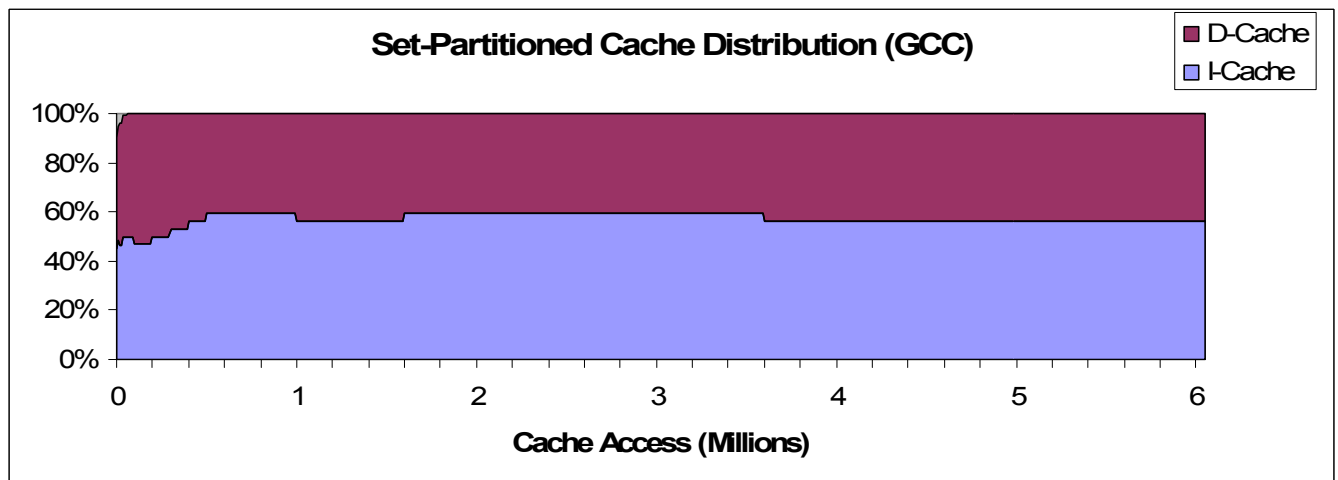


Fig. 45. SP-cache distribution over time (GCC)

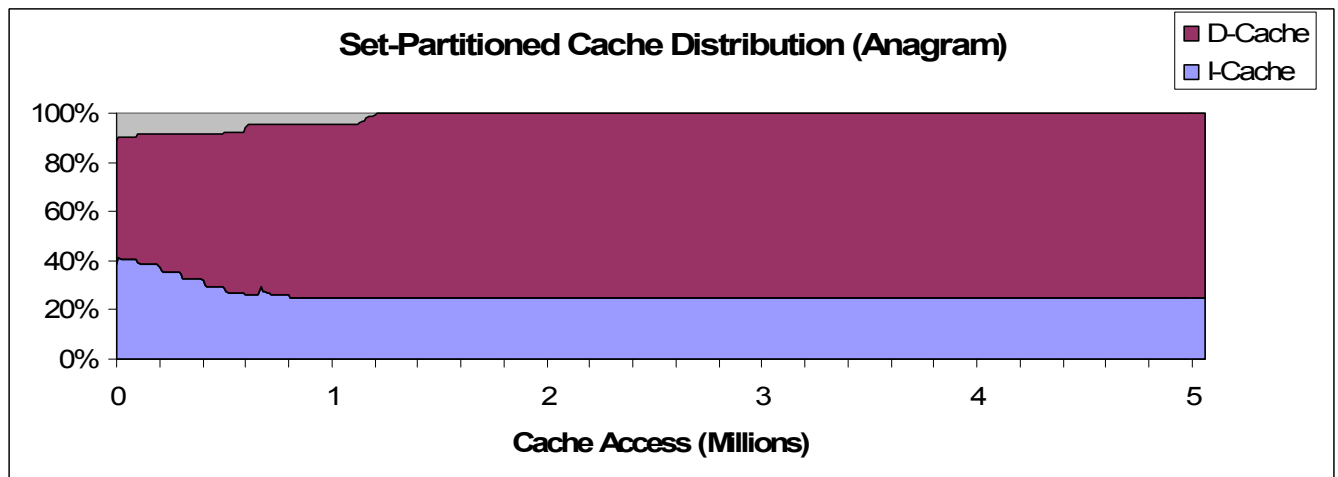


Fig. 46. SP-cache distribution over time (Anagram)

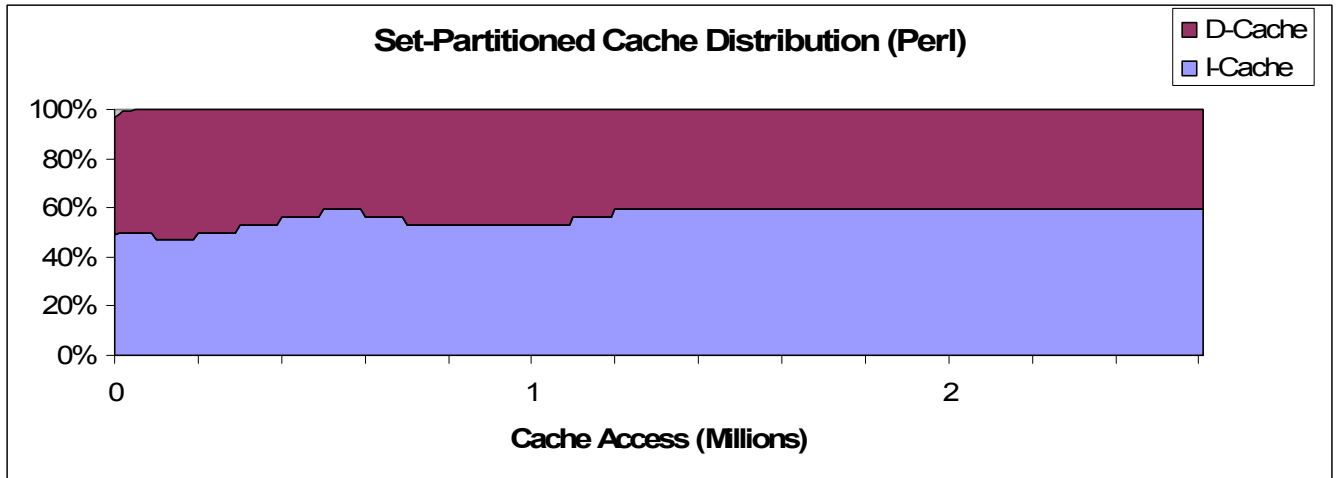


Fig. 47. SP-cache distribution over time (Perl)

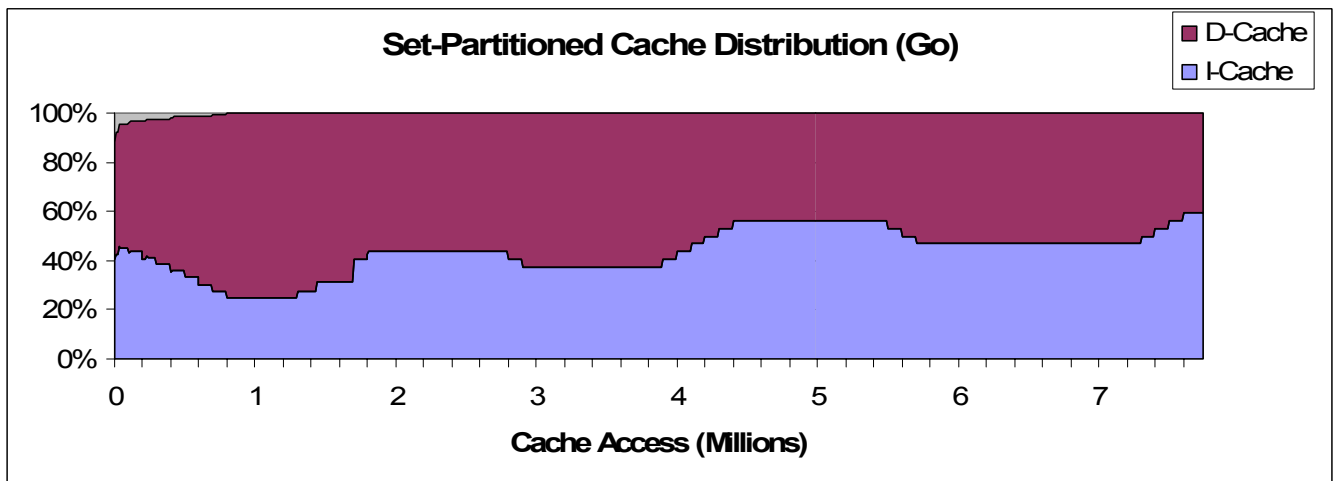


Fig. 48. SP-cache distribution over time (Go)

APPENDIX III
CACHE SIMULATOR CODE

Listing 1. Main function (cache_sim.c)

```

////////////////////////////////////
// Barnett Morrison Level-1 Cache Simulator
//
// File:    cache_sim.c
//
// Authors: Adam Barnett
//          Casey Morrison
//
// Date:    03 Nov 2006
//
// Description: This is the main file which instantiates the cache
//              and runs the simulation.
////////////////////////////////////

////////////////////////////////////
// Include Files
////////////////////////////////////
#include "cache.h"
#include "defines.h"
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <string.h>

////////////////////////////////////
// Defines
////////////////////////////////////
#ifndef REPARTITION_STEP
#define REPARTITION_STEP 0
#endif

////////////////////////////////////
// Global Variables
////////////////////////////////////
int access_num;
int i_access_num;
int d_access_num;
int i_misses;
int d_misses;
char * sim_type;

////////////////////////////////////
// Function: main()
////////////////////////////////////
int main(int argc, char *argv[]) {

    // Function prototypes
    int parse_line(char *, char *, unsigned int *);
    int access_wp_cache(struct cache *, char, unsigned int);
    int access_sp_cache(struct cache *, char, unsigned int);
    int repartition(struct cache *);
    void print_sp_set(struct cache *, unsigned int);
    void print_sp_header();

    // Process command line parameters
    char * file_name;
    char * arg1;

    // Simulation type
    arg1 = (char*)(argv[1]);
    if ((strcmp(arg1,"wp") == 0) || (strcmp(arg1,"WP") == 0)) {
        sim_type = WAY_PARTITION;
    }
    else if ((strcmp(arg1,"sp") == 0) || (strcmp(arg1,"SP") == 0)) {
        sim_type = SET_PARTITION;
    }
    else {
        printf("ERROR: Invalid argument, %s\n", arg1);
        return 1;
    }
}

```

```

}

// Memory access file
file_name = argv[2];

// Instantiate a cache
struct cache cache;

// Initialize the system
access_num = 0;
i_access_num = 0;
d_access_num = 0;
i_misses = 0;
d_misses = 0;

if (sim_type == WAY_PARTITION) {
    cache.way_partition = NUM_WAYS/2;
    cache.min_way_partition = NUM_WAYS/8;
    cache.max_way_partition = NUM_WAYS/8 * 7;
}
else {
    cache.max_i_sets = NUM_SETS/2;
    cache.min_i_sets = NUM_SETS/4;
    cache.min_d_sets = NUM_SETS/4;
    cache.cur_i_sets = 0;
    cache.cur_d_sets = 0;
}

int i, j;
for (i = 0; i < NUM_SETS; i = i + 1) {
    for (j = 0; j < NUM_WAYS; j = j + 1) {
        cache.set[i].line[j].tag = 0;
        cache.set[i].line[j].valid = INVALID;
        cache.set[i].line[j].last_used = 0;
    }
    if (sim_type == SET_PARTITION) {
        cache.set[i].type = NONE;
    }
}

// Set up for File I/O
FILE *fp;
if ((fp = fopen(file_name, "r")) == NULL) {
    printf("ERROR: Cannot open file %s\n", file_name);
    return 1;
}

// Read memory access file
char line[10]; // Ten characters: one for I or D,
// eight for address, and one for NULL terminator
char end_of_line[2]; // This is the newline character at the end of each line
char access_type; // Type of cache access (I or D)
unsigned int address; // Address of cache access
int result; // Result of cache access (0 --> miss, 1 --> hit)
while (fgets(line, 10, fp) != NULL) {
    // Update access number
    access_num += 1;

    // Process memory reference
    parse_line(line, &access_type, &address);

    // Check for invalid values
    if (access_type == INST) {
        i_access_num += 1;
    }
    else if (access_type == DATA) {
        d_access_num += 1;
    }
    else {
        printf("ERROR: Invalid cache access type: %c\n", access_type);
        return 1;
    }
    if ((address < 0x00000000) || (address > 0xFFFFFFFF)) {
        printf("ERROR: Invalid address: 0x%X\n", address);
        return 1;
    }
}

```

```

    }

    // Access cache
    if (sim_type == WAY_PARTITION) {
        result = access_wp_cache(&cache, access_type, address);
    }
    else if (sim_type == SET_PARTITION) {
        result = access_sp_cache(&cache, access_type, address);
    }

    // Process a miss
    if (result == 0) {
        if (access_type == INST) {
            i_misses += 1;
        }
        else {
            d_misses += 1;
        }
    }

    // Print statistics every N accesses
    int N = 1000;
    if ((access_num % N) == 0) {
        if (sim_type == WAY_PARTITION) {
            printf("%d %d %d %d %d %d\n",
                access_num, i_access_num, i_misses, d_access_num, d_misses,
                cache.way_partition);
        }
        else {
            printf("%d %d %d %d %d %d %d %d\n",
                access_num, i_access_num, i_misses, d_access_num, d_misses,
                cache.cur_i_sets, cache.cur_d_sets, cache.max_i_sets);
        }
    }

    // Repartition every REPARTITION_T accesses
    if ((access_num % REPARTITION_T) == 0) {
        repartition(&cache);
    }

    // Read newline character to set up for reading next line
    fgets(end_of_line, 2, fp);
}

// Return
return 0;
} // End main()

////////////////////////////////////
// Function: parse_line()
////////////////////////////////////
int parse_line(char * line, char * type, unsigned int * addr) {
    // Retrieve the access type
    *type = line[0];

    // Retrieve the address
    char temp[9] = {line[1], line[2], line[3], line[4],
        line[5], line[6], line[7], line[8], '\0'};
    unsigned long int temp_addr;
    temp_addr = strtoul(temp, NULL, 16);
    *addr = (unsigned int)(temp_addr & UINT_MAX);

    return 0;
} // End parse_line

////////////////////////////////////
// Function: access_wp_cache()
////////////////////////////////////
int access_wp_cache(struct cache * c, char type, unsigned int addr) {
    // Determine set number and tag
    unsigned int set_num, block_num, tag;
    set_num = (addr & SET_NUM_MASK) >> 4;
    block_num = (addr & BLOCK_NUM_MASK) >> 2;
    tag = (addr & TAG_MASK) >> TAG_SHIFT;

```

```

// Search the set for tag
int result = 0; // Default to miss
int line_num;

// Process an I-Cache access
if (type == INST) {
    // Determine hit or miss
    for (line_num = 0; line_num < c->way_partition; line_num += 1) {
        if ((tag == c->set[set_num].line[line_num].tag) &&
            (c->set[set_num].line[line_num].valid == VALID)) {
            // Process a hit
            result = 1;
            c->set[set_num].line[line_num].last_used = access_num;
        }
    }

    // Process a miss
    if (result == 0) {
        // Find the true least-recently used
        int lru_line = 0;
        int current_line;
        for (current_line = 1; current_line < c->way_partition; current_line += 1) {
            if (c->set[set_num].line[current_line].last_used <
                c->set[set_num].line[lru_line].last_used) {
                lru_line = current_line;
            }
        }

        // Replace LRU with new tag (i.e. fetch from L2)
        c->set[set_num].line[lru_line].tag = tag;
        c->set[set_num].line[lru_line].valid = VALID;
        c->set[set_num].line[lru_line].last_used = access_num;
    }
}

// Process a D-Cache access
else if (type == DATA) {
    for (line_num = c->way_partition; line_num < NUM_WAYS; line_num += 1) {
        if ((tag == c->set[set_num].line[line_num].tag) &&
            (c->set[set_num].line[line_num].valid == VALID)) {
            // Process a hit
            result = 1;
            c->set[set_num].line[line_num].last_used = access_num;
        }
    }

    // Process a miss
    if (result == 0) {
        // Find the true least-recently used
        int lru_line = c->way_partition;
        int current_line;
        for (current_line = c->way_partition + 1; current_line < NUM_WAYS; current_line += 1) {
            if (c->set[set_num].line[current_line].last_used <
                c->set[set_num].line[lru_line].last_used) {
                lru_line = current_line;
            }
        }

        // Replace LRU with new tag (i.e. fetch from L2)
        c->set[set_num].line[lru_line].tag = tag;
        c->set[set_num].line[lru_line].valid = VALID;
        c->set[set_num].line[lru_line].last_used = access_num;
    }
}

return result;
} // End access_wp_cache

////////////////////////////////////
// Function: access_sp_cache()
////////////////////////////////////
int access_sp_cache(struct cache * c, char type, unsigned int addr) {
    // Determine set number and tag
    unsigned int set_num, block_num, tag;
    set_num = (addr & SET_NUM_MASK) >> 4;

```

```

block_num = (addr & BLOCK_NUM_MASK) >> 2;
tag = (addr & TAG_MASK) >> TAG_SHIFT;
//printf("%c Addr: 0x%08X, Set: 0x%08X, Block: 0x%01X, Tag: 0x%08X ----- ",
//      type, addr, set_num, block_num, tag);

// Search for a hit
int result = 0; // Default to miss
int line_num;
int current_line;
int lru_line;

// Process an I-Cache access
if (type == INST) {
    if (c->set[set_num].type == INST) {
        // Determine hit or miss
        for (line_num = 0; line_num < NUM_WAYS; line_num += 1) {
            if ((tag == c->set[set_num].line[line_num].tag) &&
                (c->set[set_num].line[line_num].valid == VALID)) {
                // Process a hit
                result = 1;
                c->set[set_num].line[line_num].last_used = access_num;
            }
        }

        // Process a miss
        if (result == 0) {
            // Find the true least-recently used
            lru_line = 0;
            for (current_line = 1; current_line < NUM_WAYS; current_line += 1) {
                if (c->set[set_num].line[current_line].last_used <
                    c->set[set_num].line[lru_line].last_used) {
                    lru_line = current_line;
                }
            }

            // Replace LRU with new tag (i.e. fetch from L2)
            c->set[set_num].line[lru_line].tag = tag;
            c->set[set_num].line[lru_line].valid = VALID;
            c->set[set_num].line[lru_line].last_used = access_num;
        }
    }

    // The set is already occupied by Data
    else if (c->set[set_num].type == DATA) {
        // Determine if set should be evicted
        if ((c->cur_i_sets < c->max_i_sets) && (c->cur_d_sets > c->min_d_sets)) {
            // Re-assign set to I, Replace line 0 with an Instruction line
            c->set[set_num].type = INST;
            c->set[set_num].line[0].tag = tag;
            c->set[set_num].line[0].valid = VALID;
            c->set[set_num].line[0].last_used = access_num;

            // Invalidate other lines
            for (line_num = 1; line_num < NUM_WAYS; line_num += 1) {
                c->set[set_num].line[line_num].valid = INVALID;
                c->set[set_num].line[line_num].last_used = 0;
            }

            // Increment number of I sets
            c->cur_i_sets += 1;

            // Decrement number of D sets
            c->cur_d_sets -= 1;
        }
    }

    // The set is not claimed yet
    else {
        if (c->cur_i_sets < c->max_i_sets) {
            // Assign set to I, Replace line 0 with an Instruction line
            c->set[set_num].type = INST;
            c->set[set_num].line[0].tag = tag;
            c->set[set_num].line[0].valid = VALID;
            c->set[set_num].line[0].last_used = access_num;
        }
    }
}

```

```

        // Increment number of I sets
        c->cur_i_sets += 1;
    }
} // if (type == INST)

// Process a D-Cache access
else if (type == DATA) {
    if (c->set[set_num].type == DATA) {
        // Determine hit or miss
        for (line_num = 0; line_num < NUM_WAYS; line_num += 1) {
            if ((tag == c->set[set_num].line[line_num].tag) &&
                (c->set[set_num].line[line_num].valid == VALID)) {
                // Process a hit
                result = 1;
                c->set[set_num].line[line_num].last_used = access_num;
            }
        }

        // Process a miss
        if (result == 0) {
            // Find the true least-recently used
            lru_line = 0;
            for (current_line = 1; current_line < NUM_WAYS; current_line += 1) {
                if (c->set[set_num].line[current_line].last_used <
                    c->set[set_num].line[lru_line].last_used) {
                    lru_line = current_line;
                }
            }

            // Replace LRU with new tag (i.e. fetch from L2)
            c->set[set_num].line[lru_line].tag = tag;
            c->set[set_num].line[lru_line].valid = VALID;
            c->set[set_num].line[lru_line].last_used = access_num;
        }
    }

    // The set is occupied by Instruction
    else if (c->set[set_num].type == INST) {
        // Determine if set should be evicted
        if ((c->cur_d_sets < (NUM_SETS - c->max_i_sets)) && (c->cur_i_sets > c->min_i_sets)) {
            // Re-assign set to D, Replace line 0 with a data line
            c->set[set_num].type = DATA;
            c->set[set_num].line[0].tag = tag;
            c->set[set_num].line[0].valid = VALID;
            c->set[set_num].line[0].last_used = access_num;

            // Invalidate other lines
            for (line_num = 1; line_num < NUM_WAYS; line_num += 1) {
                c->set[set_num].line[line_num].valid = INVALID;
                c->set[set_num].line[line_num].last_used = 0;
            }

            // Increment number of D sets
            c->cur_d_sets += 1;

            // Decrement number of I sets
            c->cur_i_sets -= 1;
        }
    }

    // The set is not claimed yet
    else {
        if (c->cur_d_sets < (NUM_SETS - c->max_i_sets)) {
            // Assign set to D, Replace line 0 with a Data line
            c->set[set_num].type = DATA;
            c->set[set_num].line[0].tag = tag;
            c->set[set_num].line[0].valid = VALID;
            c->set[set_num].line[0].last_used = access_num;

            // Increment number of D sets
            c->cur_d_sets += 1;
        }
    }
} // if (type == DATA)

```

```

    return result;
} // End access_sp_cache()

////////////////////////////////////
// Function: repartition()
////////////////////////////////////
int repartition(struct cache * c) {
    // Determine the miss rates
    float i_miss_rate, d_miss_rate;
    i_miss_rate = ((float)i_misses)/((float)i_access_num);
    d_miss_rate = ((float)d_misses)/((float)d_access_num);

    // Check for adjustment in favor of I-Cache
    if (i_miss_rate - d_miss_rate > MISS_RATE_THRESHOLD) {
        if (sim_type == WAY_PARTITION) {
            if (c->way_partition < c->max_way_partition) {
                // Invalidate cache lines
                int current_set;
                int old_partition = c->way_partition;
                for (current_set = 0; current_set < NUM_SETS; current_set += 1) {
                    c->set[current_set].line[old_partition].valid = INVALID;
                    c->set[current_set].line[old_partition].last_used = 0;
                }

                c->way_partition += 1;
            }
            else {
                //printf("Repartitioning in favor of I-Cache by +0\n");
            }
        }
        else if (sim_type == SET_PARTITION) {
            if ((NUM_SETS - c->max_i_sets - REPARTITION_STEP) >= c->min_d_sets) {
                c->max_i_sets += REPARTITION_STEP;
            }
            else {
                //printf("Repartitioning in favor of I-Cache by +0 sets\n");
            }
        }
    }

    // Check for adjustment in favor of D-Cache
    else if (d_miss_rate - i_miss_rate > MISS_RATE_THRESHOLD) {
        if (sim_type == WAY_PARTITION) {
            if (c->way_partition > c->min_way_partition) {
                c->way_partition -= 1;

                // Invalidate cache lines
                int current_set;
                int new_partition = c->way_partition;
                for (current_set = 0; current_set < NUM_SETS; current_set += 1) {
                    c->set[current_set].line[new_partition].valid = INVALID;
                    c->set[current_set].line[new_partition].last_used = 0;
                }
            }
            else {
                //printf("Repartitioning in favor of D-Cache by -0\n");
            }
        }
        else if (sim_type == SET_PARTITION) {
            if ((c->max_i_sets - REPARTITION_STEP) >= c->min_i_sets) {
                c->max_i_sets -= REPARTITION_STEP;
            }
            else {
                //printf("Repartitioning in favor of D-Cache by +0 sets\n");
            }
        }
    }

    return 0;
} // End repartition()

////////////////////////////////////
// Function: print_sp_set()
////////////////////////////////////

```



```

    char type;
};
/*
  For an 8-way set associative Cache:
  +-----+-----+-----+
line[0]: |   tag   | valid | last_used |
  +-----+-----+-----+
line[1]: |   tag   | valid | last_used |
  +-----+-----+-----+
line[2]: |   tag   | valid | last_used |
  +-----+-----+-----+
line[3]: |   tag   | valid | last_used |
  +-----+-----+-----+
line[4]: |   tag   | valid | last_used |
  +-----+-----+-----+
line[5]: |   tag   | valid | last_used |
  +-----+-----+-----+
line[6]: |   tag   | valid | last_used |
  +-----+-----+-----+
line[7]: |   tag   | valid | last_used |
  +-----+-----+-----+
*/

////////////////////////////////////
// Structure: Cache
////////////////////////////////////
struct cache {
    struct cache_set set[NUM_SETS]; // Array of cache sets

    // Way-partitioned cache only
    int way_partition;           // First way dedicated to data
    int min_way_partition;      // Minimum allowed way partition
    int max_way_partition;      // Maximum allowed way partition

    // Set-partitioned cache only
    int max_i_sets;             // Partition restricts number of I sets
    int min_i_sets;             // Minimum number of I sets
    int min_d_sets;             // Minimum number of D sets
    int cur_i_sets;             // Current number of I sets
    int cur_d_sets;             // Current number of D sets
};

```

Listing 3. Way-partitioning defines (wp_defines.h)

```

////////////////////////////////////
// Barnett Morrison Level-1 Cache Simulator
//
// File:    cache.h
//
// Authors: Adam Barnett
//          Casey Morrison
//
// Date:    02 Nov 2006
//
// Description: This header file defines structures for constructing
//              various caches.
////////////////////////////////////

////////////////////////////////////
// Include Files
////////////////////////////////////
#include "wp_defines.h"
//#include "sp_defines.h"

////////////////////////////////////
// Structure: Cache Line
////////////////////////////////////
struct cache_line {
    unsigned tag: TAG_SIZE; // Cache line tag
    unsigned valid: 1;      // Valid indicator for cache line
    unsigned int last_used; // Access number during which line was last used
};
/*
  +-----+-----+-----+
  |   tag   | valid | last_used |

```

```

+-----+-----+-----+
*/
////////////////////////////////////
// Structure: Cache Set
////////////////////////////////////
struct cache_set {
    struct cache_line line[NUM_WAYS];

    // Set-partitioned cache only
    char type;
};
/*
For an 8-way set associative Cache:
+-----+-----+-----+
line[0]: |      tag      | valid | last_used |
+-----+-----+-----+
line[1]: |      tag      | valid | last_used |
+-----+-----+-----+
line[2]: |      tag      | valid | last_used |
+-----+-----+-----+
line[3]: |      tag      | valid | last_used |
+-----+-----+-----+
line[4]: |      tag      | valid | last_used |
+-----+-----+-----+
line[5]: |      tag      | valid | last_used |
+-----+-----+-----+
line[6]: |      tag      | valid | last_used |
+-----+-----+-----+
line[7]: |      tag      | valid | last_used |
+-----+-----+-----+
*/

////////////////////////////////////
// Structure: Cache
////////////////////////////////////
struct cache {
    struct cache_set set[NUM_SETS]; // Array of cache sets

    // Way-partitioned cache only
    int way_partition;             // First way dedicated to data
    int min_way_partition;        // Minimum allowed way partition
    int max_way_partition;        // Maximum allowed way partition

    // Set-partitioned cache only
    int max_i_sets;               // Partition restricts number of I sets
    int min_i_sets;               // Minimum number of I sets
    int min_d_sets;               // Minimum number of D sets
    int cur_i_sets;               // Current number of I sets
    int cur_d_sets;               // Current number of D sets
};

```

Listing 4. Set-partitioning defines (sp_defines.h)

```

////////////////////////////////////
// Barnett Morrison Level-1 Cache Simulator
//
// File:    sp_defines.h
//
// Authors: Adam Barnett
//          Casey Morrison
//
// Date:    06 Nov 2006
//
// Description: This header file defines certain parameters for the
//              lowly set associative cache with set partitioning
//              (SP-Cache).
////////////////////////////////////

// Global Constants
// #define CACHE_SIZE      4 // Size of cache (in KiloBytes)
#define CACHE_SIZE      16 // Size of cache (in KiloBytes)
// #define CACHE_SIZE      32 // Size of cache (in KiloBytes)
#define NUM_WAYS        2 // Number of ways in each cache set
#define NUM_BLOCKS      4 // Number of blocks in a cache line

```

```

#define BLOCK_SIZE      32 // Block size in bits (recommend 32)
#define NUM_SETS        (CACHE_SIZE*1024*8)/(NUM_WAYS*NUM_BLOCKS*BLOCK_SIZE)

//#define TAG_SIZE      21 // Number of Tag bits (must be an integer)
//#define SET_NUM_MASK  0x7F0
//#define BLOCK_NUM_MASK 0xC
//#define TAG_MASK      0xFFFFF800
//#define TAG_SHIFT     11

#define TAG_SIZE        19 // Number of Tag bits (must be an integer)
#define SET_NUM_MASK    0x1FF0
#define BLOCK_NUM_MASK  0xC
#define TAG_MASK        0xFFFFE000
#define TAG_SHIFT      13

//#define TAG_SIZE      18 // Number of Tag bits (must be an integer)
//#define SET_NUM_MASK  0x3FF0
//#define BLOCK_NUM_MASK 0xC
//#define TAG_MASK      0xFFFFC000
//#define TAG_SHIFT     14

//#define REPARTITION_STEP 16
#define REPARTITION_STEP 8

/*
Cache Parameters
-----

    BLOCK_SIZE * NUM_BLOCKS * NUM_WAYS * NUM_SETS = CACHE_SIZE * (1024 * 8)
      32      *      4      *      8      *      32      =      4      * (1024 * 8)

Address Breakdown (for 128 sets)
-----
    31                11 10                4 3                2 1                0
+-----+-----+-----+-----+
|           Tag           |   Set #   | Block # | Byte # |
+-----+-----+-----+-----+
*/

```

Listing 5. Global defines (defines.h)

```

////////////////////////////////////
// Barnett Morrison Level-1 Cache Simulator
//
// File:    defines.h
//
// Authors: Adam Barnett
//          Casey Morrison
//
// Date:    04 Nov 2006
//
// Description: This header file defines certain parameters for all
//              cache simulations.
////////////////////////////////////

// Global Constants
#define VALID      1
#define INVALID   0
#define INST      'I'
#define DATA     'D'
#define NONE      'N'
#define WAY_PARTITION "wp"
#define SET_PARTITION "sp"

#define REPARTITION_T 100000
#define MISS_RATE_THRESHOLD 0.01
//#define MISS_RATE_THRESHOLD 0.02

```
