Thrashing in Real Address Caches due to Memory Management

Arup Mukherjee, Murthy Devarakonda, and Dinkar Sitaram

IBM Research Division
Thomas J. Watson Research Center
Yorktown Heights, NY 10598

Abstract: Direct-mapped real address caches are used by a number of vendors due to their superior access time, simplicity of design, and low cost. The combination of virtual memory management and a direct-mapped real address cache can, however, cause significant performance degradation, particularly for data-scanning CPU-bound programs. A data-scanning program manipulates large matrices, vectors, or images (which typically occupy several pages) by repeatedly reading or writing the data in its entirety. Given a real-address direct-mapped cache, such a program can suffer repeated and systematic cache misses (*thrashing*), even if the cache is large enough to hold all the data pages simultaneously, for the following reason: Two or more virtual pages holding the program's data may be mapped to real pages whose cache addresses are in conflict. Through measurements and analysis this paper shows that the likelihood of such conflicts and the consequent thrashing is unexpectedly large for realistic cache sizes. Measurements show that a program uniformly accessing 64 Kbytes of data suffers at least one conflict in the cache (size 256 Kbytes) with 61% probability; Such a conflict increases the program's running time by as much as 86% on the measured system.

1 Introduction

High cache hit rate is the key to good performance in today's processors. Large, direct-mapped, real address caches are often used to attain this goal because of their hardware simplicity, and their ability to hold a large portion of a program's working set while avoiding the *synonym* problem. However, this paper presents measurements showing that programs repeatedly accessing a contiguous virtual address space may perform poorly due to *thrashing* in such caches even if the virtual address space being accessed is much smaller than the cache. The term thrashing here refers to the occurrence of repeated and regular cache misses which cause significant degradation of the program's performance. In a direct-mapped real address cache, thrashing occurs when two or more virtual pages of the program's address space map to real pages that in turn map to the same cache addresses, and these virtual pages are repeatedly accessed by the program.

This case study represents experiences with an Encore Multimax Model 510 which is a shared memory multiprocessor with 256 Kbytes of direct-mapped real address cache per processor. The Multimax used in our experiments has 64 Mbytes of real memory, and it runs the Mach operating system, which uses an 8 Kbyte page size and provides a 4 Gbyte address space to each process. The test program scans a contiguous virtual address space several times, each time reading a byte from every double word. The size of the address space is a small multiple of the page size. This program characterizes the access pattern of numerical applications that manipulate large matrices or vectors. When this test program was run several times some runs took about 50% to 150% longer to complete than others. Further investigation of this large disparity in running times showed that the real pages of two or more virtual pages accessed by the program were mapped to same set of cache lines in the slower runs of the program, even though the total virtual address space accessed was much smaller than the cache.

At the first glance such an occurrence of cache conflict seems highly unlikely given a fairly large cache and main memory (as in the measured system). However, besides the empirical measurements, an analysis based on the assumption that real pages are randomly allocated to virtual pages also shows an unexpectedly large probability of such cache conflicts for realistic values of main memory size and page size. Therefore, the mapping of real pages to virtual pages seems to be close to random after the system has been in use for a while after reboot, with consequent cache conflicts for programs with the characteristics described here.

Although we observed this thrashing on a multiprocessor, where a large variance in the running times of the program is noticed when several instances of the program are run simultaneously, the problem is not, to the best of our knowledge, relevant only to multiprocessors. In fact, we observed identical results even when we ran the program several times sequentially, one instance at a time.

Clearly, an n-way set-associative cache, where $n \geq 2$, avoids thrashing unless more than n virtual pages map to the same set of cache lines, and the likelihood of a cache conflict decreases with increasing n. Therefore, the chances of thrashing can be minimized by increasing set associativity in the cache. However, such a hardware solution is often unattainable or prohibitively expensive. In such cases, a software solution is the only alternative. There are two changes that can be made to the virtual memory management to reduce the probability of cache conflict: (1) reducing the page size, and (2) sorting the free page list by page addresses. Our analysis shows that the probability of cache conflict decreases with decreasing page size relative to cache size. Similarly, the probability is also reduced if the free page list is kept sorted by address. Both of these solutions have the disadvantage of penalizing those applications that do not have the characteristics of the test program. An attractive solution is to provide an operating system service that can allocate a virtual address space with real pages mapped to it in such way that they do not conflict in the cache(and the real pages remain mapped to the virtual pages for the life of the program).

In this paper, we present empirical results showing the probability that two or more real pages mapped to a program's virtual pages conflict in cache for realistic main memory and cache sizes, and the performance degradation possible from such a cache conflict for data scanning programs as described above. This paper also mathematically analyzes the probability of such cache conflicts, and compares them with the empirically measured values. Such a measurement based study of the role played by the virtual memory management in causing thrashing in a direct-mapped real address cache has not been done before. A few software techniques for avoiding the cache conflict are also discussed.

The rest of the paper is organized as follows: Section 2 covers the background information on cache design to explain why system designers prefer direct-mapped real address caches. Section 3 shows the test program used in our measurements, and empirical results from running this program on the Multimax. Section 4 presents the mathematical analysis of the cache conflict probability for a given cache and virtual memory parameters, and compares the results with the measurements. Sections 5 and 6 conclude with suggestions for avoiding or minimizing the problem.

2 Background

Cache and virtual memories are well known concepts, however, some of the terminology used in the literature is not always consistent. So, here we briefly review the relevant concepts and and clearly define related terms.

Cache memories are small, high-speed buffers to large, relatively slow main memory. The smallest amount of data that can be transferred from main memory to cache memory or vice versa is a *line* (usually a small multiple of the wordsize), and both main memory and the cache memory can be thought of as consisting of several lines. As there are many more main memory lines than there are cache lines, the cache mapping algorithm determines where a main memory line can reside in the cache. Assuming a cache of C lines and a main memory size of M lines, the two algorithms discussed in this paper are:

- Direct Mapping: In this scheme, line m in main memory is always mapped to line m *modulo* C when it is in the cache. Thus, there is only one possible location for each main memory line if it is cached.
- Set-Associative Mapping of Degree D: The cache lines are grouped into S sets of D lines each (so $S = \frac{C}{D}$). Line m in main memory can reside in any of the D lines in set m *modulo* S when it is in the cache. Consequently, all D lines must be examined to determine whether a line is in the cache. This search necessitates extra hardware, and increases the cache access time. Hence, direct mapping is the technique of choice with many vendors. Note that direct mapping is a special case of set-associative mapping where $D = 1$.

Memory lines can be mapped to cache lines either by their real addresses or by their virtual addresses. A real address cache requires simpler hardware but since programs use virtual addresses for data and instruction accesses, an address translation is required before the cache can be searched, thereby increasing the cache access time (particularly when there is a TLB miss). While a virtual address cache has an advantage in this respect, its design is compilcated by the synonym problem. The synonym problem occurs if real address is mapped into two or more address spaces, possibly at different virtual addresses. When contents of a cache line are changed using the virtual address from one address space, cached contents for all other synonymous lines, if they exist, must be invalidated. Extra hardware is required to perform the checking and invalidation. Many vendors thus prefer real address caches due to their hardware simplicity.

The general discussions of memory management in this paper assume well known notions of address spaces, virtual and real pages, and a mapping between the virtual and real pages. Each process has a virtual address space, consisting of many virtual pages, associated with it. Even though there may be further structure to the address space, such as segments, it is irrelevant to the discussion here. The main memory is organized as several real pages; Most modern operating systems use fairly large pages consisting of several memory lines. The real memory in a system is smaller than even a single virtual address space on the same system. Memory management maintains a list of free (real) pages and transparently maps real pages to virtual pages as necessary. Multi-threaded programs are assumed to have several concurrently executing threads within a single address space.

3 Experimental Results

In this section, we discuss empirically observed cache conflict probabilities and performance loss for a test program because of such conflicts. We begin with a description of the system configuration used in the experiment.

3.1 Configuration of the System Used

The system used in our experiments is a shared memory multiprocessor, namely an Encore Multimax model 510, with eight processors. It should be noted, however, that the system is a multiprocessor is irrelevant to the experiments conducted. Each processor has a two level cache. The first level is too small to be of significance in our studies, and the second level is a 256K byte, direct-mapped real address cache. The line size is 8 bytes for both caches. This second level cache is the target of the experiments described in this paper.

The operating system running on the multiprocessor is Mach 2.5 (a.k.a. Encore Mach 0.5/0.6), which uses a page size of 8 Kbytes for both virtual and real memory pages. The measured system has 64 Mbytes of real memory, and its virtual and real addresses are 32 bits wide. Mach memory management organizes free real pages as an unordered list and maps a real page from this list to a virtual page when the virtual page is first referenced. The real page corresponding to a virtual page is reclaimed when the virtual address space is discarded or when the system runs short of free real pages (quite rare in the measured system because of large real memory).

3.2 The Test Program

The test program used in the experiments is

```
PROGRAM DataScan(n,s,i)

BEGIN   <sequential code>
  Allocate n distinct 'footprints' of size s
  in virtual memory (one for each thread,
  each thread executes the parallel code)

  Reference all pages in all footprints, so that
  they are assigned pages in physical memory;

END     <sequential code>

BEGIN <parallel code, n instances, i.e., n threads>

  REPEAT i times
    Reference one byte in every doubleword of
    the footprint allocated to this thread.

    Wait until all the other threads reach
    this point (i.e. barrier synchronization).
  END <repeat loop>

  print time spent in executing the repeat loop.
END <parallel code>

<wait until parallel code completes>

BEGIN <sequential code>

  /*
   * Generate list to be used in determining how
   * many cache conflicts are present in each
   * footprint
   */

  FOR all n footprints
    print out list of cache lines
    to which the pages in the
    footprint are mapped.
  END <for loop>

END <sequential code>
```

Figure 3.0: The test program description.

shown in Figure 3.0. This program, which was originally written for an unrelated multiprocessor experiment, characterizes applications that manipulate large matrices, vectors, and images. The program first creates n threads, and then allocates a separate virtual memory "footprint" of size s (a small multiple of the page size) for each thread. The program then enters a parallel phase where each thread loops over its own footprint accessing a byte in every double word. Note that the cache line is a double word and therefore each thread accesses every cache line of its footprint. We also used a slightly modified, single-threaded version of this program to determine that results can be reproduced even with unrelated multiple Unix processes running either simultaneously or concurrently.

Since the test program performs little computation besides accessing its footprint, its running time results can be thought of as representing a worst case performance. We chose to use this program for two reasons. First, for applications with a significant computational component the results can be construed to reflect the data access component of the running time only, independent of the computational component. Second, for the new high-performance RISC processors, such as the IBM Risc System/6000, the data accessing component of a program is more significant than the computational part because of the disproportionately fast compute engines in such architectures.

3.3 Measurements

When the multithreaded test program shown in Figure 3.0 was run on the multiprocessor, using seven threads and eight pages of footprint per thread, we observed that the running times of the individual threads varied substantially. Often, the slowest thread ran three times as long as the fastest one. Thread scheduling played no role in these measurements since the number of concurrently executing threads was less than the number of processors in the multiprocessor and the system was otherwise idle. We found similar variations in running times when multiple unrelated Unix processes were used, both when they were run simultaneously on multiple processors and when they are run sequentially on a single processor. An examination of the addresses of the real pages mapped to each thread's virtual pages showed conflicts in the cache. Results from several runs of the multithreaded test program are summarized in Table 3.1. Each row of the table corresponds to a single run of the program, and shows the number of threads that have 0, 2, 3, 4, or 5 pages conflicting in cache. It can be seen that an unexpectedly large number of threads suffer from cache conflicts, and many threads have more than two pages in conflict. Of course, the running time of a thread depends on the number of pages in conflict.

We made further measurements to determine the empirical probability of cache conflicts for a given footprint size. Figure 3.1 shows the measured probability of at least one pair of conflicting pages for a given a footprint size. As the figure shows, the probability increases very rapidly as the footprint increases beyond two pages, and reaches nearly 100% for footprints of 12 or more pages. For an eight-page footprint, the probability is about 63%. The relationship illustrated in Figure 3.1 also holds for non-integral numbers of pages involved in conflicts. Non-integral numbers of pages in conflict can occur if the size of the footprint being accessed is not a multiple of the page size.

Since the running time of a program depends on exactly how many pages are conflicting in cache, we measured the probability of exactly n pages conflicting in cache (where $n = 0, 2, 3, ...$) for the test program using a footprint of 16 pages (128K bytes). The footprint size was chosen because it is

Runs	Number of threads having cache conflicts among:				
	0 pages	2 pages	3 pages	4 pages	5 pages
A	1	5	1	0	0
B	2	4	0	1	0
C	3	3	0	1	0
D	3	2	0	1	1
E	2	2	0	3	0
F	1	5	0	1	0
G	3	4	0	0	0
H	5	1	0	1	0
I	4	2	0	1	0
J	2	2	1	1	1

Table 3.1: Each row corresponds to a single run of the multithreaded test program. Seven threads, each accessing a footprint of 64 Kbytes, are used in each run. Each column shows the number of threads having a given number of pages conflicting in cache.

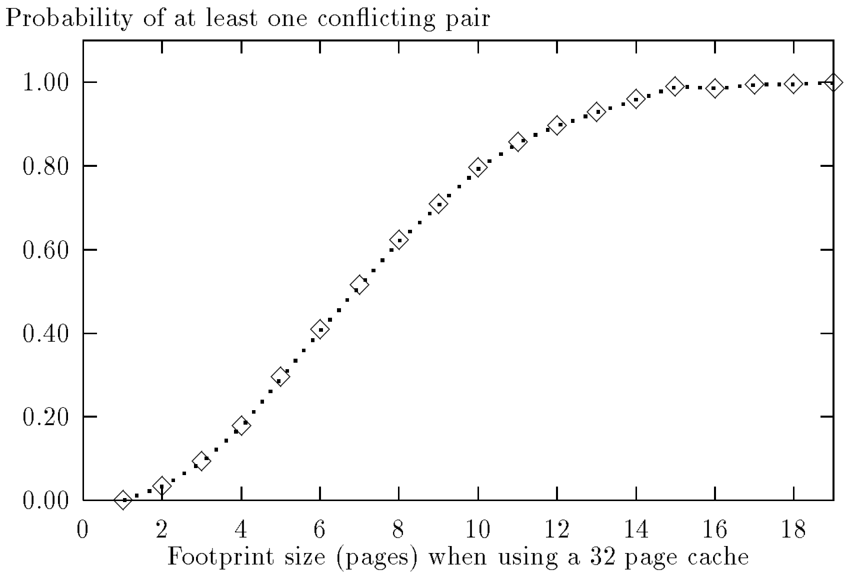

Figure 3.1: The measured probability that a footprint contains at least one pair of conflicting pages.

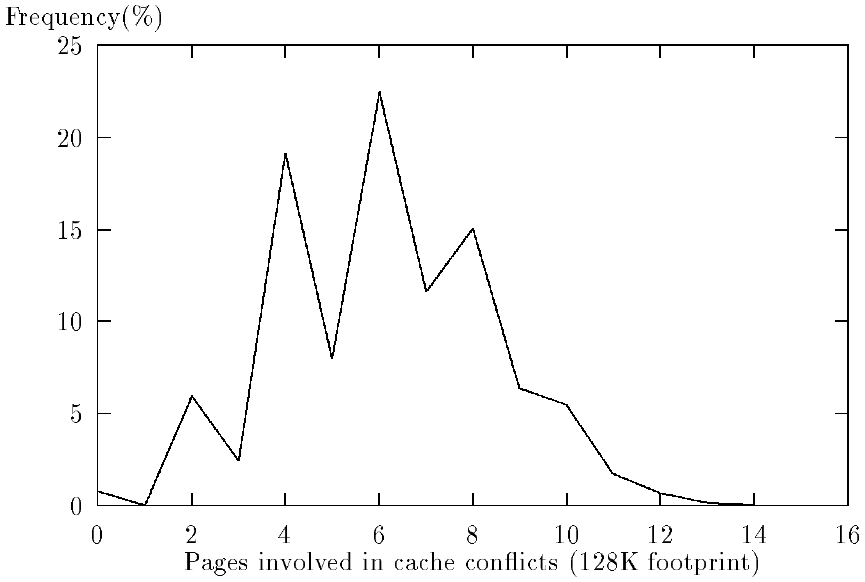

Figure 3.2: The measured probability of having exactly n pages in conflict (where $n = 0, 2, 3, ...$) for a program with 128K bytes (16 pages) footprint.

half the cache. The measurement results are shown in Figure 3.2. The probability distribution function has many peaks and valleys with the highest peak occurring at 6 pages. The test program run with a 128K of footprint suffers cache conflicts involving exactly six pages with a probability of about 24%. Note that the probability of no conflicts is nearly zero. Interestingly, this probability distribution is valid for all programs that use 128K bytes footprint, not just for the test program. Therefore, the expected running time of an arbitrary data-scanning program can be deteremined based on this distribution function, and the cache miss penalty given the ratio of computation to data access for the program.[1]

The measurements presented in Figure 3.1 and Figure 3.2 were obtained by repeatedly allocating chunks of memory, accessing all of the pages allocated (so that physical memory assignments are made to the virtual pages), and checking the address of the physical pages allocated (obtained through kernel instrumentation) for cache conflicts. A theoretical analysis of these results is deferred until the next section.

We made a third set of measurements to determine the performance penalty incurred by the test program as a function of the number of pages in cache conflict and the footprint size. The results are shown in Figure 3.3. Not surprisingly, the running time increases linearly as the number of pages in conflict increases. Note that since the test program accesses all pages in its footprint an equal number of times, cache conflicts affect the running time equally no matter which pages are involved. The performance penalty per page in conflict, however, decreases with increasing footprint size. We observed 43% and 23% increase in running time

[1] If the data access is nonuniform the run time computation becomes difficult as it is then necessary to know exactly which pages are in conflict and their access density.

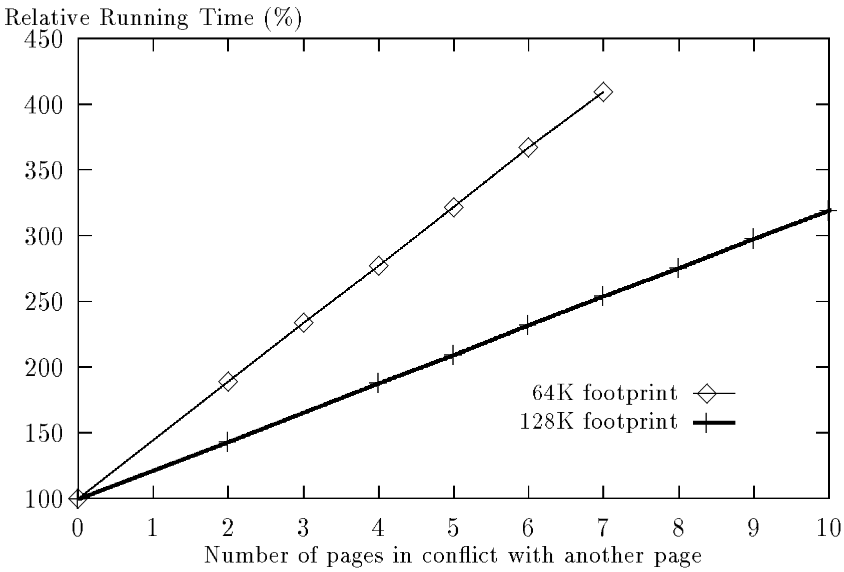

Figure 3.3: The running time penalty for the test program as the number of pages conflicting in cache increases, for different footprint sizes. Without any cache conflicts, the test program takes about 83 and 118 seconds using 64 and 128 Kbyte footprints respectively.

per page in conflict for 64K and 128K footprints respectively.

4 Analysis

In this section, we analyze the likelihood that at least two pages in a program's footprint conflict in the cache, and compare the calculated probability distribution to the empirical results from the measured system. We characterize the performance loss per conflict incurred by our test program, and show the applicability of our analysis to other programs. Subsequently, we evaluate the significance of higher order conflicts, i.e. those involving more than two pages, and analyze the frequency distribution of the amount of data involved in conflicts for one representative footprint size. Finally, we present some simulation results derived from a model based on our analysis; These results evaluate the choice of alternative main and cache memory management strategies in attempting to minimize cache conflicts.

4.1 Probability of At Least One Conflict

As a computer system is used over a period of time its free page list becomes randomly ordered. Based upon this assumption, we have calculated the probability P_n that a request for n pages of memory will result in an allocation containing at least two pages that map to a conflicting set of cache lines in a cache of C pages[2]:

[2] This probability can be expressed in closed form as $P_n = 1 - \frac{(C-1)!}{(C-n)!\ C^{n-1}}$; we prefer the recursive form, as it is easier to understand.

$$
\begin{aligned}
P_1 &= 0 \\
P_n &= P_{n-1} + (1 - P_{n-1})\,\frac{n-1}{C}
\end{aligned}
$$

(i) Memory is allocated in units of pages, and a single page cannot conflict with itself in the cache (assuming that the page size is smaller than the cache size!) because it is a sequence of contiguous memory locations.

(ii) A request for n pages will contain a conflict if there is a conflict in the first $n-1$ pages (which has probability P_{n-1}), or if the first $n-1$ pages are free of conflict (which has probability $1-P_{n-1}$) and the nth page conflicts with one of the other $n-1$ pages (which has probability $\frac{n-1}{C}$).

P_n can also be measured easily in practice as described earlier. Figure 4.1 compares the results from Figure 3.1, obtained by using the Mach vm_allocate()[3] system call to allocate blocks of each size 2000 times, to the theoretical probabilities. Note that our measurements were made after our system had been running for a while, in order to give the free page list a chance to reach a steady state[4]. Observe that for a request that is only 25% of the cache size, a cache conflict results 60% of the time, and requests greater than 50% of the cache size virtually guarantee a conflict. In other architectures, probabilities of a conflict may be even higher, due to factors such as interleaved memory-allocation schemes designed to maximize main memory throughput. Thus, a real program running on a virtual memory system using a direct-mapped real address cache is quite likely to experience an unnecessary performance degradation through cache conflicts.

[3] The standard C library call malloc() yields results that are almost identical. However, if malloc() is used in a loop to collect these results, an allocated block cannot be free()'ed prior to the next allocation – If it is free()'ed the next malloc() returns the same block. When using vm_allocate(), the allocated blocks can be deallocated prior to the next allocation with no problems.

[4] In practice we found that running two parallel kernel compilations simultaneously was sufficient to attain this state following a system reboot.

4.2 Performance Penalty Per Conflict

The performance penalty incurred due to each page involved in cache conflicts is a function of:

1. The proportion of the total number of data accesses that are made to that page. Our test program accesses each page equally, so this factor is simply $\frac{1}{FootprintSize}$ for all pages in the footprint.

2. Changes in the order in which the pages are accessed. Our test program always accesses its footprint in the same order, so this factor can be ignored (Note that such behaviour is typical of most data scanning programs).

3. $CacheMissPenalty$, a system dependent constant which reflects the ratio of the memory access time on a cache miss to that on a hit. We have measured this to be approximately 3.55 on the measured system.[5]

Thus, the running time R_n of our test program when n pages are involved in conflicts, is characterized by:

$$R_n = R_0 \cdot \left(1 + \frac{n \cdot CacheMissPenalty}{FootprintSize}\right)$$

The results presented in Figure 3.3 as well as several other measurements we made (data not shown) reflect this relationship. At first glance the analysis

[5] Our test program accesses only one byte per cache line, which is 8 bytes long, so this is the maximum possible value of $CacheMissPenalty$ on the measured system; its effective value may be slightly lower for programs that access all bits in each cache line.

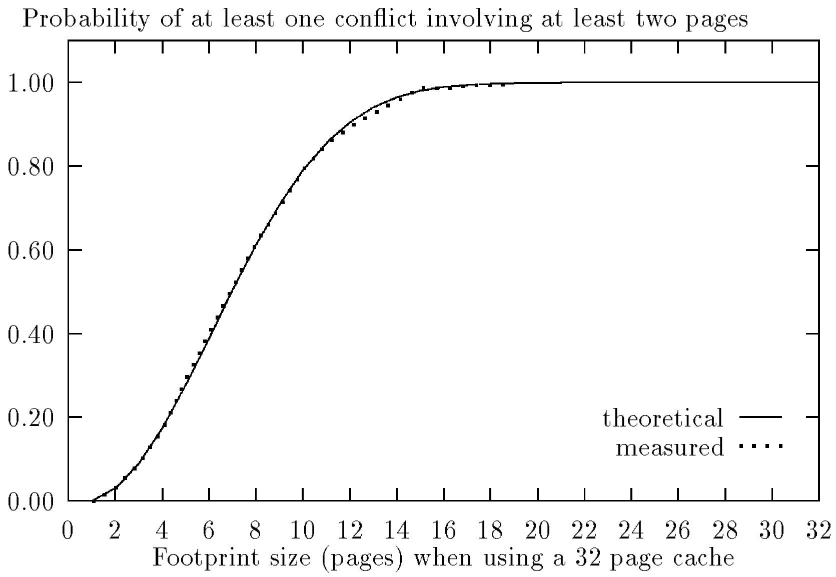

Figure 4.1: Likelihood of at least one conflict

presented here may not seem applicable to other real life, data-scanning programs that may have more computation per data access than the test program used here. However, since R_n can represent the data access component of such programs, the above model has a general applicability.

4.3 Higher Order Conflicts

We have hitherto placed the most emphasis on conflicts between only 2 pages (i.e. 2-way conflicts), although n-way conflicts where $n > 2$ may also occur. In presenting our measurements of performance degradation due to cache conflicts, we only considered the number of pages involved in conflicts (i.e. the degree of these conflicts was not considered significant) – our test program uniformly accesses its entire footprint on every loop, and thus (for example) one 4-way conflict is just as bad as two 2-way conflicts. For programs whose access patterns are not uniform, a conflict of higher degree may be much more serious, especially if the conflict involves the n most-frequently used pages. Thus, an evaluation of the significance of higher order conflicts is important.

Table 4.1 presents a summary of the probability of occurrence of n-way conflicts for n = 2,3 and 4 when using a 32 page cache. From this table, it can readily be seen that higher order conflicts do not become significant until the size of the footprint is almost equal to that of the cache, at which point no cache algorithm can efficiently avoid conflicts. *It is important to realise that these figures are more useful for evaluating the relative importance of higher order conflicts than for judging the effect of increasing the set associativity of the cache.* This is because in a practical scenario, changing the set size usually involves adjusting the total number of sets to keep the total amount of cache memory fixed. The probabilities in the table only apply when the number of sets remains unchanged - i.e. increasing the set size would produce an increase

Footprint Size (pages)	Probability of Conflict in a 32 Page Cache (%) 2-way	3-way	4-way
8	61	5	0.2
16	98.9	36	4
32	99.999	97	45

Table 4.1: The significance of higher order conflicts

in the overal cache size. The practical evaluation of the need to change set size without changing the cache size is discussed later.

4.4 Frequency Distribution of Conflicts

Empirical results for the frequency distribution of the number of conflicts occurring in an allocated memory footprint have been presented earlier (See Figure 3.2). As one might expect, the probability of having a certain total number of pages involved in conflicts depends on the number of possible combinations of conflicts that will produce that total number of pages. For example, since 2-way conflicts are the commonest type of conflict, the peaks in the graph correspond to numbers of pages that can be produced from combinations of 2-way conflicts (2, 4, 6, etc). Other non-zero values occur for totals that can be produced from 3-way conflicts (3, 6 ...), and from combinations of 2-way and 3-way conflicts (5, 7, 9, 10 ...). The highest peaks occur at points that can be produced from 2-way conflicts, 3-way conflicts and combinations (e.g. 6). Beyond 6 pages, the peaks fall off because the contraint imposed by the total footprint size comes into effect. Note that this analysis extends to include n-way conflicts, for all $n>3$, but those cases have not been considered as higher orders conflicts are much less likely to occur, as explained earlier.

The above description provides the basis from which one might compute the overall frequency distribution by a process combining the frequency distributions of all n-way conflicts, where $1 < n \leq FootprintSize$. Rather than taking this approach, we chose to exploit our assumption that the free page list is randomly ordered, and developed a simulator to mimic the effects of allocating pages from a randomly ordered free list.

4.5 Simulations

We simulated the random mapping of virtual pages to real pages by making use of the Unix pseudo-random number generator `random()`. The simulator was tested by using it to calculate the probability distribution in Figure 4.1; the results it produced were virtually indistinguishable from those in the figure. We then applied it to calculating the frequency distribution of conflicts in allocating a 128K footprint from a cache of 256K - these results are presented in Figure 4.2. Once again, the results are very close. We therefore concluded that our simulation results represent a reasonable approximation to the actual behavior of the system, and have used it to evaluate the value of changing the memory page size and the cache set size in attempting to reduce the occurrence of cache conflicts (These changes require extensive hardware and/or software modifications, and thus we could not obtain empirical measurements for them).

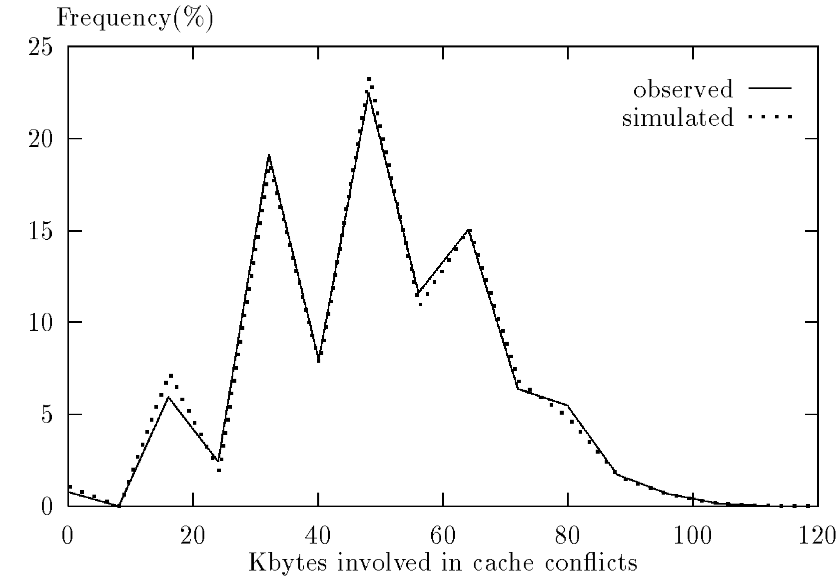

Figure 4.2: Simulated and observed frequency distributions of the amount of data in conflict when allocating a 128K footprint from a cache of 256K. Note that the observed data is the same as that presented in Figure 3.2.

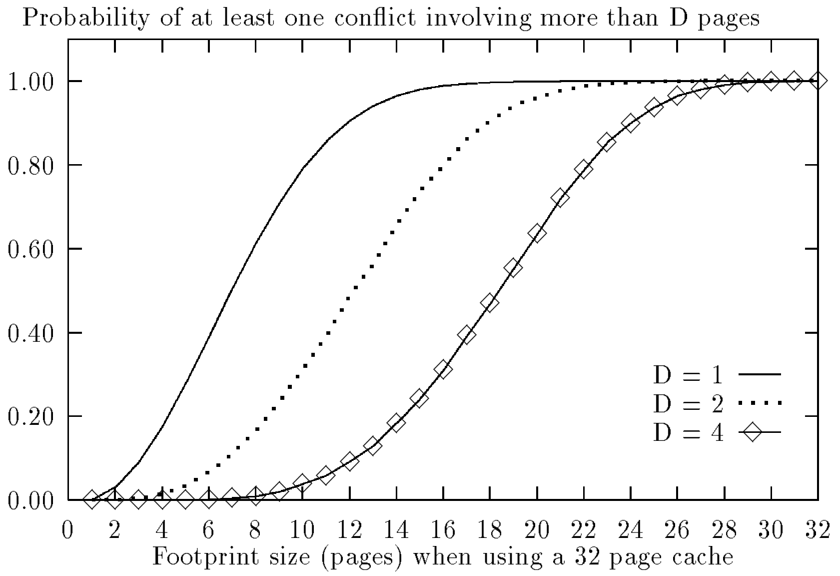

Figure 4.3: The effect of changing the set size while keeping the total amount of cache memory availabel constant. Note that the case of $D = 1$ is the data from Figure 4.1

Set Size: Probabilities have been presented earlier (Table 4.1) that can be used to evaluate the benefits of increasing set size(D) if the total number of sets(S) can be kept unchanged. We have used the simulator to evaluate the benefits of changing the set size while keeping the total *cache* size constant, as is often required in practice. In such a scenario, we have plotted the likelihood that for a given footprint size, at least one group of more than D pages will get mapped to the same set of cache lines, thereby producing thrashing (Figure 4.3). Our results reflect the dimishing returns observed earlier in [Agarwal89] (Notice that going from $D = 2$ to $D = 4$ produces only about as much benfit as going from $D = 1$ to $D = 2$). Thus we believe that, in practice, $D = 2$ would provide the best compromise between reducing cache conflicts while avoiding substantially higher overheads required for greater set-associativity.

Page Size: We have also simulated the effect that changing the page size would have upon the probability plot presented in Figure 4.1. In order to compare the probabilities of occurrence of at least one conflict *with the same severity* as that of a conflict between a pair of 8 Kbyte pages, we have plotted the likelihood that at least one conflict involving at least 16 Kbytes of data will occur. These results are presented in Figure 4.4. Note that in spite of the normalization to maintain a constant "severity" level, smaller page sizes make conflicts much more likely.

This information is supplemented by Figure 4.5, which shows the cumulative frequency distribution[6] for the amount of data involved in cache conflicts for a footprint of 128 Kbytes. As might have been expected from the data in Figure 4.4, the graph clearly shows that for small page sizes, there is a low likelihood that only a small amount of data will be involved in cache conflicts. Large page sizes offer a greater chance that there will be only a small amount of data in conflicts, and the cumulative probability rises at a slower rate for higher amounts of data in conflicts. On the other hand, there is little variation in the mean amounts of data in conflicts for all page sizes. The mean is slightly lower for the larger sizes (reflected in the more gradual rise of the cumulative frequency curve), but in changing the page size from 2 Kbytes to 16Kbytes, the mean drops by only 3.8 Kbytes (i.e. 3% of the size of the allocated footprint). This leads us to conclude that changing the page size has little or no effect on the occurrence of cache conflicts in an allocated footprint; the distribution of the occurrences remains fairly similar, and differences are attributable to granularity constraints, imposed by the page size, upon the possible values for the total amount of data involved in cache conflicts.

5 Solutions to the Problem

The harmful interaction between direct-mapped caches and virtual memory systems can be broken by a number of current methods, in hardware or in software:

(1) Utilization of set-associative caches : Use of a set-associative cache allows a program to access, without performance degradation, a group of conflicting pages as long as the size of the group is smaller than the set size of the cache. If a two-way set associative cache were to be used instead of a direct-mapped cache, two-way page conflicts would cease to degrade performance. Three-way conflicts could still cause problems, but as they occur less frequently, the importance of handling them is re-

[6] We chose to use the cumulative frequency distribution, rather than a simple frequency plot as presented earlier (Figure 4.2) because the position of the peaks in the frequency plot changes when the page size is changed, and this effect makes it difficult to compare gross effects produced by changing the page size.

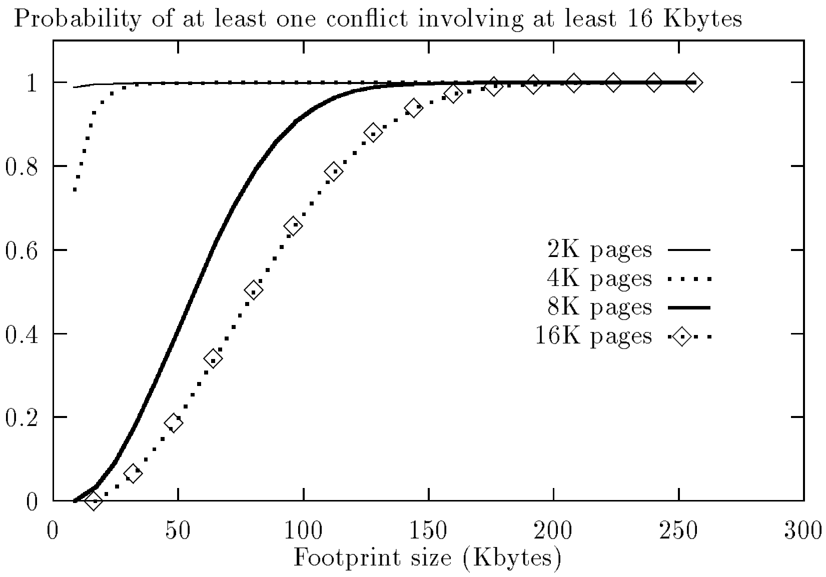

Figure 4.4: The probability of getting at least one conflict in which at least 16K of data is mapped to the same set of cache lines. Note that the data for 8K pages is the same as that plotted in Figure 4.1

duced. Similarly, if a four-way set associative cache were to be used, the problem would be almost completely eliminated, as 5-way conflicts are extremely unlikely to occur. This solution, as one might expect, has the disadvantages of requiring additional hardware and increasing the cache access time.

(2) Use of a cache that is direct-mapped by virtual address: This solution preserves the benefits of a direct-mapped cache (low cost, and good performance for sequential accesses). However, its efficacy may be reduced unless the compiler is optimized for such an environment - typically, certain ranges of virtual addresses are used for similar functions in all programs, and without appropriate modifications, use of a virtual-addressed cache may result in a greatly increased significance of inter-program cache conflicts.

(3) Maintaining an ordered free page list: If the virtual memory system were to maintain an ordered free page list, the problem would not have arisen, as contiguous virtual pages would be allocated to contiguous physical pages wherever possible. This approach is advantageous in that all existing software would be able to take advantage of the improvement, and that no additional hardware is required. Unfortunately, maintaining a sorted free page list is computationally very expensive, and therefore not feasible.

(4) Introduction of an extra system call to allocate memory free of cache-line conflicts: This approach is simple to implement, and does not require any additional hardware. Unfortunately, only programs that were rewritten to use the system call in frequently referenced sections of memory would be able to benefit from it.

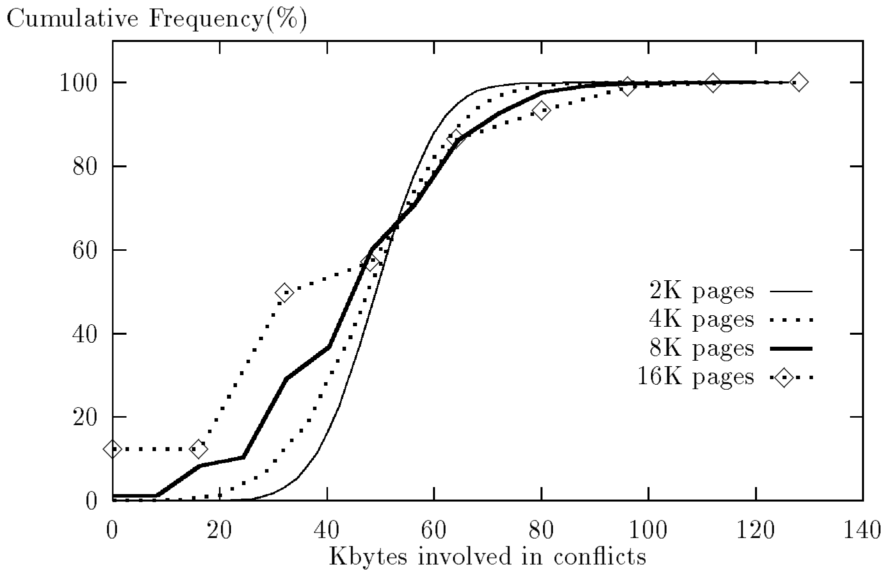

Figure 4.5: The cumulative frequency distribution of the amount of data involved in conflicts, for varying page sizes. Note that the data for 8K pages is the cumulative frequency plot of the simulation results presented in Figure 4.2. The mean amounts of data involved in conflicts are 50.0K (2K pages), 49.5K (4K pages), 48.6K (8K pages), 46.42K (16K pages).

6 Conclusion

We have demonstrated that the Unix virtual memory system may interact with a direct-mapped real address cache in a manner that produces cache thrashing that is very detrimental to performance. This interaction can produce significantly increased running times in certain classes of programs; our test program exhibited large percentage increases in running time which varied with the size of the data footprint being accessed and the number of pages invoved in conflicts with other pages. We have evaluated the significance of this problem, and have presented a means of predicting its effects on a given machine. Finally, We have also outlined several approaches to eliminating the problem, and have pointed out the shortcomings of each one.

References

[Smith82] A. J. Smith, "Cache Memories," *ACM Computing Surveys*, September 1982.

[Agarwal89] A. Agarwal, "Analysis of Cache Performance for Operating Systems and Multiprogramming," *Kluwer Academic Publishers*, Boston, 1989

[Baron88] R. V. Baron `et al`, "MACH Kernel Interface Manual," *Technical Report,* Dept. of Computer Science, Carnegie Mellon University, Feb 1988.