NOVEMBER 1991

WRL
Technical Note TN-22

Cache Replacement with Dynamic Exclusion

Scott McFarling

d|i|g|i|t|a|l **Western Research Laboratory** 250 University Avenue Palo Alto, California 94301 USA

The Western Research Laboratory (WRL) is a computer systems research group that was founded by Digital Equipment Corporation in 1982. Our focus is computer science research relevant to the design and application of high performance scientific computers. We test our ideas by designing, building, and using real systems. The systems we build are research prototypes; they are not intended to become products.

There is a second research laboratory located in Palo Alto, the Systems Research Center (SRC). Other Digital research groups are located in Paris (PRL) and in Cambridge, Massachusetts (CRL).

Our research is directed towards mainstream high-performance computer systems. Our prototypes are intended to foreshadow the future computing environments used by many Digital customers. The long-term goal of WRL is to aid and accelerate the development of high-performance uni- and multi-processors. The research projects within WRL will address various aspects of high-performance computing.

We believe that significant advances in computer systems do not come from any single technological advance. Technologies, both hardware and software, do not all advance at the same pace. System design is the art of composing systems which use each level of technology in an appropriate balance. A major advance in overall system performance will require reexamination of all aspects of the system.

We do work in the design, fabrication and packaging of hardware; language processing and scaling issues in system software design; and the exploration of new applications areas that are opening up with the advent of higher performance systems. Researchers at WRL cooperate closely and move freely among the various levels of system design. This allows us to explore a wide range of tradeoffs to meet system goals.

We publish the results of our work in a variety of journals, conferences, research reports, and technical notes. This document is a technical note. We use this form for rapid distribution of technical material. Usually this represents research in progress. Research reports are normally accounts of completed research and may include material from earlier technical notes.

Research reports and technical notes may be ordered from us. You may mail your order to:

Technical Report Distribution
DEC Western Research Laboratory, WRL-2
250 University Avenue
Palo Alto, California 94301 USA

Reports and notes may also be ordered by electronic mail. Use one of the following addresses:

Digital E-net:	`DECWRL::WRL-TECHREPORTS`
Internet:	`WRL-Techreports@decwrl.dec.com`
UUCP:	`decwrl!wrl-techreports`

To obtain more details on ordering by electronic mail, send a message to one of these addresses with the word ‘‘`help`’’ in the Subject line; you will receive detailed instructions.

Cache Replacement with Dynamic Exclusion

Scott McFarling

November 1991

Western Research Laboratory 250 University Avenue Palo Alto, California 94301 USA

Abstract

Most recent cache designs use direct-mapped caches to provide the fast access time required by modern high speed CPU's. Unfortunately, direct-mapped caches have higher miss rates than set-associative caches, largely because direct-mapped caches are more sensitive to conflicts between items needed frequently in the same phase of program execution.

This paper presents a new technique for reducing direct-mapped cache misses caused by conflicts for a particular cache line. A small finite state machine recognizes the common instruction reference patterns where storing an instruction in the cache actually harms performance. Such instructions are dynamically excluded, that is they are passed directly through the cache without being stored. This reduces misses to the instructions that would have been replaced.

The effectiveness of dynamic exclusion is dependent on the severity of cache conflicts and thus on the particular program and cache size of interest. However, across the SPEC benchmarks, simulation results show an average reduction in miss rate of 35% for a 32KB instruction cache. In addition, applying dynamic exclusion to one level of a cache hierarchy can improve the performance of the next level since instructions do not need to be stored on both levels. Finally, dynamic exclusion also improves combined instruction and data cache miss rates.

1 Introduction

In recent years, the dramatic rise of CPU speed has increasingly stressed memory system performance. Even though DRAM chips are now much denser, their speed has not kept pace with CPU cycle times. This trend increases the importance of cache design to provide the instruction and data bandwidth required by modern CPU's. Furthermore, short CPU cycle times often require both instruction and data caches to be on the same chip as the CPU since crossing chip boundaries leads to unacceptable cache access times. On-chip caches must necessarily be small. These factors leads to caches with relatively high miss rates and large miss penalties.

For a given cache size, set-associative caches have a significantly lower miss rate than direct-mapped caches. In a set-associative cache, every memory item can be stored in one of multiple cache lines. Thus, any two items can be simultaneously stored in a set-associative cache. In a direct-mapped cache, each item can be stored in only one cache line. Thus, a direct-mapped cache can have many more misses if two items are needed repeatedly in the same phase of program execution and they both must be stored in the same cache line. In spite of this issue, direct-mapped caches often have better overall performance because they have lower access times.

This paper presents a new hardware technique that improves the miss rate of direct-mapped caches, particularly instruction caches. When two instructions compete for the same cache line, the technique attempts to keep one instruction in the cache and the other instruction out of the cache. Thus, if execution alternates between the two instructions, the miss rate is halved. We call this selection of instructions to keep out of the cache *dynamic exclusion*.

The key to dynamic exclusion is the recognition of the common instruction execution patterns. These patterns are typically determined by the loops in the program. Section 3 describes the common loop structures, the resulting execution patterns, and the optimal replacement policy for these patterns. Section 4 describes a simple finite state machine that can recognize these patterns and guide a direct-mapped cache toward the optimal replacement policy. Section 5 discusses an alternate finite state machine with better performance for small caches where the reference patterns are slightly different. Section 6 describes the implications of dynamic exclusion on the next level of the cache hierarchy. In particular, it discusses how to handle misses in the next level cache and how dynamic exclusion can reduce these misses. Section 7 describes the interaction between dynamic exclusion and caches with block sizes larger than one instruction. Section 8 discusses using dynamic exclusion on data and combined caches. Finally, Section 9 gives some concluding remarks. First however, we begin with a short discussion of related work in the area of cache design.

2 Related Work

The design of caches to improve memory system performance has been studied extensively. For an overview, see Smith [Smi82]. One area of strong interest has been cache organization. In this paper, we assume caches are direct mapped. Several studies have shown that direct-mapped caches have better overall performance than set-associative caches because of their lower access times [Prz88, PHH89, PHH88, Hil87].

Many cache replacement policies have also been studied. Belady [Bel66] gave an optimal replacement policy in the context of page replacement for virtual memories. Belady's algorithm uses future information to establish a theoretical upper bound on the performance of any other replacement policy. This paper attempts to predict the future and match optimal replacement by recognizing the execution patterns caused by loops. Smith and Goodman [SG85] also used a loop model to compared the effectiveness of different instruction cache replacement policies. Loops models have also been used to improve cache performance with compiler techniques [HC89, PH90, McF89, WL91].

Jouppi [Jou90] also studied a hardware method of reducing the sensitivity of direct-mapped caches to conflicts. A small second level associative cache, called a victim cache, was used to avoid pathological conflicts between two frequently accessed items that use the same cache line. Victim caches work well for data references where the number of conflicting items may be small. For instruction references, there are usually many more conflicting items than a victim cache can hold. This is where dynamic exclusion is most effective.

3 Common Instruction Reference Patterns

To understand how direct-mapped cache performance can be improved, we need to look at the misses that occur for common instruction reference patterns. For now, we assume that each cache miss brings one instruction into the cache. Typically, misses are caused by the interference between the pairs of instructions known as *conflicting instructions* that cannot both be stored in cache at the the same time. Conflicts between three or more instructions also occur, but less frequently. Normally, when execution of two conflicting instructions alternate, there are two misses as each instruction must be brought into the cache. Our goal is to change the replacement policy so that whenever possible there will only be one miss. To explain how this can be done, we will examine the three most common sources of instruction conflicts:

1. conflict between instructions in two different loops
2. conflict between an instruction inside a loop with an instruction outside the loop.
3. conflict between two instructions within the same loop.

To guide our choice of a new replacement policy, we will compare a conventional direct-mapped cache with an optimal direct-mapped cache. By optimal direct-mapped cache, we mean that the cache stores instructions in the same place that a direct-mapped cache would, but the cache has an optimal replacement policy. With this optimal policy, the cache retains the instructions that will be used soonest in the future among those instructions that map to each location in the cache. Furthermore, we assume an instruction can be passed directly to the CPU without ever being stored in the cache. This allows the instruction in the cache to be retained if it will be used sooner in the future than the current instruction will be needed again.

3.1 Conflict Between Loops

An example of conflict between instructions in different loops can be seen in the following example, where instructions a and b map to the same cache location.

```
for i = 1 to 10
    for j = 1 to 10
        instruction a
    for j = 1 to 10
        instruction b
```

Ignoring the instructions associated with the for loops, this example has the execution sequence:

$$(a^{10}b^{10})^{10}$$

In this paper, exponents give the number of times a subsequence is repeated. The subscripts h and m refer to instructions that hit or miss respectively. For the above sequence, both conventional and optimal direct-mapped caches have the behavior:

$$(a_m a_h^9 b_m b_h^9)^{10}$$

with miss rates:

$$m_{DM} = m_{OPTDM} = 10\%$$

Every time an instruction is executed, it is either already in the cache or should be placed in the cache because it is also the next instruction to be executed. Thus, a conventional direct-mapped cache already has optimal performance. Any new replacement strategy for direct-mapped caches should not change this performance.

3.2 Conflict Between Loops Levels

The following example shows a conflict between an instruction inside a loop with another instruction outside the loop:

```
for i = 1 to 10
    for j = 1 to 10
        instruction a
    instruction b
```

Here, the behavior of an conventional direct-mapped cache is:

$$(a_m a_h^9 b_m)^{10}$$

$$m_{DM} = 18\%$$

The behavior of an optimal direct-mapped cache is:

$$a_m a_h^9 b_m (a_h^{10} b_m)^9$$

$$m_{OPTDM} = 10\%$$

In the conventional direct-mapped cache, each access to b causes two misses. Not only does instruction b miss, b knocks a out of the cache and causes a to miss the next time it is executed. In the optimal cache, instruction a is kept in the cache even when b is executed. Instruction a will only miss once. To achieve the optimal cache behavior, a new replacement method should recognize instructions that will only be executed once before some other instruction is repeated, and not add such instructions to the cache.

3.3 Conflict within Loops

The final example below illustrates the behavior when two instructions within a single loop compete for space in the cache.

```
for i = 1 to 10
    instruction a
    instruction b
```

Here, the behavior of a conventional direct-mapped cache is:

$$(a_m b_m)^{10}$$

$$m_{DM} = 100\%$$

The behavior of an optimal direct-mapped cache is:

$$a_m b_m (a_h b_m)^9$$

$$m_{OPTDM} = 55\%$$

In a conventional cache, both instructions continuously knock each other out of the cache. Neither hits. Conversely, the optimal cache selects one instruction to keep in the cache. This selected instruction will hit on later executions. To improve on the conventional direct-mapped cache, the replacement policy should recognize when two instructions are alternating and select one to be kept in the cache.

4 Dynamic Exclusion Replacement Policy

We now present a new method for improving direct-mapped cache performance. The basic idea is to recognize the patterns presented in the previous section and mimic the behavior of an optimal direct-mapped cache. The recognition process treats each cache line independently. In this section, we assume each cache line can hold a single instruction at a time. Section 7 will generalize the result to larger cache line sizes.

In a conventional direct-mapped cache, the most recent reference is always placed in the cache. As we saw in Section 3, optimal direct-mapped caches can get fewer misses by excluding some instructions from the cache. We now consider a new cache replacement policy that reduces the number of misses by dynamically determining which instructions should be excluded from the cache. The determination uses a simple finite-state machine (FSM) in conjunction with two new state bits associated with each cache line. These new bits are called *sticky* and *hit-last* and are denoted s and $h[\,]$ respectively. Figure 1 shows one cache organization that stores these new state bits. The Level 1 (L1) cache contains both bits. The Level 2 cache contains only the $h[\,]$ bit. An alternate organization that does not require the $h[\,]$ bits to be stored in the L2 cache will be discussed in Section 6. For simplicity, Figure 1 and later discussions assume dynamic exclusion is applied to the L1 cache. Application to other cache levels is also possible.

To understand the function of the dynamic exclusion state bits, consider again the patterns shown in Section 3. Both the second and third patterns require the cache to retain an instruction while a conflicting instruction is executed. The sticky bit allows the cache to do this without having to exclude instructions forever. Whenever there is a hit, the sticky bit for that cache line is set. Normally, an instruction is held in the cache while the sticky bit is set. Whenever there is a miss, the sticky bit is reset. Thus, two sequential misses to a cache line will always remove it from the cache.

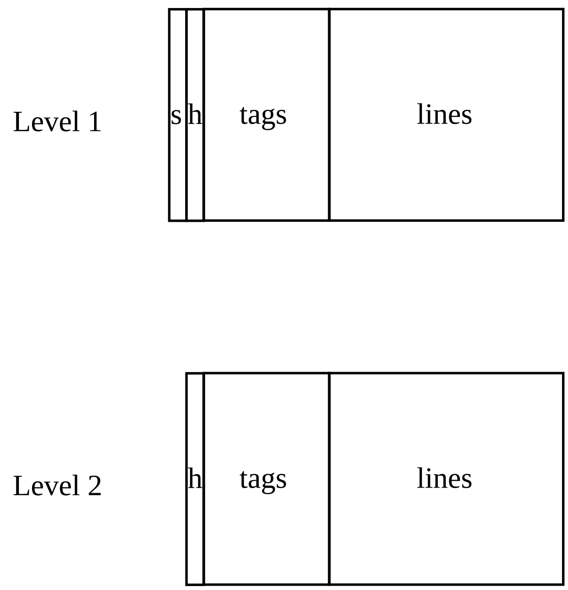

Figure 1: Direct-Mapped Cache with Dynamic Exclusion State Bits

The hit-last bit tells whether an instruction hit the last time it was in the cache. This allows some instructions to be brought immediately into the cache even when the sticky bit is set. Without the hit-last bit, the number of misses caused by switches between execution phases would double.

The hit-last bit must be remembered while an instruction is not in the cache. In addition, the value associated with an instruction is only needed when that instruction misses. Thus, the hit-last bit is logically associated with a lower level in the memory hierarchy. For now, we will assume the hit-last bit is stored in the next level cache. Methods for treating misses in the next level cache are discussed in Section 6.

Even though the value is not necessarily used, the hit-last bit could be set on every access to the first level cache. Unfortunately, the second level cache is normally too slow to be written on all these accesses. The hit-last bit in the first level cache shown in Figure 1 avoids this problem. Whenever the hit-last bit needs to be set, the L1 copy is set. This copy is then transferred to the L2 cache when the instruction in the L1 cache is replaced.

Figure 2 gives the state transition diagram for dynamic exclusion FSM. Only the states and transitions for two instructions, a and b, are shown. The behavior for other instructions is symmetrical. The notation A and B indicate instruction a or b is in the cache respectively. s and $!s$ indicate whether the sticky bit associated with the current cache line is set or reset respectively. The notation $h[x]$ refers to the hit-last bit associated with instruction x. As discussed above, $h[x]$ refers to the L1 bit when the value is changed and the L2 bit when the value is used. The semicolons in arc labels separate the input conditions on the left from state assignments on the right. As in the language C, assignment is represented by an equal sign.

The state diagram can be understood by examining its behavior for the three common

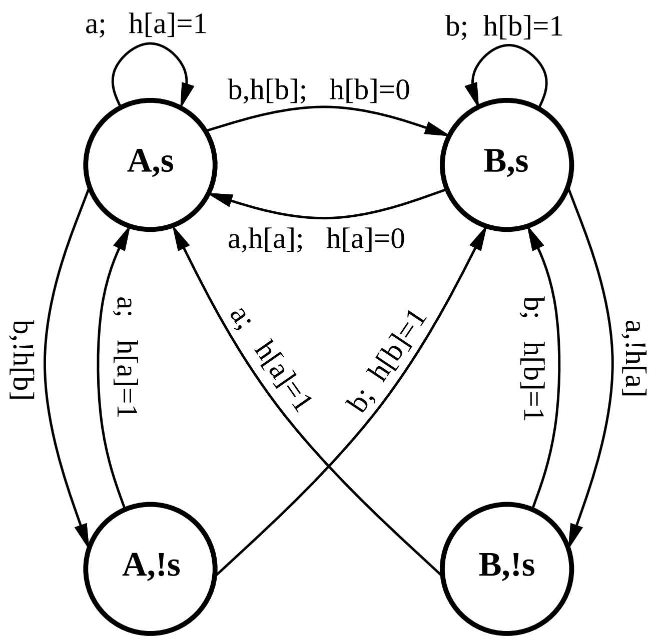

Figure 2: Dynamic Exclusion State Diagram

instruction reference patterns described in Section 3. Consider first the conflict between loops in the pattern $(a^{10}b^{10})^{10}$. The initial states of the sticky bit, the hit-last bits, and the current instruction in the cache are unknown. However, for all possible initial states, instruction a will be loaded into the cache after at most two misses. During sequential executions of a, the cache will remain in State A, s and the hit-last bit $h[a]$ will be set since a will hit several times. When b is executed the initial action depends on the initial state of $h[b]$. However, after at most two misses, b will be loaded into the cache and subsequently $h[b]$ will be set.

At this point, both $h[a]$ and $h[b]$ are set, indicating that both a and b are repeated whenever they are executed. Subsequently, the cache behavior is the same as an optimal direct-mapped cache. Whenever either a or b is executed, the instruction will either already be in the cache or be immediately loaded just as an optimal cache would. Thus, for this pattern, a direct-mapped cache with dynamic exclusion has at most two more misses than an optimal direct-mapped cache depending on the initial state.

In the conflict between loop levels pattern $(a^{10}b)^{10}$, a direct-mapped cache with dynamic exclusion again matches the behavior of an optimal direct-mapped cache, perhaps after some initial training time. The initial executions of instruction a are identical to the earlier pattern. After at most three executions, a will be in the cache and $h[a]$ will be set. Instruction b will be loaded only if $h[b]$ is initially set. However, even in this case, $h[b]$ is reset and the sticky bit will keep b from ever being loaded again. Again, a direct-mapped cache with dynamic exclusion has at most two more misses than an optimal direct-mapped cache. This is significantly better than a normal direct-mapped cache where each execution of b causes two misses.

benchmark	description
doduc	Monte Carlo simulation
eqntott	conversion from equation to truth table
espress	minimization of boolean functions
fpppp	quantum chemistry calculations
gcc	GNU C compiler
li	lisp interpreter
mat300	matrix multiplication
nasa7	NASA Ames FORTRAN Kernels
spice	circuit simulation
tomcatv	vectorized mesh generation

Figure 3: SPEC Benchmarks Used for Evaluation

Finally, in the pattern of conflict within a loop $(ab)^{10}$, a direct-mapped cache with dynamic exclusion again acts like an optimal direct-mapped cache after some initial activity to correctly set the $h[\,]$ and s bits. Depending on the initial conditions, either instruction a or b may be kept in the cache. However, eventually one instruction will be kept in. For example, if a is initially in the cache, the finite state machine will cycle between states A, s and $A, !s$. Again, after some initial misses the direct-mapped cache with dynamic exclusion has only half the misses of a normal direct-mapped cache.

Before going further, we should note that the state diagram in Figure 2 contains a small exception to the definition of $h[\,]$. The bit does not always mean that the relevant instruction hit the last time in the L1 cache. For example, in the transition $A, !s \rightarrow B, s$, the $h[a]$ bit is set even though there is no a hit. This aberration improves the cache performance by allowing more random references to get in the cache sooner. This is especially useful for data and mixed caches.

To evaluate the effectiveness of dynamic exclusion, we will use the SPEC benchmarks shown in Figure 3. These benchmarks include a mix of symbolic and numeric applications. However, to limit cache simulation time, only the first 10 million references from each benchmark were used. Results using the full reference streams are similar.

Figure 4 shows the instruction cache performance for each of the SPEC benchmarks for a normal direct-mapped cache, a direct-mapped cache with dynamic exclusion, and an optimal direct-mapped cache, all at a cache size of 32KB. The improvement varies between benchmarks largely depending on the relative frequency of the patterns discussed in Section 3. All the benchmarks with a high instruction cache miss rate show a significant improvement. Benchmarks nasa7 and tomcatv show a slight increase in misses with dynamic exclusion. This is caused by a small increase in cold-start misses while the dynamic exclusion state bits are initialized. For the full instruction reference streams, this increase is negligible.

Figure 5 shows the average instruction cache miss rate across the SPEC benchmarks for

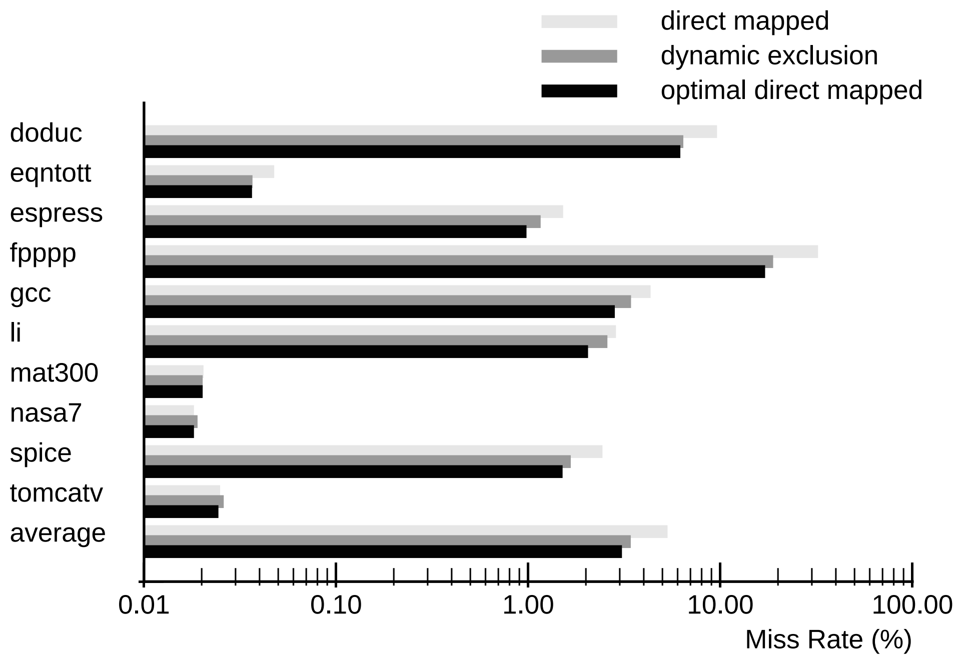

Figure 4: Instruction Cache Performance for Various Benchmarks (S=32KB)

a range of cache sizes with the same three types of caches as in Figure 4. Figure 8 shows the percentage reduction from the normal direct-mapped cache miss rate. For very large caches, the potential improvement decreases since there are no conflicts when the programs fit in the cache. For very small cache sizes, the conflicts are more likely to involve more than two instructions and thus not be recognizable by the FSM in Figure 2. The potential improvement is also smaller as the degree of conflicts increases. This is reflected by the decline in the percentage improvement for an optimal direct-mapped cache for these small cache sizes. The average improvement for dynamic exclusion peaks at 35% at a cache size of 32KB.

5 Additional Sticky State

So far, we have assumed that all conflicts for cache space involve two instructions. Clearly, conflicts among three or more instructions are also possible. For example, if the outer loop in Section 3.2 has two instructions that map to the same cache location, the execution sequence could be $(a^{10}bc)^{10}$. Similarly, if the loop in Section 3.3 has four instructions that map to one cache location the execution sequence would be $(abcd)^{10}$.

In both these new sequences, an optimal direct-mapped cache locks instruction a in the cache and keep the other instructions out. Unfortunately, the finite state machine presented in Section 4 cannot recognize these patterns because they require an instruction to be kept

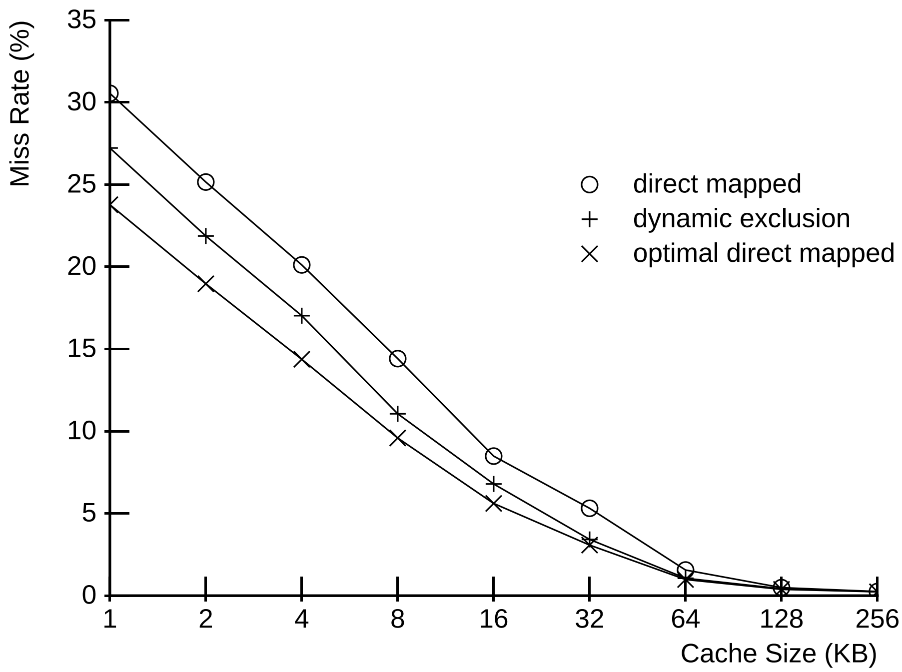

Figure 5: Instruction Cache Dynamic Exclusion Performance for Various Cache Sizes

in the cache despite multiple misses. However, we can easily add this ability by adding sticky bits. The resulting finite state machine is shown symbolically in Figure 6. Here, s is a sticky counter. If the counter has only one bit, then Figure 2 is equivalent to the state diagram in Figure 2. However, with a two bit counter the finite state machine in Figure 6 can recognize the more complex sequences discussed above.

The results using a two bit counter are shown in Figure 7. The two bit counter has better performance for small cache sizes. Here, typical patterns tend to have more instructions because a given loop is more likely to reuse the same cache locations. The two bit counter has worse performance for larger cache sizes. This is because patterns involving more than two instructions are relatively infrequent and a two bit counter takes longer to initialize. In addition, the two bit counter can confuse some patterns involving two instructions. For example, it might confuse pattern $(abcd)^{10}$ with $(ab^3)^{10}$ and keep a in the cache where keeping b would have much better performance.

6 Choices for Lower Level Caches

In Section 4, we assumed that the second level cache was large. With this assumption, the hit-last bits flushed out of the L1 cache can normally be found in the L2 cache. In this section, we consider smaller L2 caches where there is more interference between hit-last bits. In particular, we will consider three ways a cache with dynamic exclusion could

state A,s>0{
  a → state A,s=s_{max}; h[a] = 1;
  h[b],b → state B,s=s_{max}; h[b] = 0;
  !h[b],b → state A,−−s;
}

state A,s==0{
  a → state A,s; h[a] = 1;
  b → state B,s; h[b] = 1;
}

Figure 6: Dynamic Exclusion Finite State Machine

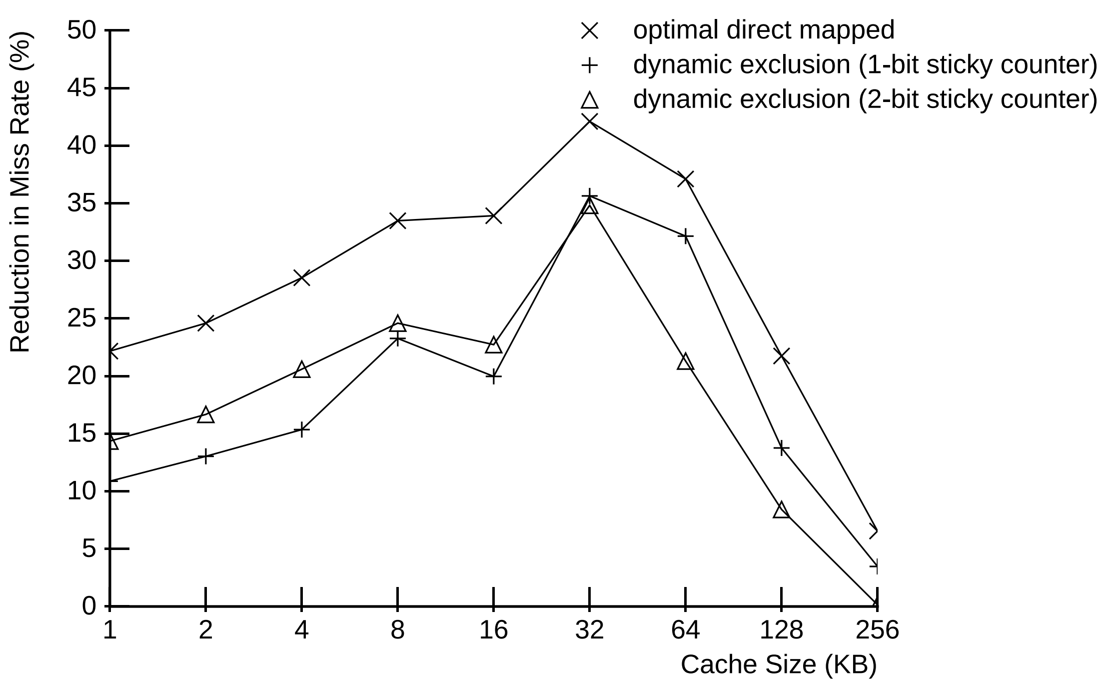

Figure 7: Instruction Cache Performance Improvement for Various Cache Sizes

respond to an L2 miss:

1. use the existing hit-last bit (hashed)

2. assume the hit-last bit is set (assume-hit)

3. assume the hit-last bit is not set (assume-miss)

The first option has a significant structural advantage. The hit-last bits previously kept in the L2 cache can be kept in the L1 cache since there is no need to insure that the current instruction matches the instruction stored in L2. This avoids the need to communicate the hit-last information between the caches and even the need for the L2 cache to know that the L1 cache is using dynamic exclusion. Also, there is no need for the original hit-last bit in L1 since the bits previously stored in L2 can be accessed directly at L1 speeds.

Figure 8 shows the L1 miss rates with dynamic exclusion using each of the three options as the L2 cache size is increased. For most L2 cache sizes, the assume-hit option has slightly fewer L1 misses. Assuming instructions will hit is usually correct. However, if the L2 cache is the same size as the L1 cache, the assume-hit option gives no improvement since the cache degenerates to conventional direct-mapped behavior. With all three schemes, most of the performance is achieved as long as the L2 cache is at least 4 times as large as the L1 cache. This is large enough to insure that most L1 misses are found in the L2 cache. This also implies that the hashing strategy needs only four hit-last bits for each cache line to get good performance.

Figures 9 and 10 show the results of the three options on the L2 miss rates. With the hashed and assume-miss strategies, all instructions stored in the L1 cache do not need to be stored in the L2 cache. This allows the L2 cache to store other instructions and get a lower miss rate. This could be particularly useful if the L2 cache is on the same chip as the CPU and needs to to be kept small. As the figures show, the assume-miss strategy is best at improving the L2 miss rate. This is because it maximizes the difference between the two caches. The hashed strategy also has a significant improvement. The assume-hit strategy does not help the L2 cache because everything in the L1 cache will also be in the L2 cache.

7 Caches with Longer Line Sizes

In the discussion so far, we have assumed each cache line contains one instruction. Larger cache lines present two problems. First, the sequence of instructions that use each tag is much different. If the dynamic exclusion state bits are updated whenever a given tag is accessed, the patterns described in Section 3 will no longer be seen. Using the same finite state machine, cache lines would rarely be excluded because there are almost always several instructions executed sequentially that use the same tag. The second problem is that even if the FSM excluded whole cache lines, this would result in poor performance as each sequential instruction generated a new miss.

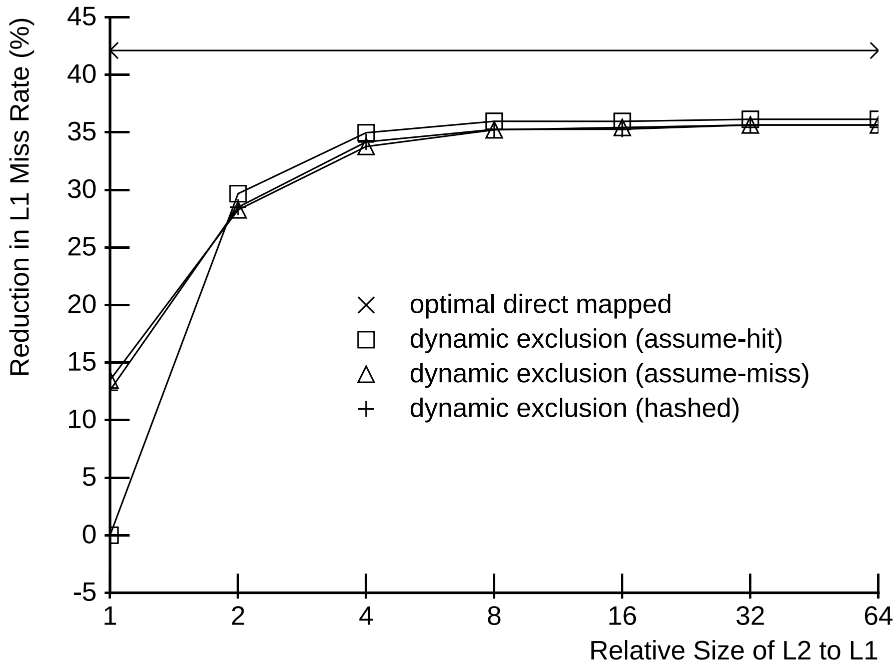

Figure 8: Dynamic Exclusion L1 Performance for Various L2 Cache Sizes (L1=32KB)

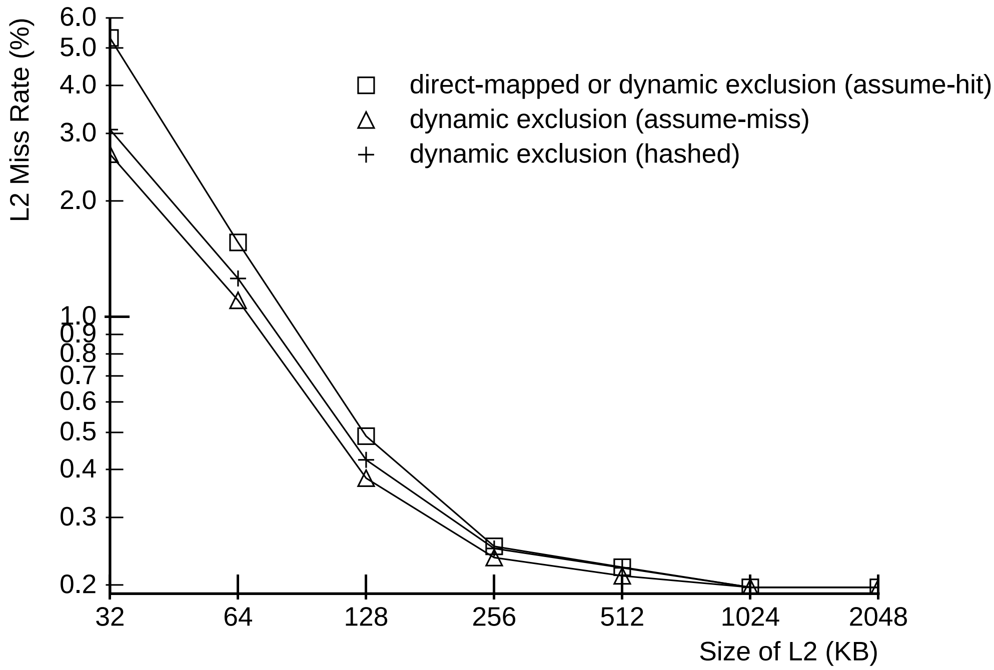

Figure 9: Dynamic Exclusion L2 Performance for Various L2 Cache Sizes (L1=32KB)

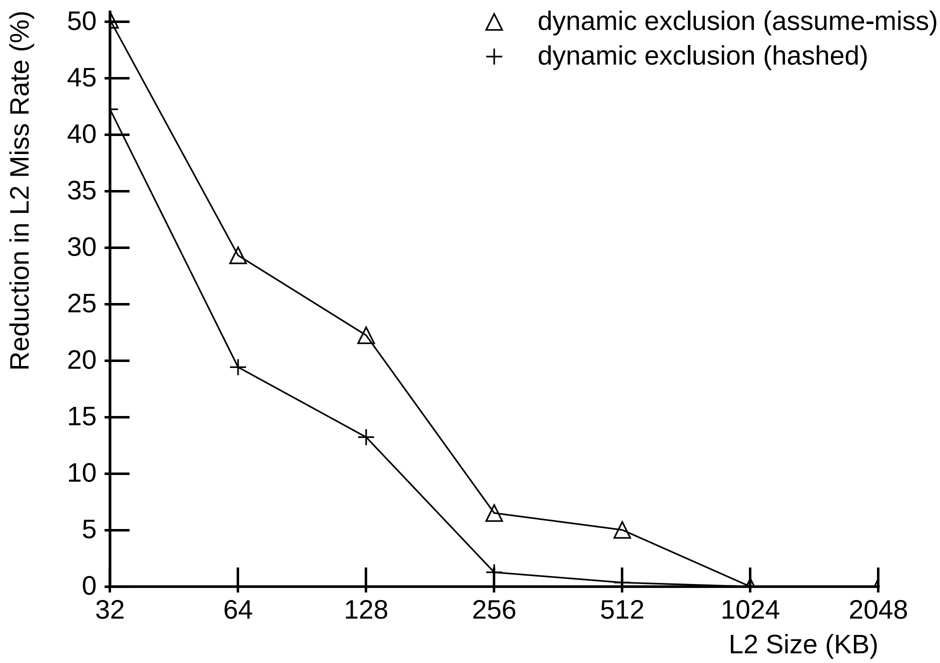

Figure 10: Dynamic Exclusion L1 Performance Improvement for Various L2 Cache Sizes (L1=32KB)

We can solve both these problems once we recognize that the pattern of references to each position within a cache line is essentially the same. Likewise, if we treat the sequential references to each cache line as one reference, the sequence of these references is essentially the same as the sequence of references to each instruction within the line. Moreover, the pattern of these line references is the same as the patterns discussed in Section 3. If each cache line holds one instruction, we were able to reduce the number of misses by excluding instructions that are only used once before a competing instruction is executed. A similar improvement is possible with larger line sizes if we recognize lines that will only be used for sequential instruction executions. These lines should be excluded from the cache, but they do need to be held somewhere so that only one miss is required for the sequential references in the line. This allows dynamic exclusion to be used without losing the benefits of spatial locality.

There are two particularly simple methods of implementing exclusion of longer cache lines:

1. use an instruction register the same size as the L1 cache line.
2. logically add a special line to the cache with its own tag to hold the most recently referenced line.

In the first alternative, missing lines are always stored in the instruction register. However, the line is only stored in the L1 cache if the dynamic exclusion FSM suggests it should

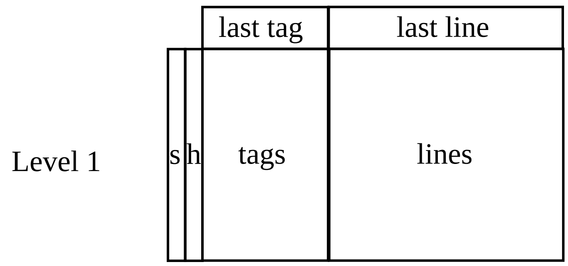

Figure 11: Dynamic Exclusion Structure with Longer Cache Lines

be. Sequential requests for the next instructions are taken from the instruction register without changing the dynamic exclusion state. It is only necessary to access the cache again when there is a taken branch or the program counter rolls over to the next line.

The structure for the second alternative is shown in Figure 11. All missing lines are stored in the special *last-line* area. Subsequent sequential references are taken from the last-line when there is a match to the *last-tag* field. Lines are only stored to the L1 cache as directed by the dynamic exclusion FSM. Finally, the dynamic exclusion state is only changed when the current instruction address does not match last-tag.

Figure 12 shows the performance of the second scheme as the cache line size increases with a 32KB instruction cache. The percentage improvement in the miss rate declines progressively from 35% with four byte lines to 23% at 64 byte lines. This loss in efficiency tracks the internal fragmentation problem of long cache lines. In particular, two instructions that do not conflict with a small line size may conflict with a longer line size. These added conflicts can prevent the FSM from finding a line that can be excluded to improve performance.

8 Data and Mixed Caches

The previous sections have only discussed instruction caches. A natural question is whether the same techniques can be applied to data caches as well. Figure 13 shows the result of using the single bit dynamic exclusion FSM to the data references from the SPEC benchmarks. Again, only the first 10 million references were used to keep simulation time reasonable. For small cache sizes there is a small improvement. However, for larger cache sizes, the dynamic exclusion FSM has slightly worse performance than a direct-mapped cache. The common data reference patterns are different than those for instructions. In addition, a normal direct-mapped cache is closer to optimal for data references than for instruction references. Thus, there is less potential for dynamic exclusion to help.

Figure 14 shows the performance of dynamic exclusion for various sizes of combined data and instruction caches with a line size of 4 bytes. For smaller cache sizes, the improvement is nearly as large as the improvement for instruction caches. For large caches, the improvement is smaller. With these benchmarks, instruction references dominate the miss rate for small caches and data references dominate for large caches.

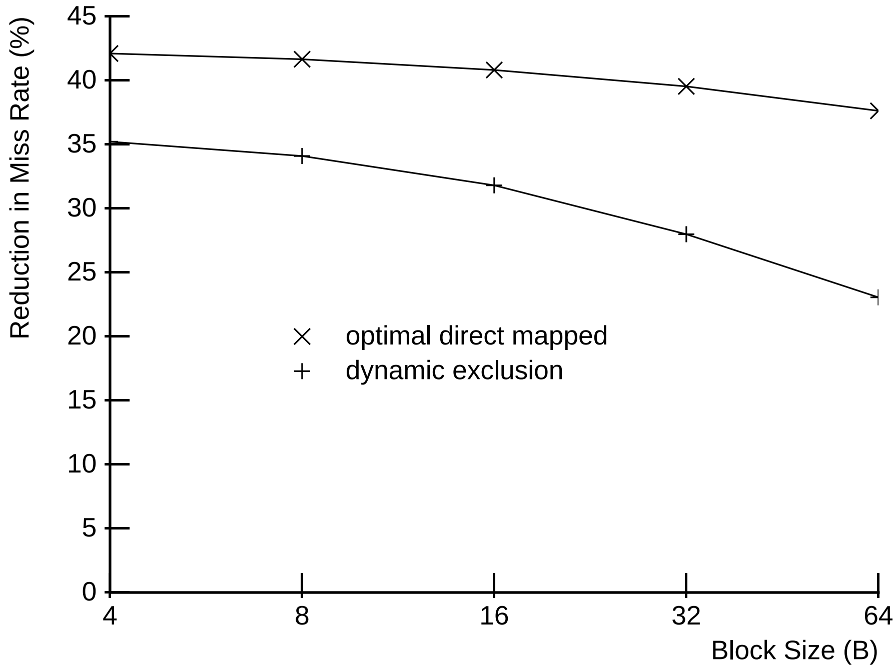

Figure 12: Instruction Cache Dynamic Exclusion Performance for Various Line Sizes (S=32KB)

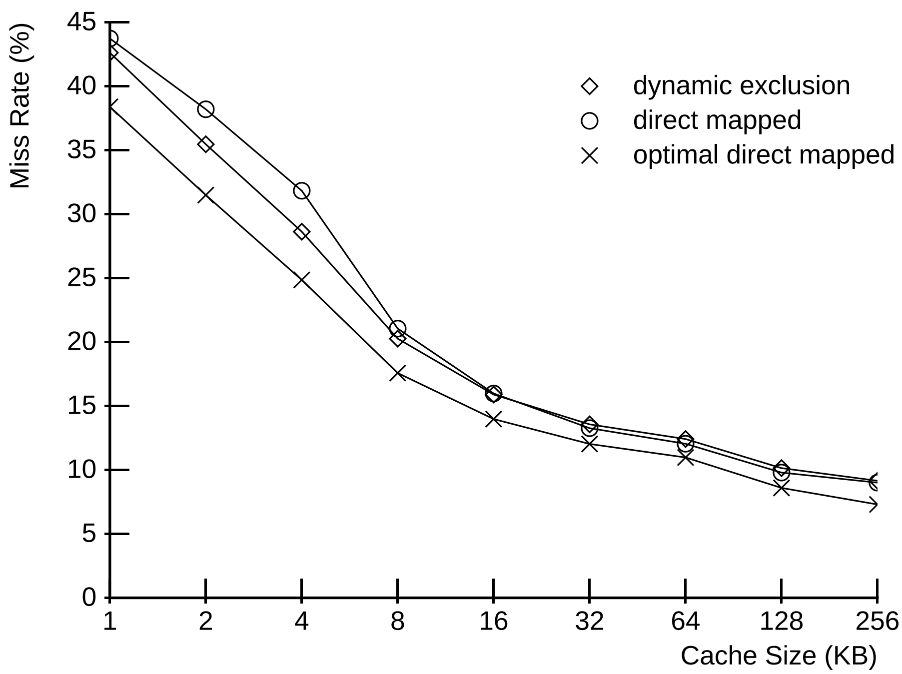

Figure 13: Data Cache Dynamic Exclusion Performance for Various Cache Sizes

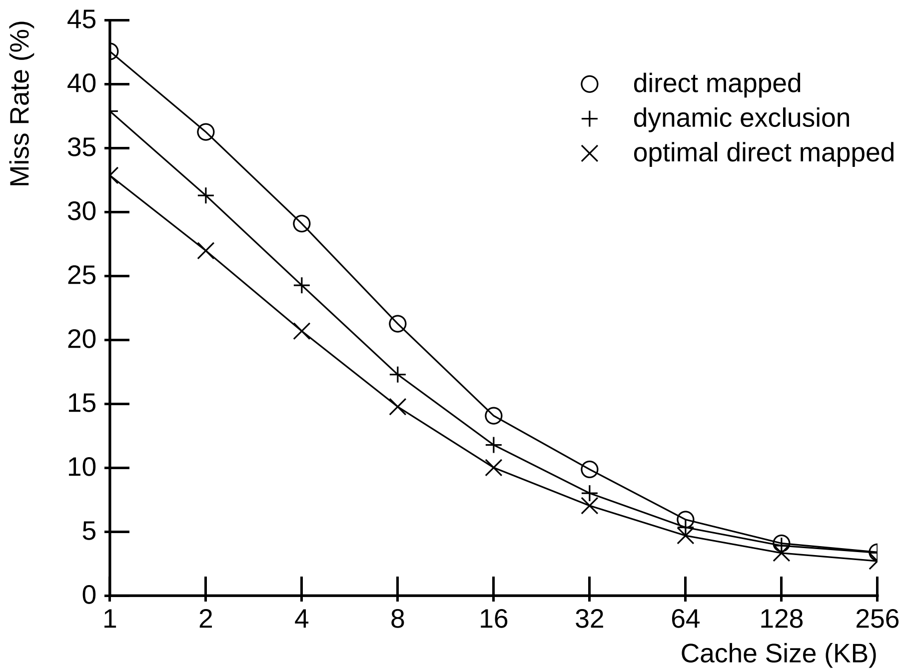

Figure 14: Dynamic Exclusion Performance for Various Combined I and D Cache Sizes

9 Conclusions

This paper has presented a new technique named dynamic exclusion that reduces the miss rate of direct-mapped caches. The technique uses a small finite state machine to recognize the common instruction reference patterns. By keeping instructions that will not hit anyway out of the cache, the remaining instructions have fewer misses. The reduction in miss rate depends on the benchmark and the cache size. However, for a 32KB instruction cache, the average miss rate for the SPEC benchmarks is reduced by 35%. In addition, dynamic exclusion can improve the performance of the second level cache since some instructions only need to be stored in the first level cache. Finally, by reducing the number of instruction misses, dynamic exclusion is also useful for combined instruction and data caches.

References

[Bel66] L.A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems J.*, 5(2):78–101, 1966.

[HC89] W. W. Hwu and P. H. Chang. Achieving high instruction cache performance with an optimizing compiler. In *Proc. 16th Sym. on Computer Architecture*, Israel, May 1989.

[Hil87] M.D. Hill. *Aspects of Cache Memory and Instruction Buffer Performance*. PhD thesis, University of California, Berkeley, November 1987.

[Jou90] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proc. 17th Sym. on Computer Architecture*, Seattle, May 1990.

[McF89] S. McFarling. Program optimization for instruction caches. In *Proceedings of ASPLOS III*, Boston, MA, April 1989.

[PH90] K. Pettis and R. C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 16–27, 1990.

[PHH88] S. Przybylski, M. Horowitz, and J.L. Hennessy. Performance tradeoffs in cache design. In *Proc. 15th Sym. on Computer Architecture*, Honolulu, Hawaii, June 1988.

[PHH89] S. Przybylski, M. Horowitz, and J. Hennessy. Characteristics of performance-optimal multi-level cache heirarchy. In *Proc. 16th Sym. on Computer Architecture*, Israel, May 1989.

[Prz88] S. A. Przybylski. *Performance-Directed Memory Heirarchy Design*. PhD thesis, Stanford University, September 1988.

[SG85] J. E. Smith and J. R. Goodman. Instruction cache replacement policies and organizations. *IEEE Transactions on Computers*, C-34(3):234–241, March 1985.

[Smi82] A. J. Smith. Cache memories. *Computing Surveys*, 14(3):473–530, September 1982.

[WL91] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, 1991.

WRL Research Reports

‘‘Titan System Manual.’’
Michael J. K. Nielsen.
WRL Research Report 86/1, September 1986.

‘‘Global Register Allocation at Link Time.’’
David W. Wall.
WRL Research Report 86/3, October 1986.

‘‘Optimal Finned Heat Sinks.’’
William R. Hamburgen.
WRL Research Report 86/4, October 1986.

‘‘The Mahler Experience: Using an Intermediate Language as the Machine Description.’’
David W. Wall and Michael L. Powell.
WRL Research Report 87/1, August 1987.

‘‘The Packet Filter: An Efficient Mechanism for User-level Network Code.’’
Jeffrey C. Mogul, Richard F. Rashid, Michael J. Accetta.
WRL Research Report 87/2, November 1987.

‘‘Fragmentation Considered Harmful.’’
Christopher A. Kent, Jeffrey C. Mogul.
WRL Research Report 87/3, December 1987.

‘‘Cache Coherence in Distributed Systems.’’
Christopher A. Kent.
WRL Research Report 87/4, December 1987.

‘‘Register Windows vs. Register Allocation.’’
David W. Wall.
WRL Research Report 87/5, December 1987.

‘‘Editing Graphical Objects Using Procedural Representations.’’
Paul J. Asente.
WRL Research Report 87/6, November 1987.

‘‘The USENET Cookbook: an Experiment in Electronic Publication.’’
Brian K. Reid.
WRL Research Report 87/7, December 1987.

‘‘MultiTitan: Four Architecture Papers.’’
Norman P. Jouppi, Jeremy Dion, David Boggs, Michael J. K. Nielsen.
WRL Research Report 87/8, April 1988.

‘‘Fast Printed Circuit Board Routing.’’
Jeremy Dion.
WRL Research Report 88/1, March 1988.

‘‘Compacting Garbage Collection with Ambiguous Roots.’’
Joel F. Bartlett.
WRL Research Report 88/2, February 1988.

‘‘The Experimental Literature of The Internet: An Annotated Bibliography.’’
Jeffrey C. Mogul.
WRL Research Report 88/3, August 1988.

‘‘Measured Capacity of an Ethernet: Myths and Reality.’’
David R. Boggs, Jeffrey C. Mogul, Christopher A. Kent.
WRL Research Report 88/4, September 1988.

‘‘Visa Protocols for Controlling Inter-Organizational Datagram Flow: Extended Description.’’
Deborah Estrin, Jeffrey C. Mogul, Gene Tsudik, Kamaljit Anand.
WRL Research Report 88/5, December 1988.

‘‘SCHEME->C A Portable Scheme-to-C Compiler.’’
Joel F. Bartlett.
WRL Research Report 89/1, January 1989.

‘‘Optimal Group Distribution in Carry-Skip Adders.’’
Silvio Turrini.
WRL Research Report 89/2, February 1989.

‘‘Precise Robotic Paste Dot Dispensing.’’
William R. Hamburgen.
WRL Research Report 89/3, February 1989.

‘‘Simple and Flexible Datagram Access Controls for Unix-based Gateways.’’
Jeffrey C. Mogul.
WRL Research Report 89/4, March 1989.

‘‘Spritely NFS: Implementation and Performance of Cache-Consistency Protocols.’’
V. Srinivasan and Jeffrey C. Mogul.
WRL Research Report 89/5, May 1989.

‘‘Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines.’’
Norman P. Jouppi and David W. Wall.
WRL Research Report 89/7, July 1989.

‘‘A Unified Vector/Scalar Floating-Point Architecture.’’
Norman P. Jouppi, Jonathan Bertoni, and David W. Wall.
WRL Research Report 89/8, July 1989.

‘‘Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU.’’
Norman P. Jouppi.
WRL Research Report 89/9, July 1989.

‘‘Integration and Packaging Plateaus of Processor Performance.’’
Norman P. Jouppi.
WRL Research Report 89/10, July 1989.

‘‘A 20-MIPS Sustained 32-bit CMOS Microprocessor with High Ratio of Sustained to Peak Performance.’’
Norman P. Jouppi and Jeffrey Y. F. Tang.
WRL Research Report 89/11, July 1989.

‘‘The Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance.’’
Norman P. Jouppi.
WRL Research Report 89/13, July 1989.

‘‘Long Address Traces from RISC Machines: Generation and Analysis.’’
Anita Borg, R.E.Kessler, Georgia Lazana, and David W. Wall.
WRL Research Report 89/14, September 1989.

‘‘Link-Time Code Modification.’’
David W. Wall.
WRL Research Report 89/17, September 1989.

‘‘Noise Issues in the ECL Circuit Family.’’
Jeffrey Y.F. Tang and J. Leon Yang.
WRL Research Report 90/1, January 1990.

‘‘Efficient Generation of Test Patterns Using Boolean Satisfiablilty.’’
Tracy Larrabee.
WRL Research Report 90/2, February 1990.

‘‘Two Papers on Test Pattern Generation.’’
Tracy Larrabee.
WRL Research Report 90/3, March 1990.

‘‘Virtual Memory vs. The File System.’’
Michael N. Nelson.
WRL Research Report 90/4, March 1990.

‘‘Efficient Use of Workstations for Passive Monitoring of Local Area Networks.’’
Jeffrey C. Mogul.
WRL Research Report 90/5, July 1990.

‘‘A One-Dimensional Thermal Model for the VAX 9000 Multi Chip Units.’’
John S. Fitch.
WRL Research Report 90/6, July 1990.

‘‘1990 DECWRL/Livermore Magic Release.’’
Robert N. Mayo, Michael H. Arnold, Walter S. Scott, Don Stark, Gordon T. Hamachi.
WRL Research Report 90/7, September 1990.

‘‘Pool Boiling Enhancement Techniques for Water at Low Pressure.’’
Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.
WRL Research Report 90/9, December 1990.

‘‘Writing Fast X Servers for Dumb Color Frame Buffers.’’
Joel McCormack.
WRL Research Report 91/1, February 1991.

‘‘A Simulation Based Study of TLB Performance.’’
J. Bradley Chen, Anita Borg, Norman P. Jouppi.
WRL Research Report 91/2, November 1991.

‘‘Analysis of Power Supply Networks in VLSI Circuits.’’
Don Stark.
WRL Research Report 91/3, April 1991.

‘‘TurboChannel T1 Adapter.’’
David Boggs.
WRL Research Report 91/4, April 1991.

‘‘Procedure Merging with Instruction Caches.’’
Scott McFarling.
WRL Research Report 91/5, March 1991.

‘‘Don’t Fidget with Widgets, Draw!.’’
Joel Bartlett.
WRL Research Report 91/6, May 1991.

‘‘Pool Boiling on Small Heat Dissipating Elements in Water at Subatmospheric Pressure.’’
Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.
WRL Research Report 91/7, June 1991.

‘‘Incremental, Generational Mostly-Copying Garbage Collection in Uncooperative Environments.’’
G. May Yip.
WRL Research Report 91/8, June 1991.

‘‘Interleaved Fin Thermal Connectors for Multichip Modules.’’
William R. Hamburgen.
WRL Research Report 91/9, August 1991.

‘‘Experience with a Software-defined Machine Architecture.’’
David W. Wall.
WRL Research Report 91/10, August 1991.

‘‘Network Locality at the Scale of Processes.’’
Jeffrey C. Mogul.
WRL Research Report 91/11, November 1991.

WRL Technical Notes

‘‘TCP/IP PrintServer: Print Server Protocol.’’
Brian K. Reid and Christopher A. Kent.
WRL Technical Note TN-4, September 1988.

‘‘TCP/IP PrintServer: Server Architecture and Implementation.’’
Christopher A. Kent.
WRL Technical Note TN-7, November 1988.

‘‘Smart Code, Stupid Memory: A Fast X Server for a Dumb Color Frame Buffer.’’
Joel McCormack.
WRL Technical Note TN-9, September 1989.

‘‘Why Aren’t Operating Systems Getting Faster As Fast As Hardware?’’
John Ousterhout.
WRL Technical Note TN-11, October 1989.

‘‘Mostly-Copying Garbage Collection Picks Up Generations and C++.’’
Joel F. Bartlett.
WRL Technical Note TN-12, October 1989.

‘‘Limits of Instruction-Level Parallelism.’’
David W. Wall.
WRL Technical Note TN-15, December 1990.

‘‘The Effect of Context Switches on Cache Performance.’’
Jeffrey C. Mogul and Anita Borg.
WRL Technical Note TN-16, December 1990.

‘‘MTOOL: A Method For Detecting Memory Bottlenecks.’’
Aaron Goldberg and John Hennessy.
WRL Technical Note TN-17, December 1990.

‘‘Predicting Program Behavior Using Real or Estimated Profiles.’’
David W. Wall.
WRL Technical Note TN-18, December 1990.

‘‘Systems for Late Code Modification.’’
David W. Wall.
WRL Technical Note TN-19, June 1991.

‘‘Unreachable Procedures in Object-oriented Programming.’’
Amitabh Srivastava.
WRL Technical Note TN-21, November 1991.

‘‘Cache Replacement with Dynamic Exclusion’’
Scott McFarling.
WRL Technical Note TN-22, November 1991.