THE "MOVING-TARGET" CACHE -- A NEW APPROACH TO PREFETCHING

by

A. Mendelson and G.M. Silberman

Technical Report #342

October 1984

* This research was supported by the Technion VPR Fund - Lawrence Deutsch Research Fund.
ABSTRACT

A new architecture for a cache memory, the Moving-Target Cache, is presented. The architecture is based on the idea of keeping in the cache the immediate vicinity of the currently accessed instructions and data.

The cache is managed by an algorithm which belongs to the prefetching class, but it is characterized by an address translation mechanism simpler than most demand class algorithms. This fact, together with the comparatively low cache size required, make the new cache particularly attractive for integration in a VLSI processor chip.

Three possible implementations of the cache are presented, together with their cost/performance trade-offs. These alternatives are evaluated by simulation, compared with each other, and with LRU. Also, all implementations are shown to be more stable than LRU, when cache flushes resulting from I/O activity are taken into account.
KEYWORDS

Moving Target, Cache Architecture, Cache Management, Prefetching.
1.0 INTRODUCTION

Cache memories are increasingly common in today's computers, as a means to improve processor performance. Moreover, memory prices are such, that the limitations on cache sizes usually stem from the hardware required to manage it, rather than by the cost of the memory itself.

Commercially available cache memories are usually managed by demand algorithms (e.g., LRU in the IBM 370/168 [1]), both because of their simplicity and high performance. Nevertheless, however simple these algorithms may be, they require relatively expensive hardware support (e.g., associative memory), in order to efficiently handle the address translation process [2].

On the other hand, several prefetching algorithms have been proposed (e.g., [3]), which would further improve memory throughput. The problem with these algorithms is the inordinate amount of hardware required for their implementation, making their cost/performance improvement over demand algorithms highly questionable.

The management algorithm associated with the Moving-Target (MT) cache architecture, belongs to the category of prefetching algorithms. Nevertheless, it is characterized by very modest hardware requirements, even when compared with demand algorithms. Also, the performance of the new
cache is comparable to that of demand-managed caches. These properties make the MT cache an attractive alternative for both conventional computer systems, and as on-chip cache for VLSI processors.

In the following section we shall describe the proposed cache architecture and its associated management algorithm, and analyze some of its properties. Section 3 will present three alternative implementations for the MT cache and its management algorithm, stressing their intuitive cost/performance trade-offs.

The search for configuration parameters for the MT cache (e.g., cache size, line size, etc) is covered in Section 4, using results from simulation runs. Section 5 then compares the performance of the proposed implementations and LRU. Special attention is given to the implementations' behavior under increasing cache flushes, resulting from higher I/O volumes.

A discussion of the results obtained in the simulation runs is given in Section 6, stressing the assumptions made on the possible hardware implementations for the MT cache. Finally, Section 7 presents a summary of this work.

Introduction
2.0 THE MOVING-TARGET CACHE--ARCHITECTURE AND MANAGEMENT

This section presents the architecture of the MT cache and its associated management algorithm. Also, the principles guiding the MT approach are briefly dealt with, and some properties of the management algorithm are given.

The MT architecture takes advantage of the time and space locality properties of programs [4]. These properties, which have been long recognized as inherent to most computer programs, specify that:

a) Information (instructions or data) being currently accessed is highly likely to be re-used in the near future (time locality); and,

b) The likelihood of a reference to the area in the vicinity of currently accessed information is very high (space locality).

In order to utilize the above properties, specially the second one (space locality), the MT cache tries to maintain in its memory the immediate vicinity ("neighborhood") of currently active program segments. Each active area in the program (i.e., instructions, data, stack, etc.) is handled using a separate Cache Sight, such as the one depicted in Figure 1. The MT cache consists of a number of such Sights, their assignment to different areas in the program depending on the sophistication

The Moving-Target Cache--Architecture and Management
of the underlying system architecture (e.g., support for segmentation, separate code/data address spaces, etc.).

Each Cache Sight is divided into three areas, as shown in Figure 1. The central area is called the Target, and the two surrounding blocks are referred to as Windows. Each of these areas are partitioned into Line Frames, reflecting the size of the basic information unit which can be moved between cache and main memory (the so-called "cache line" [5]). The Sight contains (when full) a contiguous block of information, representing a (virtual) neighborhood in the program. As a result of this, the address translation process needed to locate a certain (virtual) main memory address in the cache is greatly specified, as we shall see below.

The aim of the MT cache management algorithm is to adjust (when needed) the contents of each Sight, so that the currently accessed information is always found within the Target. Sights are individually managed and there is no overlap of information among them. Let us now consider an example which will illustrate the basic behavior of each Sight.
Example 1

We choose a sight of size 8 lines, each line consisting of 4 words. The Target comprises 4 lines, in the middle of the Sight, and the Windows are 2 lines each. The program segment which is used to illustrate the Sight operation is:

```
for i:=1 to 10 do
begin
...
end;
```

Let's assume the above program segment resides between (virtual) addresses 1000 and 1020 (decimal). Figure 2 shows the different states that the (instruction) Sight goes through, during the execution of the loop. Each state applies to a certain value of the Program Counter (PC), and the numbers represent the boundary addresses for each cache line.

Figure 2(a) shows the Sight state at the head of the loop, i.e., PC=1000. Notice that the line containing word 1000 is located in the center of the Target, and the cache management mechanism completes the line's neighborhood by (pre)fetching the lines which complete the Sight. As long as the program executes in the address range 992-1007 (i.e., within the Target), no adjustment of the Sight is made.

The Moving-Target Cache--Architecture and Management
When the PC becomes 1008, the Sight adjusts to bring this address into the Target's range, as shown in Figure 2(b). Notice that no cache miss occurs, since the line containing address 1008 is already in the cache. It is clear that the adjustment of the boundaries does not involve the movement of information within the Sight, but rather the adjustment of control registers, as we shall see later on. As a result of this adjustment, some lines are ejected from the Lower Window, and others (words 1016-1023) are (pre) fetched into the Upper Window.

The next change in the Sight's contents will occur for PC=1016, as depicted in Figure 2(c). Again, no cache miss is involved in the transition. The contents of the Sight will then remain unchanged until the loop is closed at address 1020. A return to the loop's head (at word 1000) will bring the Sight back to the state shown in Figure 2(a).

It should be noted that given the initial Sight state, as shown in Figure 2(a), no cache misses are incurred during the overall execution of the loop.

As for properties of the MT cache management algorithm, it can easily be shown that for constant Target size, increasing Sight sizes comply with the Inclusion Property (see [6]). This makes the algorithm, under the above constraint, a Stack Algorithm. Therefore, measurements of the MT cache performance via simulation may be done very efficiently, obtaining results for a whole range of Sight sizes in a single pass through the Reference String (the string of addresses produced by the CPU during im-
struction execution). For details on Stack Algorithms and their properties, refer to [6].

Another advantage of keeping a sequence of virtual addresses in each sight, is that the mapping process (processor virtual address to cache physical address) is very simple. This is particularly useful in a Virtual Memory environment (see [7] for example), since filling up the whole cache will require usually one, and at most two different address (main-memory virtual to physical) translations. The reason for this is that a sight is usually smaller than a main-memory page, and thus only loading a segment around a page boundary will result in two separate pages being referenced.

A simple mapping algorithm has a direct bearing on performance, since every reference to memory by the CPU utilizes this mechanism. Other cache systems, such as the IBM 370/168 [1], require expensive hardware support (e.g., associative memory) in order to achieve high performance and avoid slowing down the CPU. In the MT approach this is not necessary, as we shall see in the next Section.

In the following Section we shall examine three possible implementations for the MT cache architecture and management algorithm, each with different cost-performance trade-offs.

The Moving-Target Cache—Architecture and Management
3.0 IMPLEMENTATIONS OF THE MOVING-TARGET CACHE

We show here three implementations for the MT cache, all based on the same principles, but differing in their cost-performance. We start by showing a simple approach, and then work out two improvements which require slightly more hardware, but render better performance.

We will consider a single Sight cache throughout, therefore, the words "cache" and "Sight" will be used interchangeably.

3.1 COMMON FEATURES

In all our implementations we separate the cache management hardware into a **Mapping Unit** and a **Fetch Unit**. This separation is necessary to allow the CPU to continue operations (and references to the cache), while the cache management's Fetch Unit prefetched further lines from main memory. Moreover, the Fetch Unit doubles as a "Cleaning Unit" to implement a write-back scheme with "sneak" line cleaning (i.e., utilizing idle prefetch cycles to "clean" dirty lines).

Implementations of the Moving-Target Cache
The Mapping Unit is responsible for determining the (possible) location in the cache of an address being referenced. We shall call this process a Search. In all our implementations we have taken a Direct Mapping approach, i.e., a line may be present in only one location of the cache. The function we use can be expressed as:

\[
\text{Cache-line} = \left\lfloor \frac{\text{virtual-address}}{\text{line-size}} \right\rfloor \mod \text{sight size},
\]

where sight-size is the number of lines in the cache. It is clear that choosing powers of two for the line-size and sight-size parameters, make the above function very simple to implement.

In theory, our usage of direct mapping would suffer from the disadvantages inherent to this scheme in other cache systems, namely high contention over few lines. In practice, since the MT cache holds a contiguous (logical) segment, moving as execution progresses, contention over certain lines does not exist.

The second component of the cache management hardware, the Fetch Unit, is responsible for moving the Sight whenever a miss to the Target occurs. Moreover, this Unit fetches lines from main memory to the cache, updating information needed in the Search process. Moving the Sight boundaries and completing its contents will be referred to as the Fetch process.

It is obvious that a circular buffer implementation is called for in the MT cache. This facilitates moving the Sight with minimal loss of inform-
mation in most cases. This approach requires the Fetch Unit to continuously update two registers, one with the (virtual) address of the lowest (addressed) line in the Sight, and the other pointing to the line frame which contains (supposedly) this line. These registers are called the **Base Address (BA)** and **Base Frame (BF)**, respectively, and are used to determine Sight boundaries (Target and Windows).

Notice that the circular buffer implementation has no effect on the mapping function, i.e., the cache-line is determined regardless of the contents of BA and BF.

For example, consider Figure 3. This case represents a Sight around address 999, for the same parameters as Figure 2 (line-size=4, sight-size = 8, Target size = 4, Windows = 2,2).

Some sample mappings are:

\[ \lfloor 999/4 \rfloor \mod 8 = 1, \]

\[ \lfloor 985/4 \rfloor \mod 8 = 6, \]

\[ \lfloor 1007/4 \rfloor \mod 8 = 3. \]

This completes the review of those features common to all our MT implementations.
3.2 ALTERNATIVE MT IMPLEMENTATIONS

After reviewing those features common to all three alternatives, we now outline each implementation with its own peculiarities. These features effect two major functions in the MT cache, the Fetch and Search process.

3.2.1 TWO POINTERS (TP) IMPLEMENTATION

In this implementation we use two pointers (as the name indicates) to keep track of the Sight's area which contains currently relevant information. I.e., the lines between these two pointers (Top and Bottom) belong to the present segment residing in the Sight, while all other lines do not. These areas may vary in size from "null" to the "full" Sight size.

The Fetch process must update these pointers (Top and Bottom), whenever the Sight is being (logically) moved, or at the time new lines are being brought into the cache. In the first case, the area between Top and Bottom will decrease (as some information becomes irrelevant), while in the second case the area will increase (relevant lines are brought in).

Searching for information in the Sight now requires a check of the requested cache-line against the two pointers. If the access is to a line...
between the pointers, a hit takes place. Otherwise, the cache registers a miss, and appropriate action must be taken.

In order to determine if an address is in the Sight, we first apply the mapping function we described above, and then compare the result with the two pointers. A result in between (above Bottom and below Top) signifies that the information is present in the Sight. Other processing, such as Sight movement, proceeds as in the general case.

This implementation is very simple and economic, but it should be apparent that the trade-off in performance comes from cases such as the one depicted in Figure 4. In this Figure, the Sight has gone through two transitions, with the consequent updates of pointers. As a result, relevant information still in the Sight (in this case, addresses 988-991) is not recognized as such, references to it will result in misses, and the Fetch unit will eventually refetch it from main memory.

3.2.2 TWO POINTERS WITH AUXILIARY MEMORY (TP-AM) IMPLEMENTATION

In order to overcome one of the major drawbacks in the TP implementation, i.e., refetching lines already in the Sight, we add an Auxiliary Memory.
This memory will keep track of the (virtual) addresses currently residing in each line frame.

With this scheme, the Search process remains the same as in TP (utilizing only the two pointers), but the Fetch mechanism first checks the contents of a line frame before fetching it from main memory. Since the Search mechanism is still based on the two pointers, it is conceivable that an access is flagged as a miss, even though the information requested is in the Sight (such would be the case for PC = 988-991 in Figure 4(b)). We shall refer to this case as a "Dummy Miss," since it is resolved with little overhead, because the Fetch Unit checks the Auxiliary Memory before starting any information transfer.

This implementation is slightly more complex than the previous one, both because of the additional memory, and the need for the Fetch Unit to update and check its contents.

3.2.3 SIGHT-MAPPING (SM) IMPLEMENTATION

This implementation of the MT cache is similar to the Associative Memory approach in conventional caches. In it, we keep an Auxiliary Memory with the (virtual) addresses of each line in the Sight. This is similar to the TP-AM case, but now both Fetch and Search processes utilize this
memory for their function, avoiding the overhead associated with a dummy miss.

It is clear that no associative capabilities are needed here, since we utilize a direct mapping scheme. Nevertheless, this implementation is the more involved of the three shown here, since conflicting accesses to the auxiliary memory (by both Mapping and Fetch units) must be resolved.

In the next Sections we shall explore some performance issues related to the MT cache implementations we have just outlined.
4.0 CHOOSING MOVING-TARGET CACHE PARAMETERS

This Section deals with the selection of cache parameters for each implementation of the MT cache. The cache features which are checked for their effect on performance are the Sight (and cache) size, cache line size, the different ways of partitioning the Sight into Target and Windows, and the movement of Sight boundaries when Window hits occur.

Since we lack analytic tools to evaluate the performance of the cache for different parameter sets, we must resort to simulation runs. In this approach, we utilize the stack property of the MT cache management (and of LRU), in order to improve the running efficiency of the simulations. Because of the unavailability of address trace data from real programs in appropriate quantity, we resorted to synthetic address trace generation as a means to supply the input for the simulation runs.

The approach taken in the generation of synthetic address traces is very similar to that described in [8]. We define a set of parameters (e.g., I/O volume, frequency and size of branches, etc) which characterize different programs, e.g., scientific, data processing, compilers, etc. The values for these parameters are taken from studies which describe the execution profiles for different programs and/or languages (e.g., [9-15]). The traces thus produced may be saved in files or fed directly to a simulator.
Also, the trace generator accumulates "execution" profiles for the traces it generates. This feature allowed us to validate the synthetic address traces by comparing these profiles with the parameters used as input to the generator. For a detailed explanation of the address trace generator, refer to [16].

A simulator was developed to accept the address traces and simulate the cache behavior for all the MT implementations described in Section 3 above, plus LRU and B₀ (the theoretical optimum for demand-driven management algorithms - see [6]). This program is capable of simulating a multiprogramming and/or interactive computer system, with either two-level (cache-main) or three-level (cache-main-paging) storage hierarchy. The hierarchy management algorithms are implemented as external procedures, while the system environment (batch/interactive) is determined by input parameters. Overall behavior of the system is characterized by these parameters and the program load, as defined by the synthetic input traces. This simulator is described in detail in [16].

In the following we describe the programs and environments used in the process of evaluating the various cache implementations. For all our experiments we assumed both instructions and data to be of word size. Each of the MT implementations consists of one Sight each for instructions, stack and data. The results shown are totaled over these three cache components.

Choosing Moving-Target Cache Parameters
It could be argued that different portions of the programs (i.e., instructions, stack, and data) have their own characteristic behavior, therefore requiring different optimal Sight parameters. Nevertheless, we have found experimentally that instructions are the program component which has the strongest effect over a wide range of Sight parameters. For the stack area, almost every Sight chosen showed a hit ratio between 90 and 98%, with data usually having a 20 to 30% higher miss ratio than the average over all components. Also, significant improvements for the data Sight came only when its size became very large (over 4K), which made it impractical.

In light of the above observations, and in order to simplify the presentation of the results, a single set of parameters is used for all three Sights of the MT cache.

4.1 TEST PROGRAMS AND ENVIRONMENTS

The programs used in the simulation runs can be partitioned into four groups:

Choosing Moving-Target Cache Parameters
Compilers: Five different compilers were used as sources for parameters to produce a class of address traces. They are COBOL, BASIC, PL/I, Pascal and FORTRAN, as implemented in the VAX under VMS, and described in [15].

Scientific Programs: Four sets of parameters are used in this group. The first is based on [10], the second is characteristic of FFT programs, and the other two represent highly iterative and recursive processes, respectively (data extracted from [11,12]).

Business Data Processing: Two different sets of parameters generate these applications. They are characterized by numerous (but short) branches ([9]).

Interactive Tasks: The last group of programs we simulate are interactive applications as they execute from terminals. They perform short utility-like actions in sudden bursts, while having "long" periods of inactivity in between. Also, a relatively high I/O volume is associated with them. The source for our parameters was a VAX11/780 running UNIX.

The above programs were integrated into two different executing environments. The first is an interactive type system (such as UNIX), which runs an average of 25 programs concurrently. These were mostly of the inter-
active kind (see above), with a few (2-3) compilations. For this case, the traces were fed to the simulator directly from the address generator.

A multiprogramming batch system was the second environment simulated. It run an average of 7 jobs in parallel, comprising mostly scientific and business data processing applications. Few of the programs represented compilations, as would be the case in a "mostly production" type of environment. For this case, address traces were read from files produced by the generator, to break the size of the task (each program executed some 50,000 instructions which generated between 60,000 to 80,000 memory references).

It could be argued that such a short reference string is not indicative of overall program behavior. In order to check the validity of our results, few of the simulations were run for longer address traces. The results obtained were very similar (in terms of relative performances) to those for the shorter runs, with a general bias towards overall better performance for all cases (MT and LRU).

Next, we show the process by which cache parameters are chosen. The results shown are mostly averages over a large number of simulations, with (synthetic) inputs representing 50,000 instructions of single programs running in a two level memory hierarchy. The monoprogramming environment was used as an indicator for most of the parameters measured, since they depend more strongly on the program behavior than on the system environment. (an exception to this is the Sight size, as we shall see below).

Choosing Moving-Target Cache Parameters
4.2 CACHE PARAMETERS

The four parameters being considered here are line size, Sight (cache) size and partitioning, and the response (in terms of Target movement) to hits within the Sight's Windows. Of all these, it is obvious that only the first two (line and cache size) also apply to LRU. From our simulations, we observed that the line size depended on the algorithm only, and it could not be related to the other parameters. Also, the remaining three parameters were found to be independent of each other.

4.2.1 LINE SIZE

Two conflicting trends influence the cache line size. A larger line size results in a higher hit ratio, but it requires more time to bring into the cache. Also, in schemes with prefetching, it means more traffic between main memory and cache. In order to improve performance, in the case of multi-word lines, extra hardware may be introduced to fetch first the word being referenced, with the rest of the line immediately following. Nevertheless, two consecutive references to the same line but different words usually nullifies the advantage of this scheme.
The simulation results used to determine the line size examined three functions. These are the miss ratio, word traffic and the overall program execution time (which takes into account the time spent for cache management, i.e., address translation, line fetching, etc).

Results for the three proposed MT implementations are shown in Figure 5 through Figure 7. Each graph in these figures represents two implementations of the fetch process. The left side shows the functions for the case where a whole line is fetched before execution continues. The right side assumes that the referenced word is fetched first, allowing execution to proceed while the rest of the line arrives in the cache.

Corresponding results for the LRU cache are shown in Figure 8. For these experiments we assumed a fully-associative LRU cache. No traffic curve is plotted in this case, since this is a demand-driven cache.

The results obtained indicate that the smallest line size has a definite advantage for all MT implementations. Henceforth, we shall use a one-word line for all MT experiments. In the case of LRU, a two-word line with the "referenced word first" algorithm showed the most favorable results.
4.2.2 SIGHT PARTITIONING

In order to examine the effect of partitioning the Sight into different sizes of Target and Windows, two sets of experiments were performed (both used a one-word line size). In the first, we examined the effect on hit ratio by varying the size of both Windows, while keeping the Target size fixed. These results are depicted in Figure 9 for the TP-AM implementation (results for the other two are very similar). The second set of experiments explored the relation between hit ratio and Window size, this time for fixed Sight size. Figure 10 through Figure 12 show the results for the three proposed MT implementations.

From the results it is easy to see that Windows have the most (positive) effect for small Sight sizes. Also, small windows accounted for most of the performance gain. It is also worth mentioning that the extra traffic, associated with the presence of Windows, accounted for only one percent of the total traffic.

4.2.3 SIGHT (CACHE) SIZE

The effect of Sight size on the hit ratio is shown in Figure 13. In it we see the results obtained for two representative (and meaningful size)
cases, those for the COBOL and BASIC compilers. Again, the line size is taken as one word; the Windows are ten words each. As expected, the more sophisticated NT implementations (TP-AM and SM) take advantage of the larger Sight sizes, while the more primitive case (TP) suffers from the loss of information implicit in its mapping algorithm.

It is clear from the implementations of the MT cache, that the traffic will follow a pattern inverse to that of the hit ratio, thus underlining the relative merits of each implementation. Nevertheless, in order to determine the most favorable Sight size for each case, it is necessary to consider other factors besides program behavior, e.g., cache flushes resulting from multiprogramming, system and I/O activity.

4.2.4 TARGET MOVEMENT FOR WINDOW HITS

The last parameter considered here is the algorithm which controls the movement of the Target boundaries, for the case of a hit to a word in one of the Sight's windows. We considered two alternatives for adjusting the Sight boundaries in this case (hits outside the Sight or within the Target are treated equally in both cases). The first alternative readjusts the Sight boundaries so that the word being referenced (in one of the Windows) becomes part of the Target. This implies that some prefetching will be performed to complete the Sight contents with relevant information.
In the second approach, a hit to one of the Windows is treated as a hit to the Target, i.e., no boundaries are moved and no further information is fetched into the cache. The results for both approaches are shown in Table 1. The experiment used Sights of size 128 words each, with a one word Window size (other Sight sizes yield very similar results). It is clear that no definite advantage exists for either alternative. As a result, the second approach is preferred, implying slightly less traffic and simpler implementation.

<table>
<thead>
<tr>
<th>Reaction</th>
<th>Performance</th>
<th>TP</th>
<th>TP-AM</th>
<th>SM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target</td>
<td>Line Traffic</td>
<td>25410</td>
<td>16480</td>
<td>16480</td>
</tr>
<tr>
<td>Moved</td>
<td>Hit. Ratio</td>
<td>84.64</td>
<td>89.09</td>
<td>90.81</td>
</tr>
<tr>
<td>Target</td>
<td>Line Traffic</td>
<td>24997</td>
<td>16473</td>
<td>16473</td>
</tr>
<tr>
<td>Moved</td>
<td>Hit. Ratio</td>
<td>83.68</td>
<td>89.10</td>
<td>90.81</td>
</tr>
</tbody>
</table>

Table 1 — Target reaction after a Window hit.

After showing the processes by which the cache parameters are chosen, we now compare the various cache implementations. The next Section shows the results of these comparisons, for both two and three-level memory hierarchies.
5.0 PERFORMANCE MEASUREMENTS

In this Section we shall compare the performance of the three MT implementations, LRU and B₀. The criteria used in the comparisons are hit ratio, sensitivity to cache flushes (caused by increased I/O activity) and the traffic between cache and main memory, as it relates to the updating of information in main memory. Two level (cache and main memory) hierarchies are considered first and then the three level case (cache, main memory and backing store) is reviewed.

The parameters used in for all the runs in this Section are as follows:

- For the MT cache -
  - line size = 1 word,
  - window size = 10 lines.

- For the LRU and B₀ caches -
  - line size = 2 words (referenced word fetched first).

Since for the MT cache we use three Sights (instruction, stack and data), we must adjust the size of the LRU and B₀ cases accordingly. Our approach...
is to allocate three associative sets to parallel the three Sights, with the distinction that no separation of instructions, stack and data is performed (our experience showed no significant changes when this was attempted). Each set is the same size (in words) as each Sight.

5.1 HIT RATIO

We have chosen this criterion as the main indicator for relative performances between the different cache implementations. The reason for this is that overall execution time is directly proportional to it (for fixed parameters), but obtaining the latter requires accurate knowledge of the implementation details (e.g., fetch times, relative processor speed, etc). In this fashion we tried to keep the results as general as possible, with the knowledge that different caching schemes may be separately optimized at implementation time.

A multiprogramming mix of programs (as described in Section 4 above) was used to compare the three MT cache implementations, LRU and $B_0$. For the latter two, we measured hit ratios with both three and four associative sets. The results are depicted in Figure 14, with the horizontal axis graded by "partition" size (meaning Sight size for MT and set size for LRU and $B_0$). To obtain the overall cache size for each case, multiply
the partition size by three (in the case of four associative sets, add one more partition).

From the results it is clear that the more sophisticated MT implementations have a definite performance edge over LRU and even B$_0$. It is of interest to note that increasing the size of the LRU and B$_0$ caches by adding an additional set (a 33% cache size increase), only results in them slightly overtaking the TP case.

5.2 CACHE FLUSHES

The next aspect of cache performance deals with the emptying of the cache as the result of I/O and system activities. In order to study this so-called cache flush effect, we used an interactive environment for which we could control the amount of I/O activity present in the system.

First, we assumed no I/O activity took place during our simulation, and then measured the effect on the hit ratio by gradually increasing the I/O rate so that 0.5, 1.0 and 2.0% of the time a cache flush takes place. As we can clearly see from Figure 15 through Figure 18, all three MT implementations exhibit a more stable behavior than LRU. Therefore, we would expect CPU performance of a system with an MT cache to be relatively invariant with respect to the I/O rate.

Performance Measurements
It should also be noted that cache flushes have another effect on system performance, besides lowering the hit ratio. This relates to the idle time of the Fetch Unit (refer to Section 3.1), which decreases as the cache flushes become more frequent. This in turn affects the process of updating the main memory, as we shall see below. Actually, when a cache flush takes place 1% of the time, and each partition (Sight) has 256 words, the three MT implementations will fully utilize the Fetch Unit. Therefore, for systems where such an I/O volume is expected, small (64 to 128 words) Sights should be used.

From these results, it appears that utilizing separate Sights for user and system programs (or multiple sets of user Sights) does not result in significant performance improvements (as it would decrease cache flushes), and therefore, their additional cost would be unjustified.

§.3 CACHE TRAFFIC AND MAIN MEMORY UPDATING

With the presence of separate Mapping and Fetch units, the natural way to update main memory is by using the idle Fetch Unit time to perform "write-backs". Therefore, it is important to keep the amount of idle time as high as possible, to insure the prompt update of information in main memory.
Figure 19 shows the percentage of time the Fetch Unit is idle for each of the MT implementations. We have plotted only one curve for both TP-AM and SM, since both utilize similar Fetch mechanisms. This is actually only an approximation, since conflicting accesses to the mapping memory by both Mapping and Fetch would result in a slow down for the SM case.

5.4 RELATIVE CPU SPEED

In order to get some feeling on how the speed of the CPU (relative to the cache) affects performance, we measured the effect of varying the cache access time on the hit ratio.

Figure 20 shows three CPU/cache speed ratios and how they affect the hit ratio. The lowest hit ratio is obtained when we assume that the cache access time matches the addition time in the CPU (this is the relation used in all other measurements). Improving the cache access time, either by technological or architectural means, so that it corresponds to half the addition time yields the middle curve. The top curve in the Figure represents the idealized case for which access to the cache is instantaneous (zero access time).

It is clear that the MT cache will benefit from lower CPU/cache speed ratios (i.e., higher cache speeds), since it relies on prefetching for
its performance. Therefore, the faster this prefetching is done, and the
sights adjusted, the better its overall performance will be.

5.5 THREE LEVEL HIERARCHIES

Having explored cache performance in a two level hierarchy, we now turn
our attention to the three level case. The main difference, as far as
the cache is concerned, is the occurrence of cache flushes as the result
of misses at the main memory (i.e., page faults).

We set up two experiments to check the effect of the additional storage
level. In the first, a batch environment with CPU-bound jobs was used,
while the second reflected a purely interactive system. The batch jobs,
a total of seven, used an average of one CPU second in between I/O re-
quests and run for the full 50,000 instructions. Each interactive pro-
gram, on the other hand, had a 1% I/O activity and run for at most 10,000
instructions. The LRU cache consisted of four sets of 1K lines each (2
words per line), for a total of 8K words, while the MT cache had three
sights of 64 lines each (1 word per line), for a total of 192 words. Main
memory pages were 1K words each.

The results from running the batch mix are shown in Figure 21, separately
for each job in the mix. We can see that LRU has a better performance
than TP, but cannot compare to either TP-AM or SM. The first observation marks a difference from the results in the two level case, which can be explained by the following fact. LRU maps addresses on the cache based on main memory physical addresses, which leads to a better distribution of addresses in the cache and therefore better performance. Also, since the programs have relatively low I/O activity and run for very long, the effect of cache flushes is very light (page faults in main memory run about only 0.2%).

Running the interactive environment produced results identical to those of the two level case. Here, the address mapping has no effect since the programs tend to be very small. Also, higher I/O rates and more page faults contributed to up to 2% cache flushes, therefore hindering the LRU performance more than that of the MT cases. We expect \( B_0 \) to behave in the same way as LRU.

This concludes the review of the measurements performed to evaluate the three MT implementations, and how they compare to LRU and \( B_0 \). In the next Section we shall discuss the results of these measurements and their implications.
6.0 DISCUSSION

From the above results we can make the following observations:

- All three MT implementations obtain better hit ratios than either LRU or \( B_0 \). This is mainly due to the prefetching nature of the MT algorithm and the usage to greater advantage of the locality property of programs. Also, the more expensive implementations yield better hit ratios, therefore justifying the additional hardware (if we look at access time this might not be the case (see below).

- For varying amounts of cache flushes, the MT implementations show much less impact than LRU, leading to a more stable performance in systems with high I/O activity. This suggests that the addition of Sights would not yield significant performance improvement.

- The effective cache access time (i.e., search + access) is much faster than that for other algorithms. For both TP and TP-AM, only accessing the two pointers is needed to perform a search, whereas SM requires an extra cycle to access the mapping memory. This compares favorably with the three to five cycles needed in LRU caches.
• For the HT cache, addresses are mapped based on the CPU generated (virtual) addresses. This greatly simplifies the mapping process in a virtual memory environment.

• Implementation of a write-back scheme for updating main memory is compatible with the HT approach. Furthermore, the Fetch Unit already present in the HT cache can be utilized for this task. Its availability has been demonstrated, with TP showing about 60% more Fetch activity than either TP-AM or SM, and therefore less available.

• Finally, a small Sight size (64 to 128 words) coupled with window sizes of 5 to 10 words, is the most efficient HT configuration. This overall size (all three Sights total under 0.5K words) compares very favorably with any demand-driven cache.
7.0 SUMMARY

We have introduced a novel cache architecture, the Moving-Target (MT) cache. Its main advantage over traditional approaches is the simplicity of its supporting hardware and overall smaller size. Management of the MT cache is accomplished by a simple prefetching algorithm, which yields better performance than its demand-driven counterparts.

Three alternative implementations for the MT cache were presented, with progressively higher hardware requirements and hit ratio improvements to match the increased cost. Nevertheless, if in addition to hit ratio we also look at the effective cache access time (i.e., take into account the address mapping time), it is the medium complexity solution (the one called TP-AM) which shows the most promise.

Optimal cache parameters were obtained using simulation of synthetically produced instruction traces. The same technique was used to compare the performance of the three MT implementations and that for LRU and $B_0$.

With the results showing an overall advantage for the MT approach, and considering its compactness (both in memory and control), this caching scheme seems to be a good candidate for inclusion as on-chip cache in a microprocessor. In this context, the performance of the MT cache in a...
multiprocessing environment should prove as an interesting continuation for this research.

ACKNOWLEDGEMENTS

The idea for the MT cache developed during the course of a discussion with G. Frieder. Some preliminary simulations were written by X. Salinas using as input IBM/360 address traces. The results obtained in these simulations prompted us to make further investigations on the subject.
REFERENCES


References


Figure 1. Structure of a Cache Sight
<table>
<thead>
<tr>
<th>PC = 1000</th>
<th>PC = 1008</th>
<th>PC = 1016</th>
</tr>
</thead>
<tbody>
<tr>
<td>1012 - 1015</td>
<td>1020 - 1023</td>
<td>1028 - 1031</td>
</tr>
<tr>
<td>1008 - 1011</td>
<td>1016 - 1019</td>
<td>1024 - 1027</td>
</tr>
<tr>
<td>1004 - 1007</td>
<td>1012 - 1015</td>
<td>1020 - 1023</td>
</tr>
<tr>
<td>1000 - 1003</td>
<td>1008 - 1011</td>
<td>1016 - 1019</td>
</tr>
<tr>
<td>996 - 999</td>
<td>1004 - 1007</td>
<td>1012 - 1015</td>
</tr>
<tr>
<td>992 - 995</td>
<td>1000 - 1003</td>
<td>1008 - 1011</td>
</tr>
<tr>
<td>988 - 991</td>
<td>996 - 999</td>
<td>1004 - 1007</td>
</tr>
<tr>
<td>984 - 987</td>
<td>992 - 995</td>
<td>1000 - 1003</td>
</tr>
</tbody>
</table>

Figure 2. Sight states for Example 1.
<table>
<thead>
<tr>
<th>Frame Number</th>
<th>Sight Contents</th>
<th>Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>988 - 991</td>
<td>Lower Window</td>
</tr>
<tr>
<td>6</td>
<td>984 - 987</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1012 - 1015</td>
<td>Upper Window</td>
</tr>
<tr>
<td>4</td>
<td>1008 - 1011</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1004 - 1007</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1000 - 1003</td>
<td>Target</td>
</tr>
<tr>
<td>1</td>
<td>996 - 999</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>992 - 995</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3. Circular Buffer Implementation of a Sight (BA=984, BF=6)
Figure 4. Loss of Information in TP Implementation

(a) PC=1000  
(b) PC=1008  
(c) PC=997
Figure 5. Miss Ratio, Word Traffic and Execution Time vs. Line Size
- TP Implementation.
Figure 6. Miss Ratio, Word Traffic and Execution Time vs. Line Size
- TP-AM Implementation
Figure 7. Miss Ratio, Word Traffic and Execution Time vs. Line Size
- SM Implementation
Figure 8. Miss Ratio and Execution Time vs. Line Size - LRU Cache
Figure 9. Hit Ratio vs. Window Size for Fixed Target Sizes - TP-AM Implementation
Figure 10. Hit Ratio vs. Window Size for Fixed Sight Sizes - TP Implementation
Figure 11. Hit Ratio vs. Window Size for Fixed Sight Sizes - TP-AM Implementation

Figures
Figure 12. Hit Ratio vs. Window Size for Fixed Sight Sizes - SM Implementation
Figure 13. Hit Ratio vs. Sight Size for BASIC and COBOL Compilations - TP, TP-AM and SM Implementations.
Figure 14. Hit Ratio vs. Partition Size (cache size is three or four times the partition size).
Figure 15. Effect of Cache Flushes on LRU Performance (cache size is three times the set size).
Figure 16. Effect of Cache Flushes on TP Performance (cache size is three times the Sight size).
Figure 17. Effect of Cache Flushes on TP-AM Performance (cache size is three times the Sight size).
Figure 18: Effect of Cache Flushes on SM Performance (cache size is three times the Sight size).
Figure 19. Fetch Unit Idle Time
Figure 20. Varying the CPU/Cache Access Speed Ratio
Figure 21. Hit Ratios for Three Level Hierarchy in a Batch Environment